CMASan: Custom Memory Allocator-aware Address Sanitizer

Junwha Hong
Department of Computer Science
UNIST

Wonil Jang
Department of Computer Science
UNIST

Mijung Kim
Department of Computer Science
UNIST

Lei Yu
Department of Computer Science
Rensselaer Polytechnic Institute

Yonghwi Kwon
Department of Electrical and
Computer Engineering
University of Maryland

Yuseok Jeon†
Department of Computer Science
UNIST

***Abstract*—Custom Memory Allocator (CMA) replaces the standard memory allocator for various purposes, such as improving memory efficiency or enhancing security. However, memory objects allocated by CMA are vulnerable to memory bugs similar to those allocated by the standard memory allocator. Unfortunately, existing memory bug detection approaches, including Address Sanitizer (ASan), do not work properly with these CMAs because existing approaches are mainly designed for the standard memory allocator.**

This paper presents CMASan, the first CMA-aware address sanitizer designed to effectively detect memory bugs on CMA objects that ASan misses without requiring expert knowledge, manual code modifications, or changing the unique internal logic of CMAs. According to our evaluation, CMASan successfully identifies 19 previously unknown CMA memory bugs undetected by ASan, including some undetected for 9 years. Compared to ASan, CMASan incurs only an additional 9.63% overhead.

1. Introduction

Custom Memory Allocator (CMA) is used as an alternative to the standard memory allocator for various purposes [1–6], such as cache utilization, memory efficiency, and security. These CMAs are widely used in the real world. For example, among the top 100 C/C++ applications sorted by GitHub stars, CMAs have been adopted in 44% of the applications. However, objects allocated by CMAs can be vulnerable to invalid memory access due to memory bugs (e.g., buffer overflow), similar to those allocated by the standard memory allocator.

Unfortunately, existing memory bug detection approaches are designed for the standard memory allocator, and therefore, they are unable to adequately detect memory bugs related to CMAs, which have unique internal CMA logic and different CMA API names and types (e.g., clear API). To address this, several approaches have been proposed for detecting CMA-related memory bugs. These approaches can be broadly classified into two categories: symbolic execution-based approaches [7–11] and Address Sanitizer (ASan)-based [12] manual code modification approaches [13–17].

Symbolic execution-based detection approaches for CMA memory bugs are restricted by the inherent limitations [18] of symbolic execution, such as path explosion, limited handling of complex data structures, and difficulties with real-world inputs, preventing them from detecting all types of CMA memory bugs in large and complex programs.

ASan, unlike symbolic execution, performs instrumentation at compile time and verification during runtime using precise real-time data. Consequently, ASan does not suffer from the same limitation as symbolic execution, thereby allowing for identifying memory bugs in large and complicated programs. In fact, ASan is the most widely adopted approach [19] for memory bug detection, and ASan identifies a significant number of bugs in C/C++ programs. For instance, ASan has identified more than 10,000 memory bugs [20–22] across different software.

While ASan effectively detects memory bugs, it does not properly cover memory bugs related to CMA objects, as it only tracks objects allocated by standard memory allocators (e.g., `malloc()`). To address these limitations, the Shim [13, 14] approach and the ASan manual poisoning APIs [15–17] have been proposed to support the detection of CMA memory bugs through ASan. This Shim approach provides a mode to disable CMA and replace it with a standard allocator, thereby enabling ASan to treat CMA as a standard allocator. However, applying the Shim approach requires an understanding of the target application code and manual modifications. Furthermore, not all CMAs are compatible with Shim because their diverse and unique CMA internal logic can complicate precise mapping with the standard memory allocator, resulting in less accurate detection.

The ASan manual poisoning APIs support manual poisoning while maintaining CMA's logic. However, using these APIs also requires code modifications and expert knowledge of the target program and CMA. Additionally, users need to manually modify CMA to secure redzone areas for each CMA object and implement separated Quarantine zones

†. Corresponding author

to delay the freeing of CMA objects for a certain period. Consequently, the application of both the Shim approach and ASan manual poisoning APIs to real-world programs is rare due to these limitations, including the need for manual code modification and expert knowledge, and limited CMA API coverage. According to our survey of the top 100 C/C++ applications ranked by GitHub stars, only ten applications have adopted Shim. Among these, CMAs in five applications are not fully covered by Shim. Furthermore, only three applications utilize ASan manual poisoning APIs.

We present CMASan, the first CMA-aware Address Sanitizer, designed to detect CMA-related memory bugs in applications that utilize CMAs, without requiring expert knowledge, code modification, or alterations to the internal logic of the target CMAs. First, at static time, CMASan identifies the CMA APIs and instruments these APIs for memory bug detection. More specifically, CMASan identifies CMA API candidates using a widely-used static analyzer, CodeQL [23], and then categorizes these candidates through CMASan's semi-automated categorization procedure. For each identified CMA API (e.g., alloc, realloc, and free), CMASan automatically inserts the corresponding ASan instrumentation. Note that to preserve the internal logic of the CMA allocator, CMASan does not replace the allocator with ASan's allocator like ASan. Furthermore, CMASan handles special APIs that are used uniquely and frequently in CMA. For example, CMA frequently uses the clear API to release multiple objects at once, a practice observed in 20% of arena-type CMA cases, instead of individually freeing objects through the free. By handling such clear APIs, CMASan can accurately detect DF and UAF that occur after clear operations.

In runtime, for fast and accurate memory detection related to CMA objects, it is important to efficiently manage the CMA object-related metadata generated by CMASan. To achieve this, CMASan utilizes a customized two-level table to handle metadata efficiently. Additionally, CMASan carefully manages the lifecycle of the metadata to maximize its detection capability by always maintaining the most up-to-date, accurate information.

To enhance the detection capability for temporal memory violations while preserving the logic of CMA, CMASan introduces instance-specific quarantine zones to maximize the duration of objects in the freed state. Unlike ASan, CMASan's quarantine zone delays free requests by only marking them in the redzone without actually freeing them. Additionally, when multiple allocators share a single quarantine zone, it can result in inaccurate detections of Use-After-Free (UAF) and Double-Free (DF). To address these issues, CMASan proposes an individual quarantine zone approach, maintaining a separate quarantine zone for each allocator instance.

Lastly, to further improve the accuracy of CMA bug detection, it is necessary to appropriately handle the various false positive reports. For example, CMASan manages redzones without replacing existing CMAs with ASan's internal memory allocators. This approach preserves the internal behavior of CMAs, which could lead to false positives, such as metadata access on freed objects. To prevent these CMA-related false positive issues, CMASan proposes four different false positive suppression techniques.

In our evaluation, we first identify a significant number of memory accesses to CMA objects—for example, in `ncnn` [24], 87% of memory accesses are related to CMA—that ASan is unable to verify. Due to CMASan's increased detection coverage for checking memory access to such CMA objects, CMASan has successfully detected ten known CMA bugs that native ASan does not detect, without any false positive reports. In addition, CMASan identifies 19 new unknown CMA memory bugs in real-world applications. All these 19 bugs have been reported, and twelve have received confirmation. Out of these, six have been patched and assigned CVE IDs. Interestingly, CMA memory bugs detected by CMASan in `SQLite3` and `PHP` are triggered by their own unit tests. However, due to the limitations of existing approaches, these bugs remained undetected for two years in `PHP` and nine years in SQLite. Furthermore, due to several CMASan's optimization techniques, such as the two-level metadata table, CMASan incurs only an additional 9.63% performance overhead and 14.83% memory overhead on average compared to native ASan.

In summary, this paper makes the following contributions:

- We propose the first CMA-aware address sanitizer designed to improve detection capabilities, allowing for the identification of CMA memory bugs without requiring specific expert knowledge and code modification.
- We propose several techniques that handle CMA's unique logic to improve the precision of detecting CMA memory bugs and minimize false positives.
- In our evaluation, CMASan identifies 19 previously unknown real-world CMA memory bugs, of which 5 are acknowledged, 7 have been fixed, and 6 of them received CVEs.

2. Background and Motivation

2.1. Mechanism of Address Sanitizer

Address Sanitizer (ASan) pre-allocates shadow memory at a ratio of 1 byte of shadow memory for every 8 bytes of user-accessible memory for storing the redzone statuses to validate accesses during load/store instructions [25]. When allocating an object, surrounding areas are marked as redzones to detect Buffer-Overflow (BOF). ASan achieves this by replacing standard memory allocator APIs (e.g., `malloc`, `realloc`, or `calloc`) with its internal allocator APIs, which allocate extra space surrounding the requested-sized object to accommodate space for redzones. This extra space for redzones helps detect BOF memory access.

Similarly, to detect Use-After-Free (UAF) and Double-Free (DF), ASan replaces standard memory allocator APIs (e.g., free, or realloc) with its internal ASan allocator APIs. ASan also marks a freed object as a redzone, allowing the detection of subsequent accesses to and deallocation API calls on the object as UAF and DF, respectively. However,

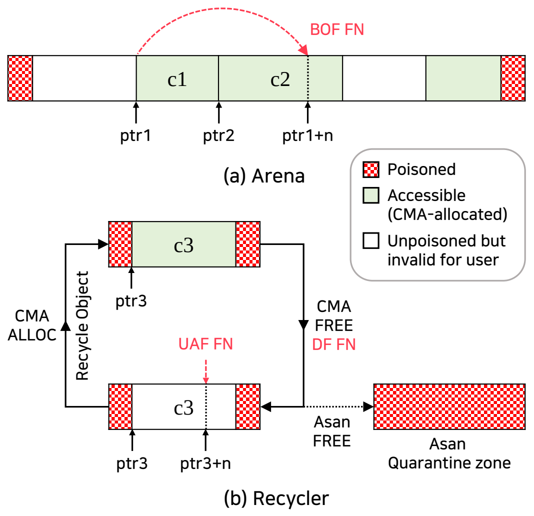

Figure 1: Two examples describe the Arena and Recycler patterns and possible false negatives.

```
// ALLOC (Arena)
void* allocate(size_t size) {
    ...
    if (FOLLY_LIKELY((size_t)(end_ - ptr_) >= size)) {
      // Fast path: there's enough room in the current block
      char* r = ptr_;
      ptr_ += size;
      assert(isAligned(r));
      return r;
    }
    ...
    return r;
}
```

```
void clear() {
    ...
    currentBlock_ = blocks_.begin();
    char* start = currentBlock_->start();
    ptr_ = start;
    end_ = start + blockGoodAllocSize() - sizeof(Block);
    ...
}
```

Figure 2: An example of the Arena pattern and clear API in `folly`

merely marking redzones (i.e., poisoning) is insufficient, as the memory allocator tends to recycle the freed memory in subsequent allocations for efficient memory utilization. This updates the shadow memory of the object as accessible (unpoisoning), making it challenging to detect UAF and DF. To address this, ASan places freed objects into a Quarantine zone, delaying their recycling until the zone reaches a certain capacity and extending the duration for bug detection.

