Practical Object-Level Sanitizer With Aggregated Memory Access and Custom Allocator

Xiaolei Wang, Ruilin Li✉, Bin Zhang✉, Chao Feng, Chaojing Tang
College of Electronic Science and Technology, National University of Defense Technology, Changsha, China
Email: xiaoleiwang, liruilin, b.zhang@nudt.edu.cn

***Abstract*—To mitigate potential memory safety vulnerabilities, recently there have been significant advances in sanitizers for pre-production bug detection. However, the limited inability to balance performance and detection accuracy still holds. The main reason is due to excessive reliance on shadow memory and a large number of memory access checks at runtime, incurring a significant performance overhead (if fine-grained memory safety detection is performed, the overhead will be even greater).**

In this paper, we propose a novel *O*bject-*L*evel *A*ddress *San*itizer *OLASan* to reduce performance overhead further while implementing accurate memory violations (including intra-object overflow) detection. Unlike previous sanitizers ignoring the correlation between memory access and objects, OLASan aggregates multiple memory accesses of same object at function level to perform on-demand targeted sanitization, thus avoiding examining most memory accesses at runtime. Specifically, OLASan characterizes various memory access patterns to identify those which can be aggregated, and implements memory safety checks with customized memory tagging.

We implement OLASan atop the LLVM framework and evaluate it on SPEC CPU benchmarks. Evaluations show that OLASan outperforms the state-of-the-art methods with 51.18%, 25.20% and 6.52% less runtime overhead than ASan, ASan−− and GiantSan respectively. Moreover, aided by customized memory tagging, OLASan achieves zero false negatives for the first time when testing Juliet suites. Finally, we confirm that OLASan also offers comparable detection capabilities on real bugs.

I. Introduction

Memory errors, such as buffer overflows, are critical issues [1], [2] in software development, accounting for the majority (70%) of severe vulnerabilities in large C/C++ codebases[3]. To mitigate these risks, numerous memory sanitizers [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14] have been developed to monitor memory accesses at runtime, such as known AddressSanitizer [4] (abbreviated as ASan). Most sanitizers use a dedicated shadow memory to label addresses in the application's memory as either *addressable* or *non-addressable*; memory accesses are deemed safe if all target bytes are addressable. To improve performance, recent research has focused on removing redundant checks and minimizing overhead, as seen in tools like ASAP [15], ASan−−[16], SANRAZOR [17], and GiantSan [18], etc.

Despite these advancements, practical challenges remain. Many sanitizers rely heavily on static analysis, which can lead to inefficiencies or missed vulnerabilities in complex scenarios. For example, ASAP [15] aggressively removes checks, risking missed vulnerabilities, while SANRAZOR [17] uses unsound patterns that cannot ensure removed checks are redundant. ASan−−[16], as discussed in [19], improves efficiency but faces challenges with false negatives in both temporal and spatial heap error detection. GiantSan [18] only provides a single-sided summary, failing to safeguard lower addresses effectively given only higher addresses, which can deteriorate efficiency in reverse traversals.

Hardware-assisted sanitizers, like HWASan [20], improve performance using memory tagging but provide only probabilistic security guarantees due to the limited entropy of hardware-generated tags. Furthermore, both software- and hardware-based approaches often overlook intra-object overflows, where buffer overflows occur between struct fields.

Our insight. Recently, location-based sanitizers have stood out among various efficient memory sanitizers due to their high compatibility, which stems from a simpler safety model. However, they typically need to check each memory access instruction separately to ensure no non-addressable bytes are accessed, which is very time-consuming. The root cause is that these sanitizers limit their view to each byte of shadow memory during memory access checking, ignoring the underlying objects. As a result, they often delay excessive checks until execution time, resulting in substantial performance overhead. In our research, we identified a key source of optimization that existing approaches underutilize—information about the memory objects being accessed and their access patterns. Specifically, accesses to large objects are not isolated events that must be verified individually at runtime. Instead, they often involve groups of similar (or identical) behaviors and relative displacement in locations visited as indicated in the code. The processing tasks delayed to execution time frequently perform redundant checks on individual members of these large groups of homogeneous accesses. Theoretically, by aggregating multiple memory accesses to the same object, more aggressive optimizations can be implemented at the function level. Consequently, we believe it is possible to leverage the information about memory objects accessed to further improve sanitizers. By recognizing and utilizing the access patterns and the context of memory objects, we can enhance sanitizer efficiency and reduce runtime overhead significantly.

Our approach. We propose OLASan, a hybrid sanitizer that integrates memory access pattern classification, dynamic profiling, and custom heap allocators with memory tagging to optimize memory sanitization. By leveraging object-level information, OLASan minimizes runtime checks through memory access aggregation and targeted sanitization while maintaining robust safety guarantees. Specifically, OLASan tackles **two main challenges**: reliably grouping memory accesses targeting the same object, and ensuring that boundary-only checks effectively detect memory violations. OLASan employs dynamic

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00213

profiling to collect runtime traces of memory access patterns, focusing on normal frequently executed paths (profiled paths). This information allows OLASan to implement selective instrumentation, particularly for array-based memory accesses, where most redundant checks can be safely removed. For paths not covered during profiling (unprofiled paths), OLASan skips memory aggregation and selective instrumentation but applies other targeted sanitization techniques, such as loop-oriented access sanitization (with limited profiling assistance), customized shadow checks, and struct access sanitization. For remaining complex or unanalyzable cases, OLASan falls back to default sanitization provided by ASan [4] to ensure safety. This hybrid approach balances performance optimization on profiled paths and comprehensive safety on unprofiled paths.

We implemented OLASan as a prototype on LLVM 15 and evaluated it with the Juliet benchmark [21] and real-world vulnerabilities, assessing its performance on the SPEC CPU2006 and CPU2017 suites. Results show that OLASan provides stronger security than leading sanitizers (e.g., HWASan, ASan, SoftBound/CETS, CryptSan, GiantSan, ShadowBound, PACMem), with reduced runtime overhead. Specifically, OLASan achieves an average runtime overhead of 69.59%, compared to 76.11%, 94.79%, and 120.77% for GiantSan [18], ASan−−[16], and ASan[4], respectively, demonstrating a clear performance advantage.

Goals, Scope, and Contributions. OLASan is designed for pre-production testing and debugging, where efficient sanitization is crucial to reduce overhead. Its primary application is in fuzzing, which often faces significant runtime costs due to comprehensive instrumentation. To address this, OLASan adopts a targeted sanitization strategy that optimizes memory checks for most access patterns while reserving default ASan's sanitization for complex or unanalyzable cases. OLASan's goal is to strike a balance between performance optimization and memory safety. It achieves this by leveraging hybrid analysis to characterize frequently executed paths and apply targeted sanitization techniques, such as selective instrumentation. For unprofiled paths during dynamic profiling, OLASan employs partial targeted sanitization or default ASan's sanitization to ensure safety. Note that OLASan focuses on optimizing performance and effectiveness of sanitization, concurrency-related memory vulnerabilities are not within its current scope. To summarize, we make the following contributions:

- To the best of our knowledge, we are the first to develop techniques—through customized allocators and sofware-based pointer tagging—to identify the object base address for memory access aggregation.
- We propose to classify memory access patterns for targeted sanitization. Additionally, we introduce a novel dynamic profiling technique for identifying access bases and bounds.
- To our best knowledge, we are the first to conduct extensive engineering modifications to port HWASan [20] to run x86_64 platform without page aliasing. This significant achievement paves the way for more efficient and accurate memory sanitization on x86_64 architectures.
- We have implemented a prototype of OLASan and evaluated it using real-world datasets. Experimental results demonstrate that OLASan outperforms existing sanitizers.

II. Preliminaries

This section introduces background knowledge relevant to OLASan and presents a motivating example.

A. Background Knowledge

1) Location based Memory Sanitizer: Sanitizers are dynamic tools that detect memory safety violations by using additional metadata to determine whether memory accesses are safe. Sanitizers can be categorized broadly into two types: **(1) Pointer-based sanitizers**, which rely heavily on pointer type information but often face compatibility issues in complex environments, and **(2) Location-based sanitizers**, which track memory addresses directly and are more widely adopted due to their simplicity. However, location-based sanitizers face efficiency challenges that OLASan aims to address.

As illustrated in Figure 1, location-based sanitizers typically allocate one-eighth of the virtual address space as shadow memory, with each shadow byte encoding the addressability of eight application bytes. Non-addressable padding regions, or 'redzones' are inserted around objects to detect spatial errors, while memory quarantine is used to detect temporal errors by delaying memory reuse. However, while location-based sanitizers determine if a byte is addressable, they cannot confirm if the byte belongs to the intended object.

