Efficient Memory Allocator for Restricting Use-After-Free Exploitations

by

Ruizhe Wang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

© Ruizhe Wang 2024

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis is based on work co-authored with my supervisors Meng Xu and N. Asokan that have been accepted for ACM CCS 2024 [76] (Chapter 3) and DIMVA 2024 [75] (Chapter 4). I conducted all the design, implementation, and experimentation described in this thesis. All authors contributed to the writing of the papers [75, 76].

Abstract

Attacks on heap memory, encompassing memory overflow, double and invalid free, use-after-free (UAF), and various heap-spraying techniques are ever-increasing. Existing secure memory allocators can be generally classicied as complete UAF-mitigating allocators that focus on detecting and stopping UAF attacks, type-based allocators that limit type confusion, and entropy-based allocators that provide statistical defenses against virtually all of these attack vectors. In this thesis, I introduce two novel approaches, SEMalloc and S2Malloc, for type- and entropy-based allocation, respectively. Both allocators are designed to restrict, but not to fully eliminate, the attacker's ability, using allocation strategies. They can significantly increase the security level without introducing excessive overheads.

SEMalloc proposes a new notion of thread-, context-, and flow-sensitive "type", `SemaType`, to capture the semantics and prototype a `SemaType`-based allocator that aims for the best trade-off amongst the impossible trinity. In SEMalloc, only heap objects allocated from the same call site and via the same function call stack can possibly share a virtual memory address, which effectively stops type-confusion attacks and make UAF vulnerabilities harder to exploit.

S2Malloc aims to enhance UAF-attempt detection without compromising other security guarantees or introducing significant overhead. We use three innovative constructs in secure allocator design: **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.

This thesis demonstrates the importance of memory security and highlights the potential of more secure and efficient memory allocation by constraining attacker actions.

Acknowledgments

I would like to thank my advisors, N. Asokan and Meng Xu for their supports and my thesis readers: Yousra Aafer and Sihang Liu.

Dedication

To my loved ones.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Heap-related vulnerabilities are serious 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 respectively, in terms of bug prevalence [48].

Secure memory allocators are an important defense against heap exploitations. State-of-the-art secure allocators can be generally classified into three categories: *complete UAF-mitigating* allocators, *type-based UAF-mitigating* allocators and *entropy-based* generic memory allocators.

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 their security issues are related to UAF, more prevalent than other types of memory errors combined [68].

While complete mitigation of UAF is possible by virtually never re-using a previously freed memory block, UAF-mitigating allocators incur substantial overheads (e.g., about 40% for MarkUs [2]), and their complexity leaves them vulnerable to new attacks [78] (albeit preventable).

Type-based allocators assign a memory pool to each object type and only allows objects of the same type to be allocated to the same memory address. Specifically, they target on restricting the attacker's exploitation ability by limiting type-confusion. The attacker now can only use the dangling pointer to obtain an object of the same type.

However, the efficacy of type-based allocators requires a careful and precise tracking and annotation of object types. Existing works are limited in identifying allocation wrappers or tracking program execution semantics, causing type-confusion attacks still powerful.

Entropy-based heap allocators aim to provide comprehensive protection against most, if not all, common heap vulnerabilities with simpler designs but may fail with a small probability. Specifically on UAF mitigation, entropy-based memory allocators typically use delayed free-lists [40, 63] to prevent the same memory block from being *immediately* re-allocated after being freed. Attackers now face a moving-target even when they manage to 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 overhead on time usage, especially compared with UAF-mitigating allocators, existing entropy-based allocators still face the challenge of entropy-loss due to heap spraying, information leak, and silent failures on (potentially repeated) trials.

In this thesis, I show that the protection provided by both types of allocators can be bypassed, individually or even when the techniques are combined. I show that the design of secure memory allocator should take the attacker's ability into consideration and both types of allocators can be further improved by adopting this concept.

	Memory Overhead	Run-Time Overhead	UAF Attack Scope	UAF Success Rate
Complete Miti.	+ ~+++	+++	●	●
Complete Miti. (w/ Special HW)	+	+	●	●
Type-Based	++	+	◍	○
Entropy-Based	++	++	○	◍
SEMalloc	++	+	◍	○
S2Malloc	++	++	○	◍

Table 1.1: Illustration of existing secure memory allocators and the two proposed allocators. The number of + symbols indicates the amount of introduced overhead. The attack scope is progressively more limited from ◍ to ◍ and ●. The attack success chance gradually decreases from ◍ to ◍ and ●.

Contributions. I improve the state-of-the-art memory allocators of both types by limiting the attack ability. Specifically, I present S2Malloc (Chapter 4), a straightforward drop-in solution that makes an UAF attack *detectable* while also protects against all other commonly observed heap memory vulnerabilities, and SEMalloc (Chapter 3) a new type-based

memory allocator SEMalloc which uses `SemaType`, a carefully designed "type", to target a sweet spot between sensitivity (which decides security) and performance (which is affected by tracking overhead). Designs of these two allocators may be even adopted to a single allocator to limit both attack scopes and success rates. An illustration of existing and proposed secure memory allocators is presented in Table 1.1 that highlights the gap the two proposed allocators cover.

Chapter 2

Background

2.1 Common Heap Vulnerabilities

Common heap vulnerabilities include ❶ buffer overflow (reads/writes to an out-of-bound memory location), ❷ double free (frees an already-freed object), ❸ invalid free (frees an invalid pointer), and, as explained later in details, ❹ use-after-free (UAF). Successful exploitation of these vulnerabilities can cause heap corruption, leading to devastating results such as denial of service (DoS), information leak, arbitrary code execution, and/or privilege escalation.

2.2 UAF Attacks

UAF is a common heap error that occurs when a program unintentionally releases a heap object but continues to use the original pointer, i.e., a dangling pointer. Abusing the dangling pointer, an attacker can create powerful exploitation primitives such as arbitrary-address-write and code execution. A typical UAF exploitation involves the following steps:

- Trigger the vulnerable `free()` function to release a heap object A, turning the pointer P that points to the freed object into a dangling pointer.
- Look for a victim heap object B having ideally the same size as the released object. Victims containing interesting data such as pointers or length are usually preferred.
- Wait for the program to allocate a new B object that could refill the memory space initially belonging to A. The attacker can now corrupt the victim object B with pointer

P. This primitive can be used to either hijack the control flow or corrupt sensitive data such as `uid` or `gid`.

2.2.1 UAF Exploitations

UAF is generally considered as a *temporal* memory error, i.e., an error that occurs following a specific temporal order of events. In the context of UAF, the events include allocation (e.g., `malloc`), de-allocation (e.g., `free`), read, and write. Fortunately (or unfortunately), for most programs, there are plenty of such events in their original code logic; all an attacker needs to do is to find and trigger the correct sequence of events to mount an attack without code injection. Figure 2.1 is a crafted example to show how a dangling pointer can be exploited differently with different event orderings.

Formally, if a new object N (accessible through a fresh pointer p) is allocated over the heap location previously occupied by a freed object O (which leaves a dangling pointer q), then one of the following cases can happen:

A a read through p breaches the confidentiality of O, although this is usually called *uninitialized read*, which is generally not a concern of UAF and can be mitigated via zeroing allocations [25, 34] or other techniques [5, 45];
B a write through p breaches the integrity of O, as the written content can be subsequently read through q which can compromise the execution context where q is used;
C a read through q breaches the confidentiality of N which can be used to leak sensitive information such as pointer addresses (to break ASLR [4, 8]) or secret data
D a write through q breaches the integrity of N, as the written content can be subsequently read through p which can compromise the execution context where p is used;
E a free through q de-allocates N entirely, and yet, the heap allocator cannot block it if p and q are the same integer representing memory addresses (a free through p is legit).

Exploit B-E can all be found in Figure 2.1. Again, note that from an attacker's point of view, exploiting a UAF bug does not require code injection. Instead, an attacker can craft a "weird machine" [18] by merely re-purposing operations involving the inadvertent alias pair (p, q) in the original code logic. Intuitively, the more operations an attacker can re-purpose, the more useful a UAF can be in launching attacks. In the extreme case where any new object can be allocated over the heap location accessible by the dangling pointer q, this UAF is effectively an arbitrary read/write exploit primitive.

On a side note, Exploit-E is different from what is conventionally known as *double free* which arises when the old pointer q is freed twice without the allocation of a new heap

```
1  struct N {long usr; long pwd; int (*fn)(void);};
2  struct O {int (*oper)(void); long u1; long u2;};
3
4  void foo(long uid, long secret) {
5    struct N *p = malloc(sizeof(struct N));
6    p->fn = __safe_function_1;
7    p->usr = uid;
8    p->pwd = secret;
9    p->fn();
10 }
11
12 void bar(long user1, long user2) {
13   struct O *x = malloc(sizeof(struct O));
14   x->oper = __safe_function_2;
15   struct O *q = x;
16   free(x);           // q is dangling
17   q->oper();
18   q->u1 = user1;
19   q->u2 = user2;
20   reply("Users: %l | %l", q->u1, q->u2);
21   free(q);
22 }
```

Exploit-B: line 13–4–6–14 → arbitrary code execution.
Exploit-C: line 13–4–5–6–16 → information leak.
Exploit-D: line 13–4–15–7 → arbitrary code execution.
Exploit-E: line 13–4–17 → p is de-allocated and dangling.

Figure 2.1: A hypothetical example to illustrate different types of UAF exploits.

```
1  struct N {long usr; long pwd; int (*fn)(void);};
2
3  void register_real(long uid, long secret) {
4    struct N *p = malloc(sizeof(struct N));
5    p->fn = __real_fn;
6    p->usr = uid;
7    p->pwd = secret;
8    p->fn();
9  }
10
11 void register_fake(long uid, long secret) {
12   struct N *x = malloc(sizeof(struct N));
13   x->fn = __mock_fn;
14   struct O *q = x;
15   free(x);           // q is dangling
16   q->fn();
17   q->usr = uid;
18   q->pwd = secret;
19   reply("Debug: %l | %l", q->usr, q->pwd);
20   free(q);
21 }
```

Figure 2.2: A hypothetical example to illustrate UAF exploits against objects of the same type.

object N. Double free vulnerabilities can be mitigated cheaply by maintaining a set of freed and yet-to-be allocated memory addresses [40, 54, 63] as a top-up of other UAF-mitigation strategies.

Type confusion. The methodology to exploit the UAF bug in Figure 2.1 is also known as *type confusion* or *type manipulation*, which is arguably the most popular way to exploit a UAF bug, especially when an object type involved contains a function pointer. However, type confusion is not the only way to exploit a UAF; and more importantly, a UAF bug can be exploited even when the two objects involved have the same type, as shown in Figure 2.2. Despite the fact that both the victim pointer p and dangling pointer q share the same type, one can still leak sensitive data via p or cause `__real_fn` to be called in `register_fake` and vice versa.

Multi-threading and race conditions. Although Figure 2.1 and 2.2 are demonstrated in a multi-threaded setting, and indeed many exploits in the wild requires some form of race condition to work [33], multi-threading is not a strict requirement to exploit a UAF bug, as long as the attackers can find a similar sequence of events in a sequential execution, as showcased in [17, 28, 32, 49–51].

2.3 Secure Memory Allocators

In this section, we introduce three lines of secure memory allocators, from providing the strongest to the weakest security guarantee (UAF-mitigating techniques A1 - A3).

2.3.1 Complete UAF-Mitigating Allocators (A1)

This line of allocators prevent heap objects from being allocated over any pointer that might be dangling, which can be achieved by tracking pointer derivation [62] or sweeping all stored pointers [2, 19, 57]. In other words, these allocators do not trust the `free` request from developers; instead, they de-allocate memory only when "absolutely" safe. Hence, allocators in this category can achieve complete protection against heap-based UAF (modulo subtle pointer propagation flows [61]). However, they arguably introduce high overheads. For example, MarkUs [2] more than doubles the run-time on the PARSEC benchmark (see §3.6).

2.3.2 Type-based Allocators (A2)

This line of work prevent heap objects that an attacker targets (*victim objects*) **from being allocated over a dangling pointer**. This is essentially a weaker version of B1 and entrusts allocators to decide which class of objects should or should never be allocated on a specific free memory chunk. Intuitively, the ideal allocator would never place a victim object over a freed memory chunk with attacker-controlled dangling pointers.

Unfortunately, this ideal allocator cannot exist, as there is no way for an allocator to tell which object can be a *victim* (i.e., a valuable target for attackers) among all allocated objects, even with information from static or dynamic program analysis. Hence, in theory, perfect UAF-mitigation is not possible with this approach.

However, if an allocator knows enough about the *semantics* of allocated objects, it can place objects of different semantics into different and isolated pools. In this way, a dangling pointer of certain semantics controlled by the attacker can only be used to access newly allocated objects bearing the same semantics. This is commonly known as *type-based allocation* which makes UAF exploitation harder by confining what attackers can do after obtaining a dangling pointer. Not surprisingly, allocators in this category [3, 72, 78], differ on their definition of semantics or type.

PUMM [79] proposes that the completion of a "task" can be a clear signal to de-allocate freed memory accumulated in the ended task such that the freed memory can be re-allocated

in a new *task* which is irrelevant to any previous tasks. As tasks can have arbitrarily-defined boundary (e.g., one iteration of a loop is one task), PUMM effectively encodes temporal information into type and in theory, can be complementary to type-based allocators [3, 72].

2.3.3 Entropy-Based Allocators (A3)

Entropy-based heap allocators strive to provide protections against almost all of the above-mentioned heap exploitations by minimizing the attacker's success rate. Ideally, the success rate should be low enough to deter attackers from even trying to attack the system. However, practical implementations face limitations in terms of memory and computation resources, as demonstrated in Guarder [63].

BIBOP. State-of-the-art entropy-based allocators typically build on the *Big Bag of Pages* (BIBOP) [29] memory management mechanism with various security enhancements. BIBOP-style allocators classify allocation sizes as classes. For each size class, one or several continuous pages are treated as a bag, and all allocations of the same size class will be assigned to a corresponding bag. Each bag is split into several slots preemptively and each allocated object will take one of them. The status of each slot is monitored by a bitmap and can be used to defend against double or invalid frees. Bag metadata is stored separately to avoid metadata-based attacks [60], and UAF only occurs within the same size class.

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

- *Guard page*: is a single-page virtual memory block not mapped to the physical memory. Therefore, any attempt to dereference an address in a guard page triggers a segmentation fault. Guard pages could be strategically placed after each bag or randomly within bags to protect against overflows or random accesses.
- *Heap canary*: 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*: 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. More slots will be requested if there are not enough free slots to satisfy this requirement. Intuitively, the entropy (reflected by r) marks a trade-off between security and performance overhead.

Evolution history. While many entropy-based allocators have been proposed, we introduce three representative works that have contributed to advancing the field.

DieHarder [54] is one of the earliest entropy-based secure allocators that adopts the BIBOP-style memory management and provides statistical protections against heap exploitations. Despite incorporating all the security features mentioned above, DieHarder has several issues compared to more recent works. These include unstable allocation entropy, predictable guard pages, and a significant impact on the execution time of the hardened program.

Guarder [63] offers multiple improvements over DieHarder, such as using a linked-list-based free block management algorithm instead of bitmap traversal, providing stable allocation entropy, and offering in-bag tunable random guard pages. Guarder significantly reduces the run-time overhead to less than 3%, making it suitable for production systems.

SlimGuard [40] further extends Guarder by reducing its memory overhead. It divides size classes into 16-byte increments instead of powers of two. However, the current implementation of SlimGuard has two security compromises: it 1) allocates blocks sequentially in the free-list, violating the claim of random allocation. 2) reuses freed blocks to store metadata, possibly due to an implementation flaw, violating the argued metadata separation practice.

2.4 Other UAF-Mitigating Techniques

UAF can also be defended by invalidating dangling pointers §2.4.1 and validating the pointer upon derefernece §2.4.2. We label them as type-B and type-C approach to mitigate UAF attacks, respectively.

2.4.1 Invalidate Dangling Pointers (B)

This line of work breaks the foundation of any UAF exploits. In literature, this has been achieved via various creative techniques, including:

- Track pointer derivation at runtime and nullify all associated pointers upon object de-allocation [34, 61, 73, 80];
- Treat pointers as capabilities to access memory (instead of integers) and `free` revokes the capability [16, 22, 25].

Prior works in this category have demonstrated complete protection against heap-based UAF but may pay the price of compatibility (e.g., CHERI [77]), kernel privilege [25], high overhead (e.g., 80% run-time overhead for DangNull [34]), or subtle complexities as shown in HeapExpo [61].

2.4.2 Validate a Pointer Upon Use (C)

This line of work checks whether a pointer is safe for read/write operation upon dereference and can detect UAF attempts on the spot. Achieving this typically requires heavy instrumentation on instructions that may access memory through a pointer which significantly outnumber `malloc` and `free` operations. This explains the high overhead [37, 52, 58] even amongst the works designed to run in production [13, 21, 31, 35, 81] (e.g., 20% run-time overhead for Vik [13]).

2.5 Widely deployed secure memory allocators

The industry has adopted security designs into their allocators. Scudo [43] is a solution from LLVM [41] and supports random allocation and delayed free-list. Hardened_malloc [27] is the default allocator of GrapheneOS [26], a privacy and security-focused Android-based OS. While applying all aforementioned designs in entropy-based allocators and providing UAF-write protections, it suffers from at least reduced randomness and increased overheads for larger blocks, making the allocation predictable and limiting its use scenarios.

2.6 Summary

Prior works in categories A1, B, and C can mitigate all heap-based UAF attacks (assuming perfect implementation) but might also incur excessive overheads or require special hardware or kernel modification. Type- or entropy-based secure allocators (categories A2, A3) incur smaller overheads at the expense of incomplete protection. Compatible allocator designs may be stacked together to provide more complete defense. For example, A2 and A3 typed allocators can be used together to limit both the attack scope and success rate.

Chapter 3

Mitigation Approach: SEMalloc

3.1 Introduction

Heap vulnerabilities are common in memory unsafe languages like C and C++. Exploiting these vulnerabilities, attackers can inflict denial-of-service, information leakage, or arbitrary code execution. Use-after-free (UAF) is a typical class of heap vulnerabilities that have received special attention due to both its prevalence and the number and variety of powerful exploits it enables [66].

UAF happens when a memory chunk is accessed after it is freed. More specifically, freeing a heap object renders all pointers to this object (or parts thereof) *dangling*. Any memory access through a dangling pointer can lead to *undefined behavior* according to the C standard [11].

There is a wealth of prior research intended to address UAF vulnerabilities (see Chapter 2 for an exposition) and pros and cons can be found in each theme of UAF-mitigation techniques. For example, some allocators suffer from incomplete protection while others may incur prohibitively high run-time or memory overhead. While no allocation strategy is unquestionably superior in mitigating UAF vulnerabilities, *type-based* allocation, which permits the reuse of memory chunks only among allocations of the same type, seems to be a promising direction and is the focus of this paper.

Although type-based allocation provides imperfect protection only, the protection is more predictable than entroy-based allocators and more importantly, the protection can be achieved with reasonable overheads. However, existing type-based allocators are either coarse-grained in its definition of type [3, 72] leading to weaker protection, or extremely

fine-grained, treating each heap object as a different “type" [78], and leading to complete protection at a very high cost. Therefore, a gap remains in the design space for type-based allocators to balance between security and overheads.

The goal of this paper is to find a sweet spot in the design space of type-based allocation that achieves sufficiently high protection without excessive overhead. More specifically, we present S2MALLOC, a type-based UAF-mitigating allocator that operates on a new definition of **type** at its core:

> Two heap objects are of the same type *if and only if* they are (a) allocated from the same allocation site (e.g., a specific `malloc` call), and (b) the allocation call is invoked under the same call stack, modulo recursion.

To avoid confusion with the conventional notion of type in programming languages, we denote our "type" definition `SemaType`. For programs hardened with S2MALLOC, UAF can only occur between heap objects of the same `SemaType`.

S2MALLOC's run-time and memory overheads are low enough to make it suitable for real-world use. For instance, on SPEC CPU 2017, SEMALLOC incurs an average run-time overhead of -0.6% which is faster than MarkUs [2], MineSweeper [19], and DangZero [25] (by giving up protection against UAF within the same `SemaType`), and is similar to TypeAfterType [72] (with improved security) and PUMM [79] (with improved usability). SEMALLOC incurs an average memory overhead of 61.0% which is much lower than FFMalloc [78] (again, by giving up protection against UAF within the same `SemaType`) but is higher than TypeAfterType due to improved type sensitivity (hence security).

Summary. We claim the following contributions:

- A callout that the “type” in type-based heap allocator can be defined differently and does not need to be a native type in the programming language;
- The design and implementation of a new type-based memory allocator SEMALLOC which uses `SemaType`, a carefully designed “type”, to target a sweet spot between sensitivity (which decides security) and performance (which is affected by tracking overhead); and
- A thorough evaluation of S2MALLOC showing that it successfully detects all real-world attacks we tested with marginal overheads.

3.2 Rethinking Type

In programming languages, *“type”* is typically considered as a token that encodes some “semantics” of an object. As briefly discussed in §2.2, a type-based heap allocator confines

the types of objects a dangling pointer might ever access [3, 72, 78]. More specifically, if a freed object is of type $\mathbb{T}$, only objects of the same type $\mathbb{T}$ can then be allocated over the free chunk. Intuitively, type-based allocation provides a *tunable* defense against UAF with a clear security and performance trade-off—all by varying the definition of "type".

While there are many ways to define types (hence the research on type systems [12]), one particularly useful angle in the context of type-based allocation is the *sensitivity* of a type, i.e., how well a type can distinguish heap allocations occurring under different execution states. The insight is that: **objects allocated under the same or similar execution states are expected to behave similarly in the program**, and such behaviors are essentially the semantics of the objects, which serve as the "type" in type-based allocation.