While ASan can effectively detect BOF, UAF, and DF with a high detection rate and low false positive rate using shadow memory and the Quarantine zone, it has limitations when dealing with objects allocated by CMAs due to its inability to recognize CMAs. ASan Manual Poisoning API, which allows for manual poisoning of CMA objects, or a Shim mode that supports replacing a CMA with a standard memory allocator, offers solutions to address the issue.

However, using ASan manual poisoning API requires tester's expertise in understanding the target application's code and its CMA structures, as well as timely poisoning. Due to these challenges, our survey of the top 100 C/C++ applications on GitHub, ranked by stars, revealed that only three have implemented the ASan manual poisoning API. Moreover, because poisoning must occur whenever a CMA free takes place, and timely unpoisoning is necessary due to quick recycling for temporal locality, the detection period for UAF and DF is limited. This limitation arises from the absence of a Quarantine zone in the manual poisoning API. Another approach, Shim mode, offers a solution by designing CMA with interfaces identical to libc, allowing ASan to recognize CMA as a libc allocator during testing [14]. However, it also requires understanding the target application's code and its CMA structures, along with having allocators with the same interface as libc. In some cases, such as in-place realloc and clear CMA APIs, where there is no direct equivalence with standard APIs (e.g., malloc, realloc, calloc, or free), discrepancies can arise between the Shim mode and the actual program behavior. These issues will be discussed in detail in §2.3. Despite efforts by developers to design CMA to support Shim mode, there may be instances where certain CMAs are overlooked. Among the surveyed 100 applications, Shim mode is applied in 10 applications, but five of them only supported the Shim mode for one type of CMA while overlooking others.

Therefore, there is a pressing need for a solution that automates redzone allocation, recognizes CMA lifecycles, and implements Quarantine zones without manual intervention. Such a solution would address the gaps in current approaches and enhance the effectiveness of memory error detection in applications using CMAs.

2.2. CMA Patterns against ASan

In this section, we thoroughly examine the limitations of ASan in detecting memory access errors by analyzing the top 100 C/C++ applications on Github ranked by star ratings. We identify 78 CMAs from 44 applications. Additionally, we identify two common CMA patterns, Arena and Recycler, within these CMAs as the root causes of ASan's false negatives on CMAs.

Arena. Among the investigated CMAs, 69% follow an Arena pattern, which involves allocating a large memory region, or arena, from the upstream allocator (e.g., standard allocator) and then subdividing this region (or arena) for the allocation requests by updating offsets from the start address. However, ASan only recognizes the arena, not each subdivided object, and thus inserts redzones solely around the whole chunk, as shown in Figure 1-(a). Therefore, ASan fails to detect BOF for adjacent objects allocated within the arena. For instance, consider the case when CMA allocates an object, $c1$, with base pointer $ptr1$ and size $sz1$, and then allocates another object, $c2$, with base pointer $ptr2$ and size $sz2$ immediately after. If a user mistakenly accesses $ptr1 + n$

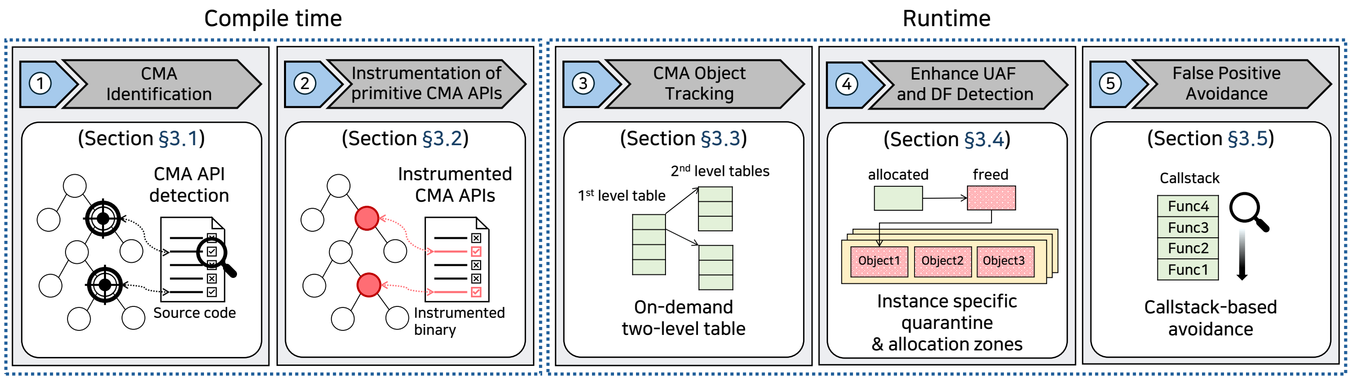

Figure 3: Overview of CMASan

where $ptr2 \leq ptr1 + n < ptr2 + sz2$, it should be identified as a BOF. However, since ASan cannot insert a redzone between CMA objects allocated within the arena, ASan misses this BOF, which results in a false negative. To detect such BOFs, a redzone area must also be allocated between objects.

Recycler. Among the investigated CMAs, 45% follow a Recycler pattern, utilizing internal data structures (e.g., free-list) to collect and reuse objects requested for deallocation. This allows a single object allocated by a standard allocator to undergo multiple lifecycles. However, as ASan cannot recognize these lifecycles, it may fail to detect UAF and DF. In Figure 1-(b), object $c3$ is initially allocated using the standard allocator, then re-allocated by CMA alloc, deallocated by CMA free, and recycled again by CMA alloc. This cycle repeats until the object enters the ASan quarantine zone via a standard free. As ASan does not poison the object as a redzone in CMA free (due to its inability to identify CMA free), both DF and UAF for object $c3$ are missed. Furthermore, even if manual poisoning is performed in CMA free using ASan manual poisoning API, the absence of a quarantine zone can hinder the detection of UAF and DF due to quick object reuse to achieve temporal locality in CMA, as discussed in §2.1.

2.3. Challenges of Applying Shim Mode

In-place realloc and clear API are the two illustrative examples where applying the Shim mode to replace a CMA with a standard allocator is challenging due to the lack of direct 1:1 correspondence between the functions.

2.3.1. In-place Realloc. Typical realloc function adjusts the size of a given object to the requested size, but it does not guarantee that the resizing will occur in the same memory location [26]. In contrast, in-place realloc is a function that changes the size of a previously allocated object without changing its memory location. For example, `MicroPython`'s `gc_realloc` supports resizing in-place through user flags [27]. If `gc_realloc` is simply replaced by ASan-supported realloc, and the object returned by realloc is allocated at a different location than the old object, it can disrupt the program logic and lead to false positives.

2.3.2. Clear APIs. In the case of Arena-type CMAs described in §2.2, the clear API, which resets arena chunks back to the beginning in one go, is frequently used. Among the 54 Arena-type CMAs investigated, 20% are found to utilize the clear API to initialize the Arena to the base pointer or to collect each object internally. However, since objects allocated by the CMA are deallocated all at once by resetting the pointer to the Arena's base pointer, it is difficult to apply the Shim approach by simply replacing the clear API with the standard allocator's free API. This is because such a replacement requires managing a list of allocated objects and releasing them all at once.

For instance, examining the example of the Arena allocator in `folly` [28] shown in Figure 2, objects are allocated incrementally from the arena chunk initialized in the allocate function as in Line 6 to 9 of the function `allocate`, by adding the `size` to `ptr_`. Once the usage of all objects is deemed complete, a clear API is used to reset `ptr_` to `start` in Line 5, allowing objects to be allocated from the base of the arena. Therefore, the usage of objects after the clear API operation should be flagged as UAF. However, since the clear API does not directly translate to a libc equivalent and cannot be simply replaced by `free`, applying a Shim to recognize the deallocation of objects after a clear API poses challenges. Indeed, among the CMAs investigated in this study, Shim is not applied to clear APIs. Therefore, a method to properly handle such clear APIs and detect related UAF and DF is necessary.

3. Design

CMASan aims to detect memory bugs in applications that utilize CMAs, by leveraging and extending ASan's heap memory bug detection capability. As shown in Figure 3, it consists of five components. First, at static time, CMASan identifies CMA APIs and categorizes them based on their objectives using CodeQL-based CMA allocation API recognition and a semi-automated categorization procedure (§3.1). Next, CMASan instruments CMA APIs to insert redzones around the objects and poison the objects after freeing, while maintaining the internal logic of CMA (§3.2).

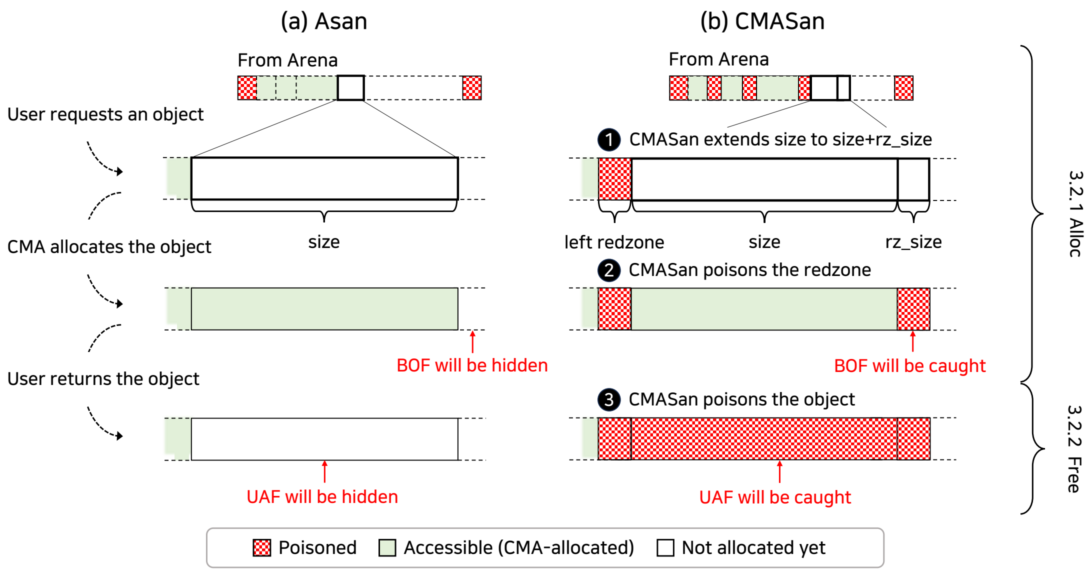

Figure 4: Visualization of CMASan instrumentation

At runtime, CMASan efficiently manages the additional metadata required for detecting CMA memory bugs by utilizing a two-level table. It also ensures proper handling of the lifecycles of these metadata (§3.3). To enhance UAF and DF detection capabilities, CMASan introduces instance-specific quarantine zones to maximize the duration of objects in the freed state. In addition, allocation zones are implemented to support the detection of invalid access following clear API calls (§3.4). Finally, CMASan avoids several potential false positives that can arise from the various internal logic of CMAs and the instrumentation itself (§3.5).