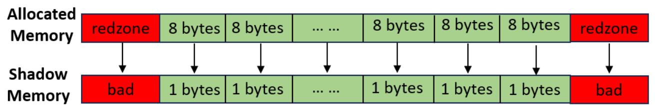


Fig. 1: Existing location-based encoding

2) Memory Tagging Extension: Memory Tagging Extension (MTE) is a hardware feature introduced in ARMv8.5 [22]. It builds on the Top Byte Ignore (TBI) feature, allowing pointer upper bits to store tags, which are ignored during address translation. MTE uses a 4-bit tag (16 values) stored separately from the application data. MTE implements a "lock-and-key" mechanism, allowing memory access only if the pointer tag (`key`) matches the memory tag (`lock`). A mismatch triggers a segmentation fault. The minimum memory unit associated with a tag is called a Tag Granule (TG), set at 16 bytes in MTE. For efficiency, existing MTE solutions [23], [24] often often rely on the Arm `IRG` instruction to randomly assign tags at allocation and deallocation. However, this approach introduces performance issues due to frequent retagging and provides only probabilistic security.

B. A Motivating example

Listing 1 presents a motivating example from the SPEC CPU2006 benchmark [25], illustrating representative memory access patterns. The code lines (1-11) demonstrate different memory displacements, assigning values from transition score arrays (`tsc`) to each pointer, such as `tpmi`. Each pointer corresponds to a specific type of transition score used in the `Viterbi` algorithm implemented in the case `456.hmmer`.

On line 12, heap memory is allocated for the `bzFile` struct. If the allocation is successful, several subsequent in-structure accesses (Lines 13-15) initialize the state variables of this struct. Upon successful initialization, a pointer to the struct is returned, indicating readiness to compress data implemented in case `401.bzip2`. Code lines (17-29) are from case `450.soplex` and describe a process of reallocating and copying string memory. Specifically, a loop-based memory access is used to reorganize a collection of strings stored in `mem` into a contiguous block `newmem` and then update indices in set to reflect new positions. Following this, a libc call `memcpy` copies data back to `mem`.

```
1  #define TMI 1
2  #define TIM 3
3  #define TII 4
4  #define TDD 6
5  void foo() {
6    int **tsc;
7    initialize(tsc);
8    int *tpim = tsc[TIM]; // offset:24
9    int *tpdd = tsc[TDD]; // offset:48
10   int *tpmi = tsc[TMI]; // offset:8
11   int *tpii = tsc[TII]; // offset:32
12   bzFile *bzf = malloc(sizeof(bzFile));
13   bzf->bufN = 0;
14   bzf->strm.bzalloc = NULL;
15   bzf->strm.bzfree = NULL;
16   ... ...
17   DataSet<int> set; // name set.
18   char *mem;          // string memory
19   spx_realloc(mem, memmax);
20   char *newmem = 0;
21   int newlast = 0;
22   spx_alloc(newmem, memSize()); // allocate memory
23   initialize_set(set);
24   for (int i = 0; i < num(); i++) {
25     const char *t = &mem[set[i]];
26     set_newmem(&newmem[newlast], t);
27     set[i] = newlast;
28     newlast += strlen(t) + 1;}
29   memcpy(mem, newmem, newlast);
30 }
```

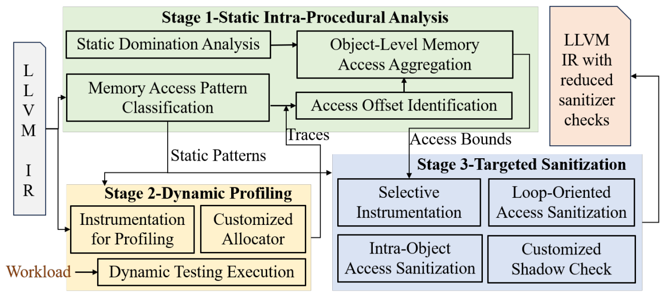
Listing 1: A simplified code extracted from SPEC2006

In practice, many functions involve multiple accesses to the same object within defined upper and lower offsets. For instance, in the example, lines 8-11 access the `tsc` object at various offsets, yet only the upper (Line 9) and lower (Line 10) offsets need to be instrumented for safe, efficient checking. Likewise, for loop-based (Lines 24-28) and libc-based (Line 29) memory accesses, upper and lower offsets can optimize instrumentation further. Standard sanitizers, like ASan [4], instrument every operation, incurring high runtime costs. This is because most sanitizers lack object-level context, leading to a compromise between accuracy and efficiency. Due to instrumentation overhead, ASan [4], HWASan [20], PACMem [26], and ShadowBound [27] struggle to detect intra-object overflows. For instance, struct `bzf` on lines 12-15 has fixed offsets and sizes for each member; if object information were available, sanitizers could consolidate per-member checks into a single object-level check, reducing instrumentation while still detecting intra-object overflows.

These insights motivated our design of **OLASan**, an object-level sanitizer that improves memory sanitization accuracy and efficiency. OLASan is primarily focused on critical safety-related memory accesses at the function level. It leverages the fact that most memory accesses can be determined to be valid within their corresponding memory object ranges. Given multiple memory accesses in a function, OLASan automatically analyzes and extracts objects and their structures. This object information allows OLASan to classify memory accesses into different patterns. Subsequently, optimized instrumentation checks are performed for these patterns in various ways, ensuring accuracy while minimizing the amount of instrumentation, thereby reducing costly runtime checks.

III. System Overview

OLASan is a hybrid analysis framework that optimizes memory sanitization by combining static analysis, dynamic profiling, and targeted sanitization techniques. As illustrated in Figure 2, OLASan involves three stages: **Stage 1 (Sec.III.A)**, OLASan performs static intra-procedural analysis to classify memory access patterns and facilitate memory access aggregation. Using domination analysis, it identifies opportunities for aggregation but relies on dynamic profiling for runtime information, such as base addresses and offsets. **Stage 2 (Sec.III.B)**, dynamic profiling collects runtime traces for profiled paths, which typically correspond to hot paths triggered by conventional unit tests. These traces are critical for enabling precise memory access aggregation and optimizing selective instrumentation, especially for array accesses. However, profiling has limited impact on loop-oriented sanitization and no effect on struct access sanitization or customized shadow checks. **Stage 3 (Sec.III.C)**, OLASan applies targeted sanitization techniques tailored to the memory access patterns identified in previous stages: For profiled paths, OLASan leverages comprehensive targeted sanitization, with a focus on selective instrumentation for array accesses. Particularly, loop-oriented access sanitization could be further refined when profiling data is available. For unprofiled paths, OLASan skips aggregation and selective instrumentation but tries to apply other three targeted sanitization (loop, struct-based and shadow checks) whenever possible. For memory operations that are complex or unanalyzable, OLASan falls back to default sanitization provided by ASan [4] to ensure comprehensive safety.



Stage 1-Static Intra-Procedural Analysis
Static Domination Analysis
Object-Level Memory Access Aggregation
Memory Access Pattern Classification
Access Offset Identification
LLVM IR with reduced sanitizer checks
L L V M I R
Traces
Static Patterns
Access Bounds
Stage 2-Dynamic Profiling
Instrumentation for Profiling
Customized Allocator
Workload
Dynamic Testing Execution
Stage 3-Targeted Sanitization
Selective Instrumentation
Loop-Oriented Access Sanitization
Intra-Object Access Sanitization
Customized Shadow Check

Fig. 2: System Overview

A. *Static Intra-Procedural Analysis*

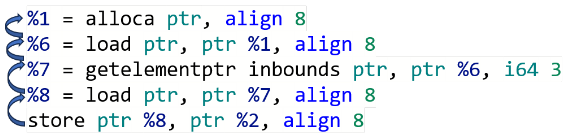
In this stage, OLASan performs intra-procedural analysis to aggregate memory accesses and identify those that can potentially be skipped in subsequent memory sanitization.

1) Memory Access Patterns Classification: Application programs exhibit a wide range of memory access patterns, from simple array accesses to complex structure accesses. OLASan begins by statically classifying all memory accesses into different patterns and identifying their access offsets. This classification is crucial for enabling targeted sanitization for different memory access patterns, a significant enhancement over previous methods. Based on a broad set of application analyses, we categorize memory accesses into four types, as shown in Table I. This classification facilitates subsequent sanitization optimizations, as discussed in later sections. While Table I may not cover all possible access patterns, the majority of observed accesses fit within these categories. For accesses that do not match any of these patterns, OLASan defaults to ASan [4] for sanitization.