Borrowing sensitivity notions from program analysis, we can define type sensitivity from the following perspectives:

Flow-sensitive. If a function is invoked in two places within the same function, a flow-sensitive type will differentiate these two function calls. To illustrate, in the code below, the two `malloc` calls are different under a flow-sensitive scheme.

```
1 void foo() {
2   void *p = malloc(sizeof(int));
3   void *q = malloc(sizeof(int));
4 }
```

Path-sensitive. If a function is reached via different control-flow paths within a function (modulo loop), a path-sensitive type will differentiate these two execution paths. To illustrate, in the code below, the `malloc` call might allocate objects of different types depending on the boolean `cond`.

```
1 void foo(bool cond) {
2   size_t len = sizeof(int);
3   if (cond) {
4     len = sizeof(long);
5   }
6   void *p = malloc(len);
7 }
```

Context-sensitive. If a function is reached via different call traces (modulo recursion), a context-sensitive type will differentiate these two calling contexts. To illustrate, in the code below, the `malloc` call under contexts [`foo` → `wrapper`] and [`bar` → `wrapper`] allocate objects of different types.

```
1 void wrapper(size_t len) {
2   void *p = malloc(len);
3 }
4 void foo() {
5   wrapper(sizeof(int));
```

```
6 }
7 void bar() {
8   wrapper(sizeof(int));
9 }
```

Thread-sensitive. If a function is invoked in different threads, a thread-sensitive type will differentiate the threads. To illustrate, in the code below, the `malloc` call under the two threads allocates objects of different types.

```
1 void *thread(void *ptr) {
2   void *p = malloc(sizeof(int));
3 }
4
5 void foo() {
6   pthread_t t1, t2;
7   pthread_create(&t1, NULL, *thread, NULL);
8   pthread_create(&t2, NULL, *thread, NULL);
9 }
```

Sensitivity in cyclic control-flow structures. In loops and recursive calls, the sensitivity are typically classified as:

- *Unbounded*, where different iterations of a loop or recursion yield different types.
- *Bounded*, where different iterations of a loop or recursion yield different types, up to a pre-defined limit.
- *Insensitive*, where different iterations of a loop or recursion yield the same type.

Finding the right sensitivity level. For a type-based heap allocator to be secure yet practical, finding the right sensitivity level is the key.

A type definition with higher sensitivity implies a smaller set of object types a dangling pointer may possibly point to. In this regard, the `glibc` allocator [24], the default allocator in most Linux-based systems, is (almost) completely insensitive. Regarding the two closely related works in this field, the type definition of Cling [3] is thread-sensitive and adopts a weaker form of context-sensitivity (the context is defined by stack pointer address, which is an approximation to call trace). Type-after-Type [72] is based on statically-inferred unqualified types native to the programming language without sensitivity add-ons.

And yet, for a type-based allocator, it is not necessarily true that more sensitivity is better. To illustrate, the type definition with the highest sensitivity is to treat every heap object as a different type. This effectively means that a heap allocator will never reclaim memory—an impractical approach, as a long-running program may allocate and free an endless number of heap objects yet the virtual memory address space has a limit (e.g., 48-bit on x86). FFMalloc [78] is a close approximation to this extreme approach and incurs a large memory overhead despite the fact that it still reclaims virtual pages. A path-,

context-, and thread-sensitive type qualifier will be extremely sensitive as well, and yet, tracking path sensitivity requires instrumentation at basic block granularity, which adds a significant overhead.

Conclusion. While a multitude of prior work has tried to address UAF vulnerabilities, the challenge of finding the right balance between the level of protection and incurred overhead remains. In the rest of this paper, we present our approach towards finding such a balance.

3.3 Capture Semantics with `SemaType`

We now introduce `SemaType`, a type qualifier [23] tailored to capture the semantics of heap allocations, and showcase how to deduce `SemaType` at runtime through a concrete example.

3.3.1 Defining `SemaType`

`SemaType` is a *thread-, context- and flow-sensitive type qualifier* over the standard type system of the underlying programming language (e.g., LLVM IR in SEMalloc) with *bounded sensitivity for recursions* and *no sensitivity for loops* (sensitivity levels defined in §3.2).

Informally, in a more operative description, two heap objects are of the same `SemaType` *if and only if* they are:

- allocated from the same allocation site (e.g., the very same `malloc` call in the source code); and
- allocated under the same call stack, modulo recursion.

In the presence of recursive calls, `SemaType` differentiates call traces inside each strongly connected component (SCC, representing a group of recursive calls) in the call graph up to a fixed limit In SEMalloc, this bound is 2^{14} different call traces overall (see Figure 3.5).

Deducing `SemaType`. In theory, the `SemaType` of every heap allocation can be deduced at compile-time by inlining all functions, converting recursive calls to loops, and creating a huge `main` function. This, however, is impractical for any reasonable-sized real-world program as analyzing a huge function can be both time- and memory-intensive in current compilers while aggressive inlining results in large binaries. Distinguishing heap allocations by thread (i.e., thread-sensitivity) at compile-time further adds complexity, as it requires more extensive function cloning to differentiate per-thread code statically.

Fortunately, `SemaType` can be deduced at runtime as well, at the cost of code instrumentation (and hence, overhead). More specifically, the dynamic deduction of `SemaType` can be facilitated with context-tracking logic automatically and strategically instrumented at compile-time.

A concrete example. To illustrate how `SemaType` can be deduced, we use the simple example in Figure 3.2. The code snippet is shown in Figure 3.1 and the figure is a conventional call graph of the program enhanced with (1) flow-sensitive edges (e.g., two edges from `a` to `e` marked as `[l] and [r]` respectively) and (2) annotations on whether the call occurs inside a loop or not (i.e., dashed vs solid edges).

3.3.2 Cyclic Control-flow Structures

Due to the existence of a recursive call group (the SCC), there are an unlimited number of call traces that can reach `malloc` from `main`. This is why we cannot enumerate all call traces to assign each call trace a `SemaType` statically. And yet, even we can track the call context at runtime, having an unlimited number of `SemaType`s for this program is not desirable either, because such an approach is, in the worst case, the same as giving each allocated heap object a different `SemaType`. As discussed in §3.2, this can be overly sensitive and may cause a significant memory overhead as in FFMalloc [78].

How can we fit unlimited number of call traces into a fixed number of `SemaType`s? We considered two simple solutions:

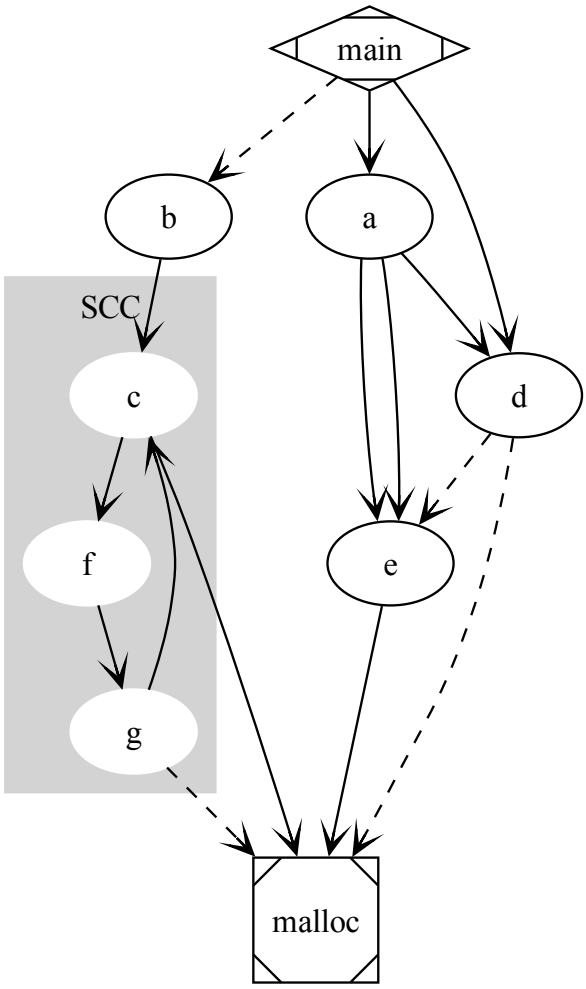
- *Bounded unrolling*: unroll the SCC to a limited depth and treat each `malloc` called from the unrolled iterations differently. Beyond the unrolled iterations, assign a single `SemaType` to the `malloc` called inside this SCC.
- *Aggregation-based hitmap*: aggregate the call trace inside the SCC to a fixed number of bits and call traces bearing the same aggregated value are considered to have the same `SemaType`.

SEMalloc uses the aggregation-based hitmap solution as it provides a slightly better security by distributing `SemaType` more uniformly across different rounds of recursion.

However, `malloc` calls occurring in different loop iterations are not differentiated by `SemaType`. Differentiating loop iterations will require path-sensitive instrumentation, i.e., instrumentations (and hence overhead) linear to the number of basic blocks; while differentiating iterations in recursive calls only requires instrumentations linear to the number of functions, which is arguably significantly smaller in most real-world programs.

```c
int main() {
  for (...) {
    b(32);
  }
  d(sizeof(int));
  a(16);
}
void a(int i) {
  d(sizeof(int));
  e(i * 16);
  e(i * 16);
}
void b(int i) {
  c(i, 64);
}
void c(int i, int s) {
  if (i > 1)
    c(i, 64);
  else
    malloc(sizeof(int));
}
void d(int size) {
  for (...) {
    malloc(size);
    e(16);
  }
}
void e(int size) {
  for (...)
    malloc(size);
}
void f(int i, int s) {
  g(i, s - 16);
}
void g(int i, int s) {
  c(i - 1, s);
  for (...)
    malloc(s);
}
```

Figure 3.1: Code snippet that yields the call graph in Figure 3.2



`SemaType` without recursive calls

A `main→a→e[l]→malloc`
B `main→a→e[r]→malloc`
C `main→a→d→malloc`
D `main→a→d→e→malloc`
E `main→d→e→malloc`
F `main→d→malloc`

`SemaType` with recursive calls

G `main→b→c→malloc`
H `main→b→c→f→g→malloc`
I `main→b→c→f→g→ c→malloc`
J `main→b→c→f→g→ c→f→g→malloc`
K `main→b→c→f→g→ c→f→g→c→malloc`
. .....
. .....
* `main→b→c→···→malloc`

Figure 3.2: Call graph (left) of a crafted program Figure 3.1 illustrating how `SemaType` (right) can be deduced. In this call graph, each node is a function and solid edges represent function calls not in a loop inside the function control-flow graph (CFG) while dashed edges represent function calls inside a loop.

3.3.3 `SemaType` Representation

`SemaType` can be represented as a composition of two values:

- `nID`: a non-recurrence identifier representing top-level call traces in the *directed acyclic* call graph, which is built by abstracting each SCC in the call graph into a node;
- `rID`: a recurrence identifier for call traces within an SCC.

The `nID` and `rID` for the current execution context are both tracked at runtime through global variables. Their values are merged together to form a `SemaType` when the execution is about to invoke a memory allocation function (e.g., `malloc`).

We assign each call site outside SCCs with a *weight* (§3.4.3). Before making a call, `nID` is incremented by the weight of the call site and decremented by the same weight upon return. This rule for `nID` generalizes to a stack of calls as well. Operationally, `nID` is the cumulative weight of all call sites in the call stack when a heap allocation happens. Our weight assignment algorithm (§3.4.3) ensures that two `SemaType` instances have the same `nID` if and only if their external SCC traces are identical (formally proved in §3.5.2).

`rID` is for intra-SCC call stack tracking. Unlike `nID`, `rID` is an aggregated value of what happened inside an SCC. `rID` is tracked with two global variables s and h, where

- s is a stack that hosts the stack pointers before a function within an SCC is called (a.k.a., a call stack), and
- h is the aggregation of stack s, representing the `rID` (§3.4.4). h is computed and stored before an SCC function calls a function not in the current SCC (outbound call), and will be cleared after the return of the initial function call to this SCC (inbound call).

Repetitive allocation. A `SemaType` only needs to be tracked if heap objects of this `SemaType` can be allocated repetitively. For one-time allocations, i.e., a `SemaType` that can only be reached in one call stack where none of the call site is in a loop (see §3.6.5 for evidence that this is rare), once an one-time object is freed, its space is never reused. Therefore, we optimize `SemaType` tracking only to those instances where re-allocation is possible, identified by the presence of at least one recursive call site in their allocation traces.

We keep track of the recursive depth l by incrementing it before executing an iterative function call and decrementing after it. Upon memory allocation, if l is not zero, we can conclude that this `SemaType` object may be recurrently allocated. l is also increased before a non-SCC function calls a SCC function (inbound call) and decreased after it.

Illustration. Revisiting the example in Figure 3.2, only **A** and **B** are non-repetitive; all other `SemaType`s need to be tracked: types **C** and **D** are repetitive because of looping while other types are repetitive due to involvement in recursive calls.

We take **F** as a case study for variable management. the `nID` is increased before calling `c` and `malloc`. The call stack s holds three stack pointers, pushed into it before each SCC function (`c`, `f`, and `g`) is called. Upon calling `malloc`, the `rID` (i.e., h) is computed. The `b`→`c` function call enters a SCC, causing l to be incremented. Upon calling `malloc`, l is non-zero indicating a recurrent allocation.

Thread sensitivity. Note that thread identifiers are not discussed here in the representation of `SemaType` despite the fact that `SemaType` is thread-sensitive. This is because the backend heap allocator does not need this information to be deduced through compiler-instrumented code at runtime. Instead, it can be queried directly by the backend allocator via a system call (e.g., `syscall(__NR_gettid)`) or even one assembly instruction if the platform supports. As a result, we do not specifically encode a thread ID in the `malloc` argument passed to the backend allocator (see §3.4.5).

3.3.4 Alternative: Path-sensitivity

`SemaType` is not path-sensitive. Although a *thread-, context- and path-sensitive type qualifier* is intriguing, we have to weaken path- to flow-sensitivity for practical reasons:

- Within a function control-flow graph (CFG), paths exponentially outnumber CFG nodes (the latter is captured by flow-sensitivity), hence adopting a path-based `SemaType` will bloat the number of `SemaType`s and allocation pools.
- Deducing execution paths requires either dynamic CFG branch tracking (non-trivial run-time overhead) or static function splitting, e.g., assign different `SemaType`s to the same `malloc` based on whether function arguments satisfies predicate X, except that devising X is undecidable.
- Empirical evaluation (§3.5) shows that `SemaType` in its current form is sufficient to defend against known exploits.

3.3.5 Instruction Insertion Summary

The number of LLVM IR instructions instrumented at different code locations are summarized in Table 3.1.

Briefly, following is a summary of instructions added:

Code location	Call	Invoke
SCC inbound edges	1	2
SCC inner edges	9	13
SCC outbound edges	16	22
Iterative node	6	12
Branch node	6	20
`malloc` call site	12	20
Duplicated invoke node	2*calls in the same bracket	

Table 3.1: Number of instructions inserted for `call`, `invoke`, and for duplicating the invoke nodes. In the call graph, we use "branch node" to denote a node with more than one incoming edges and "iterative node" to denote a node that has at least one outgoing edge annotated in dashes (i.e., the call site is in a loop). We note that a branch node can potentially also be an iterative node. In this case, both groups of instructions will be inserted.

- For an SCC inbound edge, instructions are inserted after the call site to clear s.
- For an intra-SCC edge, instructions are inserted before and after the call site to update s.
- For an SCC outbound edge instructions are inserted before the call to compute h (which is `rID`) and clear s.
- For an iterative node, instructions are inserted before and after the call to maintain `nID`.
- For a branch node, instructions are inserted before and after the call to maintain `nID`.
- For a `malloc` call site, instructions are inserted before the call to encode `rID` and `nID` into the size parameter.

Additionally, if a function is called with exception handling (via the `invoke` instruction in LLVM), additional instructions need to be inserted to handle the unwind branch and to duplicate the execution logic to make it compatible with S2Malloc. We refer the readers to §3.3.6 for details.

3.3.6 Transformation for Function Call with Exception Handling

In LLVM, regular function calls are represented with the `call` instruction. This instruction is similar to a regular function call in high-level programming languages and does not encode exception handling semantics. For calls that may throw an exception, LLVM uses the `invoke` instruction.

Different from the `call` instruction that returns the control flow to the next instruction, `invoke` terminates the control flow and jumps to two destinations that contains the regular branch and the exception handling branch (a.k.a., the unwind branch). If more than one function calls are made in one exception-handling context (e.g., more than one functions calls in the same `try` block in C++), there is still only one unwind branch that all `invoke` instructions will jump to.

When making a regular call, `nID` is decreased after the `call` returns. With the `invoke` instruction, `nID` needs to be decreased in both destination branches. The unwind branch also needs to be exclusive to each `invoke` as `nID` needs to be reduced with a different value in different sites. To achieve this, we duplicate the unwind branch and guarantee that each branch is only jumped from one `invoke` instruction.

The unwind branch might contain ϕ-instructions, whose return value is dependent to the prior basic block the control flow jumped from. To make the transformation compatible with this special instruction, we only duplicate the basic exception handling logic (first half of the unwind basic block) and insert instructions to reduce `nID` for the unwind branches here. We create a new basic block only contains the second half of each branch, and all ϕ-instructions are in the newly created basic block.

Similarly, a `invoke` destination block can have incoming edges from basic blocks that do not end with the `invoke` instruction. We need to duplicate this destination block as well to avoid always executing the inserted tracking instructions even this basic block is not jumped from a `invoke` call site. We create a new basic block and insert the `SemaType` tracking instructions here. We then replace the invoke destination to this block and link this block with the old destination, while update all ϕ-instructions accordingly.

3.4 `SemaType`-based Heap Allocation

In this section, we describe the design and implementation details of SEMalloc—a `SemaType`-based heap allcator for mitigating UAF vulnerabilities. We first introduce our threat model and explain how SEMalloc realizes dynamic `SemaType` deduction and allocates memory accordingly.

Threat model. We assume that (a) the underlying operating system kernel and hardware are trusted, (b) the targeted program is uncompromised at startup, and (c) the attacker can obtain and analyze the source code and the compiled binaries of both the targeted program and SEMalloc. Exploiting implementation bugs in SEMalloc or utilizing side-channel information, such as power usage or memory fetch time, are left out of scope.

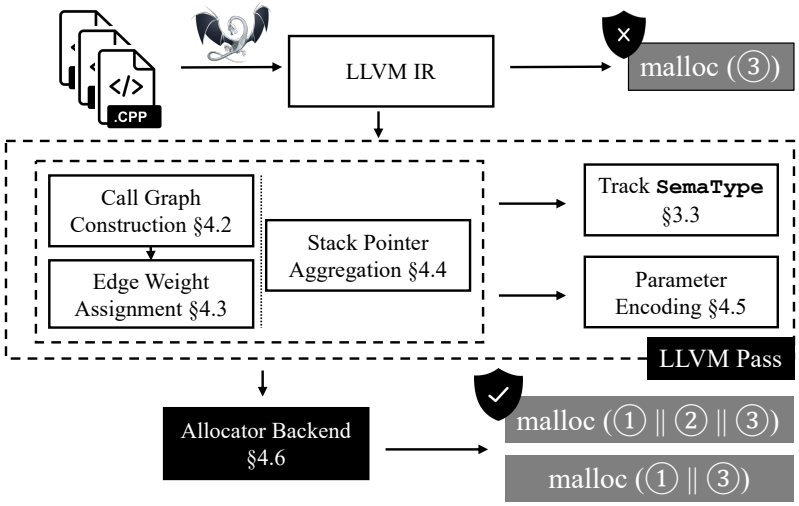


Figure 3.3: Design overview of SEMalloc (①: flags, ②: `SemaType`, ③: allocation size). The size is the parameter without SEMalloc, while SEMalloc encodes the trace information into the parameter after applying the pass.

3.4.1 Overview

SEMalloc consists of an LLVM transformation pass and a heap allocator backend. The LLVM pass analyzes the intermediate representation (IR), inserts instructions to create and instrument the tracking variables, and encodes the allocation parameters with `SemaType`-tracking information. The LLVM pass is built on top of MLTA [44] (for comprehensive and robust call graph construction) and CXXGraph [9] (for graph algorithms). The pass instruments `SemaType` tracking and encoding in the program which eventually passes `SemaType` through common heap allocation APIs, including `malloc`, `calloc`, `realloc`, `memalign`, `pthread_memalign`, and `aligned_alloc`. The heap allocator backend takes the encoded information for segregated memory allocation.

Figure 3.3 gives a comprehensive overview of SEMalloc. In the transformation pass, SEMalloc first constructs a call graph that only contains functions (nodes) and call sites (edges) relevant to `SemaType` that need to be tracked (see §3.4.2), and assigns weights on all edges and nodes in it for `nID` computation (see §3.4.3). In the call graph, an SCC is treated as a function node, and call traces within it are not considered by `nID`. Instead, intra-SCC calls are tracked in `rID` by obtaining and aggregating the stack pointers with an aggregation algorithm (see §3.4.4). They are encoded into the `size` parameter of an allocation request (see §3.4.5).

In terms of the heap allocator backend, we explain how it enforces the allocation segregation policy using the decoded `SemaType` (see §3.4.6). For simplicity, we use `malloc` to represent all functions that may request heap memory *directly* from the backend. We refer readers to §3.3.5 for a complete discussion about how the IR is transformed after applying the pass and how instructions are inserted.

3.4.2 Call Graph Construction

We start by building a call graph for the program to be hardened by SEMALLOC. While call graph is a foundational concept with mature support in modern compilers, the call graph in SEMALLOC is slightly more complicated in two aspects:

1) Flow-sensitive edges. If function `e` is called in two places by function `a`, there will be two edges from `a` to `e` in the call graph, as shown in the example in Figure 3.2.

2) Indirect calls. SEMALLOC takes special care for indirect calls whose call targets cannot be resolved at compile-time and hence do not show up in a conventional call graph. To handle indirect calls, SEMALLOC first identifies all callee candidates via MLTA [44]. MLTA uses a multi-layer type hierarchy to refine indirect call targets and is based on the observation that function pointers are stored in complicated hierarchical structures. It matches the load instructions of each layer and take functions have the same multi-layer structure type as call candidates. Subsequently, for each callee candidate identified, SEMALLOC adds an edge in the call graph and treat different callee candidates as if they are called in different places in the calling function. This is a conservative treatment for indirect calls and can lead to more `SemaType`s being derived than necessary which can result in a better security but a larger memory overhead.

