SeMalloc: Semantics-Informed Memory Allocator

Ruizhe Wang
University of Waterloo
Waterloo, Ontario, Canada
ruizhe.wang@uwaterloo.ca

Meng Xu
University of Waterloo
Waterloo, Ontario, Canada
meng.xu.cs@uwaterloo.ca

N. Asokan
University of Waterloo
Waterloo, Ontario, Canada
asokan@acm.org

Abstract

Use-after-free (UAF) is a critical and prevalent problem in memory unsafe languages. While many solutions have been proposed, balancing security, run-time cost, and memory overhead (an impossible trinity) is hard.

In this paper, we show one way to balance the trinity by passing more semantics about the heap object to the allocator for it to make informed allocation decisions. More specifically, we propose 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 makes UAF vulnerabilities harder to exploit.

Through extensive empirical evaluation, we show that SeMalloc is realistic: (a) SeMalloc is effective in thwarting all real-world vulnerabilities we tested; (b) benchmark programs run even slightly faster with SeMalloc than the default heap allocator, at a memory overhead averaged from 41% to 84%; and (c) SeMalloc balances security and overhead strictly better than other closely related works.

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

Static analysis, use-after-free, secure memory allocator

ACM Reference Format:
Ruizhe Wang, Meng Xu, and N. Asokan. 2024. SeMalloc: Semantics-Informed Memory Allocator. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670363

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 [49].

This work is licensed under a Creative Commons Attribution International 4.0 License.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670363

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 [8].

There is a wealth of prior research intended to address UAF vulnerabilities (see §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 [2, 52] leading to weaker protection, or extremely fine-grained, treating each heap object as a different "type" [55], 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 SeMalloc, 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 SeMalloc, UAF can only occur between heap objects of the same `SemaType`.

SeMalloc'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 [1], MineSweeper [15], and DangZero [20] (by giving up protection against UAF within the same `SemaType`), and is similar to TypeAfterType [52] (with improved security) and PUMM [56] (with improved usability). SeMalloc incurs an average memory overhead of 61.0% which is much lower than FFMalloc [55] (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 (§2.3);
- 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) (§3-§4); and
- A thorough evaluation of SeMalloc showing that it successfully detects all real-world attacks we tested (§5) with marginal overheads (§6).

Both the software artifact and the full appendix of the paper are available.

2 A Mini SoK on UAF

We present a mini systematization of knowledge (SoK) on techniques that exploit or mitigate UAF vulnerabilities. The purposes of this SoK is to help position SeMalloc in the research landscape, and to identify synergies among UAF-mitigating strategies and defence-in-depth opportunities.

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

Figure 1: A hypothetical example to illustrate UAF exploits.

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

2.1 Exploiting UAF Vulnerabilities

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 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 [20, 26] or other techniques [4, 33];

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 [3, 6]) 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 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" [14] 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 object N. Double free vulnerabilities can be mitigated cheaply by maintaining a set of freed and yet-to-be allocated memory addresses [30, 39, 46] as a top-up of other UAF-mitigation strategies.

Type confusion. The methodology to exploit the UAF bug in Figure 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. 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 1 and 2 are demonstrated in a multi-threaded setting, and indeed many exploits in the wild require some form of race condition to work [25], 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 [13, 21, 24, 35–37].

2.2 Mitigating UAF Vulnerabilities

Attackers' view on how to exploit UAF vulnerabilities (§2.1) also sheds light on how to mitigate UAF, which has been extensively researched. In fact, existing techniques line up nicely as layered protection against UAF vulnerabilities:

```c
struct N {long usr; long pwd; int (*fn)(void);};
void register_real(long uid, long secret) {
  struct N *p = malloc(sizeof(struct N));
  p->fn = __real_fn;
  p->usr = uid;   p->pwd = secret;
  p->fn();
}
void register_fake(long uid, long secret) {
  struct N *x = malloc(sizeof(struct N));
  x->fn = __mock_fn;
  struct O *q = x;
  free(x);            // q is dangling
  q->fn();
  q->usr = uid;   q->pwd = secret;
  reply("Debug: %l | %l", q->usr, q->pwd);
  free(q);
}
```

Figure 2: A hypothetical example to illustrate UAF exploits against objects of the same type.