TABLE I: Classification of Memory Access Patterns

Pattern	Example (from Listing 1)
Access to Array	tpim = tsc[TIM]
Access to Struct	bzf− >strm.bzfree
Access in Loop	for (int i = 0; i < num(); i++) set[i] = newlast;
Access using Libc	memcpy(mem, newmem, newlast)

2) Access Offset Identification and Challenges: For the classified access patterns, OLASan uses intra-procedural backward data flow analysis at compile time to extract memory access offsets. For each memory access, we start from the variable holding the memory address and traverse its use-def chain to identify the offset relative to the object's base address.



```
%1 = alloca ptr, align 8
%6 = load ptr, ptr %1, align 8
%7 = getelementptr inbounds ptr, ptr %6, i64 3
%8 = load ptr, ptr %7, align 8
store ptr %8, ptr %2, align 8
```

Fig. 3: Example of Backward Dataflow Analysis

Consider the example in line 8 of Listing 1, where the corresponding LLVM IR code is shown in Figure 3. Starting with the instruction `%8 = load ptr, ptr %7, align 8` (representing the operation *tsc[TIM]*), LLVM's Use-Def analysis allows us to trace back the definition of `%7` to a `getelementptr` instruction, revealing a computed address based on `%6` with an offset of 24 bytes (3 * 8). Further tracing leads us from `%6` to `%1`, indicating that `%8` accesses memory at an offset of 24 bytes from the base address of the object allocated at `%1`. Ideally, with this backward data flow analysis, all memory access addresses can be represented as `addr = object_base + offset`.

Identifying memory access offsets is **challenging** due to two primary issues. First, pointer aliasing complicates precise offset determination by overlapping memory regions. Second, pointer arithmetic and cross-function pointer propagation dynamically alter pointer semantics, making static offset computation difficult. Additionally, LLVM's alias analysis often struggles with pointer aliases, further hindering static offset calculations. To address these challenges, we propose a dynamic profiling technique to supplement static offset identification, as described in Sec. III.B.

3) Memory Access Aggregation with Domination Analysis: Once all memory accesses and their offsets within a function are identified, OLASan performs memory check pruning by aggregating memory accesses and replacing instruction-level checks with object-level checks. Formally, for a group of memory accesses to the same object (denoted as $\overrightarrow{M} = *ptr1, 2, 3, 4$ with offsets 8, 16, 24, and 32), only the accesses with the maximum and minimum offsets (i.e., $*ptr1$ and $*ptr4$) are checked. This ensures that the entire access range is covered. However, this strategy alone may lead to false negatives, particularly when memory accesses occur on different program execution paths. To address this, OLASan integrates branch-aware grouping logic in its domination analysis.

Algorithm 1 Memory Access Aggregation.

```
Input: All Memory Accesses $M$ with Bases $O$ and Offsets $S$ in $F$
Output: Sanitized Memory Accesses $M'$
 1: procedure ACCESS_AGGREGATION($M, O, S$)
 2:     Initialization: $M_o = dict()$; // key is object base and value is accesses set
 3:     for i=1 : $M.size()$ do // First step
 4:         $o_{base} = O[i]$;
 5:         $M_o[o_{base}].append(M[i])$;
 6:     for all ($base$, access_set) $\in M_o$ do // Second step
 7:         Initialization: $groups = list()$; // an empty list for each base
 8:         for $a \in$ access_set do
 9:             $group\_found \leftarrow$ False;
10:             $branch\_id$ = getBranchId($a$); // Get the branch ID being accessed
11:             for $g \in groups$ do
12:                 $dominance \leftarrow$ False;
13:                 $same\_branch \leftarrow$ True;
14:                 for $sub\_access \in g$ do
15:                     if hasDifferentBranch($g, branch\_id$) then
16:                         $same\_branch \leftarrow$ False;   break;
17:                     if $dominate(a, sub\_access)$ $or$
18:                       $post\_dominate(a, sub\_access)$ then
19:                         $dominance \leftarrow$ True;   break;
20:                 if $dominance$ $and$ $same\_branch$ then
21:                     Add $a$ to $g$;
22:                     $group\_found \leftarrow$ True;   break;
23:             if not $group\_found$ then
24:                 $groups \leftarrow groups \cup \{\{a\}\}$;
25:         for $g \in groups$ do // Third Step
26:             if $g.size() \leqslant 2$ then
27:                 Add $g$ to $M'$;
28:             $a_{max}$, $a_{min}$ = $SortByOffset(g, S)$;
29:             Add $a_{max}$, $a_{min}$ to $M'$;
30:     $return$;
```

Algorithm 1 describes the detailed steps. First, OLASan aggregates memory accesses to the same object base into a dictionary (Lines 2-5), using hybrid analysis to identify base addresses and access offsets. In the second step (Lines 6–24), OLASan analyzes memory accesses for each object individually, grouping those that share domination or post-domination relationships (Lines 18-19) and belong to the same branch (Lines 15–16). This ensures that memory accesses from different branches are not grouped together, preventing edge cases where mutually exclusive memory accesses (e.g., from distinct branches like A and B) could be incorrectly treated as part of the same group, potentially missing errors. Specifically, a `hasDifferentBranch` check validates whether a group already contains accesses from distinct branches, preventing incorrect grouping. In the final step (Lines 25–29), each memory group is processed independently. The memory accesses in a group are sorted by their offsets, and only the maximum and minimum offsets are selected for instrumentation (Line 29). For groups with fewer than two accesses (Lines 26-27), all memory accesses in the group are instrumented directly to ensure coverage.

B. Dynamic Profiling

Static calculation of memory access offsets is challenging, especially when the pointer's base address changes dynamically. Even with constant indices in a `GEP`(GetElementPtr) instruction, pointer arithmetic or aliasing can alter the base address, complicating static offset inference. To solve this, OLASan proposes dynamic profiling with instrumentation and a custom heap allocator, extracting the base addresses of memory accessing and associating them with source code info.

1) Instrumentation for Profiling: In LLVM IR, the `GEP` instruction is commonly used to calculate the result pointer for memory access by taking a base pointer and a set of indices. OLASan performs dynamic profiling at the pointer arithmetic site by instrumenting the GEP instruction and inserting a log statement after it to record the base pointer address. While this approach seems straightforward—logging base pointer addresses at `GEP` and computing offsets by subtracting these from the resulting pointer—several practical **challenges** arise. Firstly, due to pointer aliasing, arithmetic, and propagation, the recorded base pointer may not accurately reflect the original object, resulting in offsets that are not relative to the intended base. Secondly, the indices in `GEP` instructions may be input-dependent, leading to variable offsets across test cases during dynamic execution. To address these issues, we propose two strategies: **(1).** performing compile-time data flow analysis on `GEP` instructions to determine whether their indices are constant. Only `GEP` instructions with constant indices will be instrumented for profiling and optimization, while others will be left for default instrumentation checks to avoid false negatives. **(2).** introducing a custom heap allocator, allowing quick retrieval of the corresponding object base address for any given pointer, thereby accurately calculating the offset, as will be explained in detail below.

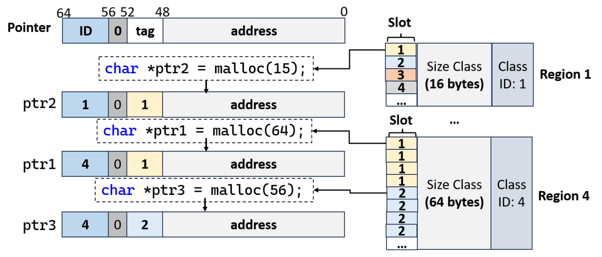


Fig. 4: Memory Organization and Allocation Examples

2) Custom Allocator: For performance reasons, many modern heap memory allocators use size classes to efficiently manage memory. For example, Scudo [28], the default memory allocator for the Android mobile OS, employs two types of allocators: a primary allocator for small-sized allocations and a secondary allocator for larger allocations (at least one page). The primary allocator is a size-class-based allocator, which reserves blocks of memory for objects of specific sizes, also known as size classes. It aligns allocations to 16 bytes to support memory tagging granules. Specifically, Scudo maps multiple memory regions, each assigned a size range (as illustrated in Figure 4). Chunks sized within a specific range are allocated from the corresponding region, with zero-permission guard pages used to separate these regions. These size ranges are referred to as class IDs in Scudo. For example, a chunk of size `0x18` is assigned the class ID 2. We build on these principles to implement our custom allocator. Notably, as shown in Figure 4, we embed the allocation's class ID into the top bits (56-64) of the pointer for subsequent use.

```
1 // assuming: tagged 'ptr' is target allocation
2 class_id = ptr >> 56;
3 ptr_raw = ptr & 0xFFFFFFFFFFFF;
4 region_base = GetRegionBeginBySizeClass[class_id];
5 class_size = ClassIdToSize[class_id];
6 base = region_base + ((ptr_raw - region_base) /
      class_size) * class_size;
7 offset = ptr_raw - base;
```

Listing 2: Object base and offset calculation

Based on pointer tagging with the Class ID, the object base address and offset of memory access can be accurately calculated, as demonstrated in Listing 2. The key steps are as follows: by extracting the embedded class ID (Line 2), we can quickly determine the starting address of the memory region where the allocation is located (Line 4) and the corresponding class size (Line 5). These details are easily accessible through our custom heap allocator.