3.1. CMA Identification

To ensure CMASan identifies all CMAs in the target application, we first recognize CMA allocator candidates using CodeQL-based CMA allocation API recognition. We then categorize these candidates and additional CMA-related functions (e.g., realloc or clear) through our semi-automated categorization procedure.

3.1.1. CodeQL-based CMA Allocation API Recognition. To identify the allocation API candidates, we extend the prototype-based rule in the `HeuristicAllocationFunction` class offered by CodeQL [29]. This rule identifies CMA allocation API candidates by checking for the presence of the "alloc" keyword in the function name, whether it returns a pointer, and whether it includes a size argument. However, this rule cannot detect various forms of CMA allocation APIs, such as those without the "alloc" keyword. To address this, we additionally handle new keywords (e.g., acquire, memory) based on our analysis of the top 100 C/C++ applications in Github. While expanding the keyword set increases detection coverage, it may also lead to the inclusion of non-CMA functions. To filter out these non-CMA functions, we further verify whether the size argument is used in comparison or pointer operations to allocate the requested size internally and whether the returned object is reachable from internal CMA metadata storage.

3.1.2. Semi-automated Categorization Procedure. The semi-automated categorization procedure first identifies alloc, realloc, and calloc APIs from the allocation API candidates by utilizing the size and pointer argument information and tracking the flow of these arguments. Then, it groups the identified functions into families based on the source locations of their definitions and collects other family functions defined in the same module (source and header files). Finally, it iterates through all functions in each CMA family to identify free and clear APIs for UAF and DF detection, as well as object accessing, size querying, and object collecting APIs for false positive avoidance, as will be discussed in §3.5. To facilitate this, CMASan implements Algorithm 1, detailed in Appendix §A, within a script, enabling users to easily conduct this semi-automated process based on the rules defined in Table 6 in Appendix §A. More specifically, CMASan provides users with the code of each function body along with the corresponding rules to determine the type of that function.

3.2. Instrumentation of primitive CMA APIs

CMASan instruments primitive CMA APIs (alloc, realloc, and free) to secure the redzone space for detecting Buffer-OverFlow (BOF) and poison (or unpoison) objects to detect Use-After-Free (UAF) and Double-Free (DF). CMASan carefully implements and places fine-grained, variable-sized redzones without modifying CMA's internal logic.

3.2.1. Alloc. Figure 4 shows an example of Arena type CMA which internally allocates a large memory chunk first and returns a smaller chunk of the requested size from this base chunk. Since it only uses the standard memory allocator for the allocation of the base chunk, ASan places redzones solely around the base chunk, not individual CMA-allocated objects. As a result, ASan could fail to detect Buffer-Overflow (BOF) bugs for these CMA objects. Unfortunately, a naive approach of poisoning around each CMA object would result in overlapping redzones with adjacent objects. To handle this, CMASan instruments the CMA allocation APIs by adjusting the allocation size argument to be the given size plus the redzone size (❶ in Figure 4) and poison the redzone area (❷).

Note that we also handle calloc and typed alloc where the arguments of those functions are not the memory size in bytes (e.g., typed alloc's input is the number of particular data type objects to be allocated). In such cases, the requested size is calculated as $num * granularity$, where num is the argument indicating the number of objects to be allocated, and $granularity$ is the object size derived from another argument or the return type. To support calloc and typed alloc, we add the redzone size to num, and the size of the redzone is calculated by dividing the byte size of the redzone by $granularity$. Additionally, when an object allocated by alloc is freed through free API, CMASan requires the precise size of the object for proper poisoning. To facilitate this, CMASan stores the object's size information in the metadata at alloc.

Position of Redzones in CMASan. Unlike ASan, which inserts redzones on both sides of an allocated object, CMASan only adds a redzone to the right side. This design choice is due to existing CMA implementations often storing metadata at the left side of the object. However, this does not mean that CMASan misses left-side BOF; the right redzone of one object naturally serves as the left redzone for the next object to be allocated, as shown in Figure 4-(b).

3.2.2. Free.

To detect the Use-After-Free (UAF) vulnerabilities, CMASan poisons the freed object memory as redzone (❸ in Figure 4). On each CMA's memory-free request, CMASan retrieves metadata to identify the allocated memory area (i.e., the base address and size, and CMASan's metadata), and then marks the state of corresponding shadow memory as poisoned. To detect Double-Free (DF) vulnerabilities, CMASan checks whether a memory-free request has the same CMA instance information as a previous memory-free request. Note that ASan detects the DF regardless of the type of allocator/deallocator used. Unfortunately, this can cause a problem in applications that utilize multiple CMAs, where a memory object might be allocated and freed by multiple different CMAs. In such cases, following ASan's approach could lead to false positive double-free detections. For example, suppose that CMA_a allocates an arena for CMA_b, and CMA_b allocates an object from the start of the arena. If CMA_b frees the object and then CMA_a frees the arena, this would be incorrectly detected as a DF if the instance information is not considered. To correctly detect DF bugs after the memory is freed, we keep the CMA ID, which could be the address of the CMA instance or an assigned family ID as outlined in §3.1.2,

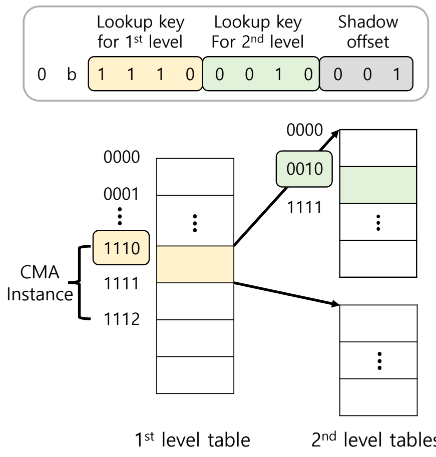

Figure 5: Two-level table for efficient object tracking

3.2.3. Realloc.

Realloc has two characteristics: it either resizes the given object at the same memory location (through in-place or general realloc), or it frees the given object and allocates a new one of the requested size. To distinguish between these two behaviors, CMASan compares the address of the old object with the address of the newly allocated object upon exit from the function (i.e., at the return instructions). If the two addresses are the same, indicating that the resizing occurred within the same memory location, CMASan only poisons the right side of the memory, similar to its allocation API instrumentation. If the addresses are different, CMASan additionally poisons the old memory to detect incorrect access to these objects. Note that CMASan secures the redzone space for the resized object by adding redzone size to the requested size, just as it does for the allocation API.

3.3. CMA Object Tracking

CMASan introduces two different types of metadata: one for each object and another for each CMA instance. This section describes how CMASan efficiently maintains the metadata for each object. The metadata for each CMA instance, including the quarantine zone and the allocation zone, will be explained in the next section (§3.4). For fast and accurate detection of memory bugs related to CMA objects, it is important to efficiently manage the metadata generated by CMASan. To achieve this, we utilize and customize a two-level table to store CMASan's metadata. Additionally, CMASan carefully manages the lifecycle of the metadata to maximize its detection capability by removing the outdated metadata properly.

3.3.1. On-demand Metadata Storage for CMA Objects.

CMASan uses a two-level table to keep track of the metadata associated with the allocated object. Note that the two-level table is also previously used by CFIXX [30] to manage frequent allocations of polymorphic objects within a small amount of area, which is a similar problem to ours. Specifically, CMAs typically allocate an arena memory

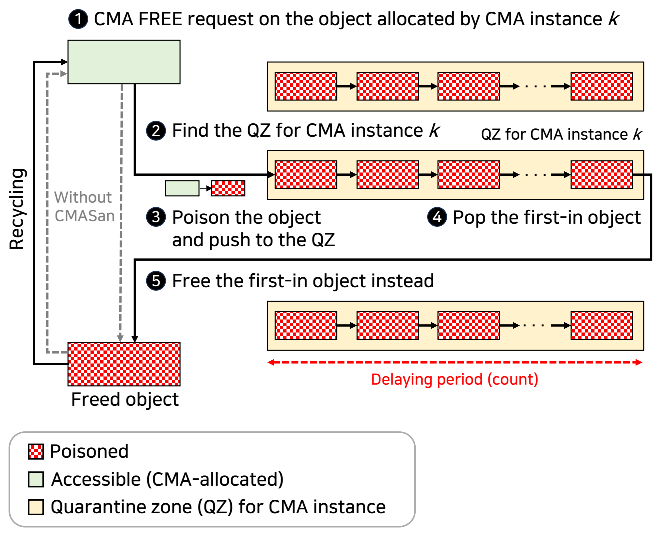
❶ CMA FREE request on the object allocated by CMA instance k
❷ Find the QZ for CMA instance k
QZ for CMA instance k
❸ Poison the object and push to the QZ
❹ Pop the first-in object
❺ Free the first-in object instead
Recycling
Without CMASan
Freed object
Delaying period (count)
Poisoned
Accessible (CMA-allocated)
Quarantine zone (QZ) for CMA instance

Figure 6: Overview of instance-specific free-delaying quarantine zone

and repeatedly allocate and recycle objects; the allocation happens within a relatively narrow memory address range. Thus, allocating a second-level table on-demand for only CMA-managed areas increases efficiency. For example, as shown in Figure 5, if the address is 11-bits long, and first-level table lookup keys, second-level table lookup keys, and shadow offset are 4, 4, and 3 bits, respectively, and a CMA instance uses memory in the address range [11100000000, 111200000000), CMASan will only need to maintain the second-level tables corresponding to the lookup keys of the first-level table, 1110 and 1111.

3.3.2. Handling Metadata Lifecycle. When the memory allocated by a CMA is fully utilized and then returned to the system, CMASan clears all related metadata to prevent potential issues (e.g., using outdated metadata). In particular, CMASan tracks a CMA object with the metadata from the CMA allocation. This metadata is removed from a two-level table if the object's memory region is no longer valid (e.g., CMAs destroy the arena or objects). This removal ensures that CMASan does not retain outdated metadata. Such collision-free metadata management helps CMASan distinguish standard objects from CMA objects based on the presence of metadata, aiding in false positive suppression §3.5.1 and information tracking in bug reports.