A) Invalidate dangling pointers, which 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 [26, 44, 53, 57];
- Treat pointers as capabilities to access memory (instead of integers) and `free` revokes the capability [12, 17, 20].

Prior works in this category have demonstrated complete protection against heap-based UAF but may pay the price of compatibility (e.g., CHERI [54]), kernel privilege [20], high overhead (e.g., 80% run-time overhead for DangNull [26]), or subtle complexities as shown in HeapExpo [44].

B1) Prevent heap objects from being allocated over any pointer that might be dangling, which can be achieved by tracking pointer derivation [45] or sweeping all stored pointers [1, 15, 41]. 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 [44]). However, they arguably introduce high overheads. For example, MarkUs [1] more than doubles the run-time on the PARSEC benchmark (see §6.2).

B2) 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 [2, 52, 55], including SeMalloc, differ on their definition of semantics or type, which we give an in-depth reflection in §2.3.

PUMM [56] 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 [2, 52] including SeMalloc.

B3) Confine allocations or reduce the level of certainty on when and which a victim object will be allocated over a dangling pointer. Allocators in this category [30, 39, 46] typically leverage on multiple randomization schemes to boost entropy, such as big bag of pages (BIBOP)-based [22] allocation and delayed freelist. Randomization provides a probabilistic defense against UAF exploits by lowering their success rates. However, entropy-based allocators can still be prone to information leaks or heap fengshui [47] which reduces the level of entropy in practice.

With that said, entropy-based allocators typically incur low overhead (both in run-time and memory). For example, Guarder reports a 3% run-time overhead and 27% memory overhead with its highest level of entropy configuration while SlimGuard [30] further reduces the memory footprint.

C) Validate a pointer upon use. 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 [28, 38, 42] even amongst the works designed to run in production [10, 16, 23, 27, 58] (e.g., 20% run-time overhead for Vik [10]).

Summary. Prior works in categories A, B1, 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 B2, B3) incur smaller overheads at the expense of incomplete protection.

Defense-in-depth: While all building blocks for a layered UAF defense has been proposed, to the best of our knowledge, there is no real-world allocator that combines them, in all or in parts. This SoK shows an opportunity to provide a defense-in-depth solution that holistically integrates memory allocators of all themes of defences.

2.3 A Reflection on Semantics And 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 [2, 52, 55]. 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 [9]), 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.

```
void foo() {
  void *p = malloc(sizeof(int));
  void *q = malloc(sizeof(int));
}
```

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.

```
void foo(bool cond) {
  size_t len = sizeof(int);
  if (cond) {    len = sizeof(long);    }
  void *p = malloc(len);
}
```

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.

```
void wrapper(size_t len) {
  void *p = malloc(len);
}
void foo() {  wrapper(sizeof(int));  }
void bar() {  wrapper(sizeof(int));  }
```

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.

```
void *thread(void *ptr) {
  void *p = malloc(sizeof(int));
}
void foo() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, *thread, NULL);
  pthread_create(&t2, NULL, *thread, NULL);
}
```

Sensitivity in cyclic control-flow structures. In loops and recursive calls, the sensitivity is 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 point to. In this regard, the default `glibc` allocator [19] in most Linux-based systems is (almost) completely insensitive. Regarding the two closely related works, the type definition in Cling [2] adopts a weaker form of context-sensitivity, where the context is defined as two innermost return addresses on the current call stack—an approximation to call trace. Type-after-Type [52] is based on statically-inferred unqualified types native to the programming language enriched by `malloc` wrapper trace.