Additional trimming and marking. With a baseline call graph, the next step is to remove nodes and edges that are irrelevant to `SemaType`, i.e., paths that do not eventually lead to a `malloc`. We also mark call sites that occur in a loop in the caller function (e.g., dashed edges in Figure 3.2) in order to distinguish recurrent allocations vs one-time allocations (see §3.3.2). Such a marked call graph enables SEMALLOC to optimize instrumentation to recurrent `malloc`s only. We remove all nodes or edges that are not (eventually) called by or (eventually) call any recurrent edge. This call graph contains only nodes and edges that eventually call `malloc` while each edge leads to at least one recurrent `SemaType` object.

Finally, we use the Kosaraju-Sharir algorithm [59] to identify SCCs and create a new call graph with each SCC being abstracted as a single node. In this way, the new call graph

is essentially a directed acyclic graph (DAG) while recursions (i.e., intra-SCC paths) are handled using `rID` (see §3.4.4).

3.4.3 Edge Weight Assignment

Recall from §3.3.3 that `nID`, which is part of the representation for `SemaType`, serves to distinguish different call stacks that end up with `malloc` modulo recursions in SCCs. As `nID` is calculated as the sum of weights per each edge in the path, these weights need to be assigned strategically to ensure that different paths yields different `nID` values.

To assign weights, we run a topological sort on the DAG for a deterministic ordering of functions and then go through each function to assign weights according to algorithm 1.

Algorithm 1: Edge weight assignment

```
nodes ← topological_sort(DAG)
for each n ∈ nodes do
    w ← 0
    for each e ∈ n.outgoing_edges do
        e.weight ← w
        w ← w + max(1, e.dst.weight)
    end
    n.weight ← w
end
```

We maintain two weights while going through each function: the function weight and the call-site weight. The function weight describes how many different `SemaType`s exists if taking this function as the program entry point. The call-site weight is the sum of the weights of all functions called before it within the function. It describes how many different `SemaType`s all previous call sites of the current function lead to. More specifically, it is an offset that guarantees that all `SemaType`s allocated through the current call site have their `nID` larger than all previous `SemaType` `nID`s to avoid collision. For example, in a function, the path weight of the first call site is zero, and the weight of the next call site is the weight of the first callee function (note the minimum weight is one, line 4 - 7). As long as the offset is computed correctly, no collision will happen.

After processing all call sites, we assign the weight of the current function as the sum of the weights of all its callees (line 8). Using the topological order, we guarantee that all

callee weights are computed before they are needed; we set the weight of `malloc` to zero as it does not call any function.

Note that weight assignment (algorithm 1) ensures a one-to-one mapping between a `nID` and an end-to-end path that reaches the `malloc` in the call DAG (see §3.5.2 for a proof). It is worth-noting, however, that tracking the path at runtime directly is possible but would incur a slightly higher overhead than tracking the `nID`, which only involves two arithmetic operations per each call site.

Optimization. To further minimize the instrumentation needed, a node can be removed from the call graph if it has only one incoming edge, i.e., the function `f` (represented by this node) is only called in one place. Essentially, removing the node has the same effect as inlining `f` into its caller (without actually transforming the code). In this situation, the call site that invokes `f` does not need to be instrumented for `nID`-related logic. And this optimization repeats until we cannot find such `f` in the call graph.

Example. We present an example of the allocated weights of a crafted program in Figure 3.4.

3.4.4 SCC Stack Pointers Aggregation

We balance between performance and security, and use an aggregation approach to track the execution path within SCCs. Before calling each function within the SCCs, we obtain the stack pointer using an LLVM intrinsic function, which is then pushed into the stack s, and will be the aggregation input to compute `rID`.

Algorithm 2: Algorithm to aggregate the stack pointers.

```
h ← 0
for each p ∈ s do
 |  h ← h ≪ 2 ;   p ← (p ≫ 6) & 0x3 ;   h ← h + p
end
h ← h & 0x3FFFF
```

`rID` is computed using algorithm 2. Initially, we set it to be zero (line 1). We then go through each stack pointer by adding the seventh and eighth least significant bits of each input to it and shift it to left by two bits (line 3-5). We specifically take these two bits as stack pointers are 8-byte aligned in the x86 clang environment [42], and we select those bits that are not identical in different call frames. Finally, we only keep the least fourteen bits of the aggregated value, which represents the most recent seven functions called within SCCs.

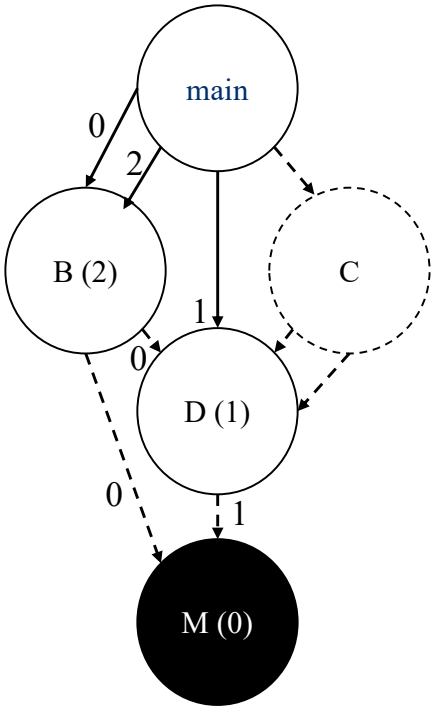


Figure 3.4: Weight assignment of a crafted example program. Dashed lines refer to one-time function calls and solid lines refer to iterative function calls. Numbers on the edges refer to edge weights and numbers in the function nodes are function weights. For example, the `nid` of `main` → D → `malloc` is 2.

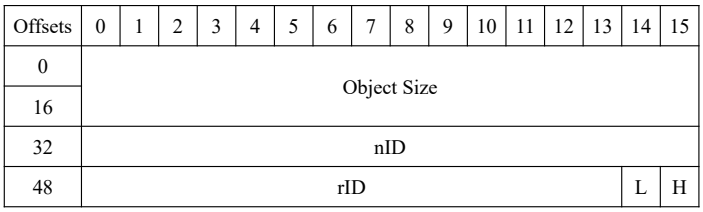


Figure 3.5: Parameter encoding rule for regular objects(L: loop identifier; H: huge block identifier).

We note that as a stack pointer is dependent on the call depth and all calls that are not returned, this algorithm accounts for the entire call trace without losing function calls older than the most recent seven.

3.4.5 Parameter Encoding

The heap allocator backend requires two pieces of information as input: allocation size (as required by all memory allocators) and `SemaType` (unique information in SEMalloc), and allocates heap objects based on them. While standard memory allocation APIs already accept the allocation size as a parameter, we need to find a way to pass `SemaType` to the backend. And SEMalloc, conceptually, has two options:

- Changing the `malloc` signature: This would involve adding a new parameter to the existing interface and hence, introducing a new function signature like `malloc(size_t size, void *semantics)`.
- Repurposing the `size_t` parameter type: This implicitly change the type of the `size` parameter with `SemaType` encoded alongside the existing size.

In SEMalloc, we take the second approach for compatibility with the existing allocation interface, and encode `SemaType` within the `malloc size` parameter using the format shown in Figure 3.5 for blocks smaller than 4GB.

We set the loop bit (L) if the number of loop layers (l) is not zero to notify the backend that it might reuse the memory freed by another object. We store the `nID` and `rID` accordingly as the `SemaType`, and we use the remaining 32 bits to store the size of the allocated object.

For larger blocks, we set the huge-block bit (H) and use all the remaining bits to store the block size. These blocks are allocated via a system call, and launching UAF attacks on them is not trivial (see §3.4.7 for details).

This design is compatible with legacy code or external libraries that are not transformed by our pass with function allocation call size up to 4GB. However, memory allocated this way do not have the loop identifier set, is not going to be released unless the block is big enough that allocated from the OS directly (see §3.4.7 for details). This indeed is not common as shown in Table 3.9 and Table 3.10, where most tests have more than 99% allocations identified with recurrent `SemaType`.

3.4.6 Heap Allocator Backend

The backend heap allocator for SEMalloc is packaged as a library that can either be preloaded at loading time or statically linked to replace the default allocator. The backend extracts and decodes the `SemaType` packed in the `size` parameter and enforce `SemaType`-based allocation by allocating objects of different `SemaType`s from segregated pools.

More specifically, SEMalloc backend adopts the BIBOP [29] style to manage block allocation inside each `SemaType` pool. BIBOP allocates blocks of the same size class together using one or more continuous memory pages, and preemptively allocate sub-pools for each size class. A block is not going to be further split or coalesced. SEMalloc is built upon this design. For each thread (hence thread-sensitivity §3.2) it allocates a global BIBOP pool for all one-time `SemaType`s and allocate individual pools for different `SemaType`s upon seeing a recurrent request for the same `SemaType`. SEMalloc uses power of two size classes, for example, all blocks with the same `SemaType` and of size 65 to 128 bytes will be allocated to the same pool.

Operationally, upon receiving a heap allocaiton request, the backend first checks the huge bit and, if applicable, allocates huge blocks using the `mmap` [38] system call. For regular blocks, if the loop bit is not set, SEMalloc will allocate it using the global pool, and it will never be released even after it is freed. If the loop bit is set, SEMalloc allocates it using the global pool if the `SemaType` is seen for the first time and otherwise create an individual pool dedicated for all following allocations with this `SemaType` A freed block in the individual pool can be reused by later allocations with the same `SemaType`.

3.4.7 Implementation Details of The Allocation Backend in SEMalloc

The heap allocator backend of SEMalloc is implemented using the `dlsym` function and is a dynamic library that can be loaded by either setting the `LD_PRELOAD` environment variable

to replace the system default memory allocator or direct linkage during compilation.

SEMalloc maintains a per-thread metadata that stores the status of all blocks allocated in this thread. If a block not allocated in this thread is freed, SEMalloc will atomically add this block to the free-list of the thread that allocates it and defers the deallocation until that thread handles a heap memory management operation. Other than the free-list, for each thread SEMalloc maintains a global pool for all one-time allocations, a lazy pool for all first-seen recurrent allocations, several individual pools for each recurrent allocations, and a map that used to locate the pool for each `SemaType`. The lazy pool and global pool are pools for all size classes, while individual pools contain a limited set of size classes (most of the time only one) that have been used to save virtual memory address space.

Upon creation, each pool is allocated with a dedicated virtual memory address range that all its memory will be allocated from. For each sub pool of the global pool or the lazy pool, SEMalloc allocates the block sequentially. For each individual pool, SEMalloc maintains a free list as well and will allocate the head of the free list if it is not empty. Otherwise, SEMalloc will also allocate the mapped memory of this individual pool sequentially.

For each block, a 16-byte metadata is stored immediately before the data. For huge block, it stores the block type (huge) and the block size. For regular small blocks, it stores the block type (regular) with one byte, the ID of the thread that allocates the block with two bytes, the pointer to the pool that allocates the block with eight bytes, and an offset if the block is allocated via `memalign`-like functions to locate the start byte of the block chunk with four bytes.

When a `malloc` call comes, SEMalloc firstly check if the block size is larger than the huge allocation threshold or not. If so, SEMalloc uses `mmap` to allocate this block and sets the header. Otherwise, SEMalloc takes the recurrent identifier. If it is not set, SEMalloc will use the global pool to allocate this block. Otherwise, SEMalloc will take `SemaType` and check if a block with it is already allocated or not using the lazy pool. If so, SEMalloc can confirm that this blocks with this `SemaType` is recurrently allocated and will allocate an individual pool for all following allocations with this `SemaType`. The map will be updated with this new entry as well. Otherwise, the lazy pool will allocate this block.

When a `free` call comes, SEMalloc will firstly take the header of the pointed block to check if it is a huge block or not If so, SEMalloc uses `munmap` to deallocate this block. Otherwise, SEMalloc will check the thread id and put it to the corresponding free list if this block is not allocated in this thread. If the block is allocated in this thread, SEMalloc then takes the pool pointer. If the pool is a global pool or lazy pool, SEMalloc will

release the taken memory to the operation system by calling `madvise`. However, this virtual memory address will not going to allocated to any blocks again. If the pool is an individual pool, SEMalloc will put this block to the pool's free list, recycling it for another block with the same `SemaType`.

3.5 Security Analysis

3.5.1 Qualitative Analysis

The key reason why type-based allocators cannot deliver perfect UAF mitigation is UAF within the same type. More specifically, to S2Malloc, this means UAF within memory objects marked with the same `SemaType`—and this is not only possible but also common due to recurrent allocations, i.e., `malloc` inside a cyclic control-flow structures such as loops or recursive calls (§3.3.2). On the other hand, memory reuse is crucial in reducing memory footprint. An allocator that places each object into a new pool and never reclaim memory is immune to UAF at the cost of a high memory waste. Therefore, intuitively, the more recurrent allocations a program have, the less effective SEMalloc is in mitigating UAF exploits, but the greater the memory saved by S2Malloc, compared to allocators that never free memory.

In this section, we sketch a qualitative explanation on how loops and recursive calls affect the security of S2Malloc.

Setup. Assume a program has N allocation sites:

- each allocation site $i \in 1..N$ can be reached via P_i traces in the flow-sensitive call graph after CFG-reduction (§3.4.2);
- each trace $T_{i,j}$ (where $j \in 1..P_i$) contains $R_{i,j}$ nodes that are reduced from call graph SCCs, i.e., recursive calls (§3.4.4);
- k out of N sites are in a loop w.r.t a function-level CFG.

SEMalloc assigns one `nID` to each trace $T_{i,j}$ and up to $2^{\#\text{bits}(rID)}$ `rID`s per trace. Thus, this program will have:

- a minimum of $\sum P_i$ `SemaType`s, the minimum occurs when the program does not contain any recursive calls, or
- a maximum of $2^{\#\text{bits}(rID)} \times \sum P_i$ `SemaType`s, the maximum occurs when all traces $T_{i,j}$ have recursive calls.

UAF-protection in different scenarios. We discuss how UAF protection in S2MALLOC can be weakened by recurrent allocations with reference to complete UAF mitigation:

- *No recurrent allocation* ($k = 0$ and all $R_{i,j} = 0$): SEMALLOC provides perfect security against UAF, a similar level of protection as complete UAF-mitigating allocators since no memory reuse exists. SEMALLOC provides strictly more protection than existing type-based allocators: in the worst case, all $\sum P_i$ `SemaType`s can the same C/C++ type which will be allocated from the same pool.
- *With `malloc` in loops* ($k \neq 0$ and all $R_{i,j} = 0$): SEMALLOC provides weaker security than complete UAF-mitigating allocators as UAF is possible within the same `SemaType` in one of the k loop allocations. Higher k means weaker UAF protection, but a smaller memory footprint. Regardless of k, SEMALLOC provides strictly more protection than existing type-based allocators, as shown in §3.2.
- *With `malloc` in one group of recursive calls only* ($k = 0$ and all $R_{i,j} = 0$ except $R_{a,b} = 1$): SEMALLOC provides weaker security than complete UAF-mitigating allocators as UAF is possible among the same `SemaType` in the recursive call group. Every `SemaType` in trace $T_{a,b}$ shares the same `nID`, and there is only limited entropy for `rID` but potentially unlimited call traces in the recursive call group. Having more SCCs in call graphs means weaker security but more memory-saving. Regardless of the number of SCCs, SEMALLOC still provides strictly more protection than existing type-based allocators, as shown in §3.2.
- *With `malloc` in both loops and recursive calls* ($k \neq 0$ and some or all $R_{i,j} \neq 0$): Security degradation comes from all sources of recurrent allocations (discussed above) as there are now more chances for two objects to be marked as the same `SemaType`. However, memory savings are also brought in due to exactly the same reasons.

Effectiveness evaluation. To show how `SemaType` diversifies heap allocation, we compare the number of different allocation sites using `SemaType`s and pure object types in the last two columns of Table 3.10 based on programs in the PARSEC3 [7] and SPEC 2017 [65] benchmarks. In programs that have complicated program contexts (e.g., 600 and 602), `SemaType` diversifies the allocations by more than 250x than the native allocation sites. Other tested programs that have the same native-typed objects allocated from different traces are also diversified accordingly.

3.5.2 Formal Analysis

SEMALLOC stops UAF attacks by defining the object types based on the path used to allocate them, and guarantees that upon the allocation call site (i.e., `malloc` call), the exe-

cuting tracing and weight assignment can uniquely identify each type. As explained in §3.3, `SemaType` is classified into two types in SEMalloc: allocated through recurrent allocation trace (RA), and non-recurrent regular allocation trace (NA). There are three possibilities if SEMalloc cannot represent each `SemaType` identically based on this classification:

1. NA1 and NA2;
2. NA1 and RA1;
3. RA1 and RA2.

Next, we explain why SEMalloc can notify the runtime wrapper to allocate blocks of different `SemaType` separately in each of the above scenario.

NA1 and NA2

A `SemaType` is classified as NA if there is no recurrent call site in its allocation trace. Thus, there is no possibility that a NA trace accidentally classified as RA. NA blocks are not to be reused by another block after freed. As a result, the two NA blocks are never going to share the same memory address.

NA1 and RA1

As explained above, a NA trace is not classified as RA thus never reuses a block allocated to others or releases itself to be used by another object. Thus, cross-category memory address reuse also will not happen.

RA1 and RA2

RA blocks release the memory space after being freed and a further block with the same `SemaType` will be allocated to this address. If two different RA objects allocated to the same virtual memory address, their `nID` must be the same (i.e., $t_1 = t_2$) upon calling `malloc`. We define the call trace of RA1 as $C1 = \alpha_1, \alpha_2, ..., \alpha_n$, and the call trace of RA2 as $C2 = \beta_1, \beta_2, ..., \beta_m$. Our target is to prove:

Theorem. Given any two recurrent call traces $C1$, $C2$, where $C1 \neq C2$. Upon the `malloc` call site β_m of $C2$, its `nID` t_2 is never the same to the `nID` of $C1$ at their `malloc` call site α_n with the value t_1.

To prove this, we first need to prove:

Lemma 1. Given any recurrent call trace $C = \gamma_1, \gamma_2, ..., \gamma_r$, its `nID` t is always the same at its `malloc` call site γ_r.

Suppose the iterative call site is γ_i. After returns γ_r and before calling it the next time, function call $\gamma_r, \gamma_{r-1}, ..., \gamma_i$ sequentially returns and $\gamma_i, \gamma_{i+1}, ..., \gamma_r$ are sequentially called. As t is increased with a specific value before a call site and is decreased with the same value after the call returns, executing the above sequence does not change the t. Thus, we have proved the lemma 1.

Lemma 2. The weight of function w_f is larger than the sum of the weights on any trace starting with it.

We can use mathematical induction to simply prove this. Suppose f does not call any functions other than `malloc`, its weight is the number of calls to `malloc` it has, and the path weights are assigned from zero to $w_f - 1$. Now, suppose f does call functions other than `malloc`, we suppose that f calls $F = f_1, f_2, ..., f_q$ sequentially. We assume that $\forall \tilde{f} \in F$, lemma 2 applies. We take an arbitrary $f_p \in F$. The weight assigned with it is $\sum_{i=1}^{p-1} wf_i$, and by assumption the maximum path weight among all call traces within f_p is $\sum_{i=1}^{p} wf_i < wf = \sum_{i=1}^{q} wf_i$ Thus, we have proved the lemma 2.

Prove the theorem. To prove the theorem, We only need to show that the sum of the weights in each trace is identical. We denote the weight of α_i as $w\alpha_i$, and the weight of β_i as $w\beta_i$. We assume that the two traces shares the first k prefixes (i.e., $\alpha_i = \beta_i, 0 < i \leq k$). Now take the $(i+1)$th call of each trace α_{i+1} and β_{i+1}, their caller is the same function. Without the lose of generality, we suppose that $w\alpha_{i+1} > w\beta_{i+1}$. According to the weight allocation algorithm algorithm 1, in the caller function, β_{i+1} is called before α_{i+1}, and $w\alpha_{i+1} - w\beta_{i+1} \geq w_\delta$, where w_δ is the weight of the function called by β_{i+1}. According to lemma 2, any function traces within δ is smaller than w_δ, thus $w\alpha_{i+1} > \sum_{k=i+1}^{m} w\beta_k$. Thus, $\sum_{k=1}^{n} w\alpha_k > \sum_{k=1}^{m} w\beta_k$. We have proved the theorem.

3.5.3 Empirical Check on Real-world Exploits

We evaluate the effectiveness of SEMalloc in stopping UAF exploits by running it with 15 real-world UAF vulnerabilities. We compare the protection results with two type-based allocators, Cling [3] and TypeAfterType [72], while other allocators used in performance evaluation (§3.6) either have theoretically complete UAF-mitigation [2, 19, 25, 78] or requires case-by-case manual annotation to work (e.g., PUMM [79]).

Vulnerability	Exp. (§2.2)	[72]	[3]†	S2Malloc
CVE-2015-6831	B	○	○	●
CVE-2015-6835	C	○	○	●
Python-24613	C	●	●	●
mRuby-4001	D	◐	◐	●
yasm-91	D/E	◐	○	●
CVE-2018-11496	D/E	○	○	●
CVE-2018-20623	C	●	◐	●
yasm-issue-91	C	◐	◐	●
mjs-issue-78	B	◐	○	●
mjs-issue-73	B	◐	○	●
CVE-2017-10686	D/E	◐	○	●
CVE-2016-3189	D	○	○	●
CVE-2009-0749	D/E	●	●	●
CVE-2011-0065	B	●	◐	●
CVE-2012-0469	B	●	◐	●

Table 3.2: SEMalloc is effective in thwarting (●) exploitation of all real-world UAF vulnerabilities evaluated while TypeAfterType [72] and Cling [3] provide no protection (○) or partial protection (◐) to most vulnerabilities. †: Cling is not open-sourced and is only analyzed conceptually.

Tested vulnerabilities are summarized in Table 3.2. They are selected from three sources: representative CVEs from DangZero [25], TypeAfterType [72], uafBench [46], and further enriched with additional vulnerabilities selected by us to cover the exploitation types discussed in §2.2. We present four representative examples here.