3.4. Enhance UAF and DF Detection

With only the instrumentation of primitive APIs, CMASan cannot detect UAF and DF bugs that occur a significant time after the free or clear API calls. To enhance UAF and DF detection, CMASan introduces quarantine zones and allocation zones without changing the internal logic of the CMA. These zones are instance-specific to accommodate applications that contain multiple CMAs.

3.4.1. Instance-specific Free-delaying Quarantine Zone. As the freed object will be unpoisoned when it is allocated as another object at the same location, CMASan cannot detect subsequent UAF and DF after the recycling. However, it does not mean that UAF and DF are no longer present. For instance, if a pointer $p1$ points to an object $o1$, and the object is freed and recycled as $o2$, $p1$ could still persist, pointing to $o2$. In this setup, any further access to $p1$ constitutes a UAF bug, as $p1$ could read from or write to $o2$. Thus, it is important to monitor the freed objects for an extended time to prevent the freed objects from being prematurely recycled. To improve the detection of UAF and DF, CMASan intentionally delays the recycling of freed objects. This allows for a longer detection window during which memory accesses to these objects that could lead to violations are more likely to be identified.

Free-delaying. ASan delays object recycling by implementing a quarantine zone. When a memory deallocation is requested, ASan adds these requested objects to the quarantine zone, preventing objects within this zone from being directly recycled as other objects. If the quarantine zone reaches its capacity, ASan frees the objects from the zone.

Unlike ASan, which uses its own internal allocator to perform actual frees and delay reuses, CMASan delays free requests by only marking them in the redzone without actually freeing them, thus not altering the allocator's logic. Specifically, when a CMA receives an object-free request, CMASan queues the free request instead of processing it directly with the CMA, and it poisons the memory region associated with the free request. This approach allows CMASan to monitor any subsequent incorrect accesses or free requests on freed objects, thus detecting potential UAF or DF. CMASan holds these CMA free requests until the quarantine zone reaches capacity. Once this occurs, CMASan begins to delay incoming free requests along with removing a request from the free request queue and forwarding it to the CMA to free up space. The capacity of the quarantine zone is determined by both the size in bytes and the number of objects to accommodate the variable sizes of objects. Importantly, CMASan's quarantine zone includes a free request queue, where it stores free request information (e.g., object address and size). This strategy is only applied to internal free APIs, not to realloc APIs, because realloc retains the content of the given object.

Instance Specific. Different CMA implementations can have notably different patterns in memory deallocation and different metadata for objects. However, using a single quarantine zone for all different CMAs can lead to several side effects. For example, freeing an object allocated by a CMA with a free API of another CMA can cause metadata corruption or undefined behavior (e.g., invalid free). Therefore, CMASan provides an individual quarantine zone for each CMA to ensure that memory objects allocated by different CMAs do not collide.

The overall operation of the quarantine zone is summarized in Figure 6: ❶ On a CMA free request for an object allocated by a specific CMA instance (e.g., k), CMASan intercepts this free request instead of sending it directly to the CMA. ❷ CMASan finds the corresponding quarantine zone that contains target objects from target CMA instance k. ❸ To detect UAF and DF, CMASan steals and poisons the object before the free call, and pushes it to the quarantine zone. If the quarantine zone is full, CMASan proceeds to steps ❹ and ❺, otherwise, it stops here. ❹ CMASan pops the first-in object from the quarantine zone, and ❺ the popped object is passed to the free function instead of the original object that is in the last-in position in the quarantine zone. In this way, CMASan successfully delays the actual free on an object, while sequentially freeing the oldest object for memory efficiency.

3.4.2. Instance-specific Allocation Zone for Clear APIs. To detect UAF and DF on objects freed by clear APIs, as discussed in §2.3.2, CMASan must first identify which objects are requested to be freed by these clear APIs. To this end, CMASan stores each allocated object in the corresponding instance-specific allocation zone. When a clear API is invoked, it marks all objects in the target zone as freed. The necessity for an instance-specific allocation zone is similar to that of a quarantine zone, as the target program can contain multiple CMAs.

3.5. False Positive Avoidance

CMASan manages redzones without replacing existing CMAs with ASan's internal memory allocators. Although this approach preserves the internal behavior of CMAs, it could lead to false positives, for example, when accessing metadata on freed objects. This section explains how CMASan avoids these potential false positives.

3.5.1. Call Stack-based Suppression. After CMASan poisons the object freed by CMA, other CMA internal codes can access these objects for operations, such as data flushing or logging. For example, retaining metadata on freed objects is a common design choice, as exemplified by `glibc's ptmalloc` [31]. Note that those accesses are legitimate while being detected as violations (i.e., false positives). To avoid a similar issue, ASan maintains an ignore list [32] to eliminate ASan checks in those functions causing false positives. However, this approach makes it difficult to handle more complicated cases where the common functions under CMAs might also be used elsewhere. Consequently, removing ASan checks from those functions could lead to false negatives. Instead, CMASan automatically extracts all the addresses of the CMA-related functions and uses those addresses to filter false positives. Specifically, when a memory corruption is detected by CMASan, it inspects the call stack of the execution to check whether a CMA-related function appears on the stack. If it does, CMASan ignores the detection result.

3.5.2. Call Stack-based Activation. CMA's realloc API can be implemented by combining alloc and free APIs. Moreover, alloc and free APIs can also be invoked in a nested manner. If we activate the instrumentations for all the APIs that can be called in a nested way, it may result in false positive reports of DF bugs in nested free calls. Similarly, nested alloc calls with different sizes may unnecessarily increase the redzone's size. Additionally, if we instrument only the inner alloc and free instead of realloc itself, CMASan would miss cases where realloc resizes an old object without invoking any inner free or alloc calls. Hence, when nested CMA API calls are detected, CMASan activates the instrumentation only for the outermost CMA call while disabling the instrumentation for all inner CMA calls (i.e., child functions). For this, CMASan inspects the call stack for every instrumentation and activates it only when no CMA function is present in its call stack. In particular, for the realloc implemented as a combination of alloc and free, CMASan's instrumentation of alloc and free will not activate, and only the outermost instrumentation in realloc will handle poisoning the given object and securing the redzone area.

3.5.3. Size Leakage Avoidance. CMAs often include size querying APIs, such as `sqlite3DbMallocSize()` from `SQLite3` [33], which return the size of a given object. Since CMASan adds the redzone size to the program-requested size at allocation, these APIs would return a larger size, including the redzones CMASan poisons. In such cases, the program could access memory beyond its requested size, potentially triggering false positive detections of BOF bugs. To handle this, since a program expects an object of the size that it initially requested to be allocated from a CMA, CMASan instruments these size-querying APIs to return the original, safely accessible memory size that applications had initially requested.

3.5.4. Object-collecting API Handling. Although CMASan intercepts the object before the free calls, these objects can still be forcibly recycled if CMA employs object-collecting APIs (e.g., garbage collector), which collects a part of allocated objects under specific conditions. This could lead to unexpected behavior as CMASan will request a free on already-freed or invalid objects. To address this, CMASan empties the corresponding quarantine zone when object-collecting APIs are called.

4. Implementation

CMASan is implemented on the top of Address Sanitizer of LLVM version 15.0.6. The total lines of code (LoC) is around 3,000. Additionally, we develop CMA Identification on top of CodeQL version 2.18.2.

4.1. CMA API Instrumentation

4.1.1. Implicit granularity inference. To infer the granularity of typed alloc, which implicitly provides the allocation granularity by the return type, CMASan extracts and records the type size from the data layout during the instrumentation of alloc API. However, LLVM 15 enables the opaque pointer

TABLE 1: The result of CMA object and family analysis.

Application	CMA Object Analysis				CMA Family Analysis				
	CMA Objects	Base Chunks	Freed Objects	load/store Checks (Ratio)	Candidates	Identified	Arena	Recycler	Max. CMA Instances
gRPC	623,109,396	30,462,031	10,107,363	242,237,783,898 (11.61%)	37	4	4	3	14,599
PHP	66,431,503	144,425	39,204,885	9,710,058,737 (30.02%)	49	7	7	3	30,754
Redis	26,867,307	77	26,846,482	4,353,464,117 (42.93%)	12	2	2	1	1
folly	1,087,644	3865	1,058,373	19,827,983 (0.06%)	12	6	5	3	32
MicroPython	839,561	957	110,415	467,351,689 (9.79%)	9	1	1	1	1
ncnn	95,741	1,176	95,741	55,053,014,784 (87.15%)	14	2	2	2	2
cocos2d-x	29,504	86	2,926	20,861,206 (0.71%)	16	6	6	0	15
Swoole	704	2	64	123,346 (0.01%)	18	4	3	2	1
Taichi	468	188	55	6,443,346 (0.00%)	10	3	2	2	2
RocksDB	97,101,151	4,844,394	-	32,875,914,933 (4.46%)	13	2	2	0	30,256
TensorFlow	92,900	10,539	-	231,364,292 (1.77%)	153	14	13	5	25,715
Godot	298	33	-	8,426 (0.00%)	62	9	8	2	33

feature [34], which eliminates the type information for the pointer and makes our implicit granularity inference difficult. As a solution, we build LLVM 15 without the opaque pointer feature for the implicit granularity inference, which only affects compiler analysis performance.

4.1.2. Avoiding Inlining for CMAs. Developers commonly choose to inline CMA APIs for optimization purposes. However, since the inline pass of LLVM precedes ASan pass, this inline optimization of CMA APIs results in omitting CMASan's instrumentations. To handle this, we traverse the Clang Abstract Syntax Tree (AST) and annotate CMAs to forcefully prevent the CMA functions from being inlined.

4.2. CMA Object Tracking

Since CMAs can recycle a freed object or clear the arena and reuse it from the start address, collisions can occur between the existing metadata and the newly allocated metadata. In such cases, CMASan overwrites the metadata for the newly allocated object while retaining the metadata on the remaining freed object region that does not overlap with the newly allocated object. This implementation allows CMASan to detect bugs from both the newly allocated object and the remaining freed object region.

4.3. Enhancing UAF and DF Detection

4.3.1. Implicit Instance Inference. In §3.1.2, CMASan categorizes each CMA family based on the CMA definition to support instance-specific Quarantine and Allocation zones. To facilitate this, we automatically infer the instance information from specific CMAs that are defined as C++ classes or structs, as these instances can be implicitly identified. To recognize this implicit instantiation, CMASan utilizes the feature of LLVM IR, which places the class (or structure) instance as the method's first argument. In the Clang AST, when the CMA API is identified as a `CXX Method Declaration`, CMASan uses the first argument as the instance argument if no explicit instance argument is recognized from the CMA identification process.

