GiantSan: Efficient Memory Sanitization with Segment Folding

Hao Ling
The Hong Kong University of Science and Technology, China
hlingab@cse.ust.hk

Heqing Huang*
City University of Hong Kong, China
heqhuang@cityu.edu.hk

Chengpeng Wang
The Hong Kong University of Science and Technology, China
cwangch@cse.ust.hk

Yuandao Cai
The Hong Kong University of Science and Technology, China
ycaibb@cse.ust.hk

Charles Zhang
The Hong Kong University of Science and Technology, China
charlesz@cse.ust.hk

Abstract

Memory safety sanitizers, the sharp weapon for detecting invalid memory operations during execution, employ runtime metadata to model the memory and help find memory errors hidden in the programs. However, location-based methods, the most widely deployed memory sanitization methods thanks to their high compatibility, face the low protection density issue: the number of bytes safeguarded by one metadata is limited. As a result, numerous memory accesses require loading excessive metadata, leading to a high runtime overhead.

To address this issue, we propose a new shadow encoding with *segment folding* to increase the protection density. Specifically, we characterize neighboring bytes with identical metadata by building novel summaries, called *folded segments*, on those bytes to reduce unnecessary metadata loadings. The new encoding uses less metadata to safeguard large memory regions, speeding up memory sanitization.

We implement our designed technique as GiantSan. Our evaluation using the SPEC CPU 2017 benchmark shows that GiantSan outperforms the state-of-the-art methods with 59.10% and 38.52% less runtime overhead than ASan and ASan--, respectively. Moreover, under the same redzone setting, GiantSan detects 463 fewer false negative cases than ASan and ASan-- in testing the real-world project PHP.

*Corresponding Author

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0385-0/24/04…$15.00
https://doi.org/10.1145/3620665.3640391

ACM Reference Format:
Hao Ling, Heqing Huang, Chengpeng Wang, Yuandao Cai, and Charles Zhang. 2024. GiantSan: Efficient Memory Sanitization with Segment Folding. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620665.3640391

1 Introduction

The freedom to manipulate memory through pointers guaranteed by unsafe languages like C and C++ leads to numerous kinds of memory safety violations. As reported in the 2022 CWE Top 25 Most Dangerous Software Weaknesses [40], for instance, out-of-bounds write, out-of-bounds read, and use-after-free rank 1st, 5th, and 7th among all weaknesses, respectively. For program reliability, researchers have proposed a series of memory sanitizing techniques [2, 9, 11, 12, 19, 23, 25, 29, 30, 34, 36, 41, 42] to detect invalid memory operations during the program execution.

Though tremendous efforts have been made to improve memory sanitization, most methods have limited compatibility, resulting in false negatives or low efficiency in many scenarios. Pointer-based methods, for instance, protect memory accesses with buffer bounds propagated along with pointer arithmetics. However, the propagation highly depends on program instrumentation with type information of pointers, which is not always available. It is a well-known issue [4, 9, 11, 23, 25, 29, 30, 35, 37, 39] that propagation often fails due to pointer-integer casting or uninstrumented external libraries without type information (e.g., third-party codes distributed in binary form). As a result, the pointer-based sanitizers cannot detect errors once the propagation fails.

Location-based methods stand out among the various memory sanitizers due to their high compatibility, which comes from a simpler safety model that does not rely on pointer information to maintain metadata. Specifically, each byte in the memory is assigned one of the two states, *addressable* or *non-addressable*, and a memory access is safe if the target bytes are all addressable. The addressability states are stored in a dedicated shadow memory and can be retrieved anytime, eliminating the need for instrumentation to propagate metadata. For compatibility considerations, memory sanitizers integrated into GCC [13], LLVM [24] compiler projects, and ANDROID [3] system are all location-based [26, 34, 35].

However, though location-based methods offer high compatibility and are fast in metadata maintenance [37], they are deficient in protecting memory operations [1] involving multiple instructions, and they require excessive runtime checks compared with other methods like pointer-based solutions. Specifically, pointer-based methods safeguard memory operations by checking whether the memory region being accessed is within a safe bound. In contrast, location-based methods do not have such a bound, and they have to break operations down into instructions and check each instruction separately to ensure no non-addressable bytes are accessed. Therefore, though location-based methods save time in metadata maintenance, they incur more runtime checks, which are time-consuming.