While SEMalloc successfully thwarts all exploits, TypeAfterType provides no defense against four exploits and only partial protections for most attacks, as the attacker can still launch attacks successfully but cannot create powerful attack primitives. Additionally, we checked all exploits since 2019 in exploitDB [55], and we are not aware of any exploitation against S2Malloc—confirming that SEMalloc can help confine UAF exploitability in practice.

Case study: mjs-issue-78 [46]. This vulnerability is in `mjs`, a restricted JavaScript engine, and can be triggered when `mjs` parses a crafted `JSON` string as shown in the test case.

While parsing, both the raw `JSON` string and intermediate outputs are stored in one buffer: field `owned_strings` within type (`struct mjs`), a context manager for an `mjs` engine. As the parser keeps appending parsed elements to the buffer (more precisely, to

`mjs->owned_string->buf`) during `mjs_mk_string`, the buffer might potentially be reallocated via mbuf_resize, causing other pointers that also refer to the same buffer to be dangling (e.g., `frozen->cur`). To summarize, the dangling pointer in this UAF vulnerability is allocated in the call trace of `mjs_json_parse` → `json_walk` → `...` → `frozen_cb` → `mjs_mk_string` → `mbuf_resize` → `realloc`.

Assuming that the memory chunk freed by `mbuf_resize` is later reallocated to a buffer in which the attacker can put arbitrary data, then the attacker-controlled object can be accessed by a dangling pointer (e.g., the `frozen->cur` through one of "`cur(f)`"). This UAF-read might lead to compromised execution states, making it a type-B exploit.

From an attackers' perspective, to exploit this UAF vulnerability, the crux is to gain control of an object that may be allocated to the memory chunk freed by `mbuf_resize`. This can be done in at least two ways based on our findings:

Exploit 1: Run an `mjs` engine in another thread and have the other `mjs` engine parse an attacker-supplied `JSON` string. In this way, the attacker-controlled buffer is allocated using exactly the same call trace as the dangling pointer. Therefore, only SEMalloc can defend against this exploit because `SemaType` is thread-sensitive while flow- and context-sensitivity is not enough.

Recall that Cling defines the "type" of an allocated object based on the two innermost return addresses on the call stack when `realloc` is invoked. This definition cannot distinguish objects allocated using exactly the same call stack on different threads. The lack of thread-sensitivity is also the reason why TypeAfterType cannot defend against this exploit, as both the freed object (which inadvertently creates dangling pointers) and attacker-controlled object are classified as the same "type", hence allowing UAF among them.

Exploit 2: An attacker may exploit another call trace `mjs_mkstr` → `mjs_mk_string` → `mbuf_resize` → `realloc` to obtain a controllable buffer potentially in the same thread where `mjs_json_parse` is invoked (e.g., by placing a `mkstr(..)` JavaScript call after the `JSON` string). In this way, the attacker-controlled buffer is allocated using a different call trace as the dangling pointer. SEMalloc mitigates this exploit by assigning different `SemaType`s to the dangling pointer and attacker-controlled buffer, eliminating the possibility of UAF among them.

Cling takes `mbuf_resize` as an allocation wrapper and treats all objects allocated through `mjs_mk_string` to have the same type. This allows UAF between the dangling pointer and attacker-controlled buffer despite that they are originated from different roots. TypeAfterType, on the other hand, further takes `mjs_mk_string` as a `malloc` wrapper as it still passes a variable length to `mbuf_resize`. This enables TypeAfterTypet to differentiate objects allocated through `frozen_cb` and `mjs_mkstr`. Hence, can mitigate this exploit.

Case study: CVE-2015-6835 [50]. This vulnerability is in the `PS_SERIALIZER_DECODE_FUNC` function, which restore a PHP session from a serialized string. During this process, `php_var_unserialize` returns a `zval` pointer, which is stored in a hashtable. However, the same pointer might be freed later and this causes the stored copy to be dangling. Through this dangling pointer, an attacker might corrupt any `zval` object that may be reallocated to the freed slot.

`zval` is a reference-counting wrapper of almost all other objects in the PHP engine. Therefore, the attacker can corrupt any `zval` object that may be reallocated to this free slot and its value can be leaked through the dangling pointer. In the PoC exploit, the attacker simply uses the PHP `echo(..)` function to dump a newly allocated `zval` through the dangling pointer, i.e., a type-C exploit.

In this PoC exploit, both the dangling pointer and victom objects are allocated through a common call trace: `php_var_unserialize` → `emalloc` → `malloc`. This is critical to understand why both Cling and TypeAfterType fail to provide protection. For Cling, this `malloc` wrapper chain implies that all `zval` objects allocated through this chain share the same type (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.

SEMalloc can mitigate this exploit because `SemaType` is not only context-sensitive but also flow-sensitive. For examples, a session initialization `zval` can never be allocated from the same pool as a `zval` created in the middle of a session.

Python-24613. [32] This vulnerability resides in the logic of parsing an array from a string and appending it to an existing array. In the `array.fromstring()` method, the Python interpreter calls `realloc` to guarantee that the allocated memory of the appended array object is big enough, and calls `memcpy` to copy the data from the string to the new array. However, if the array is appending itself, i.e, the string and the array is the same object, `realloc` essentially frees this object, and the subsequent `memcpy` copies the freed heap chunk to the new object. An attacker can exploit this vulnerability by racing to allocate objects filled with attacker-controlled malicious data over the freed chunk, making this a type-C exploit (see §2.2).

In this exploit, both the dangling pointer and the target object (i.e., the object that the attacker uses the dangling pointer to read from) are allocated through

`array_fromstring`→`realloc`. `array_fromstring` is an exposed Python API that can be called directly in a Python script, and is only called by one other function, `array_new`, which is also a Python API. All three allocators here can differentiate these two call sequences, thus providing complete protection for this vulnerability.

CVE-2012-0469. [72] This vulnerability is in the `indexedDB` module of Firefox. While a `IDBKeyRange` is freed, its reference is left in the object table. The attacker can craft an object, for example, a `vector` to reclaim the pointed space, and interpreting the crafted object using the dangling pointer can cause arbitrary code execution. This is a type-B exploit.

As this object can only be allocated by calling `new` to its constructor, Cling can stop the type confusion on the primary type, provides a partial protection. SEMalloc and TypeAfterType can further differentiate the source of the created `IDBKeyRange` object, from serialized data or explicitly created in the JavaScript script, thus providing complete protection for this vulnerability.

3.6 Performance Evaluation

We evaluate the performance of SEMalloc across a diverse range of scenarios, including macro, micro, and real-world programs. For comparative analysis, we also benchmark MarkUs [2], FFMalloc [78], TypeAfterType [72], and the glibc memory allocator [24] on the same test suites. Although Cling [3] is a closely related work, we omit it in our evaluation because its code is not available.

Based on the design features of each memory allocator, we expect that SEMalloc should:

A incur a lower run-time overhead than MarkUs,
B incur a lower memory overhead than FFMalloc,
C is on-par with TypeAfterType in run-time overhead and may incur a higher memory overhead.

Preview. The outcomes of our evaluation are in alignment with these expectations with consistent results across mimalloc-bench, SPEC CPU 2017, PARSEC 3, and real-world server programs (Nginx, Lighttpd, and Redis).

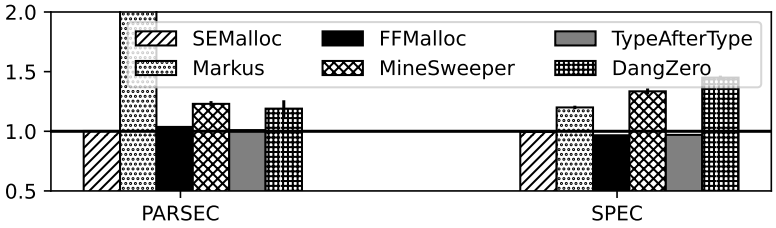


Figure 3.6: Normalized average and standard deviation of run-time overhead on PARSEC and SPEC benchmarks.

3.6.1 Evaluation Setup

All experiments except DangZero are conducted in the Ubuntu 22.04.4 environment, on a server configured with a 48-core 2.40GHz Intel Xeon Silver 4214R CPU with 128GB of system memory. DangZero experiments are executed on the same machine with QEMU-KVM as DangZero requires patching the Linux 4.0 kernel in the guest VM.

We use LLVM 15 to compile programs. For simplicity, we use the WLLVM [71] compiler wrapper to link the whole program bitcode into a single IR file. While generating the IR, we enable the compiler to track pointer types by setting the `-fno-opaque-pointers` flag, and disable constructor aliasing with `-mno-constructor-aliases` flag to simplify the call graph. We then transform the IR using our pass and compile it to generate the hardened program. We also generate unhardened programs by directly compiling the IR without running the S2Malloc-specific transformation pass.

We use a Python wrapper to measure clock time and maximum memory usage (`maxrss`) in program execution for all test programs except from DangZero-protected programs, which is additionally measured with the page table size as instructed in their paper. All results are based on five runs, normalized with respect to the corresponding glibc. We compute performance averages using geometric means, and report standard deviations as well.

While we built all related tools as instructed in the latest versions of their respective official GitHub repositories, we note that MarkUs, TypeAfterType, DangZero, and MineSweeper are not compatible with all tests. We exclude them from computing their respective average overheads and the complete list can be found at §A.2.

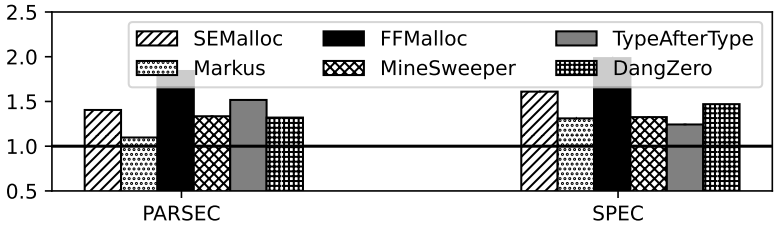


Figure 3.7: Normalized average and standard deviation of memory overhead on PARSEC and SPEC benchmarks.

3.6.2 Macro Benchmarks

We choose the widely used SPEC and PARSEC benchmark suites as macro benchmarks. They are general-purpose benchmarks with various kinds of programs that can show the performance of SEMalloc in a broad range of scenarios.

SPEC CPU2017: We use SPEC CPU2017 [65] version 1.1.9 and report the results of 12 C/C++ tests in both "Integer" and "Floating Point" test suites. We use the SPEC `runsetup` option to set up the program inputs and invoke the benchmark binary through our measurement wrapper to collect more precise overhead numbers. We note that some SPEC tests run the test executable multiple times with different inputs. We report the sum of all run-times and the largest memory usages for these tests.

PARSEC 3: We use the latest PARSEC 3 [7] benchmark, excluding two ("raytrace" and "facesim") from analysis because they are incompatible with the Clang compiler, and one ("x264") as it causes segmentation fault with glibc.

Benchmark Performance. On SPEC, SEMalloc, FFMalloc, and TypeAfterType outperform the glibc allocator (0.6%, 3.3%, and 3.0% respectively). This is explainable as pre-allocating heap pools by types reduces the number of page requests made the kernel and hence can reduce allocation latency. Placing heap objects of similar types or `SemaType`s in adjacent memory is also beneficial to cache lines. MarkUs and MineSweeper incur significant overheads (21.0% and 33.4% respectively), which is expected due to expensive pointer scanning operations. DangZero also incurs a significant 45.2% overhead even with a modified kernel presented, which does not align with our expectations, and is explained below.

All allocators incur extra memory overhead than glibc. As expected, for type-based allocators, the more sensitive the type (TypeAfterType → SEMalloc → FFMalloc) the greater the memory overhead (23.5% → 61.0% → 98.4%). MarkUs and MineSweeper incur

31.1% and 32.5% memory overheads respectively due to quarantine of freed blocks although the number here is for reference only. DangZero incurs a 47% memory overhead (including kernel memory consumption) due to the use of alias page tables.

The results on PARSEC also align with expectations that SEMalloc incurs: smaller run-time overhead (-0.4%) than MarkUs (144%) and MineSweeper (23.0%), smaller memory overhead (40.5%) than FFMalloc (84.1%), similar run-time overhead with TypeAfterType (1.0%), and smaller run-time and similar memory overheads with DangZero (19.5% and 32.3% respectively).

Abnormalities. While the overall evaluation results align with expectations A, B, C, and D, we do notice abnormalities in the results. Failed test cases and how they might affect the reported evaluation numbers in related works are summarized in §A.2. Here, we focus on discussing individual test cases that do not yield expected results.

SEMalloc allows memory reuse among allocations of the same `SemaType` while FFMalloc does not allow any virtual memory reuse, thus running SEMalloc should incur less memory overheads compared with FFMalloc. However, on the benchmarks, we observe three exceptions: "641", "644", and "fer". Test "641" and "fer" frequently call functions in external libraries that allocates heap memories causing excessive memory use. Test "644" reaches its memory usage peak at the beginning of the program that allocates a significant number of blocks together and they are all not released until the end of the program. As FFMalloc allocates blocks at a 16-byte granularity, it uses less memory to allocate them compared with SEMalloc uses the size of two size classes to allocate blocks. The observed overhead comes from the data storage instead of the way SEMalloc reuse freed blocks.

While benchmarks like SPEC and PARSEC are good indicators of the overall performance of SEMalloc, to give a clear view how long SEMalloc takes to finish the memory allocation and release calls, and how such delay would influence real-world programs, we run it on micro benchmarks (§3.6.3) and real world programs respectively (§3.6.4).

3.6.3 Micro Benchmarks

We use mimalloc-bench [15], a dedicated benchmark designed to stress test memory allocators with frequent (and sometimes only) allocations and de-allocations. We exclude one test: "mleak" that tests memory leakage instead of allocation performance, and summarize the overheads and standard deviations of the rest of tests in Figure 3.8. Individual results can be found in Table 3.5 and Table 3.6.

On average, SEMalloc introduces less execution delay compared with allocators that offer more security (i.e., MarkUs, MineSweeper, DangZero and FFMallloc) and perform

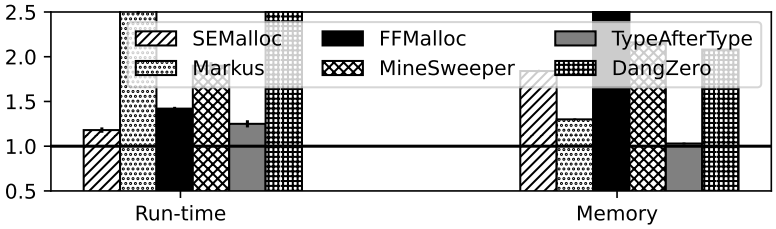


Figure 3.8: Normalized average and standard deviation of run-time and memory overhead on mimalloc-bench.

slightly better than TypeAfterType. For memory overhead, SEMalloc cuts the memory usage by more than half compared with FFMalloc, which aligns with our expectations and make it a possible approach for real-world programs.

3.6.4 Performance on real-world programs

We evaluate three real-world performance of SEMalloc using Nginx (1.18.0), Lighttpd (1.4.71) and Redis (7.2.1).

For network servers, we use ApacheBench (ab) [67] 2.3 to evaluate their throughput with 500 concurrent requests, and take the Nginx default 613 bytes root page as the requested page. On Redis, we use the same settings as how its performance is measured in mimalloc-bench [15]. The results are summarized in Figure 3.9 and Figure 3.10. Overhead numbers can be found in Table 3.7 and Table 3.8. While running Nginx, MarkUs consumes a significant amount of memory possibly due to an implementation error. DangZero is not compatible with Redis and Lighttpd, and MineSweeper is not compatible with Nginx. Running them causes segmentation faults and hence we exclude them from the analysis. PUMM incurs negligible run-time overheads for the two web servers but an abnormal 43% overhead for Redis, possibly due to an implementation bug or an incomplete program profiling that misidentifies the "task". Albeit this outlier, the results align with our expectations for SEMalloc:

- a lower run-time overhead than MarkUs,
- a lower memory overhead than FFMalloc,
- on-par with TypeAfterType in run-time overhead but may incur a higher memory overhead.

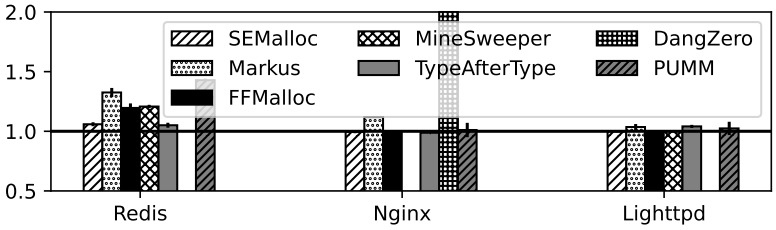


Figure 3.9: Normalized average and standard deviation of throughput overhead on three real-world programs.

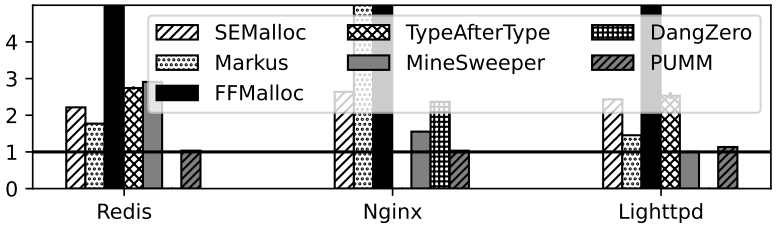


Figure 3.10: Normalized average and standard deviation of memory overhead on three real-world programs.

3.6.5 On Recurrent Allocations

In SEMalloc, a `SemaType` only needs to be tracked dynamically if heap objects of this `SemaType` are allocated recurrently, i.e., through loops or recursions (see §3.3.2, §3.3.3, and §3.4.6). For non-recurrent allocations, once an object is freed, its space is never reused. In two extreme cases,

- if a program itself involves absolutely zero recurrent heap allocations (but the dependent libraries may allocate heap memories) SEMalloc behaves exactly like FFMalloc [78];
- if there is only one execution context where heap allocation can happen (i.e., a single `SemaType`), SEMalloc behaves exactly like the glibc heap allocator [24].

Fortunately, most programs are not written in these extreme cases. And yet, this observation leads us to wonder how prevalent recurrent heap allocations are in common benchmark programs that evaluate heap allocators. Needless to say, programs that have a more diverse set of recurrent allocations can benefit more from the fact that SEMalloc attempts to strike a sweet spot in security, performance, and memory overhead in the context of UAF mitigation.

To give more insights on profiling recurrent allocations in real-world programs, we list

key statistics pertinent of each PARSEC and SPEC test relevant to SEMalloc's internals in Table 3.9 and Table 3.10. In particular, we use recurrent allocation percentage to describe how many allocations are one-time allocations. For most programs that frequently allocate blocks, over 99% of the allocations are effectively captured and allocated to individual `SemaType` pools. These pools handles a significant amount of memory reallocation (as shown in the last column), which improves memory efficiency and thus explains why empirically SEMalloc incurs a lower memory overhead than FFMalloc.

However, we observe three exceptions: "620," "bod," and "fer". They often call functions from external libraries (such as those linked with the `-lm` flag in the `math.h` library) that allocate heap memory as well. These external libraries are not transformed by SEMalloc, leading to untracked heap allocations that are handled in the global non-releasing pool (like FFMalloc). Therefore, adopting SEMalloc for a program that heavily depends on external libraries for heap allocations may not be ideal, and the developers can opt to recompile the depended libraries with SEMalloc for better compatibility.

TEST	SEMalloc	[2]	[78]	[19]	[72]	[25]
600	2.69 (0.00)	1.65 (0.00)	3.98 (0.00)	1.74 (0.01)	1.30 (0.00)	1.85 (0.00)
602	1.08 (0.00)	1.06 (0.00)	1.27 (0.02)	1.61 (0.02)	-	1.32 (0.00)
605	1.00 (0.00)	1.28 (0.00)	1.02 (0.00)	1.01 (0.00)	1.00 (0.00)	1.25 (0.00)
619	1.00 (0.00)	1.00 (0.00)	1.02 (0.00)	1.00 (0.00)	1.00 (0.00)	1.00 (0.00)
620	3.27 (0.00)	1.98 (0.00)	18.04 (0.00)	2.17 (0.17)	1.28 (0.00)	3.32 (0.00)
623	1.58 (0.00)	2.23 (0.00)	3.54 (0.00)	1.77 (0.02)	1.00 (0.00)	1.50 (0.00)
625	1.07 (0.00)	1.12 (0.00)	1.53 (0.00)	1.08 (0.00)	1.01 (0.00)	1.09 (0.00)
631	1.00 (0.00)	1.00 (0.00)	1.01 (0.00)	1.00 (0.00)	1.00 (0.00)	1.00 (0.00)
638	1.00 (0.00)	1.07 (0.00)	1.01 (0.00)	1.06 (0.00)	1.03 (0.02)	-
641	4.29 (0.00)	1.23 (0.01)	3.98 (0.00)	1.65 (0.05)	1.89 (0.00)	1.92 (0.00)
644	4.46 (0.00)	1.76 (0.00)	1.76 (0.00)	1.42 (0.00)	2.18 (0.00)	-
657	1.00 (0.00)	1.00 (0.00)	1.02 (0.00)	1.00 (0.00)	1.01 (0.00)	-
Avg	1.61 (0.00)	<u>1.31 (0.00)</u>	1.98 (0.00)	1.33 (0.01)	**1.24 (0.00)**	1.47 (0.00)
bla	1.02 (0.00)	1.00 (0.00)	1.12 (0.00)	1.01 (0.00)	1.00 (0.00)	1.00 (0.00)
bod	2.50 (0.00)	1.23 (0.01)	3.14 (0.03)	1.76 (0.01)	10.21 (0.00)	-
can	1.21 (0.00)	1.35 (0.00)	1.23 (0.00)	1.43 (0.00)	-	1.24 (0.00)
ded	1.05 (0.00)	1.07 (0.00)	1.05 (0.00)	1.99 (0.00)	1.07 (0.00)	-
fer	1.81 (0.00)	-	1.76 (0.00)	-	-	-
flu	1.04 (0.00)	1.00 (0.00)	1.13 (0.00)	1.02 (0.00)	1.00 (0.00)	1.01 (0.00)
fre	0.95 (0.00)	0.99 (0.00)	1.05 (0.00)	0.96 (0.00)	1.09 (0.00)	1.02 (0.00)
str	1.47 (0.00)	1.09 (0.00)	1.65 (0.00)	1.08 (0.00)	1.00 (0.00)	1.07 (0.00)
swa	2.37 (0.00)	1.08 (0.00)	9.77 (0.01)	1.87 (0.07)	-	3.83 (0.00)
vip	1.47 (0.00)	1.08 (0.00)	2.92 (0.00)	1.53 (0.03)	-	-
Avg	1.41 (0.00)	**1.10 (0.00)**	1.84 (0.00)	1.33 (0.01)	1.52 (0.00)	<u>1.32 (0.00)</u>

Table 3.3: Normalized average runtime overheads (and standard deviations) of SEMalloc on SPEC and PARSEC. We indicate the best scheme in bold and the second best underlined to show how SEMalloc hits the sweet spot in the tradeoff between run time and memory use.