4.3.2. Quarantine Zone and Allocation Zone. Since both the Quarantine and the Allocation zones require memory that accumulates over time, a size limit is necessary. For the Quarantine zone, we set the default size limit as 256KB like ASan, and the default object count limit as 8,192. Regarding the Allocation zone, CMASan tracks up to 10,000 recently allocated objects by default. Note that these constraints are determined empirically and can be adjusted by users.

4.4. Call stack-based Avoidance

To enable false positive suppression as discussed in §3.5, CMASan checks for the presence of a CMA in the call stack at runtime. To implement this, CMASan extracts the start and end addresses of the CMA functions from the target binary and libraries at the program start-up using `llvm-objdump` [35], and checks whether each frame pointer in the call stack falls within the pre-extracted CMA function address range. However, if a CMA is in a shared library, the address of the function will be changed after the library is actually loaded. Thus, CMASan resolves the addresses of CMAs in shared libraries by inspecting the dynamic libraries loaded by default and intercepting every `dlopen`.

5. Evaluation

In this section, we evaluate CMASan on five aspects: detection coverage (§5.2), performance overhead (§5.3), false positive avoidance (§5.4), bug detection capability (§5.5), and real-world unknown bug finding (§5.6).

5.1. Evaluation Setup

All of our evaluations are conducted in the following environment: a 32-core Intel(R) Core(TM) i9-13900KF CPU with 128GB DDR4 RAM and an NVIDIA GeForce RTX 4090 GPU, running on Ubuntu 22.04.1 LTS (GNU/Linux 6.2.0).

Evaluation set. To evaluate detection coverage (§5.2), performance overhead (§5.3), and false positive avoidance (§5.4), we select the top 30 GitHub repositories ranked by stars for both C and C++, totaling 60 repositories. Then,

TABLE 2: Performance and memory overhead of CMASan compared to ASan.

Application	Performance Overhead (%)	Memory Overhead (%)
gRPC	3.55	10.99
PHP	21.76	7.85
Redis	9.50	33.46
folly	9.39	3.16
MicroPython	3.15	4.93
ncnn	22.44	99.09
cocos2d-x	0.13	1.02
Swoole	4.64	0.68
Taichi	2.27	0.43
RocksDB	2.81	5.90
TensorFlow	28.80	9.80
Godot	7.13	0.68
Average	**9.63**%	**14.83**%

we select 12 repositories as our final evaluation target repositories, which meet the following criteria: (1) buildable using the Clang version used to implement CMASan (i.e. Clang 15.0.6), (2) utilizes at least one CMA, and (3) provides workloads (e.g., unit test or benchmark).

Comparison target preparation. For a pair comparison, we utilize ASan included in LLVM version 15.0.6 to implement CMASan. Both ASan and CMASan are built using the recommended options [36] (`-fno-omit-frame-pointer` and `-fno-optimize-sibling-calls`) and `-O2` optimization option. Additionally, version 3e073da of Goshawk [8] is used for comparison.

5.2. Detection Coverage

In this evaluation, we evaluate CMASan against real-world applications to verify that applying CMASan can indeed increase the detection coverage compared to native ASan for various CMA objects and operations. To do this, we measure the number of CMA APIs and CMA objects used in each target application and count the number of memory access checks by CMASan on the CMA objects, as well as the proportion of quarantine zone usage.

First, to measure this, we identify CMA alloc API candidates (generated from CodeQL-based CMA allocation API recognition §3.1.1) used by the target applications. Then, we categorize these alloc candidates using our semi-automated procedure, as mentioned in §3.1.2. For this target categorization, using our predefined rule, an assigned person spends approximately 10 minutes for each application to manually categorize each CMA family function. This is a one-time cost that only needs to be incurred once. It is an additional step for increasing the accuracy of CMA memory bug detection—specifically, reducing both false positives and false negatives—by recognizing all relevant functions in a CMA.

In Table 1, the number of CMA objects (`CMA Objects` in Table 1) and base chunks (`Base Chunk`) indicates that several CMA objects are generated from one base chunk. This indicates that in the Arena pattern, a single large base chunk is divided into multiple smaller objects. Additionally, in the Recycler pattern, a single base chunk is reused as a CMA object across multiple lifecycles.

The significant number of freed CMA objects (`Freed Objects`) in nine applications indicates a high potential for UAF bugs and the importance of the Quarantine zone. The `load/store checks` (and `Ratio`) in the table, which measure the number of ASan checks associated with CMA objects (and their proportion of `load/store checks` related to CMA objects out of all `load/store checks`), reveal that a significant number of checks are performed in all programs (except `Godot`). This indicates that many CMA-related memory accesses are not verified by ASan but can be checked through CMASan. For further details on the total number of `load/store checks`, please refer to Appendix §C.2.

As shown in Table 1, applications commonly use Arena (`Arena` in Table 1) and Recycler (`Recycler`) patterns, which ASan does not handle. We also find that some CMA families utilize a combination of Arena and Recycler. For instance, `gRPC` utilizes a total of 4 different CMA families, all utilizing Arena-type CMAs, with 3 of these families also using Recycler-type CMAs. The `Identified` column under `CMA Family Analysis` in Table 1 denotes the number of CMAs at static time, and `Max. CMA Instances` refers to their number at runtime. A CMA family can be defined in the form of a class or structure, allowing multiple creations (i.e., even with a single definition, multiple instances can exist at runtime). For instance, in `gRPC`, 4 CMA families are generated 14,599 times during our evaluation.

5.3. Performance Overhead

In this evaluation, we measure the additional overhead generated by CMASan compared to ASan. To do this, we utilize the wall time provided by each benchmark and unit test. If wall time is not provided, we measure the time using the `/usr/bin/time -v` command. Memory overhead is assessed by measuring the peak resident set size to determine the maximum usage of physical memory.

Table 2 shows the performance and memory overhead of CMASan. The average overhead of CMASan is 1.096 times (performance overhead) and 1.148 times (memory overhead) compared to ASan, showing a minor increase in overhead. However, `ncnn` and `Redis` have notably higher memory overhead, which is due to the relatively higher percentage of total memory usage accounted for by CMA in these two applications. This is caused by the allocation of large CMA objects, leading to an increased demand to save related information to the CMASan metadata table. Additionally, the delayed freeing of CMA objects due to our quarantine zone (i.e., need to secure additional new memory regions) also contributes to the memory overhead observed in `ncnn` and `Redis`. Additionally, `PHP` and `TensorFlow` show relatively high-performance overhead compared to other applications due to our use of CMASan's FP Avoidance approach based on call stack tracing. Applications like `TensorFlow` and `PHP` have higher overhead than other programs because of the need to suppress many false positives. This performance

TABLE 3: False positive rates without and with FP Avoidance. The percentages (%) represent the ratio of workloads that failed due to false positives, and the fraction next to the percentage indicates the number of failed workloads relative to the total number of workloads (`(# of failed workloads)/(# of total workloads)`). `crashed` indicates that target test applications are terminated due to false positives before the test.

Application	Without FP Avoidance	With FP Avoidance
gRPC	10.68% (110/1030)	0%
MicroPython	32.32% (298/922)	0%
Redis	crashed	0%
PHP	crashed	0%
TensorFlow	crashed	0%

and memory overhead also arise from additional processing by CMASan, which manages the CMA-related metadata table and shadow memory of CMA objects—tasks that ASan does not handle. For example, `ncnn` shows both high performance and memory overhead due to the allocation and use of large CMA objects, which can incur relatively high metadata management and poisoning overhead. Note that ASan itself incurs 2.01 times memory overhead compared to native applications, mainly due to the large shadow memory. CMASan, which builds on top of this, adds an extra 14.83% memory overhead to the ASan's 2.01 times overhead (i.e., 2.31 times memory overhead compared to native).

5.4. False Positive Avoidance

In this section, we evaluate the effectiveness of CMASan's false positive avoidance approaches (§3.5). For this evaluation, we disable all CMASan's four false positive avoidance approaches (from §3.5.1 to §3.5.4) mentioned in §3.5. As shown in Table 3, 5 out of 12 applications show false positives, while CMASan successfully suppress all these false positives. Three applications (`Redis`, `PHP`, and `TensorFlow`) are terminated due to false positives before the test. Additionally, `gRPC` reports false positives in 110 out of 1030 tests (10.68%), and `MicroPython` reports false positives in 298 out of 922 tests (32.32%).

We analyze the main reasons for these false positives. First, four applications, except for `TensorFlow`, access the redzone area due to metadata-related operations, such as freed segment management, data flush, and logging. These false positives can be prevented by enabling CMASan's call stack-based suppression feature (§3.5.1). Additionally, since `PHP` implements realloc as a combination of alloc and free, activating our instrumentation for both realloc and free, results in false positive DF detection, which can be prevented through CMASan's call stack-based activation mentioned in §3.5.2. CMAs in `Redis` and `PHP` provide size querying APIs that return the size of objects, including redzone sizes. As a result, they access beyond the program-requested size, leading to BOF false positives that can be prevented through the size leakage avoidance approach (§3.5.3). Lastly, `MicroPython`, `PHP`, and `Tensorflow` utilize object-collecting APIs (e.g., garbage collector). Thus, our quarantine zone frees objects already freed (collected) by CMAs, leading to false positive DF detection. These issues can also be prevented through CMASan's object-collecting API-aware false positive avoidance approach (§3.5.4).

TABLE 4: Known CMA (shim) bugs from `PHP` and `ImageMagick`.

Bug ID	Application	Type	ASan	CMASan
Issue 11028	PHP	BOF	✗	✓
Issue 10581	PHP	UAF	✗	✓
CVE-2015-2787	PHP	UAF	✗	✓
CVE-2019-13307	ImageMagick	BOF	✗	✓
Issue 1714	ImageMagick	BOF	✗	✓
CVE-2019-17541	ImageMagick	UAF	✗	✓
Issue 1621	ImageMagick	BOF	✗	✓
Issue 1644	ImageMagick	BOF	✗	✓
CVE-2018-8804	ImageMagick	DF	✗	✓
CVE-2021-3962	ImageMagick	UAF -> DF	✗	✓

5.5. Bug Detection Capability

In this section, we evaluate whether CMASan accurately detects existing CMA memory bugs. Due to the absence of proper CMA memory bug detection approaches, most known memory bugs are not related to CMA. Therefore, with our best effort, as shown in Table 4, we collect ten existing CMA bugs previously reported in `PHP` and `ImageMagick`. These bugs consist of BOF, UAF, and DF. The notation `UAF->DF` indicates that the DF bug occurs following a UAF bug. All these CMA memory bugs are detected through the ASan Shim logic.