Additionally, our custom heap allocator not only assists with base address and offset calculations but also improves memory error detection capabilities. Unlike the original HWASan [20], which uses random tag assignment and detects errors probabilistically by comparing pointers and memory tags (a method that may result in tag collisions between adjacent objects), our allocator employs a deterministic round-robin pattern for tag assignment. Specifically, for N-bit tags (with the default being 4 bits), each tag repeats every $2^N - 1$ slots. As illustrated in Figure 4, this ensures that the tag of any object in each region will not collide with that of a known number of neighboring slots. Consequently, our allocator provides robust protection against buffer underflows and overflows, with each memory slot having two implicit spatial guards composed of the surrounding $2^N - 1$ objects on both sides, each with a different tag. The effective size of the guards depends on the size class S: each object is protected by $(2^N - 1) \times S$ guard bytes. For example, our smallest size class (16 bytes) provides bi-directional guards of 240 bytes, while the largest size class (262 KB) offers 3.75 MB of guard coverage. Similar to HWASan [20], allocated tags are stored both in shadow memory as memory tags and in the top bits of pointers as pointer tags. However, in our case, the pointer tag is stored in the top 48-52 bits of the pointer. By using the embedded class ID information, our allocator can effectively detect all memory accesses exceeding the corresponding class size, including large offset accesses to other regions or slots.

3) Dynamic Testing Execution: In this phase, the instrumented program is executed with a specific workload to collect memory access profiling traces. Two considerations are important here: **(1).** Sanitizers like HWASan typically abort on detecting errors. To allow for complete trace collection, we re-implement HWASan to 'alert users' rather than abort, ensuring uninterrupted execution. **(2).** To capture comprehensive traces, the workload should cover diverse execution paths. Like SANRAZOR [17], OLASan uses default test cases to record dynamic patterns, and our observations indicate that runtime overhead is concentrated on hot paths, which are generally well-covered by default cases. If certain memory accesses are not encountered during profiling, OLASan will skip memory access aggregation and selective instrumentation for those accesses. However, other three targeted sanitization techniques will still attempt to handle and optimize them as much as possible. This conservative fallback maintains full protection without relying solely on profiling coverage.

C. Targeted Sanitization

The final stage of OLASan optimizes performance by applying targeted sanitization to identified memory access patterns, including those based on arrays, structs, loops, and libc calls. Each pattern offers unique optimization opportunities. OLASan's targeted sanitization applies checks selectively, enhancing performance without compromising detection accuracy. This section outlines the strategies for each pattern.

1) Selective Instrumentation: Within the targeted sanitization framework, the first optimization focuses on selective instrumentation. Instead of instrumenting every memory access indiscriminately, we strategically place sanitization checks where they are most needed. For regular array accesses, our approach involves analyzing access patterns and inserting checks specifically at the array boundaries. This method leverages results from static domination analysis and memory access aggregation to identify the most critical points for instrumentation, as described by motivating example (Sec.II.B). By focusing checks on these boundaries, we aim to detect memory errors effectively while minimizing the number of checks required. This targeted approach ensures that we catch common errors efficiently without imposing unnecessary overhead.

```
bool loopStart = false;           +
uintptr_t InitialAddress = 0;     +
uintptr_t EndAddress = 0;         +
for (int i = 0; i < num(); i++) {
  if (!loopStart) {
    InitialAddress = set[i];      +
    loopStart = true;}            +
  set[i] = newlast;
  EndAddress = set[i];            +
}
SanitizeCheck(InitialAddress, EndAddress) +
```

Listing 3: Example of Access Sanitization in Loop

2) Loop-Oriented Access Sanitization: Loop-oriented access sanitization addresses the challenge of repetitive memory accesses within loops by significantly reducing the number of checks while maintaining comprehensive coverage. Traditional sanitizers often insert checks for every iteration of a loop, leading to substantial performance overhead, especially in loops with numerous iterations. In contrast, OLASan identifies memory accesses in loops and applies reduced but comprehensive checks that cover all iterations. Specifically, for each memory access in a monotonic loop (i.e., loops with predictable, consistent access patterns), OLASan only checks the start and end addresses. For example, consider the memory access $set[i]$ from Listing 1, instead of inserting potentially hundreds or thousands of checks for each iteration, OLASan's optimization reduces the number of checks to just two through recording and checking the start and end access address (as shown in Listing 3). In reality, the mem $[set[i]]$ and other memory accesses in loop, will also be sanitized in same way, greatly enhancing efficiency. This boundary-based checking strategy is effective for loops with monotonic access patterns, ensuring that intermediate accesses are safe if the boundaries are verified. Additionally, dynamic profiling and static domination analysis refine this strategy by identifying accesses to the same object within loops during execution, allowing OLASan to selectively remove redundant checks while maintaining memory safety. By focusing on these boundary checks, OLASan not only reduces the performance overhead but also ensures that out-of-bounds errors are effectively captured. This approach is particularly beneficial in tight loops with high iteration counts, where the overhead of per-iteration checks can be prohibitively expensive.

```
1  memcpy(cs->bitcounter, currMB->bitcounter, 3960);
2  // optimize sanitization for libc call
3  class_id = cs->bitcounter >> 56;
4  ptr_tag = (cs->bitcounter >> 48) && 0x000F;
5  ptr_raw = cs->bitcounter && 0xFFFFFFFFFFFF;
6  class_size = ClassIdToSize[class_id];
7  if (class_size < 3960) {
8    ErrorAlert();
9  }
10 tag_first = getShadow(ptr_raw);//memory tag
11 if (tag_first != ptr_tag)
12 {
13   ErrorAlert();
14 }
15 tag_last = getShadow(ptr_raw + 3960);
16 if (tag_last != ptr_tag) {
17   if (!PossiblyShortTagMatchesCheck())
18       ErrorAlert();
19 }
```

Listing 4: Example of Customized Sanitization for Libc Call

3) Customized Shadow Check for Libc Calls: To efficiently handle memory accesses via `libc` function calls such as `memcpy`, OLASan introduces customized shadow checks tailored specifically for these cases. This approach leverages our understanding of `libc` function semantics. By embedding the object's class size into the pointer's high bits and easily restoring it, our approach allows for a significant reduction in the number of runtime checks required. For instance, when handling a `memcpy` call that copies 3960 bytes, instead of performing a safety check every 8 bytes, OLASan first extracts the target object's class size from the pointer and compares it with the length of the memory operation. If the restored class size is less than 3960 bytes, an error is flagged. Otherwise, only the start and end of the memory access are checked with software-based memory tagging (Lines 10-15) for safety, as demonstrated in Listing 4. Specifically, the pointer tag is compared with the memory tags at the start and end positions to ensure they are equal (Lines 11, 16). By focusing on the start and end boundaries of the memory region affected by the `libc` function, OLASan ensures robust error detection and optimized performance.

4) Struct Access Sanitization: Struct access sanitization in OLASan focuses on optimizing intra-object memory checks by leveraging the natural grouping of memory accesses within structured data types (where it is applicable), such as structs. Instead of instrumenting individual field accesses, OLASan applies a holistic approach, checking the struct's boundary as a whole. Once constant offsets of specific struct fields are identified and validated as safe, OLASan skips instrumentation for these fields, significantly reducing overhead while maintaining safety. Specifically, for struct field accesses identified by `GEP` instructions, OLASan performs static checks to validate their source object types. Using metadata extracted from the struct definition, OLASan calculates field offsets and verifies whether they fall within valid boundaries. For nested structs, OLASan recursively adjusts metadata to account for deeper hierarchies, maintaining accuracy. This approach minimizes runtime costs by consolidating per-field checks into a single struct-level check. OLASan's struct access sanitization is applicable and particularly effective for structs with predictable, fixed memory layouts. For cases involving non-constant or complex offsets, OLASan defaults to ASan's [4] conventional sanitization to ensure robustness.

IV. Evaluation

This section addresses four key questions to assess the effectiveness and efficiency of OLASan:

- **RQ1:** Does OLASan have a performance advantage?
- **RQ2:** Can OLASan effectively detect real bugs?
- **RQ3:** How do individual customized components affect time overhead?
- **RQ4:** What are the characteristics of OLASan's profiling-based removed checks through memory access aggregation?

Benchmark. We evaluate OLASan's performance using the SPEC CPU2006 [25] and CPU2017 [29] benchmarks(RQ1), conducting an ablation study with the same benchmarks to evaluate the impact of OLASan's different optimization components(RQ3). We then use the Juliet Test Suite [21], Magma [30], and Linux Flaw Project [31] to evaluate OLASan's detection capabilities(RQ2). Finally, we perform additional statistical and manual analysis to better understand the characteristics of profiling-based removed checks(RQ4).