TEST	SEMalloc	[2]	[78]	[19]	[72]	[25]
600	2.69 (0.00)	1.65 (0.00)	3.98 (0.00)	1.74 (0.01)	1.30 (0.00)	1.85 (0.00)
602	1.08 (0.00)	1.06 (0.00)	1.27 (0.02)	1.61 (0.02)	-	1.32 (0.00)
605	1.00 (0.00)	1.28 (0.00)	1.02 (0.00)	1.01 (0.00)	1.00 (0.00)	1.25 (0.00)
619	1.00 (0.00)	1.00 (0.00)	1.02 (0.00)	1.00 (0.00)	1.00 (0.00)	1.00 (0.00)
620	3.27 (0.00)	1.98 (0.00)	18.04 (0.00)	2.17 (0.17)	1.28 (0.00)	3.32 (0.00)
623	1.58 (0.00)	2.23 (0.00)	3.54 (0.00)	1.77 (0.02)	1.00 (0.00)	1.50 (0.00)
625	1.07 (0.00)	1.12 (0.00)	1.53 (0.00)	1.08 (0.00)	1.01 (0.00)	1.09 (0.00)
631	1.00 (0.00)	1.00 (0.00)	1.01 (0.00)	1.00 (0.00)	1.00 (0.00)	1.00 (0.00)
638	1.00 (0.00)	1.07 (0.00)	1.01 (0.00)	1.06 (0.00)	1.03 (0.02)	-
641	4.29 (0.00)	1.23 (0.01)	3.98 (0.00)	1.65 (0.05)	1.89 (0.00)	1.92 (0.00)
644	4.46 (0.00)	1.76 (0.00)	1.76 (0.00)	1.42 (0.00)	2.18 (0.00)	-
657	1.00 (0.00)	1.00 (0.00)	1.02 (0.00)	1.00 (0.00)	1.01 (0.00)	-
Avg	1.61 (0.00)	1.31 (0.00)	1.98 (0.00)	1.33 (0.01)	**1.24 (0.00)**	1.47 (0.00)
bla	1.02 (0.00)	1.00 (0.00)	1.12 (0.00)	1.01 (0.00)	1.00 (0.00)	1.00 (0.00)
bod	2.50 (0.00)	1.23 (0.01)	3.14 (0.03)	1.76 (0.01)	10.21 (0.00)	-
can	1.21 (0.00)	1.35 (0.00)	1.23 (0.00)	1.43 (0.00)	-	1.24 (0.00)
ded	1.05 (0.00)	1.07 (0.00)	1.05 (0.00)	1.99 (0.00)	1.07 (0.00)	-
fer	1.81 (0.00)	-	1.76 (0.00)	-	-	-
flu	1.04 (0.00)	1.00 (0.00)	1.13 (0.00)	1.02 (0.00)	1.00 (0.00)	1.01 (0.00)
fre	0.95 (0.00)	0.99 (0.00)	1.05 (0.00)	0.96 (0.00)	1.09 (0.00)	1.02 (0.00)
str	1.47 (0.00)	1.09 (0.00)	1.65 (0.00)	1.08 (0.00)	1.00 (0.00)	1.07 (0.00)
swa	2.37 (0.00)	1.08 (0.00)	9.77 (0.01)	1.87 (0.07)	-	3.83 (0.00)
vip	1.47 (0.00)	1.08 (0.00)	2.92 (0.00)	1.53 (0.03)	-	-
Avg	1.41 (0.00)	**1.10 (0.00)**	1.84 (0.00)	1.33 (0.01)	1.52 (0.00)	1.32 (0.00)

Table 3.4: Normalized average memory overheads (and standard deviations) of SEMalloc on SPEC and PARSEC. We indicate the best scheme in bold and the second best underlined to show how SEMalloc hits the sweet spot in the tradeoff between run time and memory use.

TEST	SEMalloc	[2]	[78]	[19]	[72]	[25]
alloc-test*	1.20(0.02)	2.83(0.08)	1.51(0.02)	1.53(0.04)	-	7.13 (0.14)
cscratch	1.01(0.00)	1.00(0.00)	1.03(0.00)	1.05(0.02)	1.00 (0.00)	1.62(0.05)
cthrash	1.01(0.00)	1.01(0.00)	1.03(0.00)	1.03(0.01)	1.00 (0.00)	1.75(0.05)
glibc-simple	1.16(0.00)	3.41(0.04)	1.29(0.00)	1.70(0.07)	1.33 (0.01)	16.39(0.12)
malloc-large	3.28(0.00)	3.96(0.01)	3.25(0.01)	3.19(0.01)	-	2.62(0.02)
rptest*	0.84(0.13)	8.43(0.07)	3.46(0.01)	9.94(0.12)	1.23 (0.00)	4.31(0.05)
mstress	1.28(0.01)	2.94(0.02)	2.53(0.02)	3.63(0.09)	1.32 (0.23)	3.90(0.02)
rbstress	1.03(0.00)	1.03(0.00)	1.02(0.00)	1.04(0.01)	-	-
sh6bench	0.95(0.00)	7.73(0.13)	1.51(0.00)	2.08(0.05)	1.36 (0.00)	-
sh8bench	0.89(0.00)	0.00(0.00)	1.60(0.02)	3.50(0.13)	1.36 (0.03)	-
xmalloc-test*	1.40(0.04)	1.33(0.02)	0.33(0.02)	0.47(0.06)	1.49 (0.03)	4.50(0.11)
Avg	**1.18 (0.03)**	2.52 (0.04)	1.42 (0.02)	1.90 (0.04)	1.25 (0.04)	4.00 (0.06)

Table 3.5: Normalized run-time overheads (and standard deviations) of SEMalloc on mimalloc-bench (results of * marked tests use built-in measurements). We indicate the best scheme in **bold** and the second best underlined to show how SEMalloc hits the sweet spot in the tradeoff between run time and memory use.

TEST	SEMalloc	[2]	[78]	[19]	[72]	[25]
alloc-test	2.72(0.00)	1.43(0.00)	58.99(0.00)	2.52(0.05)	-	13.45(0.01)
cscratch	1.99(0.01)	1.00(0.01)	8.04(0.01)	1.32(0.01)	1.01 (0.01)	1.01(0.01)
cthrash	1.98(0.01)	1.00(0.01)	7.97(0.01)	1.30(0.01)	1.01 (0.01)	1.02(0.01)
glibc-simple	3.67(0.01)	1.01(0.01)	8.02(0.00)	2.52(0.08)	1.01 (0.01)	3.46(0.01)
malloc-large	0.78(0.00)	0.78(0.00)	0.91(0.00)	0.80(0.00)	-	1.00(0.00)
rptest	3.56(0.00)	10.09(0.01)	8.28(0.01)	1.89(0.01)	1.28 (0.01)	1.01(0.01)
mstress	1.43(0.01)	1.00(0.01)	8.04(0.00)	1.89(0.05)	1.00 (0.00)	1.03(0.00)
rbstress	1.27(0.00)	1.09(0.00)	1.71(0.01)	1.18(0.00)	-	-
sh6bench	1.23(0.00)	1.06(0.00)	2.09(0.00)	1.36(0.05)	1.01 (0.00)	-
sh8bench	1.02(0.00)	0.00(0.00)	1.30(0.01)	7.30(0.28)	0.97 (0.01)	-
xmalloc-test	3.37(0.01)	1.01(0.01)	11.19(0.01)	13.53(0.82)	1.03 (0.01)	7.07(0.01)
Avg	1.84 (0.00)	1.30 (0.01)	5.31 (0.01)	2.16 (0.03)	**1.03 (0.01)**	2.08 (0.01)

Table 3.6: Normalized average memory overheads (and standard deviations) of SEMalloc on mimalloc-bench (results of * marked tests use built-in measurements). We indicate the best scheme in **bold** and the second best underlined to show how SEMalloc hits the sweet spot in the tradeoff between run time and memory use.

	Redis	Nginx	Lighttpd	Average
SEMalloc	1.06 (0.02)	0.99 (0.01)	1.00 (0.01)	1.02 (0.01)
MarkUs	1.33 (0.04)	1.12 (0.05)	1.04 (0.03)	1.16 (0.04)
FFMalloc	1.20 (0.04)	0.99 (0.02)	1.00 (0.00)	1.06 (0.00)
MineSweeper	1.21 (0.01)	-	1.04 (0.00)	1.12 (0.00)
TypeAfterType	1.05 (0.02)	0.99 (0.00)	1.00 (0.00)	**1.01 (0.00)**
DangZero	-	2.90 (0.01)	-	2.90 (0.01)
PUMM	1.43 (0.01)	1.01 (0.00)	1.02 (0.06)	1.14 (0.00)

Table 3.7: Normalized average throughput (and standard deviations) of SEMalloc on three real-world programs (results of * marked tests use built-in measurements). We indicate the best scheme in **bold** and the second best underlined to show how SEMalloc hits the sweet spot in the tradeoff between run time and memory use.

	Redis	Nginx	Lighttpd	Average
SEMalloc	2.22 (0.01)	2.64 (0.02)	2.43 (0.02)	2.42 (0.02)
MarkUs	1.77 (0.01)	303.11 (0.02)	1.45 (0.02)	9.20 (0.02)
FFMalloc	5.92 (0.02)	15.48 (0.01)	11.55 (0.02)	10.19 (0.02)
MineSweeper	2.74 (0.04)	-	2.53 (0.10)	2.63 (0.06)
TypeAfterType	2.91 (0.03)	1.55 (0.02)	1.00 (0.02)	1.65 (0.02)
DangZero	-	2.37 (0.01)	-	2.37 (0.01)
PUMM	1.03 (0.01)	1.03 (0.01)	1.13 (0.01)	**1.06 (0.01)**

Table 3.8: Normalized average memory overheads (and standard deviations) of SEMalloc on three real-world programs (results of * marked tests use built-in measurements). We indicate the best scheme in **bold** and the second best underlined to show how SEMalloc hits the sweet spot in the tradeoff between run time and memory use.

Test	Δ size	# alloc sites	# CG nodes	# CG edges	# CG SCCs
600	8.97%	80	1,355	14,068	14
602	3.06%	91	2,330	23,128	12
605	0.06%	16	9	22	0
619	1.01%	0	0	0	0
620	3.37%	1,105	1,498	110,380	5
623	1.23%	30	1,387	4,273	25
625	1.01%	66	55	319	0
631	0.37%	0	0	0	0
638	8.97%	10	1,249	13,271	7
641	2.47%	90	96	323	0
644	5.17%	202	83	367	1
657	1.90%	10	66	221	2
bla	1.36%	0	0	0	0
bod	4.48%	55	654	809	0
can	3.07%	3	47	55	0
ded	4.98%	40	41	110	0
fer	7.78%	161	125	355	1
flu	7.75%	12	10	20	0
fre	10.29%	64	37	150	0
str	21.65%	22	14	35	0
swa	0.84%	5	12	32	0
vip	24.06%	573	2,758	37,108	25

Table 3.9: Program profile of each SPEC and PARSEC test.

Test	# allocs	# rec. pools	# rec. allocs	ave. allocs per rec. pool	mem. leak	# native objs	# SemaType objs
600*	47,799,547	4,470	99.89%	10,693	0.0%	30	7943
602*	76,739,284	1,410	99.97%	58,990	0.0%	5	1280
605	1,005,766	8	100.00%	125,719	0.0%	14	14
619	6	0	0.00%	0	0.0%	1	1
620	458,738,506	698	98.60%	648,045	41.6%	567	1026
623	138,362,610	455	100.00%	304,079	1.6%	2	193
625*	3,245	28	97.75%	113	0.6%	4	87
631	3	0	0.00%	0	0.0%	1	1
638	42,945,318	135	100.00%	613,486	0.0%	5	835
641	53,759,694	648	99.87%	82,855	54.0%	35	144
644	1,533,183	56	100.00%	27,378	0.0%	39	57
657*	32	5	31.25%	2	0.0%	6	22
bla	8	0	0.00%	0	0.0%	1	1
bod	371,834	35	94.62%	10,053	34.9%	32	76
can	21,141,425	12	100.00%	1,761,785	0.0%	3	5
ded	1,717,291	28	100.00%	61,328	0.0%	27	27
fer	521,103	104	86.5%	4,335	9.8%	76	76
flu	229,910	1	100.00%	229,899	0.0%	10	10
fre	441	6	2.72%	2	0.0%	23	35
str	8,835	35	99.57%	251	0.0%	14	15
swa	48,001,799	20	100.00%	2,400,089	0.3%	5	20
vip	2,380,317	283	99.99%	8,410	2.5%	44	320

Table 3.10: Number of allocations, iterative allocations, and iterative pools for each SPEC and PARSEC test. We highlight that SEMalloc can efficiently identify `SemaType`s and cause negligible memory leakage for most programs. Tests with * have more than one input. We only report the input that triggers the most allocations.

Chapter 4

Entropy-Based Approach: S2Malloc

4.1 Introduction

While effective in defending against various heap exploits, entropy-based allocators are not ideally suited to protect against UAF attacks. 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 [40, 63] 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, existing entropy-based allocators still face the challenge of entropy loss, further compromising the probabilistic protection. To illustrate how entropy loss can occur:

1. *Entropy-loss due to heap spraying.* If an attacker could control multiple dangling pointers, one of them will likely point to a newly allocated block. This is especially popular in scenarios where the creation and access of the dangling pointer happen in different threads or loops [20, 74].
2. *Entropy loss due to information leak.* An attacker could estimate the block size or even type based on the pointers address — blocks of the same size range have the same top address bytes [40, 63]. An attacker can exploit this information to launch UAF exploits strategically, i.e., only when a target category of dangling pointer is obtained.

3. *Entropy loss due to silent failures and repeated trials.* A failed exploitation attempt raises no signals to the heap allocator. Whether the victim application enters into a human recognizable erroneous state (e.g., crashing) is completely decided by the application logic. Unfortunately, long-running services are usually equipped with resilience features such as auto-restart on exceptions. These features actually work in favor of the attackers as now they can probe live systems by repeatedly launching the same attack until succeed.

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.
- 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.
- Through various real-world CVEs and benchmarks, we show that S2Malloc can successfully detect all attacks 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.

4.1.1 Adversary model

We assume that the attacker can analyze the source code and binary executable to determine the implementation details of the victim 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.

We do, however, 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 [40, 63].

Additionally, we allow attackers to use any existing heap feng-shui [64] technique (e.g., heap spray) to prepare or manipulate the layout of heap to facilitate UAF-exploits. And attackers can retry an exploit as long as previous attempts fail silently. These assumptions make our adversary model stronger than those assumed in entropy-based allocators and on-par with the adversary models in UAF-mitigating allocators [2, 78].

4.1.2 Challenge 1: entropy loss

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

Figure 4.1 is an example, abstracted from mRuby issue 4001 [17], 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` object (`DATA_PTR(copy)`) (line 7) and allocates new memory to it (line 9 and 11). If an invalid argument is passed, calling `io_get_open_fptr` would throw an exception (line 10), making `DATA_PTR(copy)` a dangling pointer to the freed object.

Using this vulnerability, the attacker can allocate a string object to take the freed memory space. The attacker can then close the file in Ruby, which will set the first word of the pointed memory to `INF`. If this memory is taken by a string object, its length will be overwritten to `INF` that allows arbitrary memory read and write.

In the above scenario, random (i.e., non-sequential) allocation or delayed free-lists available in existing entropy-based allocators [40, 54, 63] merely increases the attack difficulty: as long as the attacker can wait, the memory chunk referred to by the dangling pointer will eventually be re-allocated, allowing the UAF exploit to proceed after some delay.

Furthermore, the entropy diminishes if an attacker is allowed to repeat the same attack *without penalty* (e.g., when a failed attempt does not crash the target program or trigger

```
1 mrb_io_initialize_copy(mrb_state *mrb, mrb_value copy) {
2   mrb_value orig;
3   struct mrb_io *fptr_copy, *fptr_orig;
4
5   fptr_copy = (struct mrb_io *)DATA_PTR(copy);
6   if (fptr_copy != NULL) {
7     mrb_free(mrb, fptr_copy);
8   }
9   fptr_copy = (struct mrb_io *)mrb_io_alloc(mrb);
10  fptr_orig = io_get_open_fptr(mrb, orig);
11  DATA_PTR(copy) = fptr_copy;
12 }
```

Figure 4.1: Example UAF attack based on mRuby issue 4001 [17]

attention). Similarly, if attackers have the ability of spraying the heap with either dangling pointers of victim objects, the probability of success increases significantly.

This motivates us to design an allocator in a way that 1) actively searches for UAF attempts and raises signals if the 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 could prevent the attacker from being able to deterministically locate the string length even if the attacker manages to obtain a dangling pointer to a string object originally pointed to by `DATA_PTR(copy)`. In addition, any attempts of writing 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 actually happens.

4.1.3 Challenge 2: information leak

Existing entropy-based allocators [40, 63] create a huge memory pool for each block size range, resulting in the leakage of block size via their address, possibly revealing the victim software's internal state to the attacker. For example, each BIBOP bag is assigned an 8 Gigabytes virtual memory pool in SlimGuard [40]. Any objects belonging to this bag will be allocated from this pool. As a result, obtaining a known sized block will be sufficient enough to infer the size of any blocks sharing the upper 32-bit address. Further, block addresses in Guarder [63] are aligned by their size. Block size would possibly be inferred just 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 the sub-bag boundaries unpredictable.

4.2 Design and Implementation

Now we explain the design and implementation of S2Malloc and how it thwarts the types of heap attacks in §2.1.

4.2.1 Architectural overview

At its core, S2Malloc adopts BIBOP to manage memory blocks. An overview of S2Malloc is illustrated in Figure 4.2. S2Malloc maintains a per-thread metadata (Ⓐ), stored in a memory chunk requested directly from the kernel. Huge blocks are obtained or released from the OS directly (Ⓑ), and are 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 (§4.2.3).

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 corresponding offset table cell stores an offset indicating where the data stored in the bag starts (§4.2.2). Otherwise, the slot is free and the offset table stores the location of the FBC (§4.2.5).

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 current 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 comes, (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 current 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. In this example, we assume that each block stores at most 7 bytes of data. The heap canary is set to the 9th

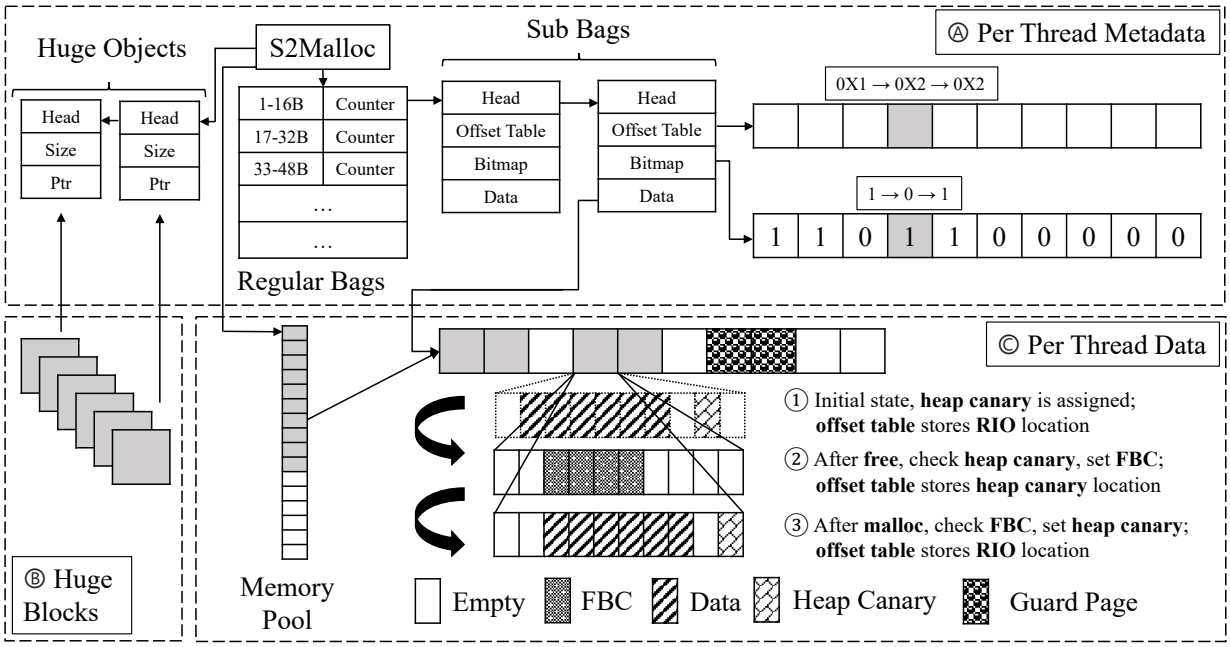


Figure 4.2: Overview of S2Malloc with an example of free and malloc. Ⓐ, Ⓑ, and Ⓒ show three S2Malloc segments, stored in segregated memory. ①, ②, and ③ show how an allocated bag slot is freed and then allocated.

byte initially as the offset is one and is then set to the 10th after reallocation as the offset is changed to two.

4.2.2 Randomized in-slot offset (RIO)

In all existing secure memory allocators, allocated objects store their data from the first byte of the allocated slot. Alternatively, we propose that the object will be stored with a random offset p, and the first p bytes of the 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 the attacker cannot accurately re-use a freed pointer and arbitrarily read or write the target memory.