For this evaluation, we perform the bug detection capability evaluation in two configurations: one with ASan's Shim logic disabled and the other with CMASan applied and Shim logic disabled. As shown in Table 4, when Shim is disabled, ASan fails to adequately detect CMA memory bugs. However, CMASan successfully detected all known CMA memory bugs. This evaluation highlights CMASan's effectiveness in detecting existing CMA bugs without the necessity of manually implementing separate Shim logic.

5.6. Real-world Unknown Bug Detection

We apply CMASan to various real-world applications, including calculator (`qhull`), parser (`RapidJSON`), and database (`SQLite3`) that contains CMAs to evaluate its capability to detect new unknown bugs. As a result, as shown in Table 5, we identify 19 previously unknown bugs from seven applications. Specifically, the bugs in `MicroPython` and one of the bugs in `PHP` are discovered through fuzzing using the AFL++ with CMASan, while other bugs are detected during unit tests. We report all 19 bugs to their corresponding project developer. As of the time of this paper's writing, twelve of these have been confirmed, and six have been patched and assigned CVE IDs.

We compare these bugs with existing works, ASan and Goshawk, to check whether they also detect these bugs under the same conditions. ASan is unable to detect CMA bugs as it does not handle Arena and Recycler CMA patterns

TABLE 5: Unknown bugs detected by CMASan (A: Arena, R: Recycler).

ID	Bug ID	Application	CMA Pattern	Bug Type	Status	CMASan	Asan	Goshawk
1	CVE-2023-7152	MicroPython	A+R	UAF	patched & CVE	✓	✗	✗
2	CVE-2023-7158	MicroPython	A+R	BOF	patched & CVE	✓	✗	✗
3	CVE-2024-8946	MicroPython	A+R	BOF	patched & CVE	✓	✗	✗
4	CVE-2024-8947	MicroPython	A+R	UAF	patched & CVE	✓	✗	✗
5	CVE-2024-8948	MicroPython	A+R	BOF	patched & CVE	✓	✗	✗
6	Issue #13004	MicroPython	A+R	BOF	reported	✓	✗	✗
7	Issue #13046	MicroPython	A+R	BOF	patched	✓	✗	✗
8	Issue #13220	MicroPython	A+R	BOF	reported	✓	✗	✗
9	Issue #13428	MicroPython	A+R	BOF	reported	✓	✗	✗
10	Issue #136-1	qhull	A+R	UAF	confirmed	✓	✗	✗
11	Issue #136-2	qhull	A+R	UAF	confirmed	✓	✗	✗
12	PR #2213	RapidJSON	A	BOF	reported	✓	✗	✗
13	PR #2244	RapidJSON	A	UAF	reported	✓	✗	✗
14	PR #2256	RapidJSON	A	UAF	reported	✓	✗	✗
15	CVE-2023-7104	SQLite3	A+R	BOF	patched & CVE	✓	✗	✗
16	Issue #13230	PHP	A+R	UAF	confirmed	✓	✗	✗
17	GHSA-rwp7-7vc6-8477	PHP	A+R	UAF	confirmed	✓	✗	✗
18	Issue #8501	Taichi	A+R	BOF	reported	✓	✗	✗
19	Issue #5734	ncnn	A+R	BOF	confirmed	✓	✗	✗

as mentioned in §2.2. More specifically, the allocators in `MicroPython`, `SQLite3`, `PHP`, `qhull`, `Taichi` and `ncnn` employ Arena and Recycler CMA patterns that are difficult for ASan to address. As for `RapidJSON`, it uses the Arena CMA pattern and the clear API. As described in §2.3.2, the clear API is a difficult type for ASan to address, even when utilizing its Shim or ASan manual poisoning API. However, CMASan detects two UAF CMA memory bugs (Bugs ID 13 and 14) that occurred after clear API is used to access freed chunks, as well as a BOF (Bug ID 12) that is obscured by the Arena CMA pattern.

In the case of Goshawk, it does not support BOF detection and only handles UAF and DF detection. However, all Goshawk's UAF and DF detection failed due to the following four main reasons: (1) failed CMA API detections (Bug IDs 10 and 11), (2) limited clear API UAF detection (Bug IDs 13 and 14), (3) not treating realloc as free by design (Bug IDs 1 and 4), and (4) complex paths that are difficult to manage (Bug ID 16 and 17). More specifically, in failing CMA API detection (case 1), Goshawk's NLP initially classifies it correctly as a candidate but then removes it during the subsequent verification process. Regarding complex paths that are difficult to manage (case 4), they can not handle target paths due to the considerable distance between free and access operations.

These unknown bugs detected through CMASan show the effectiveness of the CMASan's internal approaches, such as quarantine zone and clear API handling. For example, in the case of `MicroPython` bug (Bug ID 1) and `qhull` bug (Bug ID 11), they are not detected if the Quarantine zone is not applied. This is because the CMA immediately reallocates the chunk after freeing it, making it impossible to detect the UAF bug without a Quarantine zone (i.e., without delay).

Furthermore, as expected, unique CMA patterns, such as clear API, are not sufficiently tested in practice. In our evaluation, by handling the clear API of `RapidJSON`'s Arena CMA, two bugs (Bug IDs 13 and 14) are detected. Although these bugs have been consistently triggered during their unit test and perftest, these bugs remained undetected due to the lack of handling for the clear API. These cases demonstrate the importance of CMASan, which is able to be aware of and test both the Arena and the clear API.

While Shim requires expert knowledge and manual effort from developers, `SQLite3` and `PHP` have successfully implemented Shim testing despite the complex relationship of the Lookaside allocator and Zend allocator with the DB and zend heap context. Even with the implementation of Shim for CMA, ASan is unable to detect CMA bugs because developers accidentally failed to activate the Shim in certain sections. For example, we find `SQLite3` (Bug ID 15) bugs that have been present in testing workloads for nine years and a `PHP` bug (Bug ID 16) that is also triggered by their testing workloads and undetected for two years. This highlights that even if Shim is applied to CMA, additional manual Shim configurations are needed, and failing this step could lead to false native, However, CMASan is able to automatically test all CMAs without the manual effort of applying Shim. Moreover, apart from Shim applied CMAs in `SQLite3` and `PHP`, we discover additional uncovered CMAs (by Shim) in `SQLite3` and `PHP`: one pcache in `SQLite3` and six others in `PHP`.

6. Discussion

Manual analysis. Although CMASan does not require an understanding of the target program and its CMAs, nor does it need manual code modifications to the target programs, the CMA identification process within CMASan does involve some manual analysis (one-time cost) to categorize CMA APIs. Testers can bypass this manual analysis by directly providing the CMA API lists to CMASan (e.g., by referring to official documents or directly using the CodeQL results). However, we recommend using our CMA identification process to ensure all CMAs in the repository are accurately recognized. Although this is not our main contribution, to assist users in the quick and accurate categorization of CMA APIs, CMASan provides a script that implements the rules in Table 6 and procedures in Algorithm 1 described in the Appendix.

Size Modification. CMASan preserves the logic of CMAs. However, there is one exception: when allocating a CMA object, CMASan modifies the size to be larger than requested to secure space for the redzone. This approach, which returns a size larger than requested, can lead to side effects. However, in our evaluation with twelve real-world applications, only three show minor side effects. More specifically, PHP and Redis show false positives due to access based on the increased size returned by size querying APIs, which CMASan can avoid using CMASan's size leakage avoidance technique as mentioned in §3.5.3. RocksDB optimizes the database based on the total allocation size, so CMASan's size increment affects optimization performance, leading to some unit test failures (resolvable by adjusting thresholds or redzone size). This alteration, however, only affects performance and does not compromise the application's correctness.

CMA Internal Bugs. While our false positive suppression (§3.5.1) helps CMASan operate accurately, it may miss CMA internal bugs. CMASan aims to detect memory bugs on objects allocated by applications through CMAs. Therefore, bugs occurring within the CMA are out of CMASan's scope. However, it is worth noting that CMASan can detect such internal CMA bugs by turning off false positive suppression (with `halt_on_error=0`).

Performance-critical Application Testing. CMASan incurs approximately 1.92 times performance overhead compared to native. More specifically, the performance overhead from ASan itself (about 1.752x) combined with the additional performance overhead from CMASan (around 1.096x) results in a total of about 1.92 times performance overhead. Such performance overhead can cause side effects during the testing of performance-critical applications. To measure these side effects, we test our 12 selected evaluation target applications and two additional performance-sensitive applications (e.g., `Nginx`, `SQLite3`) through manual or unit tests. However, CMASan's performance overhead does not cause any special issues except for a few `PHP` unit tests. These failed `PHP` test cases are due to timeouts caused by ASan and CMASan's performance overhead, but after adjusting these test configurations, all test cases passed successfully.

7. Related Work

CMA Memory bug detection techniques are classified into static bug detection approaches [7–11] that utilize symbolic execution for bug detection and dynamic bug detection approaches [12, 17, 37] that perform runtime detection through additional instrumentation or code modification.

Static CMA Memory Bug Detectors. HOTracer [7] is a CMA out-of-range memory bug detection approach for binary-only programs. After defining the patterns of CMA allocator functions (e.g., the return value is a heap pointer), HOTracer identifies any function matching at least one of these patterns as a candidate for CMA allocator. HOTracer then identifies pairs of suspicious heap operations (i.e., allocation, access) and verifies them through symbolic execution. Heapster [10] is designed to identify CMA internal bugs (not bugs in CMA utilizing applications) in monolithic firmware images. Heapster identifies allocator and deallocator APIs through static analysis that starts with basic memory operations, such as `memcpy`. After defining malicious allocator behavior (e.g., chunks overlapping), Heapster uses symbolic execution to detect CMA internal bugs. NLP-EYE [11] leverages an NLP-based approach for identifying CMA APIs. Then, it uses symbolic execution to check whether a memory operation leads to incorrect memory usage, such as null pointer dereference, UAF, or DF. Goshawk [8] and Sparrowhawk [9], similar to NLP-EYE, utilize NLP [38], to identify CMA APIs and further enhance accuracy through additional static analysis verification. Goshawk is able to detect UAF and DF bugs except for BOF and realloc-related UAF and DF bugs (due to limited realloc modeling).