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 [55] 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 prior works have tried to address UAF vulnerabilities, the challenge of finding the right balance between the level of protection and incurred overhead remains. Certainly manually porting the program has the potential of achieving the trinity of optimizing all memory, run-time, and security [58], in the rest of this paper, we present our approach towards finding such a balance without this aid.

3 Capture Semantics with `SemaType`

We now introduce `SemaType`, a type qualifier [18] tailored to capture the semantics of heap allocations, and showcase how to deduce `SemaType` at runtime through a concrete example.

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 §2.3).

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

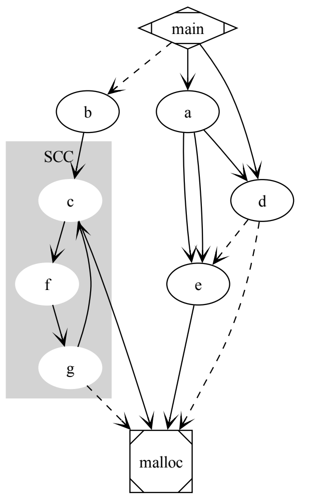
- allocated from the same allocation site (e.g., the very same `malloc` call in the source code); and
- allocated under the same call stack, modulo recursion.

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 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. The code snippet is shown in §A.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).



`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: Call graph (left) of a crafted program §A.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 corresponding function CFG while dashed edges represent function calls inside a loop.

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 §2.3, this can be overly sensitive and may cause a significant memory overhead as in FFMalloc [55].

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; call traces with the same aggregated value are deemed to have the same `SemaType`.

SeMalloc uses the aggregation-based hitmap solution as it provides 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. This helps to reduce the performance impact caused by instrumentation.

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* (§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 (§4.3) ensures that two `SemaType` instances have the same `nID` if and only if their external SCC traces are identical (formally proved in §A.3).

`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` (§4.4). h is computed and stored before an SCC function calls a function not in the current SCC (an outbound call), and is cleared after the call to this SCC (an inbound call) returns.

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 §7 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, only **A** and **B** are non-repetitive; all other `SemaType`s need to be tracked: types **C**—**F** are repetitive because of looping while other types are repetitive due to involvement in recursive calls.

We take **H** 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 §4.5).

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 (§5) shows that `SemaType` in its current form is sufficient to defend against known exploits.

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 (e.g., cache and power usage) is out of scope.

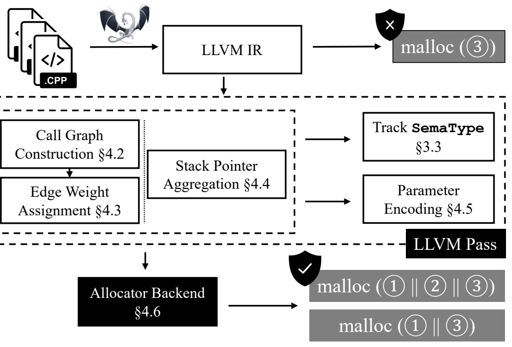


Figure 4: Design overview of SeMalloc (①: flags, ②: `SemaType`, ③: allocation size). The size is the parameter without SeMalloc, while SeMalloc encodes the trace information into the parameter after applying the pass.

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 [32] (for comprehensive and robust call graph construction) and CXXGraph [7] (for graph algorithms). The pass instruments `SemaType` tracking and encoding in the program which eventually passes `SemaType` through common heap allocation APIs. The heap allocator backend takes the encoded information for segregated memory allocation.

Figure 4 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 §4.2), and assigns weights on all edges and nodes in it for `nID` computation (see §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 §4.4). They are encoded into the `size` parameter of an allocation request (see §4.5).

We explain how the allocator backend enforces allocation segregation using `SemaType` in §4.6. For simplicity, we use `malloc` to represent all functions that may request heap memory *directly* from the backend. We refer readers to §A.2 for a complete discussion about how the IR is transformed after applying the pass and how instructions are inserted.