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, guarantee minimal in-slot offset entropy. 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 [69], and to be compatible with special data structures, such as atomic objects that need to be stored to align with the registers and cache lines. The offset of each block will be stored separately in an offset table. The minimal entropy e increases as b becomes larger to avoid introducing high memory overhead for small objects and to provide stronger protections for large objects, observing the fact that larger objects have more complicated structures and are more likely to be targeted.

We adopt the Permuted Congruential Generator (PCG) [56] algorithm following the design of [40] to generate all random numbers. With negligible execution time – much faster than existing random number generators, such as LCG and Unix XorShift, PCG generates hard-to-predict numbers. While it does not provide cryptographic security, the only known attack towards PCG requires three consecutive PCG outputs to recover the seed [10]. However, in S2Malloc, none of the generated numbers is accessible to the user and can only be obtained by sweeping the memory. As a result, PCG is sufficient enough to provide the required level of entropy.

4.2.3 Random bag layout (RBL)

As with existing secure allocators, S2Malloc employs BIBOP-style management for small-size blocks. Blocks larger than 64 kilobytes are mapped and unmapped from the OS directly, and are managed using a linked list. Blocks smaller than 64 kilobytes are further classified as small, medium, and large blocks to decrease the number of size classes. Small bags contain blocks smaller or equal to 1 kilobyte, and a bag is created every 16 bytes (16 bytes granularity). For example, the first small bag takes blocks smaller than 16 bytes, and the second small bag takes blocks of (16, 32] bytes (without taking RIO into consideration). Medium bags contain blocks within the range of 1 kilobyte and 8 kilobytes with the granularity of 512 bytes; large bags contain blocks within the range of 8 kilobytes and 64 kilobytes with the granularity of 4 kilobytes. In total, S2Malloc has 64 small bags, 14 medium bags, and 14 large bags.

S2Malloc obfuscates the virtual memory allocation and stops linking block sizes to their addresses. Instead of allocating a dedicated virtual memory pool for each bag, (as shown in prior works [40, 63]), 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 would request corresponding memory from the pool. We use a bump pointer to track the available memory in the pool and linearly allocate pool memory to sub-bags.

Secure Allocators	Guard Pages	Rand. Alloc.	Segre. MD	Heap Can.	Ptr Inval.	UAF Miti.	UAF Detect	Overheads Mem.	Overheads 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	●	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 4.1: Overview of existing secure memory allocators and S2Malloc to illustrate how S2Malloc fills the gap (MD: metadata). Memory and run-time overheads are measured by running the PARSEC benchmark [7]. Note that overheads of MarkUs and FFmalloc (numbers marked with $*$) are reported in [78] instead of measured by us. The details for getting other overhead numbers are presented in §4.5. With that said, the performance numbers shown here are for a qualitative illustration on the scale of overhead only. For quantitative comparisons, please refer to details in §4.5.

S2Malloc randomly places guard pages within sub-bags to thwart overflow, spraying, and random pointer access. If a sub-bag is randomly allocated with a guard page, one of its pages will be unmapped (protected) randomly using the `mprotect` system call. Any slots within this protected page will be marked as allocated in the bitmap to avoid allocating them to the program. Any accesses to these slots are thus invalid and result in a segmentation fault. The tunable guard page rate can be configured in an environment variable. We note that the memory pool allocation is not deterministic and cannot be predicted due to the random guard pages.

S2Malloc guarantees that the block size leakage occurs only if the known block and the victim block reside on the same memory page: adjacent blocks may not be within the same sub-bag, and RIO guarantees that blocks start at addresses that cannot be deterministically predicted. On the contrary, two blocks are highly likely of the same size range in SlimGuard if their address difference is smaller than 8GB.

4.2.4 Hardening heap canaries

Canary is a small data block put after the allocated memory to detect overflow, initially introduced in StackGuard [14] to protect the stack. Canary has now been adopted to protect the heap [53]. At the time a memory slot is being allocated, the canary will be set

to a specific value. This value will be checked at the time this slot is freed, and memory overflow will be detected if the canary value changes.

However, in previous designs, this value is set to be either globally identical [63] or is binded with slots [40], and could be trivially broken by a knowledgeable attacker. We follow the design of previous works to use the secure MAC of the memory address as the canary [36, 47]. Specifically, we take the CMAC-AES-128 encrypted block address as the canary implemented using AES-NI [30] (on x86 CPUs) or Neon [6] (on AARCH CPUs) 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.

Specifically 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.

4.2.5 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, the attack attempt is not detectable either and given the fact that the same attack can be retried, the attacker will succeed eventually.

To detect such attempts, we put a canary of length c in each free block. The canary value is also computed using CMAC-AES-128. This value will be 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 rate of detection. FBC guarantees that accessing a freed block is not risk-free. Its protection rate is analyzed in §4.4. In Figure 4.2, we further illustrate how the two kinds of canaries (FBC and regular heap canary) are set and cleared when an allocated block is first 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 set to zero in the Linux environment [38]. We take this advantage and use the zeros as the initial FBC with the following benefits:

- Until being accessed, an unused slot will remain unmapped to the physical memory, decreases memory overhead.
- An unused slot is exempted from computing a secure canary value and writing to the corresponding memory field.

- The whole slot will be checked instead of only the canary bytes, increasing the detection rate.

Treading off the computation cost of the encrypted canaries, S2MALLOC always zeros out the contents of small blocks and will check the whole block before being allocated to a new object, bringing both security and computation benefits.

4.2.6 Summary and comparison

Table 4.1 summarizes S2MALLOC and selected state-of-the-art secure heap allocators along the two defense lines that are closely related to S2MALLOC.

Being an entropy-based allocator, S2MALLOC is inherently closer to this line of work [40, 54, 63] with a nearly identical set of heap exploitation protection features *except* UAF protection. S2MALLOC provides a much stronger security assurance in the presence of UAF vulnerabilities. In particular, S2MALLOC addresses the two entropy-loss cases (discussed in §4.1.2 and §4.1.3) with RIO (§4.2.2) and RBL (§4.2.3), respectively, and hence, providing much higher effectiveness on UAF mitigation. In addition, S2MALLOC is designed to actively monitor the integrity of the heap 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 (§4.2.4) and FBC (§4.2.5).

On the other hand, Table 4.1 also shows a sheer contrast between entropy-based allocators and UAF-mitigating allocators. Notably, although providing a theoretically complete mitigation guarantee toward UAF, UAF-mitigating allocators [2, 78] significantly impair program efficiency and are hard to be deployed in time-sensitive use cases. In contrast, as will be presented in §4.5, S2MALLOC incurs a significantly lower overhead that 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 §4.3).

4.3 On The Formal Modeling of Probabilistic Use-After-Free Detection

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 §4.1.1):

① The goal of the attacker is to modify 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 to execute, 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. In real-world settings, attackers usually have an even lower success rate as memory slots can be allocated to other objects, which gives S2Malloc more chances to check FBCs and detect UAF attempts.

Notation. We denote the victim object size as s which will be placed in a block of size b. Within a victim object, the sensitive data starts at the s_1 byte, and with length l. The RIO entropy is e and obviously, $b \geq e + s$. The length of the FBC is c. The block-level entropy bit is n, i.e., each allocation of the victim object will fall in one of $r = 2^n$ blocks.

4.3.1 Success rate of attack and defense per single attempt

In a block hosting a victim object, the first byte of the sensitive field is in the interval $[s_1, b - s + s_1)$. A reasonable attacker will always try to modify l bytes of data starting at some byte within the interval. A smarter attacker will further leverage the knowledge that memory allocations are 16 bytes aligned (a convention from `glibc`). This implies that the RIO of a block is randomly chosen from one out of $1 + \lfloor \frac{b-s}{16} \rfloor$ positions. Thus, if the attacker attempts to write l bytes through the dangling pointer with a randomly guessed RIO value, the chance of success per trial is:

$$
\begin{aligned}
A &= \frac{1}{r} \rhd \text{ the correct block } \lhd \times \frac{1}{1 + \lfloor \frac{b-s}{16} \rfloor} \rhd \text{ the correct in-block RIO value } \lhd \\
&= \frac{1}{r(1 + \lfloor \frac{b-s}{16} \rfloor)}
\end{aligned}
\tag{4.1}
$$

S2Malloc puts an FBC randomly to any c consecutive bytes in the block with the same probability. Hence, the probability that an FBC is modified by an l-byte write in the same block is reduced to the probability of selecting one l-byte chunk and one c-byte chunk randomly from a b-byte block and the two chunks overlaps by at least one byte.

$$
\begin{aligned}
D &= \frac{2\sum_{i=0}^{c-2}(l+i) + \sum_{\_=c-1}^{b-(l+c-1)}(l+c-1) \rhd \text{ number of overlaps } \lhd}{(b-l+1)(b-c+1) \rhd \text{ number of ways to place } l\text{-byte and } c\text{-byte } \lhd} \\
&= \frac{b(l+c-1) - (l-1)^2 - (c-1)^2 - cl + 1}{(b-l+1)(b-c+1)}
\end{aligned}
\tag{4.2}
$$

The above equation holds when $b \geq l + 2(c-1)$, which represents the most practical cases (i.e., lengths of both sensitive field and FBC are small) and favors the attacker. In fact, if either l or c is large enough relative to b, any l-byte write to the block will almost always corrupt the FBC and can be detected by S2Malloc.

4.3.2 Strategy S1: repetitive UAF-writes to the same address

In this strategy, the attacker first obtains a dangling pointer (②) and holds the pointer for arbitrarily long. Every time the attacker notices a victim object is allocated (④), an l-byte UAF-write at the same offset through the same dangling pointer is conducted (⑤). This is essentially repetitive UAF-writes to the same address.

As S2Malloc only detects UAF attempts when a victim object is allocated, we use round i to represent the i-th allocation of a victim object *after* the attacker obtains the dangling pointer and conducts the UAF-write. Effectively, after round i, S2Malloc should have checked FBCs i times to catch the UAF attempt.

We denote $\mathbb{P}_e^i$ to represent the probability that the program execution ever reaches round i. By definition, $\mathbb{P}_e^1 = 1 - A$, i.e., when the attacker's first UAF-write is not successful

in achieving the goal (①). Suppose the execution has reached round i, the probability that the repetitive UAF-writes is detected at this particular round is

$$\begin{aligned} &\frac{2d+1}{r} \rhd \text{ the FBC of the overridden block is checked } \lhd \times \\ &\qquad D \rhd \text{ the FBC of the overridden block is corrupted } \lhd \end{aligned} \tag{4.3}$$

Based on this, we can derive the inductive definition for $\mathbb{P}_e^i$:

$$\begin{aligned} &\mathbb{P}_e^{i+1} = \mathbb{P}_e^i \rhd \text{ reaches round } i \lhd \times (1 - \frac{2d+1}{r} \times D) \rhd \text{ undetected } \lhd \\ &\times (1 - A) \rhd \text{ unsuccessful attack attempt } \lhd \end{aligned} \tag{4.4}$$

Limiting program execution to an upper bound of K rounds, the chance of attacker and S2MALLOC wins, respectively, is:

$$\mathbb{P}_{attack}^K = A + \sum_{i=1}^{K} (\mathbb{P}_e^i \times A) \qquad :: \qquad \mathbb{P}_{detect}^K = \sum_{i=1}^{K} (\mathbb{P}_e^i \times \frac{2d+1}{r} \times D) \tag{4.5}$$

4.3.3 Strategy S2: UAF-writes through fresh dangling pointers

Unlike §4.3.2, the attacker does not hold a dangling pointer indefinitely, instead, the attacker obtains a fresh dangling pointer (②) if a prior UAF-write attempt is not successful. After obtaining a dangling pointer, if the attacker notices a victim object is allocated (④), an l-byte UAF-write through the fresh dangling pointer is conducted (⑤). This essentially means that every UAF-write is likely on a different address.

More importantly, as S2MALLOC only detects UAF attempts when a victim object is allocated, this strategy effectively creates a turn-based game between the attacker and S2MALLOC where in each round, the attacker makes the move of obtaining a dangling pointer and conducting a UAF-write while S2MALLOC makes the move of checking FBCs and allocating a new victim object (if FBCs checked are intact). The game ends when either the attacker or S2MALLOC wins.

We use round i to represent the i-th round of the game. In each round, the attacker makes the first move and S2MALLOC follows. We denote $\mathbb{P}_e^i$ to represent the probability that the program execution ever reaches S2MALLOC's turn in round i (to be consistent with the notation in §4.3.2). By definition, $\mathbb{P}_e^1 = 1 - A$, i.e., the attacker's first UAF-write is not successful in achieving the goal (①).

To calculate the detection rate by S2Malloc, we rephrase the question to a classical combinatorics question: there exists r balls in a box where each time the attacker picks one ball randomly (i.e., the block referred by the dangling pointer), colors it with probability D (i.e., corrupts the FBC in the block), and puts the ball back to the box. A ball cannot be uncolored once it is colored (because the attacker does not undo a UAF-write). We use $\mathbb{Q}_i$ to denote the probability that an arbitrary ball in the box is not colored (i.e., a block with its FBCs integral) after i rounds.

$$\begin{aligned}\mathbb{Q}_i &= (\frac{r-1}{r} \rhd \text{ ball not selected } \lhd + \frac{1}{r} \times (1-D) \rhd \text{ ball selected but not colored } \lhd)^i \\ &= (\frac{r-D}{r})^i\end{aligned} \tag{4.6}$$

Therefore, at round i, there will be, by expected value, $r \cdot Q_i$ balls remain uncolored in the box. The detection rate of S2Malloc at round i will be the same as the probability of selecting $2d+1$ *consecutive* balls from a string of r balls where at least one of the selected balls is colored. The detection rate is denoted as $\mathbb{P}_d^i$ and calculated as:

$$\mathbb{P}_d^i = 1 - \frac{Q_i r - 2d}{r - 2d} \cdot \prod_{i=0}^{2d} (\frac{Q_i r - i}{r - i}) \tag{4.7}$$

Based on this, we can derive the inductive definition for $\mathbb{P}_e^i$:

$$\begin{aligned}\mathbb{P}_e^{i+1} = \mathbb{P}_e^i &\rhd \text{ reaches round } i \lhd \times (1-\mathbb{P}_d^i) \rhd \text{ undetected } \lhd \times \\ &(1-A) \rhd \text{ unsuccessful attack attempt } \lhd\end{aligned} \tag{4.8}$$

Limiting program execution to an upper bound of K rounds, the chance of attacker and S2Malloc wins, respectively, is:

$$\mathbb{P}_{attack}^K = A + \sum_{i=1}^{K} (\mathbb{P}_e^i \times A) \qquad :: \qquad \mathbb{P}_{detect}^K = \sum_{i=1}^{K} (\mathbb{P}_e^i \times \mathbb{P}_d^i) \tag{4.9}$$

4.3.4 Strategy S1-spray: repetitive UAF-writes to the same address with spraying

This strategy operates similarly to the strategy in §4.3.2: the attacker first obtains a dangling pointer (②) and holds the pointer for arbitrarily long. However, generalized

from §4.3.2 in which the attacker conducts a UAF-write after one allocation of a victim object (④), the attacker waits until there are m victim objects newly allocated and alive (i.e., not freed yet) and then issues the UAF-write. Effectively, the attacker is trying to diligently spray the heap by victim objects to increase its chance of success. The UAF-write is still an l-byte memory write at the same offset through the same dangling pointe (⑤). This is essentially repetitive UAF-writes to the same address, similar to §4.3.2.

Consistent with the analysis in §4.3.2, we still use round i to represent the i-th allocation of a victim object *after* the attacker obtains the dangling pointer and conducts the UAF-write. Effectively, after round i, S2Malloc should have checked FBCs i times to catch the UAF attempt.

We denote $\mathbb{P}_e^i$ to represent the probability that the program execution ever reaches round i. By definition, $\mathbb{P}_e^1 = 1 - mA$, i.e., when the attacker's first UAF-write is not successful in achieving the goal (①). Note that the attacker success rate increases as there are m victim objects alive and the attack succeeds as long as the sensitive field in any one of them is overridden by the UAF-write—this is essentially the advantage of heap spraying.

And yet, consistent with §4.3.2, this UAF-write can corrupt one FBC at most. Hence, suppose the execution has reached round i, the probability that the repetitive UAF-writes is detected at this particular round is still Equation 4.3. With attacker's success rate improved, the inductive definition for $\mathbb{P}_e^i$ in this strategy will be:

$$\begin{aligned}\mathbb{P}_e^{i+1} = \mathbb{P}_e^i \rhd \text{ reaches round } i \lhd \times (1 - \frac{2d+1}{r} \times D) \rhd \text{ undetected } \lhd \times \\ (1 - mA) \rhd \text{ unsuccessful attack attempt } \lhd\end{aligned} \tag{4.10}$$

Limiting program execution to an upper bound of K rounds, the chance of attacker and S2Malloc wins, respectively, is:

$$\mathbb{P}_{attack}^K = mA + \sum_{i=1}^{K}(\mathbb{P}_e^i \times mA) \qquad :: \qquad \mathbb{P}_{detect}^K = \sum_{i=1}^{K}(\mathbb{P}_e^i \times \frac{2d+1}{r} \times D) \tag{4.11}$$

4.3.5 Strategy S2-spray: UAF-writes through fresh dangling pointers with spraying

In this strategy, the attacker still sprays the heap such that overriding the sensitive field in any the m victim objects achieves the goal (like the strategy in §4.3.4). Similar to §4.3.3,

the attacker does not hold a dangling pointer indefinitely, instead, the attacker obtains a fresh dangling pointer (②) if a prior UAF-write attempt is not successful and use it to launch a UAF-write attack in the current round. In each round, the attacker makes the first move and S2Malloc follows. We denote $\mathbb{P}_e^i$ to represent the probability that the program execution ever reaches S2Malloc's turn in round i (to be consistent with the notation in §4.3.2 , §4.3.3, and §4.3.4). By definition, $\mathbb{P}_e^1 = 1 - mA$, i.e., the attacker's first UAF-write is not successful in achieving the goal (①) even after spraying m victim objects.

Similar to §4.3.3, we use $\mathbb{Q}_i$ to denote the probability that an arbitrary ball in the box is not colored (i.e., a block with its FBCs integral) after i rounds. $\mathbb{Q}_i$ and $\mathbb{P}_d^i$ can be computed using the same formula as in §4.3.3

Based on this, we can derive the inductive definition for $\mathbb{P}_e^i$:

$$\begin{aligned}\mathbb{P}_e^{i+1} = \mathbb{P}_e^i \rhd \text{ reaches round } i \lhd \times (1 - \mathbb{P}_d^i) \rhd \text{ undetected } \lhd \times \\ (1 - mA) \rhd \text{ unsuccessful attack attempt } \lhd\end{aligned} \tag{4.12}$$

Limiting program execution to an upper bound of K rounds, the chance of attacker and S2Malloc wins, respectively, is:

$$\mathbb{P}_{attack}^K = mA + \sum_{i=1}^{K} (\mathbb{P}_e^i \times mA) \quad :: \quad \mathbb{P}_{detect}^K = \sum_{i=1}^{K} (\mathbb{P}_e^i \times \mathbb{P}_d^i) \tag{4.13}$$

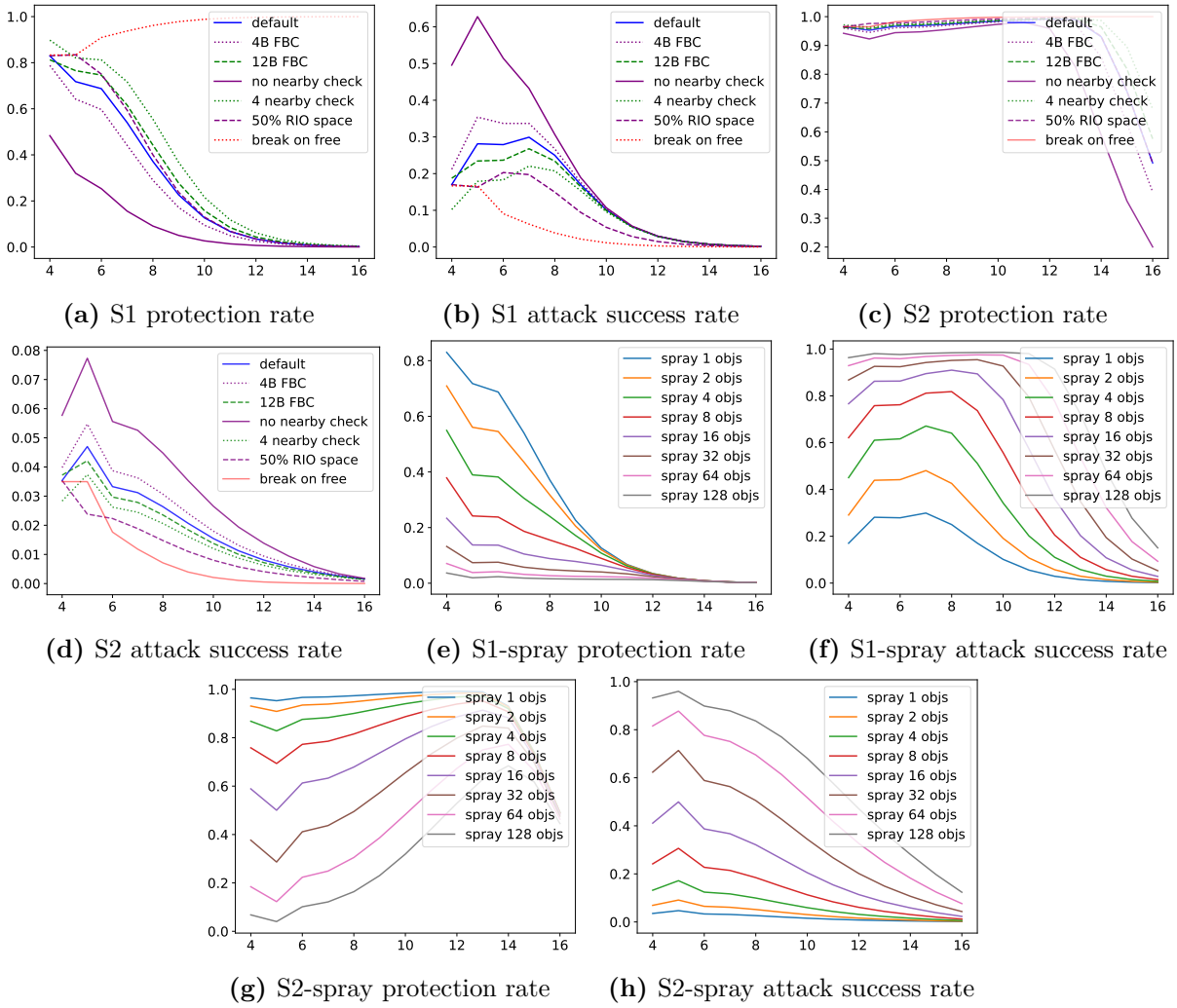
4.4 Security Evaluation

In this section, we show the robustness of S2Malloc towards UAF exploitations in different scenarios. We first present the results from our formal modeling and then show how S2Malloc mitigates real-world UAF attacks.