All mentioned approaches commonly utilize symbolic execution for CMA memory bug detection. However, symbolic execution is unable to detect all types of memory bugs in large and complex programs due to its inherent limitations, such as path explosion and the limited handling of complex data structures and real-world input. Since CMASan is the ASan-based approach that performs runtime checks, It does not suffer from the same limitation as symbolic execution, thereby allowing for the identification of all CMA bugs in large and complicated programs.

Dynamic CMA Memory Bug Detectors. ASan [12], the most widely used approach to detect memory bugs, is limited to detecting CMA memory bugs, as ASan only handles the standard library allocator and memory bugs related to the object allocated from it. Therefore, the Shim [13] approach and ASan manual poisoning API [15] have been proposed to detect CMA memory bugs through ASan. This Shim technique allows users to replace the CMA allocator with the standard memory allocator. If CMA's internal logic is identical to that of the standard memory allocator, a Shim approach to detect CMA bugs would be appropriate. CMA, however, has various internal logic, unlike the standard library allocator, such as inplace-realloc, which assigns an object to a fixed address, or the clear API, which frees all objects at once. This makes direct mapping with certain functions of the standard allocator challenging. However, CMASan, without the challenging or impossible requirement of directly mapping to the standard memory allocator, maintains and handles the unique logic of CMA, thus identifying all types of CMA memory bugs.

The ASan Manual Poisoning APIs support manual poisoning through the insertion of `__asan_poison_memory_region` and `__asan_unpoison_memory_region` APIs by users into appropriate parts of the code according to the lifecycle of CMA while maintaining CMA's logic. However, using these APIs requires code modifications and expert knowledge of the target program and CMA. For example, in the case of Arena CMA, users need to manually modify CMA to secure redzone areas. Additionally, traditional ASan utilizes a Quarantine zone to delay the freeing of objects for a certain period, thus increasing the chance of detecting UAF and DF memory bugs. However, when using the ASan manual poisoning API while maintaining the original logic of CMA, the Quarantine zone cannot be directly utilized without modifying CMA to delay object-free, making the detection of CMA's DF and UAF memory bugs difficult.

Valgrind [39] is a widely used framework for detecting memory bugs. However, Valgrind occurs around 10 times more overhead [40] compared to ASan and has limited detection coverage (e.g., Valgrind cannot detect stack out-of-bounds, global out-of-bounds, and use-after-return) than ASan. Nevertheless, unlike ASan, Valgraind is able to detect memory bugs in binary-only programs. Similarly to ASan's manual poisoning API, Valgrind provides specialized APIs such as `VALGRIND_DESTROY_MEMPOOL`, `VALGRIND_MEMPOOL_ALLOC`, and `VALGRIND_MEMPOOL_FREE` to detect CMA memory bugs.

However, like ASan's Manual Poisoning API, these APIs have limitations, including the need for expert knowledge and limited CMA bug detection capabilities. The API-based methods provided by ASan and Valgrind, including ASan's Shim, all require expert knowledge and manual source code modifications. Due to these inconveniences, most programs do not apply these APIs or Shim approaches (for example, only 5 out of the top 100 C/C++ applications based on Github stars fully utilize them). However, because CMASan preserves the CMA logic and performs automatic instrumentation at compile time, it eliminates the need for additional effort and expert knowledge for manual code modification.

8. Conclusion

The custom memory allocator is widely used in real-world applications and is also exposed to the same memory vulnerabilities as the standard memory allocator. However, existing CMA memory bug detection approaches do not properly perform due to several restrictions, including limited detection capability, the requirement for manual code modifications, and the necessity for expert knowledge. This paper proposes the first CMA-aware Address Sanitizer, CMASan, by understanding and preserving the various and unique internal logic of CMA, effectively detects all types of CMA-related memory bugs without needing expert knowledge or manual code modifications. Our evaluation shows that CMASan identifies 19 new unknown CMA memory bugs, previously undetected by native ASan. Additionally, CMASan, compared to native ASan, only occurs 9.63% performance overhead. The open-source version of CMASan is available at https://github.com/S2-Lab/CMASan.

Acknowledgement

This work was supported by a Korea Internet & Security Agency (KISA) grant funded by the Korean government (PIPC) (No. 1781000003). This work was also supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2024-2021-0-01817) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation). This work was also supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2024-00341722, RS-2024-00437306 and RS-2024-00341722, Development of Cyber Resilience Method for Intelligent Service Robots, Development of Integrated Platform for Expanding and Safely Applying Memory-Safe Languages, and AI-Based Automated Vulnerability Detection and Safe Code Generation). This work was also supported by NSF (2426653 and 2427783). We gratefully acknowledge their support.

References

[1] Alan Kelly, "Simpleperf case study: Fast initialization of TFLite's Memory Arena," https://blog.tensorflow.org/2023/08/simpleperf-case-study-fast.html, 2023.

[2] Google, "C++ Arena Allocation Guide," https://protobuf.dev/reference/cpp/arenas/.

[3] "Zend Memory Manager," https://www.phpinternalsbook.com/php7/memory$_m$anagement/zend$_m$emory$_m$anager.html/, 2017.

[4] Volker Hilsheimer, "A fast and thread-safe pool allocator for Qt - Part 1," https://www.qt.io/blog/a-fast-and-thread-safe-pool-allocator-for-qt-part-1, 2019.

[5] OpenSSL, "OPENSSL secure malloc," https://www.openssl.org/docs/man1.1.1/man3/OPENSSL$_s$ecure$_m$alloc.html.

[6] T. B. Ferreira, R. Matias, A. Macedo, and L. B. Araujo, "An experimental study on memory allocators in multicore and multithreaded applications," in *2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, 2011, pp. 92–98.

[7] X. Jia, C. Zhang, P. Su, Y. Yang, H. Huang, and D. Feng, "Towards efficient heap overflow discovery," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 989–1006.

[8] Y. Lyu, Y. Fang, Y. Zhang, Q. Sun, S. Ma, E. Bertino, K. Lu, and J. Li, "Goshawk: Hunting memory corruptions via structure-aware and object-centric memory operation synopsis," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 2096–2113.

[9] Y. Lyu, W. Gao, S. Ma, Q. Sun, and J. Li, "Sparrowhawk: Memory safety flaw detection via data-driven source code annotation," in *Information Security and Cryptology: 17th International Conference, Inscrypt 2021, Virtual Event, August 12–14, 2021, Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 129–148. [Online]. Available: https://doi.org/10.1007/978-3-030-88323-2$_7$

[10] F. Gritti, F. Pagani, I. Grishchenko, L. Dresel, N. Redini, C. Kruegel, and G. Vigna, "Heapster: Analyzing the security of dynamic allocators for monolithic firmware images," in *2022 IEEE Symposium on Security and*

Privacy (SP). IEEE Computer Society, 2022, pp. 1559–1559.

[11] J. Wang, S. Ma, Y. Zhang, J. Li, Z. Ma, L. Mai, T. Chen, and D. Gu, "Nlp-eye: Detecting memory corruptions via semantic-aware memory operation function identification," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019, pp. 309–321.

[12] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "{AddressSanitizer}: A fast address sanity checker," in *2012 USENIX annual technical conference (USENIX ATC 12)*, 2012, pp. 309–318.

[13] "Unified allocator shim," https://chromium.googlesource.com/chromium/src/base/+/master/allocator/README.md.

[14] Hanno Böck, "Custom Memory allocators," https://fuzzing-project.org/tutorial-malloc.html.

[15] Google, "Address Sanitizer Manual Poisoning," https://github.com/google/sanitizers/wiki/AddressSanitizerManualPoisoning.

[16] P. C. Amusuo, R. A. C. Méndez, Z. Xu, A. Machiry, and J. C. Davis, "Systematically detecting packet validation vulnerabilities in embedded network stacks," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 926–938.

[17] K. K. Bozdoğan, D. Stavrakakis, S. Issa, and P. Bhatotia, "Safepm: A sanitizer for persistent memory," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 506–524.

[18] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[19] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "Sok: Sanitizing for security," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1275–1295.

[20] Dmitry Vyukov, "Address/thread/memorysanitizer slaughtering c++ bugs," https://www.slideshare.net/sermp/sanitizer-cppcon-russia.

[21] Google, "Address sanitizer found bugs," https://github.com/google/sanitizers/wiki/AddressSanitizerFoundBugs.

[22] Z. Y. Ding and C. Le Goues, "An empirical study of oss-fuzz bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 131–142.

[23] GitHub, "CodeQL," https://codeql.github.com/.

[24] Tencent, "ncnn," https://github.com/Tencent/ncnn.

[25] Google, "Address Sanitizer Algorithm," https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm.

[26] cppreference.com, "realloc," https://en.cppreference.com/w/c/memory/realloc.

[27] "MicroPython," https://micropython.org/.

[28] "An open-source C++ library developed and used at Facebook," https://github.com/facebook/folly.

[29] GitHub, "Class HeuristicAllocationFunction," https://codeql.github.com/codeql-standard-libraries/cpp/semmle/code/cpp/models/interfaces/Allocation.qll/type.Allocation$HeuristicAllocationFunction.html.

[30] N. Burow, D. McKee, S. A. Carr, and M. Payer, "Cfixx: Object type integrity for c++ virtual dispatch," in *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[31] "MallocInternals," https://sourceware.org/glibc/wiki/MallocInternals.

[32] "Sanitizer special case list," https://releases.llvm.org/15.0.0/tools/clang/docs/SanitizerSpecialCaseList.html.

[33] "Dynamic Memory Allocation In SQLite," https://www.sqlite.org/malloc.html.

[34] LLVM, "Opaque Pointers," https://releases.llvm.org/15.0.0/docs/OpaquePointers.html.

[35] ——, "llvm-objdump - LLVM's object file dumper," hhttps://llvm.org/docs/CommandGuide/llvm-objdump.html.

[36] "Clang 15.0.0 documentation AddressSanitizer," https://releases.llvm.org/15.0.0/tools/clang/docs/AddressSanitizer.html.

[37] "Memcheck: a memory error detector," https://valgrind.org/docs/manual/mc-manual.html#mc-manual.mempools.

[38] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a" siamese" time delay neural network," *Advances in neural information processing systems*, vol. 6, 1993.

[39] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[40] "AddressSanitizer Comparison Of MemoryTools," https://github.com/google/sanitizers/wiki/AddressSanitizerComparisonOfMemoryTools.