The root cause of the excessive check issue is the low *protection density* caused by the inefficient shadow memory encoding. The protection density is the number of bytes safeguarded by one piece of metadata. Each byte in the memory has two different states: addressable or non-addressable. Technically, it requires at least one bit to distinguish the two states. Therefore, on average, location-based methods must load and decode one shadow byte for every eight memory bytes. The protection density can be slightly increased according to memory alignment: some consecutive bytes must be both addressable or non-addressable, and thus their states can be merged. However, most objects are only guaranteed to be 8-byte aligned, and this optimization is limited to a few neighboring bytes.

Figure 1 illustrates the shadow encoding with the low protection density in the most widely deployed sanitizer, ADDRESSSANITIZER (a.k.a. ASan) [34]. It partitions the virtual memory space into a sequence of aligned segments and employs one 8-bit integer (called the *segment state* in this paper) to encode all byte states within the segment. Segments are sized at 8 bytes so that no two objects share the same segment [2]. Checking a memory region containing S bytes requires loading $\lceil \frac{S}{8} \rceil$ segment states, which results in significant runtime overhead. For instance, checking whether a 1KB region contains a non-addressable byte requires loading 128 segment states in ASan. A past study [42] shows that ASan is about 2× slower than native execution, and about 80% of the overhead comes from excessive runtime checks and metadata loadings.

[1]In this paper, a memory operation refers to a series of instructions manipulating the memory region of the same object. For example, "memset(p, 0, 1024)" is one memory operation manipulating 1024 bytes and consists of at least $1024/8 = 128$ MOV instructions related to p in a 64-bit system.

[2]ASan assumes all objects are 8-byte aligned, which is satisfied in most cases due to the basic assumption of heap allocation.

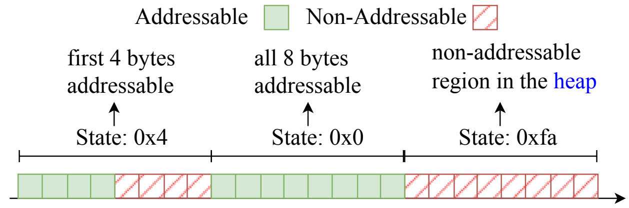
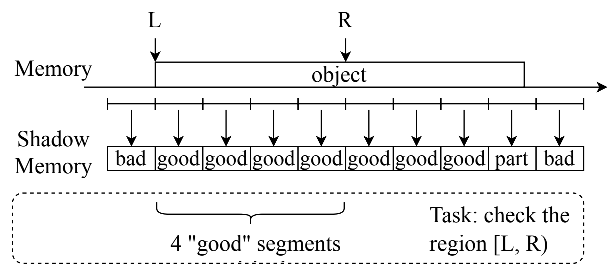
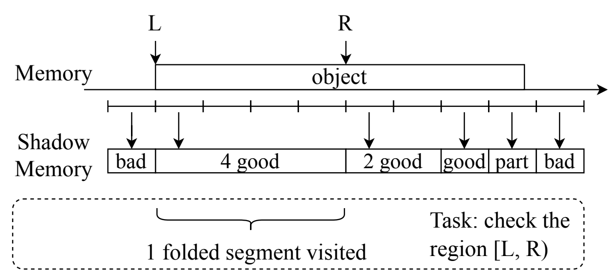


Figure 1. Shadow encoding in ASan. Objects are 8-byte aligned. The addressable bytes within a segment must occupy a prefix of the segment. By default, ASan uses eight different state codes for addressable bytes within the segments and reserves the other state codes for other purposes (e.g., recording why the bytes are non-addressable).



(a) Existing location-based encoding. "good": all bytes in the segment are addressable; "bad": all bytes are non-addressable; "part": only some bytes in the segment are addressable (partially good).



(b) Segment Folding: build a summary for "good" segments. Only one folded segment needs to be visited instead of four unfolded ones for the region $[L, R)$.