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) 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 [32], i.e., by matching the function type hierarchically. 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) in order to distinguish recurrent allocations vs one-time allocations (see §3.2). This 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 [43] 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 §4.4).

4.3 Edge Weight Assignment

Recall from §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.

```
1 nodes ← topological_sort(DAG)
2 for each n ∈ nodes do
3     w ← 0
4     for each e ∈ n.outgoing_edges do
5         e.weight ← w
6         w ← w + max(1, e.dst.weight)
7     end
8     n.weight ← w
9 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 exist 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 and `e.dst` is a node whose weight has been computed in a previous round, 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 §A.3 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.

4.4 SCC Stack Pointers Aggregation

We use an aggregation approach to track the execution path within SCCs. Before calling each function within the SCCs, we obtain and push the stack pointer into the stack s, which is the aggregation input to compute `rID`.

Algorithm 2: Aggregation of stack pointers.

```
1 h ← 0
2 for each p ∈ s do
3     h ← h ≪ 2 ;  p ← (p ≫ 6) & 0x3 ;  h ← h + p
4 end
5 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 7th and 8th least significant bits (LSBs) of each input to it and shift it left by two bits (line 3-5). We specifically take these two LSBs as stack pointers are 8-byte aligned in the x86 clang environment [31], 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.

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

Offsets	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	Object Size															
16																
32	nID															
48	rID														L	H

Figure 5: Parameter encoding rule for regular objects (L: loop identifier; H: huge block identifier).

recent seven. The current parameter maximizes tracking depth with a minimal 2-bit entropy to differentiate call frames. If recursive calls are shallow or many functions are involved in an SCC, it will be preferable to track fewer layers with more bits taken per pointer.

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 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 63 bits to store the block size. These blocks are allocated via a system call, and launching UAF attacks on them is not trivial (see §A.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 does not have the loop identifier set, and is not going to be released unless the block is big enough that allocated from the OS directly (see §A.7 for details). This indeed is not common as shown in §A.11, where most tests have more than 99% allocations identified with recurrent `SemaType`.

Tunable parameters. While Loop (L) and huge (H) indicators are both 1-bit and are not tunable, other parameters can be tuned to fit specific program or security requirements.

- *`size`* (default 32-bit): A smaller `size` leaves more room for `nID` and `rID`, but may impair functionality: if legacy code or external libraries that are not transformed by SeMalloc incur larger allocations, the overflowed bits will be taken as `SemaType`, causing the allocated block to be smaller than required. The 32-bit default is an empirical number based on programs we evaluated.
- *`nID`* (default 16-bit): Ideally, the size of `nID` should be the smallest number that satisfies $2^{sizeof(nID)} \geq \sum P_i$ for a target program, where P_i represents the number of traces to reach an allocation site i in the flow-sensitive call graph after DAG-reduction (§4.2). We take the 16-bit default to support all tested programs and benchmarks (see §6).
- *`rID`* (default 14-bit): Given a 64-bit pointer, the size of `rID` is passively decided by `size` and `nID`. However, the calculation of `rID` is tunable, as discussed in §4.4.

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 BIBOP [22] for 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 §2.3) it allocates a global BIBOP pool for one-time `SemaType`s and 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` [29] 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`. We refer readers to §A.7 for additional implementation details of this runtime backend.

5 Security Analysis

We provide both qualitative analysis and empirical evidence on the effectiveness of SeMalloc in mitigating UAF exploits.

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 SeMalloc, 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 (§3.2). 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 SeMalloc, 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 SeMalloc.

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 (§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 (§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 SeMalloc 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 $\sum P_i$ `SemaTypes` 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 §2.3.
- *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 §2.3.
- *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.

Table 1: SeMalloc is effective in thwarting (●) exploitation of all real-world UAF vulnerabilities evaluated while TypeAfterType [52] and Cling [2] provide no protection (○) or partial protection (◐) to most vulnerabilities. †: Cling is not open-sourced and is only analyzed conceptually.