Appendix A. Rules and Algorithms for CMA Categorization

Algorithm 1: CMA API categorization algorithm

```
Input  : $P_{ALLOC}$
$P_{ALLOC} \leftarrow$ a set of ALLOC candidates;
Output : $V$
Initialize a map $V$ with the key $loc$ and the entry $\{func$, $type$, $id\}$;
foreach function $f$ and location $l \in P_{ALLOC}$ do
    if $f$ satisfies $R_{size}$ then
        Annotate the size argument of $f$;
        if $f$ has an additional size argument then
            Annotate the granularity argument of $f$;
            Insert <$l$, {$f$, $CALLOC$, $-1$}> to $V$;
        end
        else if $f$ resizes a given object then
            Annotate the object argument of $f$;
            Insert <$l$, {$f$, $REALLOC$, $-1$}> to $V$;
        end
        else
            Insert <$l$, {$f$, $ALLOC$, $-1$}> to $V$;
        end
    end
end
$id \leftarrow 0$ ;
foreach key $l$ and entry $E \in V$ do
    if $E.id == -1$ then
        $E.id \leftarrow id$ ;
        For E.func, collect a family function set $F$ by $R_{family}$ ;
        foreach function $ff$ and location $ll \in F$ do
            if $V$ has $ll$ with entry $EE$ then
                $EE.id = id$ ;
            end
            else if $ff$ satisfies $R_{object}$ then
                Annotate the object argument of $f$;
                Insert <$ll$, {$ff$, $FREE$, $id$}> to $V$ ;
            end
            else if $ff$ satisfies $R_{clear}$ then
                Insert <$ll$, {$ff$, $CLEAR$, $id$}> to $V$ ;
            end
            else if $ff$ satisfies one of type $T$ in $R_{fp}$ then
                Annotate the object argument of $f$ if any;
                Insert <$ll$, {$ff$, $T$, $id$}> to $V$ ;
            end
        end
    end
    $id \leftarrow id + 1$;
end
foreach entry $E \in V$ do
    if $E.func$ satisfies $Rule_{instance}$ then
        Annotate the instance argument of $E.func$;
    end
end
```

Appendix B. Pseudo Code of Metadata Management

CMASan utilizes the same redzone-based poisoning approach as ASan. However, the information available from shadow memory alone is insufficient for detecting CMA memory bugs. To address this, additional metadata—such as object size and redzone size (for poisoning freed objects), allocation status and CMA instance information (for UAF/DF detection), and alloc/free traces (for bug tracing)—is managed in the CMASan's two-level metadata table as shown in Algorithm 2. At the exit of allocation functions (`at_alloc_exit`), CMASan creates a chunk (`info`) and stores the start address (`ptr`), object size (`size`), redzone size (`rz_size`), and CMA ID (`cid`) in this chunk, then sets the allocation status to `ALLOCATED`. This chunk is stored in a two-level table spanning the object's address range (Lines 1-7). At the entry of free functions (`at_free_entry`), CMASan retrieves the metadata from the two-level table using the object's address (Line 10). If the allocation status is already `FREED` and the metadata has the same CMA ID, it reports a double free (Lines 12-14). Otherwise, CMASan updates the allocation status to `FREED` and poisons the object region using the size information from the metadata (Lines 15-18). When

Algorithm 2: Pseudo code of metadata management in alloc, free, and ASan check functions.

```
Data: ObjectInfo: ptr, size, rz_size, status, cid, alloc_trace, free_trace
1  Function at_alloc_exit(ptr, size, rz_size, cid):
2      create a new ObjectInfo $info$;
3      set $ptr, size, rz\_size, cid$ in $info$;
4      update $info.alloc\_trace$;
5      $info.status \leftarrow$ ALLOCATED;
6      insert $info$ into the two-level table;
7      poison the redzone for the range
         $[ptr + size, ptr + size + rz\_size)$;
8  ;
9  Function at_free_entry(ptr, cid):
10     lookup $info$ by $ptr$ from the two-level table;
11     if $info$ exists then
12         if $info.status$ == FREED and $info.cid$ == $cid$ then
13             report double free error;
14         end
15         update $info.free\_trace$;
16         $info.status \leftarrow$ FREED;
17         poison the redzone for the range
             $[ptr, ptr + info.size)$;
18     end
19 ;
20 Function asan_check(region):
21     if region is poisoned then
22         if region has associated ObjectInfo $info$ then
23             use $info$ for bug report;
24         end
25         else
26             use default ASan info for bug report;
27         end
28     end
```

TABLE 6: Rules for the CMA validation

$Rule_{size}$ (size argument)	There is an integer (size) argument used for the object allocation by: (a) Pointer arithmetic (including array indexing) (b) Comparison with internal metadata If the product of two arguments satisfies the above rules, one argument is size, and the other is granularity.
$Rule_{family}$ (family matching)	The given two functions are defined in the same source file, spanning both header and source files.
$Rule_{object}$ (object argument)	There is a pointer argument to be stored in the internal storage, which is used for allocating an object in the matched ALLOC candidates
$Rule_{clear}$ (Clear API)	It destroys the CMA family by releasing all the resources the CMA allocated or resets the base pointer to the first position.
$Rule_{fp}$ (false positives)	Object accessing API: The given function accesses objects allocated by the CMA family Size querying API: The given function receives an object argument and returns the size of the object Object collecting API: The given function collects the objects into internal structures under specific conditions (i.e., reference counting).
$Rule_{instance}$ (instance argument)	There is an argument that serves as the source for the internal storage, responsible for storing and providing the object

checking the validity of memory accesses (`asan_check`), if a memory bug is detected, the memory access address is used to determine whether it is related to an object managed by the two-level table (Line 22). If so, corresponding metadata is extracted and used (Line 23) to generate the bug report (Lines 20-28).

Appendix C. Additional Evaluation Results

C.1. Memory Overhead

TABLE 7: Aggregated peak resident set size of CMASan and ASan for each workload.

Application	CMASan (MB)	ASan (MB)	# process
gRPC	2,199,582	1,981,835	9189
PHP	553,752	513,460	14121
Redis	3,115	2,334	70
folly	72,104	69,898	3047
MicroPython	16,232	15,469	957
ncnn	2,025	1,017	1
cocos2d-x	1,357	1,343	1
Swoole	369	367	2
Taichi	87,349	86,975	339
RocksDB	1,216,685	1,148,862	11463
TensorFlow	2,754	2,508	1
Godot	3,326	3,303	1

Table 7 shows the resident set size (RSS) measured and aggregated from all processes (i.e., `# process`) executed while running the given workload for each application with CMASan and ASan applied. CMASan does not incur significantly more memory overhead compared to ASan. The main reason is that the most memory-consuming 2nd-level table (128MB per table) is allocated on demand. Note that the 1st-level table only requires 64MB of memory.

C.2. Load/store Checks

TABLE 8: Number of load/store checks on CMA objects and all objects.

Application	CMA Object load/store Checks	All load/store Checks
gRPC	242,237,783,898	2,086,871,852,186
PHP	9,710,058,737	32,346,399,996
Redis	4,353,464,117	10,141,977,621
folly	19,827,983	34,224,267,807
MicroPython	467,351,689	4,762,795,646
ncnn	55,053,014,784	63,169,000,513
cocos2d-x	20,861,206	2,921,250,379
Swoole	123,346	960,544,384
Taichi	643,346	52,300,562,797
RocksDB	33,164,420,791	743,727,824,901
TensorFlow	231,364,292	13,075,270,979
Godot	8,426	279,604,507,191

C.3. Long-term Evaluation of False Positive Avoidance

TABLE 9: Fuzzing test results (24 hours) for the fuzzer-supported program.

Application	# Fuzzers	# Seeds	FP ratio (%)
gRPC	11	2,380	0%
PHP	8	9,016	0%
RocksDB	1	374	0%

To evaluate whether false positives are consistently suppressed by CMASan's false positive avoidance features, we additionally conduct long-term tests using a fuzzing tool and check if false positives are consistently suppressed. For this, we perform long-term fuzzing (i.e., 24 hours) on three (i.e., gRPC, PHP, and RocksDB) of our 12 selected target evaluation applications that offer fuzzing drivers (i.e., `# fuzzers`) or seeds (i.e., `# seeds`) for fuzzing. As shown in Table 9, the existing false positives in gRPC and PHP observed in previous evaluation (Table 3) are still effectively suppressed, and no additional false positives are observed during this evaluation.

Appendix D. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

D.1. Summary

CMASan is a novel memory bug detection tool specifically designed for applications using Custom Memory Allocators (CMAs). Unlike existing tools, CMASan is tailored to handle the unique characteristics of CMAs, ensuring accurate detection without requiring code modifications. CMASan efficiently tracks memory allocations, deallocations, and accesses, identifying various memory bugs such as buffer overflows, use-after-free, and double-free errors. Through its advanced techniques, CMASan minimizes false positives and provides a reliable solution for improving the security and reliability of CMA-based software.

D.2. Scientific Contributions

- Development of the first CMA-aware address sanitizer
- Introduction of novel techniques for CMA-specific memory bug detection
- Demonstration of improved detection capabilities in real-world applications

D.3. Reasons for Acceptance

1) The paper addresses a critical gap in memory bug detection tools by introducing the first CMA-aware address sanitizer. This contribution is significant because CMAs are widely used in various applications, and existing tools often fail to adequately detect memory bugs in these contexts.
2) Moreover, CMASan introduces innovative techniques specifically tailored to handle the unique characteristics of CMAs. These techniques, such as CMA API identification, metadata management, quarantine zones, and false positive suppression, demonstrate a deep understanding of the challenges associated with CMA-based memory bug detection.

D.4. Noteworthy Concerns

1) Compatibility: CMASan's compatibility with a wide range of CMAs and programming languages should be thoroughly evaluated. While the paper has focused on C/C++, its applicability to other languages or specialized CMAs might require additional adaptations.

Appendix E. Response to the Meta-Review

Since CMASan is developed based on ASan, it is compatible with other languages supported by ASan (e.g., Rust) and can support different CMA patterns as well. However, although the CodeQL used by CMASan supports a wide range of languages, it does not yet support languages (e.g., Rust) that ASan can be applied, requiring additional work. In the case of different CMA pattern supporting, besides the Arena and Recycler patterns, other CMA patterns (e.g., logging allocators) also exist. However, most of these CMA patterns are covered by ASan as their alloc and free functions are essentially wrappers of the standard library's malloc/free functions. The remaining major patterns, Arena and Recycler, which ASan cannot handle but CMASan covers, are widely used in real-world applications, with 71% of CMA-utilizing applications (among the top 100 programs on GitHub) relying on these patterns. Although CMASan currently covers most of the patterns that ASan does not, we will leave the identifying and addressing new additional patterns that CMASan does not currently cover as future work.