Figure 2. Folded segments reduce metadata loadings.

This paper addresses the low protection density issue to improve the efficiency of location-based memory sanitization. Despite the various segment states, almost all segments visited during the execution are "good" segments (i.e., the segments without non-addressable bytes) because most memory operations are safe and only manipulate addressable bytes. Inspired by this observation, our key insight is to build a summary for "good" segments to help reduce segment state loadings, thus increasing the sanitizing efficiency. We call the summarizing process "*segment folding*".

Let us illustrate our insight with Figure 2. Figure 2a shows how existing methods work: when accessing a memory region, they need to check all segments to ensure all accessed bytes are addressable. Checking the region $[L, R)$ involves 4 segments, and all those segments are "good" since this region is safe to access. Figure 2b shows how the segment folding works: it builds a summary of the "good" segments and uses the summary of segments to speed up the checking of the region $[L, R)$. However, the folding is not free: storing the summary needs extra shadow memory space.

To reduce the shadow memory required to store the summary, we design the binary folding strategy: a folded segment only summarizes 2^x "good" segments for some integer x. In a modern 64-bit system, x cannot exceed 64 because the maximum object size is less than 2^{64}. As a result, six shadow bits are sufficient to record the folding degree x. Combined with the 8-byte alignment optimization, all the segment states and the folding degree x can be recorded in one 8-bit integer. As a result, the new shadow memory encoding with segment folding is compact enough to build upon the shadow memory widely adopted by existing location-based methods.

We present GiantSan, a dynamic memory error detector with a novel shadow encoding based on segment folding. To the best of our knowledge, GiantSan is the first location-based method that can safeguard a sequential region of arbitrary size in $O(1)$ time. We evaluate GiantSan on SPEC CPU 2017, the industry-standard CPU-intensive benchmark suite. GiantSan reduces the geometric mean runtime overhead down to 46.04%, compared with 74.89% and 112.58% in the state-of-the-art location-based designs ASan-- [42] and ASan [34], respectively. The promising result indicates that GiantSan outperforms its competitors.

To sum up, this work makes the following contributions:

- We formulate and summarize the *low protection density* issue of location-based sanitizers.
- We introduce the segment folding algorithm to increase protection density significantly.
- We implement our approach as a tool named GiantSan and provide empirical evidence that it outperforms the state-of-the-art methods with less runtime overhead.

2 Technical Background

This section introduces fundamental knowledge about existing memory sanitizing techniques.

2.1 Existing Solutions for Memory Safety

There are two categories of memory safety violations: 1) **Spatial Errors**: access memory locations outside the allocated region of objects, and 2) **Temporal Errors**: access an object when it is not valid (e.g., unallocated or deallocated).

Although many memory safety violation detecting tools have been proposed [5–7, 9, 11, 12, 19, 21, 22, 25, 30, 33, 34, 36, 41, 42], many only provide partial memory safety guarantees. Some, like Softbound [29], Delta Pointers [22], TailCheck [15], and LFP [9, 11], only support the detection of spatial errors. In contrast, other trends of existing work, like CETS [30] and PTAuth [12], only support the detection of temporal errors.

All sanitizers need extra metadata to model the memory and validate whether one memory region can be accessed. Among the existing efforts to provide a full safety guarantee, there are two main philosophies:

- **Pointer-based**: Pointer-based methods [6, 9, 11, 12, 15, 21, 22, 25, 30, 33] model the memory from the perspective of pointers by tracking the memory region safe to access for each pointer. They encapsulate the pointer and a tag in a new pointer representation, and they use the tag as the bound for the safe region or as the index for retrieving the bound.
- **Location-based**: Location-based methods [5, 7, 19, 34, 36, 41, 42] model the memory from the perspective of memory bytes by recording which byte is addressable. The byte states are recorded in a compact shadow memory, and location-based methods inspect the shadow memory to check the state of each accessed byte.

The core difference between the two philosophies is *the dependence on the data type information of pointers*. Specifically, whenever pointer arithmetic creates a new pointer, pointer-based methods need to convert it into the new pointer representation and propagate the tag from the source pointer to the new pointer. Therefore, in pointer-based methods, all instructions must be aware of whether they are manipulating pointers so that the tag is propagated correctly and not misused. In contrast, memory protection in location-based methods only depends on the metadata binding to the memory address instead of pointers.