Vulnerability	**Exp.** (§2.1)	**[52]**	**[2]**†	**SeMalloc**
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	●	◐	●

Effectiveness evaluation. To show how `SemaType` diversifies heap allocation, we compare the number of different allocation sites using `SemaTypes` and pure object types in the last two columns of §A.11 based on programs in the PARSEC3 [5] and SPEC 2017 [48] 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.

5.2 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 [2] and TypeAfterType [52], while other allocators used in performance evaluation (§6) either have theoretically complete UAF-mitigation [1, 15, 20, 55] or requires case-by-case manual annotation to work (e.g., PUMM [56]).

Tested vulnerabilities are summarized in Table 1. They are selected from three sources: representative CVEs from DangZero [20], TypeAfterType [52], uafBench [34], and further enriched with additional vulnerabilities selected by us to cover the exploitation types discussed in §2.1. We present two representative examples here and two more in §A.6.

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 [40], and we are not aware of any exploitation against SeMalloc—confirming that SeMalloc can help confine UAF exploitability in practice.

Case study: mjs-issue-78 [34]. 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 SemaTypes 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 [36]. 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.

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 two type-based allocators: FFMalloc [55] and TypeAfterType [52], two complete UAF-mitigating allocators: MarkUs [1] and MineSweeper [15], and the glibc memory allocator [19] on the same test suites. We additionally compare with DangZero [20] on all benchmarks and PUMM [56] on targeted server programs (§6.4) as both are state-of-the-art UAF-mitigating allocators with low overheads despite DangZero requires kernel privilege and PUMM requires annotation and profiling. Although Cling [2] 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 incur a:

A lower run-time overhead than MarkUs and MineSweeper,
B lower memory overhead than FFMalloc,
C similar run-time overhead compared with TypeAfterType and potentially a higher memory overhead,
D smaller run-time overhead and similar memory overhead with DangZero,
E similar run-time and memory overheads with PUMM.

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).

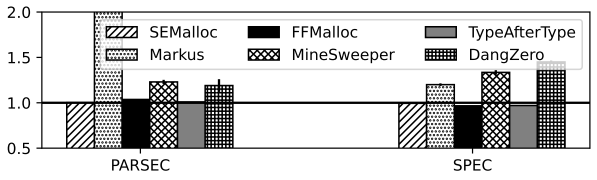


Figure 6: Normalized average and standard deviation of run-time overhead on PARSEC and SPEC benchmarks.

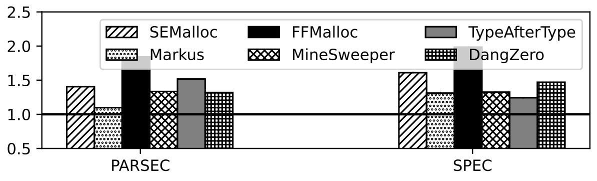


Figure 7: Normalized average and standard deviation of memory overhead on PARSEC and SPEC benchmarks.

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 [51] 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 SeMalloc-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.8. Individual test results are at §A.9. Maximum working set size is at §A.10.

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 [48] version 1.1.9 and report the results of 12 C/C++ tests in both "Integer" and "Floating Point" test suites. We note that some tests run the executable multiple times with different inputs. We report the sum of time and the max of memory use for these tests.

PARSEC 3: We use the latest PARSEC 3 [5] benchmark, excluding two ("raytrace" and "facesim") from analysis because they are incompatible with the Clang compiler, and one ("x264") as it causes a segmentation fault with glibc.

Benchmark Performance. We provide the run-time and memory overheads as well as standard deviations of these benchmarks in Figure 6 and Figure 7. Performance results of individual programs are in §A.9.