4.4.1 Parameterized protection rates

In this section, we illustrate how the protection and attack success rates vary with different parameter configurations using the two attack strategies. Assuming the victim field is a pointer of 8 bytes, The set of tunable parameters include ① block size, ② FBC length, ③ RIO entropy, ④ break on free, and ⑤ number of free blocks with FBCs checked.

For FBC-checking, we take 0 and 4 nearby blocks as a comparison to the default setting (2 nearby blocks). For FBC length, we take 4 and 12 bytes as a comparison with the default



(a) S1 protection rate **(b)** S1 attack success rate **(c)** S2 protection rate

(d) S2 attack success rate **(e)** S1-spray protection rate **(f)** S1-spray attack success rate

(g) S2-spray protection rate **(h)** S2-spray attack success rate

Figure 4.3: Parameterized security evaluation (x-axis: logarithmic block size/byte, y-axis: protection/attack success rate/%).

(8 bytes). For random offset entropy, we reserve 50% of block size for RIO compared to the default (25%). We also evaluate the influence on the two rates with break-on-free.

We provide estimates of the rates by assuming that an attacker re-tries an attack for 500 times at max even if the attack is still not detected by the defender. We also assume that overwriting the heap canary does not trigger any alarm. We analyze the protection and attack success rates of blocks with sizes ranging from 16 bytes to 64K bytes as larger blocks are resistant to UAF attacks. All these simplifications favor the attacker.

Figure 4.3a shows how the protection rate changes with a different set of tunable parameters while Figure 4.3b shows how the attack success rate changes using strategy S1. We observe that adopting a more secure setting increases the protection rate and decreases the attack success rates for both small and large blocks. However, as the block size increases, both rates decrease as it is less likely to overwrite either the target field or FBC. Both the attacker and the defender are likely not to succeed, and the rates are thus not summed up to one (in fact, not even close to one).

Figure 4.3c shows how the protection rate changes with the same set of tunable parameters while Figure 4.3d shows how the attack success rate changes using strategy S2. Similarly, we observe that a more secure setting increases the protection rate and decreases the attack success rates. However, the attacker would leave numbers of overwritten FBC in the memory pool in this strategy making the protection rate higher.

Note that other entropy-based allocators provide zero protection as they do not detect failed UAF-write attempts.

4.4.2 Protection rates with heap spray

Heap spraying boosts attackers' chance of overwriting a sensitive field in a victim object and hence, increases the attack success rate and decreases the protection rate. (see detailed strategies and formal analysis in §4.3.4 and §4.3.5). We use the default settings of S2Malloc and evaluate how both attack and defense rates change of strategies (S1 and S2) with different numbers of sprayed objects and object sizes. Figure 4.3e and Figure 4.3f show the protection and attack success rates of reusing the same pointer (strategy S1-spray). Figure 4.3g and Figure 4.3h show the protection and attack success rates using a fresh pointer (strategy S2-spray).

Spraying the heap with target variables diminishes the random allocation entropy, thus decreases the protection rate and increases the attack success rate as FBC will not be overwritten if the pointed block is allocated thus making the attack not detectable. RIO

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

entropy is not influenced by heap spray. An effective protection can be achieved by adopting a more secure configuration (e.g., checking more nearby FBCs, larger RIO or block entropy), with marginal performance degradation (see §4.5.5) to make the spraying less effective.

4.4.3 Illustrate the protection rates

We take the mRuby issue 4001 (shown in Figure 4.1) as an example and show how its protection rate is computed. The size of object `mrb_io` is 16 bytes, In each run, the attacker's goal 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 4.2, 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.

4.4.4 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

Vulnerability	Attack pattern	[63]	[54]	[40]	S2Malloc	[72]	[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 4.3: Summary of how different memory allocators defend against eight exploitation techniques on seven vulnerabilities. Vanilla BIBOP allocator and Scudo [43] are vulnerable to all attacks and behave similarly to Guarder [63] (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).

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 4.3.

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.

[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.

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 4.4 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 `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 4.5 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

```
HashTable *SESSION_VARS;

PS_SERIALIZER_DECODE_FUNC(char* p, char *endptr) {
  char *cursor = p;
  hash_table_t *ht = INIT_HASHTABLE();
  while (cursor < endptr) {
    // for each item in the serialized stream
    zval *name = PARSE_NEXT_ZVAL(&cursor);

    zval *parsed = php_var_unserialize(&cursor, ht, endptr);
    zval *stored = emalloc(sizeof(zval));
    COPY_ZVAL(parsed, stored);
    SESSION_VARS[name->value.str.val] = stored; // refcount = 1

    zval_dtor(name);
    zval_dtor(parsed);
  }
  free(ht);
}
zval* php_var_unserialize(char** pp, hash_table_t* ht, char* endptr) {
  char *cursor = *pp;
  zval *rval = emalloc(sizeof(zval));
  INIT_ZVAL(rval); // refcount = 1

  while (cursor < endptr) {
    // for each element in the serialized item
    zval *tmp;
    if (*cursor == 'R') {
        // reference to a previously parsed zval
        cursor++;
        tmp = PARSE_NEXT_ZVAL(&cursor);
        tmp = ht[HASH_ZVAL(tmp)];
    } else {
        // parsing a new zval from the cursor
        tmp = PARSE_NEXT_ZVAL(&cursor);
    }
    rval->value.ht[HASH_ZVAL(tmp)] = tmp;
    zval_dtor(tmp);
  }

  *pp = cur;
  ht[HASH_ZVAL(rval)] = rval;
  return rval;
}
void zval_dtor(zval *p) {
    p->refcount--;
    if (p->refcount == 0) { free(p); }
}

/* ---------------------------------------------------------------- */

int zend_echo_handler(char *name) {
  zval *obj = SESSION_VARS[name];
  obj->refcount++;
  zend_print_variable(obj); // UAF read
}
```

Figure 4.4: Adapted code snippets to illustrate CVE-2015-6835 and its exploits.

```
1  typedef union _zvalue_value {
2      long lval;
3      double dval;
4      struct { char *val; int len; } str;
5      HashTable *ht;
6      zend_object_value obj;
7  } zvalue_value;
8
9  typedef struct _zval_struct {
10     zvalue_value value;
11     zend_uint refcount;
12     zend_uchar type;
13     zend_uchar is_ref;
14 } zval;
```

Figure 4.5: Type definition of `zval`.

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.

4.5 Performance Evaluation

In order to evaluate the performance and memory overhead of these allocators, we run various benchmarks trying to provide a complete understanding of their performance.

We firstly run two macro benchmarks – PARSEC and SPEC (§4.5.1), and then use the mimalloc-bench and glibc micro-benchmark to evaluate the performance of running two most frequent heap memory management functions: `free()` and `malloc()` (§4.5.2). We then evaluate their performance on real-world programs using two servers: Nginx and Lighttpd, and two databases: Redis and SQLite (§4.5.3). We then discuss how multi-threading impairs each of them performance (§4.5.4). In the end, we show how different parameter values influence the performance of S2Malloc (§4.5.5). We also show the working set size (WSS) of each allocator on SPEC, PARSEC, and real-world programs (§B).

Experiment setup. Experiments are performed on both x86 and AARCH servers for macro benchmarks. The performance of benchmarks is measured only on the x86 server. The x86 server is configured with 64-bit 160-core 2.40GHz Intel Xeon E7-8870 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. On both machines, benchmarks are measured in the Docker environment with Debian 11, kernel version 5.15.0. We measure the overheads using the GNU `time` binary [39] and setting the `LD_PRELOAD` environment variable to substitute the system default allocator.

We obtain SlimGuard, Guarder, and DieHarder from their corresponding GitHub repository. We use SlimGuard with commit `81f1b0f` as a later erroneous commit prevents us from using `LD_PRELOAD` to replace the system allocator. We use the up-to-date version of the other two memory allocators (Guarder: `9e85978`, DieHarder: `640949f`). In order 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 (this actually 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 ($e = 0.25b$). For blocks smaller than a memory page (4096 bytes), we zero it out 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 and standard deviations of five runs.

4.5.1 Macro benchmarks

PARSEC. We first evaluate the performance of S2Malloc using the PARSEC [7] benchmark. We exclude three network tests (netdedup, netferret, and netstreamcluster)

	x86		AARCH	
	SPEC	PARSEC	SPEC	PARSEC
S2Malloc	**12% (4.1)**	**2.8% (1.0)**	**16% (2.3)**	**1.8% (0.9)**
SlimGuard	17% (7.6)	4.4% (1.5)	7.7% (6.9)	2.6% (1.3)
DieHarder	31% (2.1)	2.1% (1.3)	-	2.5% (0.7)
Guarder	3.5% (5.5)	2.4% (11)	-	-

(a) Run-time Overhead (std. $*10^{-3}$)

	x86		AARCH	
	SPEC	PARSEC	SPEC	PARSEC
S2Malloc	**37% (0.2)**	**27% (1.9)**	**38% (0.0)**	**28% (0.7)**
SlimGuard	57% (3.8)	23% (2.2)	57% (2.1)	24% (0.2)
DieHarder	59% (0.0)	21% (1.2)	-	21% (0.8)
Guarder	56% (0.1)	58% (0.7)	-	-

(b) Memory Overhead (std. $*10^{-3}$)

Table 4.4: Normalized run time and memory overheads for state-of-the-art entropy-based secure allocators on SPEC and PARSEC benchmarks. We report geometric averages and standard deviations of the run-time overhead over five runs.

and one test (x264) that fails to compile in the baseline scenario, and only report the result of the rest 12 benchmarks. Additionally, we exclude "raytrace" from execution for the AARCH sever as it cannot compile. We refer to each PARSEC test using the first three letters of its name.

SPEC CPU2017. We use SPEC CPU2017 [65] version 1.1.9. We report the results of 12 C/C++ only tests in both "Integer" and "Floating Point" test suites with the default OpenMP settings of four parallel threads. All reported SPEC overheads are "reportable" following the SPEC documentation [65].

Results. We measure the performance of S2Malloc and three other entropy-based allocators (SlimGuard, Dieharder and Guarder) on both benchmarks with the x86 machine. On the AARCH machine, we exclude Guarder from analysis, noticing that Guarder relies on AES-NI [30], an Intel CPU extension, not supported on AARCH machines. We report the average and standard deviations of overheads in Table 4.4. Averages and standard deviations of each test in both settings are shown in Table 4.5. Missing columns in the figures indicate the corresponding execution runs into error. A complete list of erroneous

executions and explanations is listed in §A.1.

For the SPEC benchmark, on the x86 machine, S2Malloc introduces 11.5% run-time overhead, smaller than two allocators – SlimGuard and DieHarder, and introduces the least memory overhead at 37.4%. On the AARCH machine, S2Malloc introduces similar 15.5% run-time overhead, larger than the SlimGuard due to the fact that 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.

We observe S2Malloc and other memory allocators introduce larger overheads for tests with frequent heap memory management operations, for example, "ded" in PARSEC and "620" in SPEC. We investigate the costs of running `malloc()` and `free()` in the following section.

4.5.2 Micro benchmarks

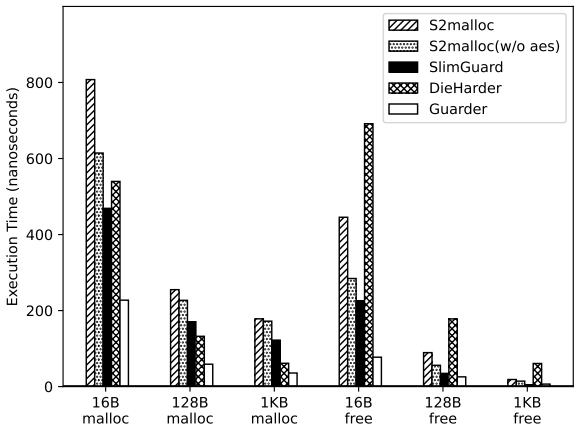


Figure 4.6: Execution time of glibc-simple

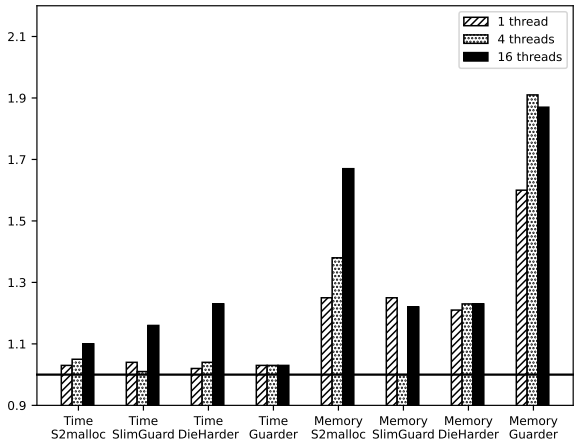


Figure 4.7: Run-time and memory overheads of running PARSEC with multi-threads

To further understand the overheads introduced by S2Malloc, we investigate its performance using mimalloc-bench [15], composed of real-world and calibrated programs that allocate and free heap memory frequently. The results are shown in Table 4.6. Individual results are reported in Table 4.9 and Table 4.8.. Running three tests with SlimGuard never returns (marked as "-"). They are excluded from computing the SlimGuard average overheads.

All secure memory allocators incur larger overheads compared to running real-world (see §4.5.3) 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 runtime overhead with S2Malloc) counts the CPU ticks instead of seconds of finishing each call.

We take one test, `glibc-simple` from mimalloc-bench, from the glibc micro-benchmark suite [70] to further investigate the delays incurred in 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 (`N * S`) is always 1000 MB. Results are presented in Figure 4.6, and is the average of 100 runs.

Generally, S2Malloc takes more time to execute `malloc()` than all other compared memory 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 should be a standard adopted by all memory allocators. Although using hardware acceleration, the canary value is computed in each `malloc()` and `free()` calls, introducing a nonnegligible computation tax. After disabling this feature and using a fixed value as the canary, following the implementation of Guarder, although the execution time of both calls is still longer than Guarder and SlimGuard, it is comparable to others and the increased overhead is expected as S2Malloc introduces extra security guarantees. For example, in 16B `malloc` call, S2Malloc is 31% slower than SlimGuard and is 26% slower than SlimGuard in 16B `free`.

4.5.3 Performance on real world programs

To evaluate the performance of S2Malloc in real-world environments, we run two servers: Nginx (1.18.0), and Lighttpd (1.4.71), and two databases: Redis (7.2.1), and SQLite (3.25.2) on the x86 machine. We use ApacheBench (ab) [67] 2.3 to test the throughput and delays using the Nginx default root page, of 613 bytes, as the requested page with 500 concurrent requests. On Redis, we use the same settings as its performance is measured in mimalloc-bench [15]. We use sqlite-bench [1] to measure the performance of SQLite. We report the results of performing random read and write operations in Table 4.10 and Table 4.11. We observe that 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 on Lighttpd. Applying all secure memory allocators increases memory consumption.

4.5.4 Performance with multi-threading

We run the memory allocators on the PARSEC benchmark with 4 and 16 threads separately using the x86 machine. We exclude the test "ray" from analysis as it cannot be executed with multiple threads and "vip" as running it using Guarder with 16 threads causes a segmentation fault. Results are reported in Figure 4.7.

We observe that as the number of threads increases, S2Malloc gradually introduces more run-time and memory overheads, as we use atomic instructions and maintain per-thread metadata. SlimGuard and DieHarder use single global metadata, and use `lock` to achieve multi-thread compatibility. While increasing the number of threads does not introduce extra memory overheads on the one hand, `lock` introduces more run-time overheads on the other hand. Guarder uses per-thread metadata but fails to use atomic instructions to update the metadata, causing racing conditions if multiple threads are handling adjacent blocks.

4.5.5 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. For the nearby checking range, we take 0 and 4 blocks as a comparison to the default setting: 2. For random allocation entropy, we take 4 bytes and 12 bytes as a comparison to the default 8 bytes. For random offset entropy, we reserve 50% of block size for RIO compared to the default 25%. Table 4.7 shows how different parameters influence the overheads.

We observe that changing the nearby checking range does not introduce observable differences for the memory overhead. The introduced delta is possibly due to server fluctuations. Using a larger nearby checking range introduces a larger run-time overhead, as S2Malloc needs to compute and check more canary values. Using a larger random allocation entropy or RIO introduces both larger memory and run-time overheads.

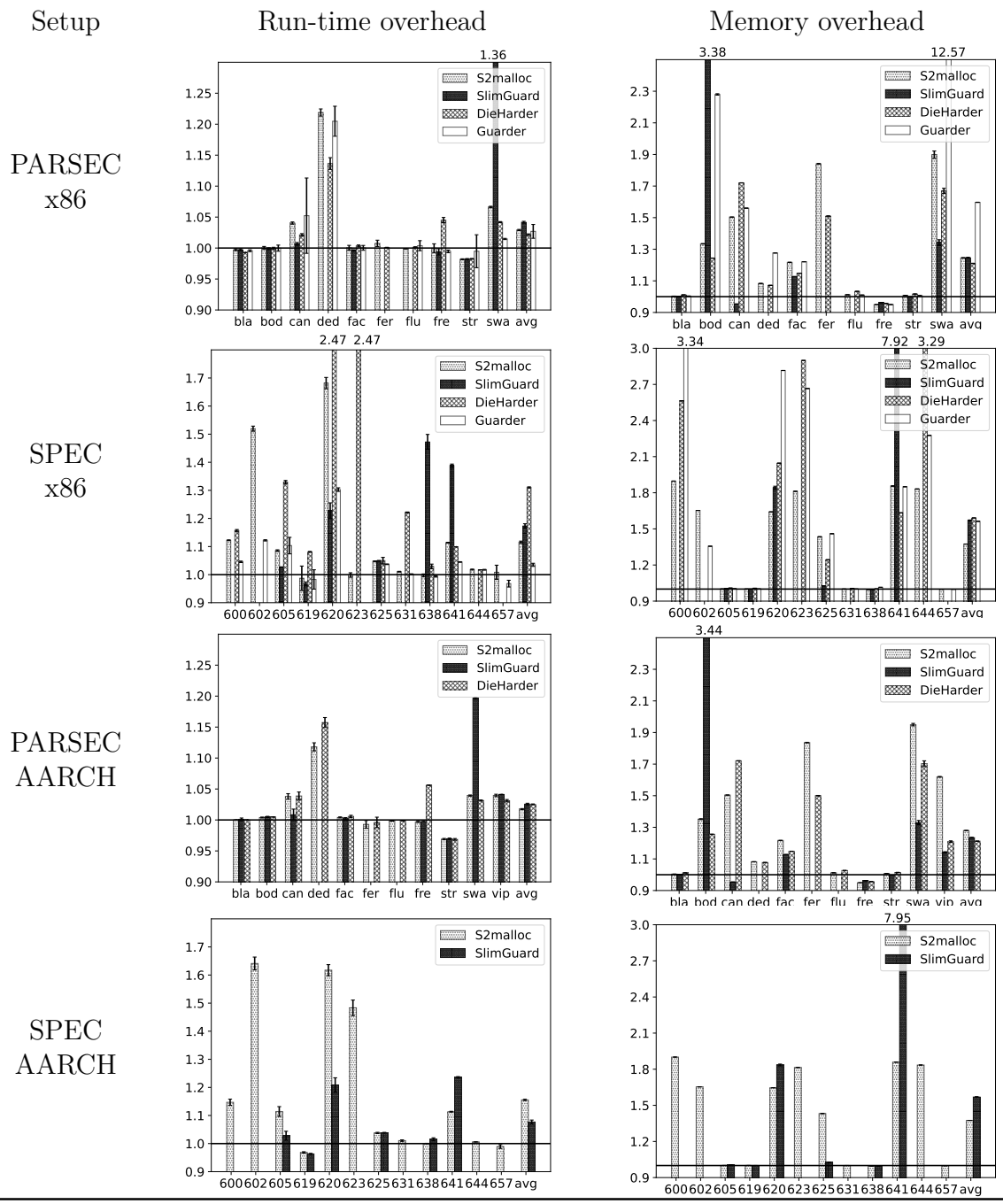


Table 4.5: Average and standard deviation of run time and memory overhead on PARSEC and SPEC benchmarks (x86 and AARCH).

	Run-time Overhead	Memory Overhead
S2Malloc	189%	343%
SlimGuard	298%	250%
DieHarder	229%	92%
Guarder	56%	980%

Table 4.6: 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 4.7: Normalized memory and run-time Overhead changes compared with the default settings.