Hardware and Setup. All the experiments were conducted on a server configured with Ubuntu20.04, Intel Xeon-6254 CPU, 128GB RAM, and 1T SSD storage. For fair evaluation and comparison, we compared OLASan with several known sanitizers, includes GiantSan [18], PACMem [26], CryptSan [32], ASan−− [16], HWASan [20], ASan [4] and Softbound/CETS[33]. As for configuration, we use the default settings listed in the ASan documentation for all ASan-based implementations and OLASan. For ASan, ASan−−, HWASan and GiantSan, we disable the functionality that we do not implement, such as use-after-scope detection. Also, to ensure fairness in our comparisons, we configured all tools to disregard detected errors by setting `halt_on_error=false`, preventing early termination of the evaluation due to the widely reported errors existing in the SPEC benchmark.

A. Performance Study

Setting. In this section, we evaluate the performance of OLASan using the SPEC CPU2006 and CPU2017 benchmarks. To put the performance of OLASan into better perspective, we compare the overhead to related state-of-the-art sanitizers. As we know, each SPEC benchmark is shipped with a training workload, a testing workload, and a reference workload. Following the convention, we use the training workload to profile these programs and obtain the dynamic traces (see Sec.III.B). After the targeted sanitization implemented by OLASan, the reference workload is used to measure the performance of the sanitized programs. We do not use test workload since it leads to a much shorter execution time compared with the reference and training workload. Regrettably, due to code compatibility issues, we had to exclude several cases from the SPEC benchmarks, as OLASan failed to compile them due to these codes being too old. Finally, 18 SPEC C benchmarks are selected to measure performance overhead. For each tool, we executed the benchmarks ten times and reported the average results to mitigate the impact of randomness.

TABLE II: Performance Study on SPEC

	Benchmark	ASan	ASan−−	GiantSan	OLASan	OLASan++[b]
SPEC2006	401.bzip2	198.18%	152.24%	RE[a]	200.99%	225.32%
	445.gobmk	192.23%	132.43%	163.5%	183.59%	197.00%
	450.soplex	197.27%	236.12%	212.58%	170.67%	203.42%
	453.povray	230.42%	279.79%	224.29%	235.29%	275.95%
	456.hmmer	200.28%	218.68%	222.81%	204.99%	223.16%
	458.sjeng	208.05%	171.5%	165.64%	190.04%	212.78%
	470.lbm	204.27%	142.6%	200.37%	169.58%	206.78%
	471.omnetpp	199.79%	RE[a]	221.32%	167.59%	252.29%
	473.astar	218.78%	249.57%	142.86%	154.62%	209.86%
SPEC2017	500.perlbench_r	238.24%	225.46%	200.56%	264.7%	277.3%
	502.gcc_r	399.25%	292.5%	278.91%	230.96%	295.63%
	505.mcf_r	222.04%	141.42%	127.82%	130.98%	165.15%
	508.namd_r	231.38%	168.23%	107.46%	133.05%	192.98%
	520.omnetpp_r	292.92%	258.94%	196.79%	130.81%	238.99%
	531.deepsjeng_r	206.45%	162.4%	141.18%	128.59%	176.45%
	538.imagick_r	196.56%	176.8%	136.47%	115.71%	187.29%
	541.leela_r	209.11%	186.2%	145.61%	135.41%	194.93%
	619.lbm_s	128.71%	116.5%	105.63%	105.03%	134.63%
	Average	220.77%	194.79%	176.11%	169.59%	214.99%

[a]Runtime Error, [b]OLASan with Intra-object Level Check.

Results. Table II shows the runtime overhead of OLASan, ASan, ASan−− and GiantSan when evaluating SPEC benchmarks on the x86-64 architecture. Each column represents the ratio compared to the native execution, which is used to indicate runtime overhead. The performance evaluation on the SPEC benchmarks demonstrates the effectiveness of OLASan in optimizing memory access checks compared to ASan, ASan−−, and GiantSan. OLASan achieves an average overhead of 169.59%, which is lower than ASan's 220.77%, ASan−−'s 194.79%, and GiantSan's 176.11%. This reduction in overhead highlights OLASan's ability to minimize unnecessary memory checks through object-level sanitization, thereby enhancing runtime efficiency. Notably, OLASan outperforms other sanitizers across several benchmarks, such as `450.soplex` and `538.imagick_r`, where it shows a considerable reduction in overhead.

OLASan++, an enhanced version of OLASan incorporating configurable intra-object level checks, further strengthens the sanitizer's capability by offering fine-grained detection of intra-object overflows. Despite introducing additional checks, OLASan++ maintains a competitive average overhead of 214.99%, which is still favorable compared to ASan (220.77%). The inclusion of intra-object level checks allows OLASan++ to detect subtle memory errors that might be overlooked by other sanitizers, providing a higher level of memory safety. This demonstrates a trade-off between performance and detection accuracy, showcasing OLASan++ as a robust solution for scenarios requiring stringent memory safety without significant performance compromise.

The performance discrepancy (stand at 235.29% and

105.03% respectively) between SPEC CPU2006 and CPU2017 benchmarks stems from complex loop patterns in CPU2006 benchmarks like `456.hmmer` and `471.omnetpp`. For instance, `456.hmmer` features irregular loop structures characterized by data-dependent iteration patterns, non-contiguous memory accesses, and complex pointer arithmetic. Similarly, `471.omnetpp` includes nested loops with interdependent conditions, where memory access patterns vary across iterations. Such irregularities hinder OLASan's optimizations, which are more effective on predictable patterns like sequential accesses.While these benchmarks highlight edge cases for OLASan, our experiments with larger, real-world applications reveal that such patterns are rare in practice. OLASan successfully optimizes the majority of program paths in typical software environments, where predictable and repetitive memory access patterns are more common.

We also evaluated OLASan's compilation time compared to ASan [4], and ASan−− [16] using the above selected 18 SPEC benchmarks. The results show that, ASan increases compilation time by an average of 69.14% relative to LLVM, while ASan−− and OLASan introduce higher overheads of 141.28% and 211.09%, respectively. OLASan's increased compilation time is primarily due to its additional static analysis and targeted sanitization, which are essential for identifying and optimizing memory access patterns. This added compilation time represents a trade-off for improved accuracy and reliability. In practice, OLASan can compile large applications like Nginx in less than one minute, demonstrating that the overhead is practical for deployment scenarios. Furthermore, OLASan can finish the compilation of most evaluation benchmarks in minutes. Note that compilation is a one-time process during the build phase. This makes the compilation overhead acceptable for most use cases, especially in scenarios where sanitization's overhead is a priority.

B. Security Evaluation

In this section, we compare the detection capabilities of OLASan against known sanitizers. Specifically, we firstly use the Juliet Test Suite [21] used in previous work [10], [16] to test a wide range of synthetic bugs. To further systematically evaluate the security and functionality correctness of OLASan, we utilize the widely used Linux Flaw project [31] and Magma [30] to demonstrate effectiveness with realistic bugs.

1) Evaluation on Juliet Test Suite: Juliet Test Suite contains hundreds of test cases to detect memory safety errors. We select the bug categories that are relevant to spatial and temporal memory errors, including heap overflow, use after free, etc. Juliet contains cases that wait for an external signal (e.g., sockets), and some test cases include a randomized version (triggered with probability). Most previous works exclude test cases that depend on external input (e.g., `fgets`) to easily enable automation, thus the limited number of evaluated test cases is used, e.g., 11531 and 5364 for PACMem [26], CryptSan [32], respectively. In contrast, for our evaluation, we design a new automation framework with a dummy server to provide external inputs (e.g., `socket`) and does not exclude any test cases, resulting in a total of 15752 test cases listed in Table III. However, for the evaluation of Softbound/CETS, additional cases leading to compilation errors had to be excluded, resulting in only 3970 cases.

Results. The results of the selected tests are presented in Table IV, which shows the percentage of detected tests. Numbers in bold indicate the presence of false positives. From Table IV, we can make the following observations: **1).** In all selected tests, OLASan passed without producing any false positives or false negatives. **2).** Most existing sanitizers have much more false negatives mainly due to their initial design and engineering implementation. From a design perspective, previous sanitizers (ASan, HWASan, CryptSan, PACMem, GiantSan) cannot detect intra-object overflow. Also, ASan cannot detect cross-object overflow that skips redzones, or access an object that has been freed for a while. And HWAsan cannot detect invalid free bugs. In addition, more false negatives are due to imperfect implementation. Specifically, ASan fails to detect some heap buffer overflows from CWE 122, due to lacking instrumentation for `wcsncpy, wmemset`, etc. We also found that the function `__asan_alloca_poison` used by ASan to set the redzones does not work in some cases, causing false negatives. For HWAsan, it does not handle stack variables allocated by the `ALLOCA` and some common memory access functions such as the `snprintf` and `strncat` functions. These incomplete handling account for most of bugs missed by HWASan. Like HWASan, SoftBound/CETS does not handle some common memory access functions such as the `strncat` functions, causing false negatives. For GiantSan, it misses some buffer overflows occurred in the loop. Take `Stack_*_Overflow__CWE131_loop_09` in CWE121 for example, GiantSan overrides the loop iterator and does not insert safety checks for memory accesses in loop. **3).** None of the approaches except SoftBound/CETS have false positives. Regarding false positives, there are some flaws in the prototype implementation of SoftBound+CETS, and thus it has a high false positive rate. **4).** Aided by our custom memory tagging, OLASan is the only sanitizer able to detect all 72 intra-object overflow cases in `CWE` 121 and 122, such as `CWE122*overrun_memcpy_17.c`. Although Softbound/CETS claims to detect intra-object overflow conceptually, the security tests have shown that the implementation cannot. Overall, on the Juliet, OLASan performs better in detecting spatial and temporal memory vulnerabilities than other sanitizers in Table IV.