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 `SemaTypes` 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.8. 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

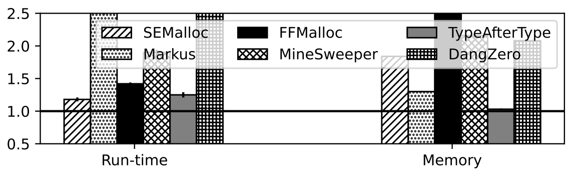


Figure 8: Normalized average and standard deviation of run-time and memory overheads on mimalloc-bench.

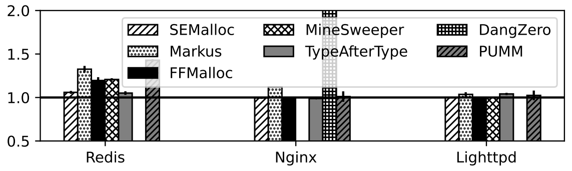


Figure 9: Normalized average and standard deviation of throughput overhead on three real-world programs.

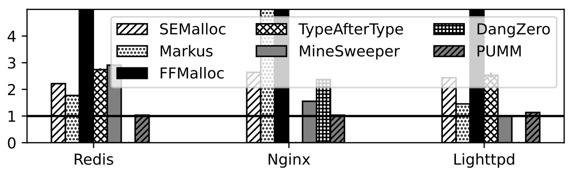


Figure 10: Normalized average and standard deviation of memory overhead on three real-world programs.

observed overhead comes from the data storage instead of the way SeMalloc reuse freed blocks.

6.3 Micro Benchmarks

We use mimalloc-bench [11], 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 8. Individual results can be found in §A.9.

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

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) [50] 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 [11].

The results are in Figure 9 and Figure 10. Overhead numbers can be found in §A.9. 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 these outliers, the results align with our expectations for SeMalloc set earlier in the beginning of §6.

7 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 §3.2, §3.3, and §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 [55];
- 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 [19].

Fortunately, most programs are not written in these extreme cases as shown in the last two columns of §A.11. 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.

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 handle a significant amount of memory reallocation (as shown in the fourth to the last column), which improves memory efficiency and thus explains why empirically SeMalloc incurs a lower memory overhead than FFMalloc and limits memory leakage.

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 dependent libraries with SeMalloc for better compatibility.

8 Concluding Remarks

Type is a loosely defined concept in security research and is often subject to different interpretations. In this paper, we look at "type" through the lens of heap allocators. To a heap allocator, a type is an encoding of information about the to-be-allocated object. Intuitively, the more information (i.e., semantics) a heap allocator knows, the better decisions it can make. While conventional memory allocators only take object size as the semantics, we argue that `SemaType` can be a useful extension to object size, and, more importantly, can be deduced cheaply at runtime (memory overhead of SeMalloc is not related to deducing `SemaType`).

Through SeMalloc, we show that `SemaType` can be used to balance security, run-time cost, and memory overhead in UAF mitigation. And yet we believe that `SemaType` is applicable beyond SeMalloc. For example, when applied to a performance-oriented memory allocator, `SemaType` might help the allocator to partition heap pools strategically to exploit cache coherence. Alternatively, combined with other software fault isolation (SFI) strategies (e.g., red-zoning or pointer-as-capabilities), `SemaType` might help specify and enforce finer-grained data access policies.

Acknowledgement. This work is funded in part by NSERC (RGPIN-2022-03325), the David R. Cheriton endowment, and research gifts from Intel Labs.