Unfortunately, the type information of pointers is not always available. For example, programs can use external libraries distributed in binary form without type information, and all values are treated as integers. Moreover, even with the source codes available, the type information of pointers may not be available since the pointer-integer casting can eliminate the type information. The casting converts pointers into integers, and later, pointers are manipulated by integer arithmetic instead of pointer arithmetic. As a result, it is challenging to distinguish between the customized pointer representation and the native integers, which might result in tag misuse or tag propagation failure [4, 9, 11, 23, 25, 29, 30, 35, 37, 39].

Once the pointer tag is lost due to propagation failure, the pointer-based methods cannot protect the pointer and all new pointers derived from it. Some efforts attempt [2, 9, 11, 21] to recover from the tag loss by obtaining a new tag based on the pointer values from dedicated data structures, e.g., shadow memory, similar to the location-based methods. However, location-based methods only require distinguishing two states of bytes with a compact shadow memory. In contrast, keeping tags to distinguish different objects requires a much larger shadow memory. Large shadow memory causes excessive memory consumption and significantly affects runtime efficiency due to a high memory footprint [34, 37].

One of the most representative efforts in tag reobtaining is the Baggy Bound Checking (BBC) [2]. To avoid large shadow memory footprints, it rounds allocation sizes up to a power of two to reduce the total variety of tags. As a result, it cannot detect errors within the rounded-up allocation size. For example, it cannot detect the out-of-bound access "*p[700]*" for a buffer "*char p[600]*" because the buffer is rounded up to "*char p[1024]*". Therefore, due to the tolerance of many spatial violations, BBC is less suitable for testing [2, 37].

Due to their high dependence on pointer type information, pointer-based methods are less compatible in the complicated real-world testing environment. In contrast, location-based methods are much more widely adopted because they only need to know which memory address is being accessed. That is why general-purpose compiler projects like LLVM and GCC only integrate location-based methods. However, location-based methods have their own efficiency issue, which we aim to address in this paper, discussed in the following.

2.2 Location-based checking with shadow memory

Shadow memory is a technique to monitor and maintain the states of bytes in the memory, widely used in memory safety sanitizers [5, 17, 34–36, 41, 42]. It is the most efficient data structure to implement location-based methods. Location-based methods partition the virtual memory space into fixed-sized segments and use shadow memory to record the segment state, which encodes the states of bytes within the segment. Specifically, shadow memory is an array of shadow units, each of which stores a piece of metadata. We use the notation m to represent the global array, N for the number of segments, and S_{shadow} for the size of each segment. The following is how shadow memory is declared:

```
ShadowUnitType m[N];
```

Given a memory address p, the state of the segment covering the address p can be loaded by:

```
m[(intptr_t)p/S_shadow]
```

Location-based methods can only detect whether a byte is addressable but cannot guarantee that the byte belongs to the desired object. Most existing location-based methods integrate *redzones* [19, 34, 36, 41, 42] and *memory quarantine* [1, 19, 34, 36, 41, 42] to detect sophisticated memory errors. Specifically, *redzones* are non-addressable paddings between objects (for spatial error detection), and *memory quarantine* delays the re-allocation of memory regions to ensure that an object's memory region is not addressable during a particular time (for temporal error detection).

Runtime Checks. Before accessing w bytes starting from an address p, location-based methods safeguard the memory access by checking whether all w target bytes are addressable. The metadata indicating the addressability of bytes comes from the shadow memory. The metadata only has a limited bit width (e.g., 8 bits) to enable compact shadow memory and can not hold much information. As a result, w is small in existing location-based methods so that the byte states can be encoded with a limited bit width.