2) Evaluation on Real Applications: To further assess OLASan's detection capabilities, we validate it on the real-world vulnerability benchmark. The evaluation focuses on the comparison with the common-used ASan [4] and most recent GiantSan [18], ShadowBound [27]. This assessment included 30 reproducible real-world vulnerabilities collected from Linux Flaw Project [31], MAGMA dataset [30] and previous work ShadowBound [27]. Details of the vulnerabilities are presented in Table V. These vulnerabilities spanned a diverse set of 17 applications, encompassing servers, video encoders, language interpreters, widely adopted libraries, and

TABLE III: Description of Juliet Test Suite

CWE Name	Vulnerability Type	Number of Samples
CWE121	Stack Buffer Overflow	4896
CWE122	Heap Buffer Overflow	3777
CWE124	Buffer Underwrite	1440
CWE126	Buffer Overread	2004
CWE127	Buffer Underread	2000
CWE415	Double Free	818
CWE416	Use After Free	393
CWE761	Invalid Free	424
Total	–	15752

TABLE IV: Comparison of Memory Violation Detection

Name (#cases)	OLASan (15752)	GiantSan (15752)	PACMem (11531)	CryptSan (5364)	HWASan (5364)	ASan (15752)	SoftB/CETS (3970)
CWE121	100%	*82.14%*	*98.82%*	*98.5%*	*82.9%*	*83.74%*	**77.7%**
CWE122	100%	*83.92%*	*99.01%*	*97.4%*	*94.6%*	*83.92%*	**73.7%**
CWE124	100%	*80.18%*	100%	100%	*81.9%*	*80.18%*	**82.5%**
CWE126	100%	*82.89%*	100%	100%	*99.7%*	*82.89%*	**96.5%**
CWE127	100%	*91.01%*	100%	100%	*75.9%*	*91.01%*	**78.4%**
CWE415	100%	100%	100%	100%	100%	100%	100%
CWE416	100%	*90.41%*	100%	100%	*50.9%*	*90.41%*	**51.3%**
CWE761	100%	*91.56%*	100%	100%	*0%*	*91.56%*	100%

TABLE V: Vulnerability Detection on Real Applications.

CVE	Type	Program	ASan	GiantSan	ShadowBound	OLASan
CVE-2006-6563	HBO[a]	proftpd	✓	✓	✓	✓
CVE-2009-2285	HBO	libtiff	✓	✓	✓	✓
CVE-2013-4243	HBO	libtiff	✓	✓	✓	✓
CVE-2013-7443	HBO	sqlite	✓	✓	✓	✓
CVE-2014-1912	HBO	python	✓	✓	✓	✓
CVE-2015-8668	HBO	libtiff	✓	✓	✓	✓
CVE-2015-9101	HBO	lame	✗	✗	✓	✓
CVE-2016-1762	HBO	libxml	✓	✗	✓	✓
CVE-2016-1838	HBO	libxml	✓	✓	✓	✓
CVE-2016-10095	SBO	libtiff	✓	✓	✗	✓
CVE-2016-10270	HBO	libtiff	✓	✗	✓	✓
CVE-2016-10271	HBO	libtiff	✓	✗	✓	✓
CVE-2017-7263	HBO	potrace	✓	✓	✓	✓
CVE-2017-14409	SBO[b]	mp3gain	✓	✓	✗	✓
CVE-2018-20330	HBO	libjpeg-turbo	✓	✓	✓	✓
CVE-2019-7310	HBO	poppler	✓	✓	✓	✓
CVE-2019-9021	HBO	php	✓	✓	✓	✓
CVE-2019-9200	HBO	poppler	✓	✓	✓	✓
CVE-2019-10872	HBO	poppler	✓	✓	✓	✓
CVE-2020-19131	HBO	libtiff	✓	✓	✓	✓
CVE-2020-19144	HBO	libtiff	✓	✓	✓	✓
CVE-2021-3156	HBO	sudo	✗	✗	✓	✓
CVE-2021-4214	HBO	libpng	✗	✗	✓	✓
CVE-2021-26259	HBO	htmldoc	✓	✓	✓	✓
CVE-2021-32281	HBO	gravity	✓	✓	✓	✓
CVE-2022-0080	HBO	mruby	✗	✗	✓	✓
CVE-2022-0891	HBO	libtiff	✓	✗	✓	✓
CVE-2022-0924	HBO	libtiff	✗	✗	✓	✓
CVE-2022-28966	HBO	wasm3	✓	✓	✓	✓
CVE-2022-31627	HBO	php	✓	✓	✓	✓

[a]Heap Buffer Overflow, [b]Stack Buffer Overflow.

UNIX utilities. For each vulnerability, we collected the corresponding Proof of Concept (PoC) to trigger crashes, and evaluated the instrumented programs with each sanitizer.

Results. In Table V, we present the evaluation results on vulnerabilities selected by us. OLASan successfully detects all cases, demonstrating its capability to detect real bugs. Results also showed that, ShadowBound has better detection capability than ASan and GiantSan. There are two vulnerabilities that have not been detected by ShadowBound, mainly because ShadowBound's design is unable to detect stack overflow ones. As shown in the Table V, ASan fails to detect 5 PoCs because the illegal memory access crosses the RedZone instrumented by ASan and access another valid object. Because GiantSan is based on ASan, their detection results on most programs are the same, either detected or missed. Interestingly, compared to ASan, GiantSan has 4 more false negatives. Take `CVE-2016-1762` as an example, ASan detects the heap overflow at line 535 of `parserInternals.c`, however GiantSan fails to detect it. Our manual check found that the safety check logic inserted by GiantSan is not perfect and general, failing to catch such situation. So, although GiantSan has improved performance, it has sacrificed some detection capabilities, which is not what we want.

It is worth mentioning that, while no unknown vulnerabilities were found during our experiments. However, evaluation results on the Juliet Test Suite and real-world applications show that OLASan also flagged several complex memory access patterns that were overlooked by existing sanitizers. These findings suggest OLASan's potential to uncover new types of vulnerabilities, especially in cases involving intra-object overflows or rare access patterns. Future work will further explore OLASan's capacity to detect previously unknown vulnerabilities through extended testing.

C. Performance Overhead Buildup and Ablation Study

1) Overhead Buildup: To better understand the source of the overhead, we measure the impact of multiple different components separately. Specifically, we conducted a detailed breakdown of the runtime overhead using the SPEC benchmarks. Figure 5 illustrates the overhead sources categorized into two primary components: **Pointer Untagging** and **Safety Check**. Given the x86_64 platform, OLASan incurs significant overhead from pointer untagging due to the necessity of removing high bits from pointers before each memory access. This operation is critical to ensure compatibility and correctness when using tagged pointers for memory safety.

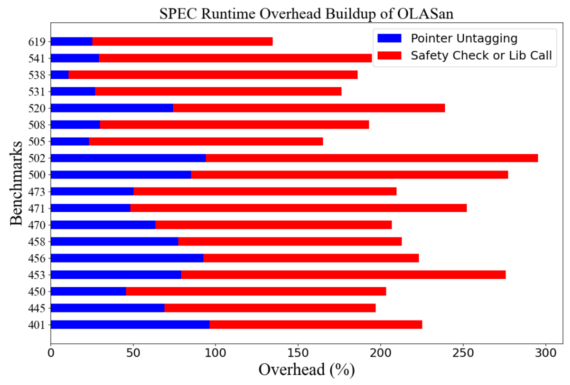


Fig. 5: SPEC runtime overhead buildup of OLASan.

Results. From Figure 5, we observe that for most benchmarks, the majority of the overhead is attributed to safety checks, which may also call library calls. For instance, benchmarks such as `500.perlbench_r`, `502.gcc_r` exhibit high overheads primarily due to the extensive safety checks implemented by OLASan. On the other hand, pointer untagging also contributes a substantial portion of the overhead, particularly in benchmarks like `401.bzip2`, `445.gobmk`. This indicates that while OLASan's safety mechanisms are effective, optimizing pointer untagging operations could further reduce the overall performance overhead.