References

[1] Sam Ainsworth and Timothy M. Jones. 2020. MarkUs: Drop-in Use-After-Free Prevention for Low-level Languages. In *IEEE S&P*. 578–591.
[2] Periklis Akritidis. 2020. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *USENIX Security*.
[3] Alejandro Guerrero. 2022. N-day exploit for CVE-2022-2586: Linux kernel nft_object UAF.
[4] Cristiano Giuffrida Alyssa Milburn, Herber Bos. 2017. Safelnit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities.
[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. *The PARSEC Benchmark Suite: Characterization and Architectural Implications*. Technical Report TR-811-08. Princeton University.
[6] Blaze Labs. 2022. The never ending problems of local ASLR holes in Linux.
[7] Matteo Botticci. 2022. *ZigRazor/CXXGraph: Release v0.2.2*.
[8] C Language Working Group. 2023. Programming languages — C.
[9] Luca Cardelli. 1996. Type Systems. *ACM Computing Surveys*.
[10] Haehyun Cho, Jinbum Park, Adam Oest, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. 2022. ViK: practical mitigation of temporal memory safety violations through object ID inspection. In *ASPLOS*. Association for Computing Machinery, New York, NY, USA.
[11] daanx. 2024. Suite for benchmarking malloc implementations.
[12] Thurston H. Y. Dang, Petros Maniatis, and David Wagner. 2017. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *USENIX Security*. Vancouver, Canada.
[13] Daniel Teuchert, Cornelius Aschermann, Tommaso Frassetto, Tigist Abera. 2018. Use after free in File#initilialize_copy.
[14] Thomas Dullien. 2017. Weird Machines, Exploitability, and Provable Unexploitability. *IEEE Trans. on Emerging Topics in Computing* (2017).
[15] Márton Erdős, Sam Ainsworth, and Timothy M. Jones. 2022. MineSweeper: A "Clean Sweep" for Drop-in Use-after-Free Prevention. Association for Computing Machinery, New York, NY, USA, 212–225. https://doi.org/10.1145/3503222.3507712
[16] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. 2020. PTAuth: Temporal Memory Safety via Robust Points-to Authentication.
[17] 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. 2020. Cornucopia: Temporal Safety for CHERI Heaps. In *IEEE S&P*. 608–625.
[18] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A Theory of Type Qualifiers. In *PLDI*.
[19] Free Software Foundation, Inc. 2024. The GNU Allocator.
[20] Floris Gorter, Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2022. DangZero: Efficient Use-After-Free Detection via Direct Page Table Access. In *ACM CCS*. Association for Computing Machinery, New York, NY, USA.
[21] Hanno Böck. 2017. use after free with malformed input file in yasm_intnum_destroy().
[22] David R. Hanson. 1980. A Portable Storage Management System for The ICON Programming Language. *Software: Practice and Experience*.
[23] Mohannad Ismail, Andrew Quach, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. 2022. Tightly Seal Your Sensitive Pointers with PACTight.
[24] John Leitch. 2015. array.fromstring Use After Free.
[25] Moshe Kol. 2023. Racing Against the Lock: Exploiting Spinlock UAF in the Android Kernel. In *OffensiveCon*.
[26] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2021. Preventing Use-after-free with Dangling Pointers Nullification. In *NDSS*.
[27] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. 2022. PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication. In *ACM CCS*. Association for Computing Machinery, New York, NY, USA.
[28] Hao Ling, Heqing Huang, Chengpeng Wang, Yuandao Cai, and Charles Zhang. 2024. GIANTSAN: Efficient Memory Sanitization with Segment Folding. In *ASPLOS*. Association for Computing Machinery, New York, NY, USA.
[29] Linux Foundation. 2024. mmap(2) - Linux manual page.
[30] Beichen Liu, Pierre Olivier, and Binoy Ravindran. 2019. SlimGuard: A Secure and Memory-Efficient Heap Allocator. In *MiddleWare*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3361525.3361532
[31] LLVM Project. 2024. X86CallingConv.td.
[32] Kangjie Lu and Hong Hu. 2019. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *ACM CCS*. Association for Computing Machinery, New York, NY, USA, 1867–1881.
[33] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *CCS*.
[34] Manh Nguyen. [n. d.]. UAF Fuzzing Benchmark.
[35] National Vulnerability Database. 2015. CVE-2015-6831.
[36] National Vulnerability Database. 2015. CVE-2015-6835.
[37] National Vulnerability Database. 2018. CVE-2018-11496.
[38] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*.
[39] Gene Novark and Emery D Berger. 2010. DieHarder: Securing The Heap. In *ACM CCS*. Association for Computing Machinery, New York, NY, USA.
[40] OffSec Services Limited. 2024. Exploit Database - Exploits for Penetration Testers, Researchers, and Ethical Hackers.
[41] Chanyoung Park and Hyungon Moon. 2024. Efficient Use-After-Free Prevention with Opportunistic Page-Level Sweeping. In *NDSS*.
[42] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *USENIX ATC*.
[43] M. Sharir. 1981. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*.
[44] Zekun Shen and Brendan Dolan-Gavitt. 2020. HeapExpo: Pinpointing Promoted Pointers to Prevent Use-After-Free Vulnerabilities. In *ACSAC*.
[45] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. 2019. CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C++. In *NDSS*.
[46] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. 2018. Guarder: A Tunable Secure Allocator. In *USENIX Security*.
[47] Alexander Sotirov. 2007. Heap Feng Shui in Javascript. In *Black Hat Europe*.
[48] Standard Performance Evaluation Corporation. 2017. SPEC 2017.
[49] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *IEEE S&P*.
[50] The Apache Software Foundation. 2023. Apache HTTP Server Documentation: ab - Apache HTTP Server Benchmarking Tool.
[51] Tristan Ravitch. 2023. whole program llvm.
[52] Erik van der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. 2018. Type-After-Type: Practical and Complete Type-Safe Memory Reuse. In *ACSAC*.
[53] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-Free Detection. In *EuroSys*.
[54] 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. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE S&P*.
[55] Brian Wickman, Hong Hu, Insu Yun, DaeHee Jang, JungWon Lim, Sanidhya Kashyap, and Taesoo Kim. 2021. Preventing Use-After-Free Attacks with Fast Forward Allocation. In *USENIX Security*.
[56] Carter Yagemann, Simon P Chung, Brendan Saltaformaggio, and Wenke Lee. 2023. PUMM: Preventing Use-After-Free Using Execution Unit Partitioning. In *USENIX Security*.
[57] Yves Younan. 2015. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*.
[58] Jie Zhou, John Criswell, and Michael Hicks. 2023. Fat Pointers for Temporal Memory Safety of C. *Proc. ACM Program. Lang.*