	S2Malloc	**SlimGuard**	**DieHarder**	**Guarder**
cfrac	3.103	2.939	3.326	1.848
espresso	2.223	6.935	2.073	1.293
barnes	0.999	1.000	1.024	0.998
leanN	2.194	-	3.297	1.498
larsonN	9.313	23.250	7.033	1.748
mstressN	3.500	4.333	2.367	1.833
rptestN	5.383	18.131	3.243	1.212
gs	1.599	1.616	1.333	1.184
lua	1.579	2.364	1.337	1.200
alloc-test1	3.612	3.103	4.314	1.783
sh6benchN	5.392	-	40.675	2.228
sh8benchN	8.384	-	26.302	3.038
xmalloc-testN	2.640	4.052	5.136	1.519
cache-scratch1	1.000	1.000	1.007	1.000
glibc-simple	3.370	1244.895	2.801	1.824
glibc-thread	7.008	14.153	4.279	2.980
redis	1.006	1.004	1.008	0.999
average	2.891	3.981	3.298	1.561

Table 4.8: Normalized runtime overheads of mimalloc-bench

	S2Malloc	**SlimGuard**	**DieHarder**	**Guarder**
cfrac	2.008	1.320	2.539	85.164
espresso	23.822	386.400	3.480	99.456
barnes	1.034	1.014	1.115	1.045
leanN	1.865	-	1.900	1.988
larsonN	11.189	5.667	0.986	17.966
mstressN	4.230	3.853	2.098	2.750
rptestN	12.056	5.119	3.224	5.661
gs	1.548	3.218	1.748	1.683
lua	1.760	1.353	1.333	1.494
alloc-test1	2.156	1.481	1.511	28.857
sh6benchN	1.103	-	1.697	3.109
sh8benchN	3.430	-	1.043	8.004
xmalloc-testN	580.270	8.197	3.032	1238.115
cache-scratch1	1.461	1.250	1.564	1.422
glibc-simple	3.592	239.045	3.778	260.030
glibc-thread	15.690	3.237	3.823	75.580
redis	1.231	1.099	1.249	1.203
average	4.340	4.502	1.922	10.800

Table 4.9: Normalized memory overheads of mimalloc-bench

	Nginx			Lighttpd		
	Throughput	Memory	p50	Throughput	Memory	p50
S2Malloc	9705.369	7867.2	56	11050.830	9425.6	44
Guarder	9496.210	13724	52	11146.580	11088.8	44
SlimGuard	6159.014	4935.2	81	11153.358	6428.0	44
DieHarder	8769.120	7396.8	57	11128.114	13626.4	44
Glibc	9564.754	3400.0	52	10742.708	5069.6	44

Table 4.10: Throughput (request/second), memory consumption (KB), and delays (msec) for servers.

	Redis		SQLite-Read		SQLite-Write	
	Throughput	Memory	Throughput	Memory	Throughput	Memory
S2Malloc	218460.294	44688.8	771247.8791	17234.4	25146.73118	208592.0
Guarder	221245.016	46548.8	790263.9482	17451.2	25735.65366	193408.8
SlimGuard	219986.106	47397.6	766518.4731	14921.6	24456.21604	137659.2
DieHarder	221733.848	52419.2	781983.1092	19115.2	24613.93050	377160.8
Glibc	218155.764	50908.8	796812.7490	5914.4	25439.72566	132984.8

Table 4.11: Throughput (request/second) and memory consumption (KB) on databases

Chapter 5

Conclusion

The threat of UAF always need to be considered as long as a memory unsafe programming language is used. While all memory allocators are not ideal in both security and overheads, I have shown that limiting the attacker's ability can be a possible direction.

I have shown that SEMalloc (Chapter 3), a UAF-specialized memory allocator, can restrict the dangling pointer and the target object to be the same `SemaType` with almost no performance degradation. The proposed `SemaType` can balance security, run-time cost, and memory overhead in UAF mitigation, and can be used beyond memory allocation to help enforce finer-grained data access policies.

I have shown that S2Malloc (Chapter 4), a drop-in solution for not only UAF threats but for other heap vulnerabilities, can fill the gap of expolitation attempt detection without compormising security and performance. The three innovative primitives (RIO, FBC, and RBL) incur only marginal performance overhead that makes S2Malloc pratical to even production systems.

These approaches strike a balance between security and performance: mitigating the risk of UAF vulnerabilities while maintaining system efficiency. It remains for future work to explore alternative avenues for achieving constraints on potential attacks. However, it is essential to remain vigilant and continue exploring advancements in memory safety techniques to stay ahead of evolving threats in software security.

References

[1] Hajime Tazaki. sqlite-bench | SQLite Benchmark. https://github.com/ukontainer/sqlite-bench/tree/master.

[2] Sam Ainsworth and Timothy M. Jones. 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, May 2020.

[3] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *19th USENIX Security Symposium*, pages 177–192, Washington, DC, 09 2010.

[4] Alejandro Guerrero. N-day exploit for CVE-2022-2586: Linux kernel nft_object UAF. https://www.openwall.com/lists/oss-security/2022/08/29/5, 2022.

[5] Cristiano Giuffrida Alyssa Milburn, Herber Bos. Safelnit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.

[6] ARM Developer. Neon. https://developer.arm.com/Architectures/Neon.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical Report TR-811-08, Princeton University, January 2008.

[8] Blaze Labs. The never ending problems of local ASLR holes in Linux. https://www.blazeinfosec.com/post/never-ending-problems-aslr-linux/, 2022.

[9] Matteo Botticci. Zigrazor/cxxgraph: Release v0.2.2. https://doi.org/10.5281/zenodo.5878832, January 2022.

[10] Charles Bouillaguet, Florette Martinez, and Julia Sauvage. Practical seed-recovery for the pcg pseudo-random number generator. *IACR Transactions on Symmetric Cryptology*, 2020(3):175–196, September 2020.

[11] C Language Working Group. ISO/IEC 9899:2023 (E) Programming languages — C. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3088.pdf.

[12] Luca Cardelli. Type Systems. *ACM Computing Surveys (CSUR)*, 28(1), 1996.

[13] Haehyun Cho, Jinbum Park, Adam Oest, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. ViK: practical mitigation of temporal memory safety violations through object ID inspection. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Vancouver, Canada, April 2022.

[14] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium (Security)*, San Antonio, TX, January 1998.

[15] daanx. mimalloc-bench | Suite for benchmarking malloc implementations. https://github.com/daanx/mimalloc-bench.

[16] 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, Canada, August 2017.

[17] Daniel Teuchert, Cornelius Aschermann, Tommaso Frassetto, Tigist Abera. Use after free in File#initilialize_copy. https://github.com/mruby/mruby/issues/4001, April 2018.

[18] Thomas Dullien. Weird Machines, Exploitability, and Provable Unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 8(2), 2020.

[19] Márton Erdős, Sam Ainsworth, and Timothy M. Jones. 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 (ASPLOS)*, Vancouver, Canada, April 2022.

[20] Exploit Database. PHP DateTime - Use-After-Free - PHP dos Exploit. https://www.exploit-db.com/exploits/36158.

[21] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. PTAuth: Temporal Memory Safety via Robust Points-to Authentication. *CoRR*, March 2020.

[22] Nathaniel Filardo, Brett F Gutstein, John Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clark, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Massinghi, Robert M Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M Jones, Simon W Moore, Peter G Neumann, and Robert N M Watson. Cornucopia: Temporal Safety for CHERI Heaps. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*. IEEE Computer Society, May 2020.

[23] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, GA, June 1999.

[24] Free Software Foundation, Inc. The GNU Allocator (The GNU C Library). https://www.gnu.org/software/libc/manual/html_node/The-GNU-Allocator.html.

[25] Floris Gorter, Koen Koning, Herbert Bos, and Cristiano Giuffrida. DangZero: Efficient Use-After-Free Detection via Direct Page Table Access. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, Los Angeles, CA, November 2022.

[26] GrapheneOS. GrapheneOS: the private and secure OS. https://grapheneos.org/.

[27] GrapheneOS. Hardened malloc. https://github.com/GrapheneOS/hardened_malloc, 2024.

[28] Hanno Böck. use after free with malformed input file in yasm_intnum_destroy(). https://github.com/yasm/yasm/issues/91, 2017.

[29] David R. Hanson. A Portable Storage Management System for The ICON Programming Language. *Software: Practice and Experience*, 10(6):489–500, 1980.

[30] Intel Corporation. Intel Advanced Encryption Standard Instructions (AES-NI). https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html.

[31] Mohannad Ismail, Andrew Quach, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. Tightly seal your sensitive pointers with pactight, 2022.

[32] John Leitch. array.fromstring Use After Free. https://bugs.python.org/issue24613, July 2015.

[33] Moshe Kol. Racing Against the Lock: Exploiting Spinlock UAF in the Android Kernel. In *OffensiveCon*, 2023.

[34] 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.

[35] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, Los Angeles, CA, November 2022.

[36] Hans Liljestrand, Zaheer Gauhar, Thomas Nyman, Jan-Erik Ekberg, and N. Asokan. Protecting the stack with paced canaries. In *Proceedings of the 4th Workshop on System Software for Trusted Execution (SysTEX)*, Ontario, Canada, October 2019.

[37] Hao Ling, Heqing Huang, Chengpeng Wang, Yuandao Cai, and Charles Zhang. GI-ANTSAN: Efficient Memory Sanitization with Segment Folding. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Diego, CA, April 2022.

[38] Linux Foundation. mmap(2) - Linux manual page. https://man7.org/linux/man-pages/man2/mmap.2.html, 2023.

[39] Linux Foundation. time(1) - Linux manual page. https://man7.org/linux/man-pages/man1/time.1.html, 2023.

[40] Beichen Liu, Pierre Olivier, and Binoy Ravindran. SlimGuard: A Secure and Memory-Efficient Heap Allocator. In *Proceedings of the 20th International Middleware Conference (Middleware)*, Davis, CA, December 2019.

[41] LLVM Project. Clang: a C language family frontend for LLVM. https://clang.llvm.org/.

[42] LLVM Project. llvm-project/llvm/lib/Target/X86/X86CallingConv.td at main · llvm/llvm-project. https://github.com/llvm/llvm-project/blob/main/llvm/lib/Target/X86/X86CallingConv.td#L586-L588.

[43] LLVM Project. Scudo Hardened Allocator. https://llvm.org/docs/ScudoHardenedAllocator.html, 1024.

[44] Kangjie Lu and Hong Hu. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, November 2019.

[45] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[46] Manh Nguyen. UAF Fuzzing Benchmark. https://github.com/strongcourage/uafbench.

[47] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[48] National Vulnerability Database. 2022 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html.

[49] National Vulnerability Database. CVE-2015-6831. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6831, 2015.

[50] National Vulnerability Database. CVE-2015-6835. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6835, 2015.

[51] National Vulnerability Database. CVE-2018-11496. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-11496, 2018.

[52] 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.

[53] Nick Nikiforakis, Frank Piessens, and Wouter Joosen. Heapsentry: Kernel-assisted protection against heap overflows. In Konrad Rieck, Patrick Stewin, and Jean-Pierre Seifert, editors, *Detection of Intrusions and Malware & Vulnerability Assessment*, volume 7967 LNCS, pages 177–196. Springer, jul 2013.

[54] Gene Novark and Emery D Berger. DieHarder: Securing The Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, November 2010.

[55] OffSec Services Limited. Exploit Database - Exploits for Penetration Testers, Researchers, and Ethical Hackers. https://www.exploit-db.com/.

[56] Melissa E. O'Neill. PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation. Technical Report HMC-CS-2014-0905, Harvey Mudd College, September 2014.

[57] Chanyoung Park and Hyungon Moon. Efficient use-after-free prevention with opportunistic page-level sweeping. In *Proceedings of the 2024 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2024.

[58] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, June 2012.

[59] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

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

[61] Zekun Shen and Brendan Dolan-Gavitt. HeapExpo: Pinpointing Promoted Pointers to Prevent Use-After-Free Vulnerabilities. In *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC)*, 2020.

[62] 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.

[63] 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.

[64] Alexander Sotirov. Heap Feng Shui in Javascript. *Black Hat Europe*, 2007.

[65] Standard Performance Evaluation Corporation. SPEC CPU® 2017. https://www.spec.org/cpu2017/.

[66] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[67] The Apache Software Foundation. Apache HTTP Server Documentation: ab - Apache HTTP Server Benchmarking Tool. https://httpd.apache.org/docs/2.4/programs/ab.html, 2023.

[68] The Chromium Projects. Memory safety. https://www.chromium.org/Home/chromium-security/memory-safety/.

[69] The GNU Project. Aligned Memory Blocks (The GNU C Library). https://www.gnu.org/software/libc/manual/html_node/Aligned-Memory-Blocks.html.

[70] The GNU Project. Standalone Glibc-Benchtests. https://github.com/xCuri0/glibc-benchtests.

[71] Tristan Ravitch. A wrapper script to build whole-program LLVM bitcode files. https://github.com/travitch/whole-program-llvm, November 2023.

[72] Erik van der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. Type-After-Type: Practical and Complete Type-Safe Memory Reuse. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018.

[73] 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, RS, April 2017.

[74] Vulners. Internet Bug Bounty: Use After Free Vulnerability in unserialize() - bugbounty database. https://vulners.com/hackerone/H1:73235.

[75] Ruizhe Wang, Meng Xu, and N. Asokan. S2malloc: Statistically Secure Allocator for Use-After-Free Protection And More. In *Proceedings of the 2024 Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, Lausanne, Switzerland, July 2024.

[76] Ruizhe Wang, Meng Xu, and N. Asokan. SEMalloc: Semantics-Informed Memory Allocator. In *Proceedings of the 2024 ACM Conference on Computer and Communications Security (CCS)*, Salt Lake City, UT, October 2024.

[77] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37, 2015.

[78] Brian Wickman, Hong Hu, Insu Yun, DaeHee Jang, JungWon Lim, Sanidhya Kashyap, and Taesoo Kim. Preventing Use-After-Free Attacks with Fast Forward Allocation. In *Proceedings of the 30th USENIX Security Symposium (Security)*, Online, August 2021.

[79] Carter Yagemann, Simon P Chung, Brendan Saltaformaggio, and Wenke Lee. PUMM: Preventing Use-After-Free Using Execution Unit Partitioning. In *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, August 2023.

[80] 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.

[81] Jie Zhou, John Criswell, and Michael Hicks. Fat pointers for temporal memory safety of c. In *Proceedings of the 2023 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Orlando, FL, June 2023.

Appendices

A List of Failed Tests and Corresponding Exceptions on PARSEC and SPEC

A.1 List of Failed Tests and Corresponding Exceptions on PARSEC and SPEC in S2Malloc

PARSEC x86.

- SlimGuard: ray(mmap error), flu(mmap error), fer(SIGSEGV), ded (SIGSEGV)

SPEC x86.

- SlimGuard: 600 (false positive double free), 602(SIGSEGV), 623(SIGSEGV), 631(SIGSEGV), 644(SIGSEGV), 657(SIGSEGV)
- DieHarder: 602(time out), 657(time out)

PARSEC AARCH.

- SlimGuard: ray(mmap error), flu(mmap error), fer(SIGSEGV), ded (SIGSEGV)

SPEC AARCH.

- SlimGuard: 600 (false positive double free), 602(SIGSEGV), 623(SIGSEGV), 631(SIGSEGV), 644(SIGSEGV), 657(SIGSEGV)

- DieHarder: All benchmarks (Too many open files error)

While running SPEC with DieHarder, test 602 runs unacceptably long, and possibly never returns. We killed the execution after running 1.5 hours, and we also mark it as invalid. As a comparison, running 602 on the AARCH machine with the default memory allocator only takes about 350 seconds. Exceptions here are triggered due to the incompatibility between the allocators and the tests. For example, SlimGuard only supports `malloc`, `free`, `realloc` and `memalign` while other allocation functions such as `calloc` and `aligned_alloc` are not supported.

NOTE: a PARSEC test is referenced by the first three letters of its name.

A.2 List of Failed Tests and Corresponding Exceptions on PARSEC and SPEC in SEMalloc

PARSEC. (referenced by the first three letters of its name)

- MarkUs:
  - fer (`SIGSEGV`)
- MineSweeper:
  - fer (`SIGSEGV`)
- TypeAfterType:
  - can (incomplete type tracking support)
  - fer (LLVM pass exception)
  - swa (incomplete type tracking support)
  - vip (incomplete type tracking support)
- DangZero:
  - bod (`SIGSEGV`)
  - ded (`SIGSEGV`)
  - fer (`SIGSEGV`)
  - vip (assertion fail)

SPEC.

- TypeAfterType:
  - 602 (LLVM pass exception)
- DangZero:
  - 638 (`SIGSEGV`)
  - 644 (`SIGSEGV`)
  - 657 (`SIGSEGV`)

Mimalloc-Bench.

- TypeAfterType:
  - alloc-test (incomplete type tracking support)
  - malloc-large(incomplete type tracking support)
  - rbstress (configuration failure)
- DangZero:
  - rbstress (`SIGSEGV`)
  - sh6bench (`SIGSEGV`)
  - sh8bench (`SIGSEGV`)

Real-world programs.

- MineSweeper:
  - Nginx (`SIGSEGV`)
- DangZero:
  - Redis (`SIGSEGV`)
  - Lighttpd (`SIGSEGV`)

Exceptions here are triggered due to the incompatibility between the allocators and the tests. For example, TypeAfterType does not track allocation functions such as `calloc` and `aligned_alloc`, and DangZero does not allocate aligned memory when `posix_memalign` or `memalign` is called.

MarkUs and MineSweeper fail to run on one PARSEC test ("fer"). We exclude it from computing the average overheads for MarkUs and MineSweeper. However, we should expect smaller overheads for them given that this failed test does not allocate blocks frequently

and running it with other allocators incurs smaller overheads Similarly, for TypeAfterType, compiling “vips”, “swa”, “can”, “fer”, and “602” fails due to its incompatibility with the `using` keyword in C++ and incomplete support for variable type casts. Running it with “bod” uses more than ten times of memory than the baseline, potentially due to an implementation bug.

Additionally, running “bod”, “ded”, “fer”, “638”, “644”, and “657” with DangZero fails due to segmentation faults and “vip” fails due to an assertion failure. This incompatibility causes the observed overhead to be larger than those reported in the paper. However, we note that the per-test numbers are close to their reported numbers (see Table 3.3 and Table 3.4). The failed SPEC tests all have small overheads that cause the average overheads to become larger.

B Maximum Working Set Size (WSS) of Each Benchmark Test

Macro benchmark. SPEC and PARSEC in Table 5.1.

Real-world programs. Three real-world programs (Redis, Nginx, Lighttpd) in Table 5.2.

We note that we do not compare with DangZero as most of its memory overheads comes from the page table management and cannot be reflected using WSS, and we do not run mimalloc-bench as its tests are time-sensitive and many tests, such as `glibc-simple`, do not access the allocated memory.

Test ID	SEMalloc	Markus	FFMalloc	MineSweeper	TypeAfterType
600	0.65 (0.00)	1.39 (0.74)	1.59 (0.62)	1.10 (0.69)	0.98 (0.03)
602	1.44 (0.32)	1.19 (0.11)	2.69 (0.69)	1.00 (0.00)	-
605	1.02 (0.20)	1.12 (0.18)	2.96 (1.62)	1.53 (0.53)	1.04 (0.03)
619	0.49 (0.10)	0.49 (0.10)	1.72 (0.84)	1.08 (0.85)	0.93 (0.06)
620	1.53 (0.18)	1.63 (0.29)	3.37 (1.37)	1.63 (0.00)	0.98 (0.06)
623	1.17 (0.29)	1.08 (0.14)	1.92 (0.28)	1.33 (0.00)	1.02 (0.03)
625	1.62 (0.22)	2.88 (1.82)	3.00 (0.00)	1.62 (0.22)	1.01 (0.00)
631	3.28 (2.38)	1.13 (0.23)	2.15 (0.20)	3.50 (2.60)	1.04 (0.00)
638	2.08 (1.32)	1.92 (1.61)	2.42 (0.92)	1.25 (0.14)	1.00 (0.00)
641	0.75 (0.08)	0.53 (0.00)	0.62 (0.09)	0.57 (0.08)	1.00 (0.03)
644	1.25 (0.14)	1.25 (0.14)	1.92 (0.14)	2.00 (1.37)	1.00 (0.00)
657	3.35 (1.40)	0.88 (0.14)	3.03 (1.46)	0.80 (0.16)	1.01 (0.00)
Avg	1.24 (0.04)	<u>1.09 (0.05)</u>	2.00 (0.18)	1.21 (0.20)	**1.00 (0.02)**
bla	1.40 (0.20)	1.30 (0.24)	2.70 (0.24)	1.40 (0.20)	1.01 (0.13)
bod	1.13 (0.16)	1.80 (1.11)	2.93 (1.18)	1.27 (0.13)	1.43 (1.46)
can	1.28 (0.00)	1.28 (0.00)	1.87 (0.21)	1.28 (0.00)	-
ded	1.00 (0.00)	1.07 (0.13)	1.60 (0.13)	1.00 (0.00)	1.15 (0.43)
fer	1.45 (0.55)	-	1.51 (0.24)	-	-
flu	1.34 (0.95)	1.71 (1.16)	1.62 (0.70)	0.69 (0.00)	2.01 (2.82)
fre	1.10 (0.44)	1.51 (1.02)	1.76 (0.46)	1.02 (0.72)	1.47 (1.54)
str	1.01 (0.15)	1.07 (0.24)	1.90 (1.14)	1.43 (1.22)	1.49 (1.60)
swa	1.69 (1.53)	1.69 (1.19)	3.52 (1.22)	1.83 (1.13)	-
vip	0.89 (0.09)	2.77 (0.38)	1.69 (0.09)	1.27 (0.90)	-
Avg	<u>1.14 (0.16)</u>	1.39 (0.23)	1.94 (0.09)	**1.11 (0.09)**	1.15 (0.69)

Table 5.1: Normalized maximum WSS (and standard deviations) of SeMalloc on SPEC and PARSEC. We indicate the best scheme in bold and the second best underlined.

Test ID	SEMalloc	Markus	FFMalloc	MineSweeper	TypeAfterType
Redis	0.99 (0.32)	1.12 (0.04)	1.17 (0.08)	1.14(0.04)	1.00 (0.00)
Nginx	1.02 (0.04)	1.08 (0.03)	1.03 (0.08)	-	1.00 (0.00)
Lighttpd	0.97 (0.27)	0.74 (0.38)	1.08 (0.07)	1.06 (0.00)	1.00 (0.00)
Avg	<u>0.96 (0.15)</u>	**0.93 (0.18)**	1.09 (0.03)	1.10(0.02)	1.00 (0.00)

Table 5.2: Normalized maximum WSS (and standard deviations) of SeMalloc on three real-world programs. We indicate the best scheme in bold and the second best underlined.