2) Ablation Study on Targeted Sanitization Contribution: To demonstrate the effectiveness of OLASan's targeted sanitization strategies, we break down the contributions of each optimization technique (introduced in Sec.III.C) on the overall performance. Figure 6 demonstrates the ratio of checks optimized by profile-based selective instrumentation, loop-oriented sanitization, struct access sanitization, and customized shadow check across various benchmarks, respectively.

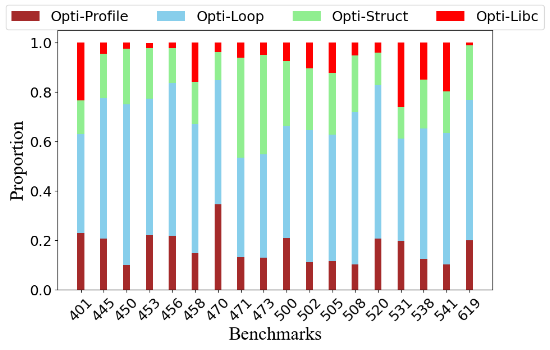


Fig. 6: The proportion of optimized checks handled by different OLASan's targeted sanitization. X-labels are SPEC benchmarks. Opti-profile refers to selective instrumentation.

Results. As depicted in Figure 6, the profile-based elective instrumentation performs well in benchmarks such as `401.bzip2`, `453.povray` and `470.lbm`, where it significantly lowers the overhead by eliminating redundant checks identified during the profiling phase. Loop-oriented sanitization plays a crucial role in benchmarks with intensive loop operations, such as `450.soplex`, `508.namd_r` and `520.omnetpp_r`, effectively minimizing the overhead by applying loop-specific optimizations. The struct access sanitization and customized shadow check for libc calls also demonstrate substantial contributions in benchmarks like `458.sjeng` and `531.deepsjeng_r`, respectively. These optimizations leverage the specific memory access patterns of structures and libc function calls to fine-tune the sanitization process, thus enhancing performance.

Overall, the targeted sanitization techniques proposed in OLASan collectively contribute to a significant reduction in performance overhead, validating the effectiveness of our approach. By focusing on the specific memory access patterns and eliminating unnecessary checks, OLASan achieves a more efficient and precise memory safety enforcement, leading to better performance compared to traditional sanitizers.

D. *Characteristics of Profiling based Removed Checks*

Next, we conducted statistical and manual analysis of memory safety checks removed by OLASan's selective instrumentation through profiling and aggregation, on both SPEC benchmarks and the real Nginx application. By comparing with ASan−− [16], we identify several characteristics of profiling-based removed checks, as outlined below.

1) Conservative Profiling Approach: Figure 7 illustrates the number of removed checks across 18 SPEC benchmarks for both ASan−− [16] (blue) and OLASan's profiling-based (red). Compared to ASan−−, OLASan's profiling-based method removes significantly fewer checks, a result that aligns with our conservative design philosophy. While ASan−− relies on static analysis to remove a broader range of checks, dynamic profiling only help eliminates checks for memory accesses that are observed in prior executions. This conservative strategy minimizes the risk of accidentally removing checks for unobserved memory access patterns.

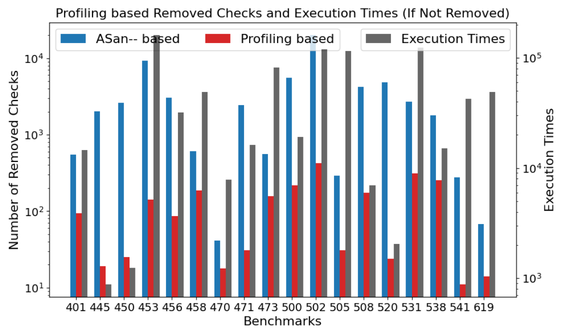


Fig. 7: Statistics of Removed Sanitization Checks on SPEC

2) Concentration of Removed Checks in Hot Paths: Despite removing fewer checks, our profiling-based method specifically targets checks associated with high-frequency memory accesses, as shown by the gray bar in Figure 7 (right axis). This indicates that most of the removed checks reside within hot paths, where repeated sanitization checks would introduce considerable runtime overhead. We also extended our analysis to a large-scale application Nginx and the results (show in Figure 8) follow a similar pattern to that observed in the SPEC benchmarks: *fewer checks are removed compared to ASan−−, yet the removed checks correspond to frequently executed memory accesses*. By selectively reducing checks in these performance-critical paths, OLASan's profiling-based approach could help reduce the runtime cost of sanitization.

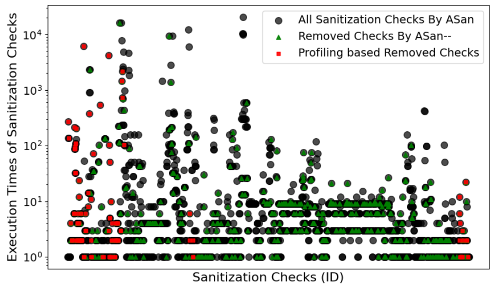


Fig. 8: Statistics of Removed Sanitization Checks on Ngnix.

3) Distribution of Removed Checks: A manual inspection of the removed checks reveals a strong concentration within linear, non-branching paths. This distribution suggests that OLASan's profiling-based method selectively removes checks in straightforward code sections, avoiding more complex control-flow regions where unique edge cases could arise. By retaining checks in branching or conditional paths, OLASan ensures comprehensive coverage for edge cases and complex scenarios, while optimizing for performance in linear, frequently executed sections.

V. Discussion

Memory Overhead and Implementation Complexity. OLASan enhances memory access checks by using object-level information, but its reliance on profiling and memory tagging adds implementation complexity and incurs memory overhead. This overhead, while manageable in typical scenarios, could pose issues in memory-constrained environments.

Future optimizations should focus on minimizing both memory consumption and implementation complexity to facilitate integration into large-scale software projects.

Handling Non-Contiguous Memory Access Patterns. While OLASan effectively optimizes loops with contiguous memory access patterns, it does not yet address non-contiguous memory structures like linked lists. For such cases, OLASan defaults to ASan's [4] default sanitization to ensure safety, though at the cost of reduced optimization. Addressing this limitation would involve refining its analysis capabilities to recognize and optimize diverse loop structures, thus extending its utility across a broader range of applications.

Edge Cases and Concurrency Related Vulnerabilities. Although OLASan has shown robust performance, edge cases with complex or atypical memory access patterns may still lead to suboptimal sanitization or false negatives. Although our tests did not reveal such instances, their potential in more complex applications underscores the need for future work to identify and address these cases systematically. Additionally, OLASan's design focuses are not on concurrency-related memory vulnerabilities. Integrating OLASan with concurrency-focused tools like ThreadSanitizer [34] could enable it to address this kind of vulnerabilities.

VI. Related Work

Below we discuss the most relevant works that broadly categorized into: 1) Sanitizer Optimization, 2) Pointer Tagging and 3) Hardware Expansions.

Sanitizer Optimization Numerous approaches [26], [16], [35], [36], [15], [17], [37], [18] have been proposed to optimize sanitizers and reduce runtime overhead. Some methods, such as GiantSan [18], employ location-based strategies with probabilistic quarantine mechanisms to improve performance, but they remain limited in detecting intra-object overflows and rely heavily on location-based assumptions that introduce inefficiencies in certain scenarios. Other optimizers, such as ASAP [15], focus on profiling to identify hot code for more efficient checking, though this performance-driven focus can result in missed memory errors. Meanwhile, ASan−−[16] and SANRAZOR[17] apply custom optimizations to reduce ASan's [4] checking overhead, yet cannot achieve an order-of-magnitude reduction. OLASan, by contrast, uses an object-level approach to aggregate memory accesses, reducing redundant checks and thus achieving higher efficiency and accuracy. Unlike static-only methods, OLASan's dynamic profiling allows it to identify access patterns in frequently executed paths, ensuring that optimizations apply to areas with the most significant runtime impact.

Pointer Tagging Traditional pointer-based solutions [38], [33], [39] require extra instructions to propagate metadata (e.g., bound) along pointer arithmetics. With the proliferation of large bit-width systems (e.g., 64-bit), recent methods [40], [41], [37], [26] leverage the pointer's upper spare bits to propagate metadata, enabling memory safety checks without adding separate metadata storage. Approaches like PACMem [26] utilize ARM's pointer authentication codes to encode safety information directly in the pointer. While effective, these methods often involve complex encoding and decoding processes that introduce additional arithmetic operations and can hinder performance, especially in high-frequency access patterns. In contrast, OLASan integrates object-level information with customized memory tagging, enabling efficient memory safety checks that reduce the need for complex pointer manipulation. This method enhances both intra- and inter-object protection without the performance penalties seen in metadata-heavy schemes, providing a simpler and faster safety model.