A Appendix

A.1 Code Example

This appendix is removed due to the page limitation. Please refer to our full appendix.

A.2 Instruction Insertion Summary

The number of LLVM IR instructions instrumented at different code locations are summarized in Table 2.

Table 2: 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.

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	

Briefly, following is a summary of instructions added:

- 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 SeMalloc. We refer the readers to §A.5 for details.

A.3 Formal Analysis

This appendix is removed due to the page limitation. Please refer to our full appendix.

A.4 List of Supported Allocation APIs

SeMalloc supports the following allocation APIs: `malloc`, `calloc`, `realloc`, `memalign`, `pthread_memalign`, and `aligned_alloc`.

A.5 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.

A.6 Additional Exploitation Case Studies

Python-24613. [24] 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 §2.1).

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. [52] 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.

A.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`.

Memory leakage. In **??**, we use the third to the last column to list memory leakage caused by SeMalloc. We show that programs not using memory-allocating external libraries have negligible memory leakage.

A.8 List of Failed Tests and Influences on the Average Overheads

This appendix is removed due to the page limitation. Please refer to our full appendix.

A.9 Run-time and Memory Overhead Details of Each Benchmark Test

This appendix is removed due to the page limitation. Please refer to our full appendix.

A.10 Maximum Working Set Size (WSS) of Each Benchmark Test

This appendix is removed due to the page limitation. Please refer to our full appendix.

A.11 Recurrent Allocation Statistics on PARSEC and SPEC Tests

This appendix is removed due to the page limitation. Please refer to our full appendix.