Example 1. *ASan [34] uses $S_{shadow} = 8$, and 8-bit signed integers as the ShadowUnitType. $m[p] = 0$ means the p-th segment is a "good" segment (i.e., all bytes in this segment are addressable), and $m[p] = k$ $(1 \leq k \leq 7)$ means the p-th segment is a k-partial segment (i.e., only the first k bytes in this segment are addressable). ASan creates one runtime check for all memory accesses with $w \leq 8$:*

```
int8_t v = m[p / 8];
if (v != 0 and (p & 7) + w > v) {
  ReportError(p, w)
}
access [p, p + w)
```

The maximum allowable value of w determines the protection density: larger w means more bytes can be safeguarded by the metadata, thus resulting in fewer metadata loadings and runtime checks. However, for memory efficiency, location-based methods need to use compact shadow memory, which cannot allocate a large bit width for a piece of metadata, and inefficient shadow encoding can only employ small w and limits the protection density.

2.3 Problems and Challenges

In this section, we demonstrate how protection density affects sanitizing efficiency by presenting two protection principles used in different sanitizers: 1) *operation-level protection* aims to protect a memory operation consisting of multiple instructions as a whole, and 2) *instruction-level protection* safeguards each instruction separately. We discuss why operation-level protection requires a high protection density and generates fewer runtime checks. We also discuss the challenges in enabling operation-level protection in location-based methods.

A memory operation is a series of memory accesses toward the allocated region of one single object. Table 1 shows four types of commonly used runtime checks based on the semantics of memory operations, all associated with the pointer p. For example, constant propagation can tell that $p[0], p[10]$, and $p[20]$ are all memory accesses towards p with constant offsets. Operation-level protection safeguards all three instructions at once by testing $[\&p[0], \&p[21]) \subseteq \text{bound}(p)$. Similarly, the memset and bounded loop require only one check under operation-level protection. In contrast, the instruction-level protection checks all instructions

Table 1. Difference between operation-level protection and instruction-level protection on the pointer p. The *Analysis Method* column shows the static analysis used to identify the operations in the source codes. N in the fourth case is the size of vec.

Analysis Method	Example	# Checks (operation-level)	# Checks (instruction-level)
Constant Propagation	p[0] + p[10] + p[20]	1	3
Predefined Semantics	memset(p, 0, N)	1	$\Theta(N)$
Loop Bound Analysis	for (auto i = 0; i < N; i++) p[i] = foo(i);	1	N

Must-alias Analysis	p[0] = 10
	for (auto i : vec)
	p[i] = foo(i);
		1 slow check + N fast checks (with bound cached)	N+1 slow checks (with nothing cached)

executed separately. For example, $p[0]$, $p[10]$, and $p[20]$ involve three instructions, and the instruction-level protection checks each of them separately.

The operation-level protection requires much fewer checks than the instruction-level protection. However, it needs to efficiently check memory regions of arbitrary sizes, which, unfortunately, is not available in existing location-based methods, as discussed in Section 2.2. Therefore, existing location-based methods utilize instruction-level protection.

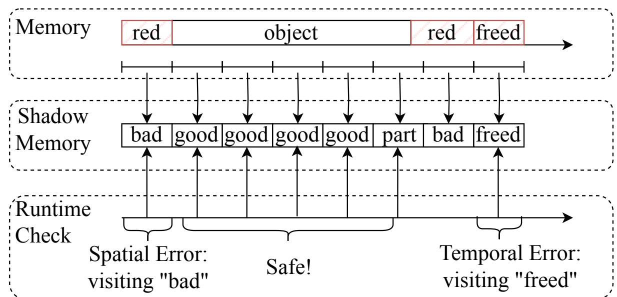
Moreover, the operation-level protection can also reduce metadata loadings with caching. The operation-level protection can cache the bound of p for future memory accesses on p, as listed in the fourth case of Table 1. Once the bound of p is loaded when checking $p[0] = 10$, the bound can be cached in a local variable and used to check all instructions in the loop. In contrast, the instruction-level protection separately checks each instruction, and the metadata loaded can only safeguard the corresponding instruction. Caching metadata with low protection density cannot help speed up future checks because it does not contain much information.

Summary. Existing location-based methods have the following deficiencies of the instruction-level protection, all caused by the low protection density. We attempt to address the deficiencies by increasing protection density.