Hardware-Assisted Sanitizers Recent hardware-based sanitizers, such as HWASan [20], CryptSan [32], and others [23], [42], [28], [43], take advantage of CPU features like Arm's Memory Tagging Extension (MTE) to support metadata storage and memory checking in hardware. These tools, while efficient, rely on random memory tagging, resulting in only probabilistic memory corruption detection due to tag collision risks. Specifically, MTSan [23] ensures that neighboring objects have different tags. [43] performs extensive static analysis to deterministically protect the stack through tag forgery prevention. However, this design results in a reported overhead that is higher even than MemTagSanitizer [44] and is hence unsuitable for production use. Additionally, such approaches also struggle to detect intra-object overflows, as they rely on coarser-grained tagging models. In comparison, OLASan's object-level sanitization addresses both inter- and intra-object memory errors directly, using object-level aggregation and targeted sanitization to maintain high detection rates without excessive runtime checks. OLASan's design thus provides a more comprehensive memory safety model, reducing the runtime burden associated with repetitive checks.

VII. Conclusion

In this paper, we presented OLASan, a novel Object-Level Sanitizer designed to minimize performance overhead while maintaining high standards of memory safety. We also introduce a customized memory tagging mechanism that further enhances OLASan's capabilities. By aggregating memory accesses at the function level and applying targeted sanitization, OLASan effectively addresses key limitations of existing sanitizers, such as excessive runtime checks and false negatives, particularly in cases of intra-object memory violations. Our evaluation on SPEC CPU benchmarks demonstrates that OLASan achieves lower runtime overhead compared to state-of-the-art sanitizers, while preserving robust detection capabilities. Additionally, we validated OLASan's effectiveness on the Juliet test suites and through real-world bug detection, confirming its reliability in diverse scenarios.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments. This work was partially supported by the China Postdoctoral Science Foundation (Grant GZC20233547). Bin Zhang was supported by the Natural Science Foundation of China (Grant No. 62302506).

REFERENCES

[1] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "Sok: Sanitizing for security," *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1275–1295, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:48364047

[2] L. Szekeres, M. Payer, T. Wei, and D. X. Song, "Sok: Eternal war in memory," *2013 IEEE Symposium on Security and Privacy*, pp. 48–62, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:2937041

[3] A. Rebert and C. Kern, "Secure by design: Google's perspective on memory safety," Google Security Engineering, Tech. Rep., 2024.

[4] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:11024896

[5] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," *International Symposium on Code Generation and Optimization (CGO 2011)*, pp. 213–223, 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:15972853

[6] E. Stepanov and K. Serebryany, "Memorysanitizer: Fast detector of uninitialized memory use in c++," 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:8119283

[7] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. USENIX Association, 2005, p. 2.

[8] F. Gorter, E. Barberis, R. Isemann, E. van der Kouwe, C. Giuffrida, and H. Bos, "Floatzone: Accelerating memory error detection using the floating point unit," in *USENIX Security Symposium*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:260777853

[9] K. Serebryany, C. Kennelly, and e. Mitch Phillips, "Gwp-asan: Sampling-based detection of memory-safety bugs in production," *arXiv*, 2024. [Online]. Available: https://arxiv.org/pdf/2311.09394

[10] Y. Jeon, W. Han, N. Burow, and M. Payer, "Fuzzan: Efficient sanitizer metadata design for fuzzing," in *USENIX Annual Technical Conference*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:219708043

[11] N. Hasabnis, A. Misra, and R. C. Sekar, "Light-weight bounds checking," in *IEEE/ACM International Symposium on Code Generation and Optimization*, 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:16496367

[12] G. J. Duck, R. H. C. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers," in *Network and Distributed System Security Symposium*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:4960605

[13] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: architectural support for spatial safety of the c programming language," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008. [Online]. Available: https://api.semanticscholar.org/CorpusID:8254904

[14] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *USENIX Security Symposium*, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:14320211

[15] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder, "High system-code security with low overhead," *2015 IEEE Symposium on Security and Privacy*, pp. 866–879, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:291402

[16] Y. Zhang, C. Pang, G. Portokalidis, N. Triandopoulos, and J. Xu, "Debloating address sanitizer," in *USENIX Security Symposium*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:252972132

[17] J. Zhang, S. Wang, M. Rigger, P. He, and Z. Su, "Sanrazor: Reducing redundant sanitizer checks in c/c++ programs," in *USENIX Symposium on Operating Systems Design and Implementation*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:236992631

[18] H. Ling, H. Huang, C. Wang, Y. Cai, and C. Zhang, "Giantsan: Efficient memory sanitization with segment folding," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Association for Computing Machinery, 2024, p. 433–449.

[19] Z. Lin, Z. Yu, Z. Guo, S. Campanoni, P. Dinda, and X. Xing, "Camp: Compiler and allocator-based heap memory protection," 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:265469289

[20] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory tagging and how it improves c/c++ memory safety," *ArXiv*, vol. abs/1802.09517, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:3544711

[21] "Software assurance reference dataset," *Online Blog*, 2017. [Online]. Available: https://samate.nist.gov/SARD/test-suites.

[22] "Memory tagging extension: Enhancing memory safety through architecture," *Online Blog*, 2019 (accessed July 12, 2024). [Online]. Available: https://developer.arm.com//media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[23] X. Chen, Y. Shi, Z. Jiang, Y. Li, R. Wang, H. Duan, H. Wang, and C. Zhang, "Mtsan: A feasible and practical memory sanitizer for fuzzing cots binaries," in *USENIX Security Symposium*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:260777798

[24] "38.7 memory related tunables," *Online Blog*, Accessed Aug 12, 2024). [Online]. Available: https://www.gnu.org/software/libc/manual/htmlnode/Memory-Related-Tunables.html.

[25] "Spec cpu2006 documentation," *Online Blog*, 2011. [Online]. Available: https://www.spec.org/cpu2006/Docs/

[26] Y.-F. Li, "Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication," *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:252167812

[27] Z. Yu, G. Yang, and X. Xing, "ShadowBound: Efficient memory protection through advanced metadata management and customized compiler optimization," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[28] pcc, "scudo: Add initial memory tagging support," *Online Blog*, 2019. [Online]. Available: https://reviews.llvm.org/D70762

[29] "Standard performance evaluation corporation," *Online Blog*, 2022. [Online]. Available: https://www.spec.org/cpu2017/,

[30] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *ACM SIGMETRICS Performance Evaluation Review*, vol. 49, pp. 81 – 82, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:221137226

[31] "Linux flaw project," *Online Blog*, 2017. [Online]. Available: https://github.com/mudongliang/LinuxFlaw.

[32] K. Hohentanner, P. Zieris, and J. Horsch, "Cryptsan: Leveraging arm pointer authentication for memory safety in c/c++," *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:258615821

[33] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "Cets: compiler enforced temporal safety for c," in *International Symposium on Mathematical Morphology and Its Application to Signal and Image Processing*, 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:914358

[34] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *WBIA '09*, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:4132940

[35] G. J. Duck and R. H. C. Yap, "Effectivesan: type and memory error detection using dynamically typed c/c++," *ACM SIGPLAN Notices*, vol. 53, pp. 181 – 195, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:4918751

[36] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, and A. DeHon, "Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:14648609

[37] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, "Sgxbounds: Memory safety for shielded execution," *Proceedings of the Twelfth European Conference on Computer Systems*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:17268329

[38] N. Burow, D. P. McKee, S. A. Carr, and M. Payer, "Cup: Comprehensive user-space protection for c/c++," *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:4655395

[39] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "Softbound: highly compatible and complete spatial memory safety for c," in *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:248719

[40] A. U. S. Gopal, R. Soori, M. Ferdman, and D. Lee, "Tailcheck: A lightweight heap overflow detection mechanism with page protection and tagged pointers," in *USENIX Symposium on Operating Systems Design and Implementation*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:259257573

[41] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida, "Delta pointers: buffer overflow checks without the checks," *Proceedings of the Thirteenth EuroSys Conference*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:4938031

[42] F. Gorter, T. Kroes, H. Bos, and C. Giuffrida, "Sticky tags: Efficient and deterministic spatial memory error mitigation using persistent memory tags," in *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2024, pp. 217–217.

[43] H. Liljestrand, C. C. Perez, R. Denis-Courmont, J.-E. Ekberg, and N. Asokan, "Color my world: Deterministic tagging for memory safety," *ArXiv*, vol. abs/2204.03781, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:248069169

[44] "Memtagsanitizer," *Online Blog*, 2024. [Online]. Available: https://llvm.org/docs/MemTagSanitizer.html