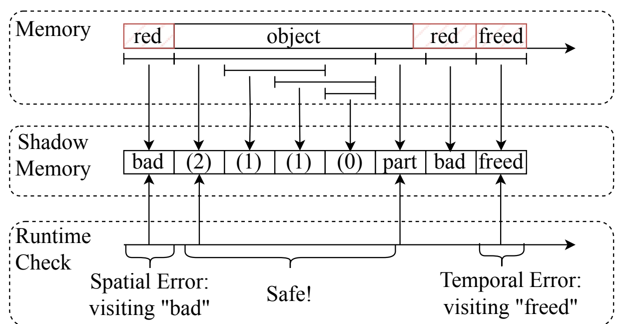
- Inefficient in safeguarding large memory regions.
- Inefficient in caching history.

3 GiantSan in a Nutshell

We present GiantSan, a novel location-based sanitizer enabling operation-level protection. Our main observation is that most segments being visited during execution are "good" segments, so characterizing and protecting good-segment-only memory regions with a customized summary suffice in most cases. For example, in Figure 3a, the "Safe!" region requires loading 5 segment states. In contrast, in Figure 3b, GiantSan combines nearby "good" segments to avoid visiting "good" segments repeatedly and conducts only 2 checks. Figure 4 demonstrates two key phases of GiantSan:



(a) Shadow memory technique. "good" means all bytes in the segment are addressable; "part" means partially good - only the first several bytes in the segment are addressable; "bad" and "freed" represent non-addressable segments maintained by redzones and memory quarantine.



(b) "(i)" indicates that this segment is a folded segment combining two consecutive folded segments with the folding degree "$(i-1)$". In particular, "(0)" indicates "good" segments. A segment with code "(i)" summarizes 2^i consecutive "good" segments.

Figure 3. High-level comparison between GiantSan and existing approaches: the majority of consecutive segments can be folded and checked as a whole.

- **The runtime support library** hooks all objects' allocation and deallocation to initialize the metadata in shadow memory (Section 4.1) during the execution.
- **The instrumentation system** inserts checks to protect memory operations. Operation-level protection

(Section 4.4) requires different instrumentation logic for consecutive region checks (Section 4.2) and history caching (Section 4.3).

The runtime support library sets the metadata in the shadow memory. Specifically, to implement the runtime support library, we first need to design metadata modeling the memory by answering the following question:
Question 1: How to fold segments and encode the folded segments in the shadow memory?
Solution: GiantSan employs the recursive binary folding strategy: two consecutive "good" segments, or two consecutive folded segments with the same size, are combined to form a new folded segment. As illustrated in Figure 3b, the (1)-folded segment combines two "good" segments, and the (2)-folded segment combines two (1)-folded segments. The folded segments summarize addressable regions, speeding up the segment checks, and only the folding degree (i) needs to be recorded. We discuss the details in Section 4.1.

GiantSan utilizes the optimized shadow memory to safeguard memory regions. To solve the deficiencies discussed in Section 2.3, we face two main questions:
Question 2: How to efficiently safeguard given memory regions with arbitrary sizes?
Solution: Safeguarding a memory region is simplified into checking whether the folding degree is large enough. More specifically, if we want to check whether N consecutive segments contain non-addressable bytes, we can check whether the first and last $2^{\lfloor \log_2 N \rfloor}$ segments are folded, significantly reducing the required metadata. We place the details of locating the folded segments in Section 4.2.
Question 3: How to build a cache to speed up further checks?
Solution: GiantSan caches the *last* folded segment visited for a given pointer, which can be considered as a temporary bound for all accesses checked. The bound helps reduce the metadata loadings for future accesses on the same pointer. We discuss the caching algorithm in Section 4.3.

4 Design

In this section, we present the design of GiantSan, an efficient location-based sanitizer with high protection density. Like existing location-based methods, GiantSan needs redzones and memory quarantine for sophisticated errors.

Figure 4 illustrates GiantSan's general workflow. The runtime support library hooks the object's allocation to update the shadow memory, and the instrumentation uses the shadow memory to safeguard memory regions. Sections 4.1, 4.2, and 4.3 present detailed solutions to the three questions mentioned in Section 3. Section 4.4 describes how to generate operation-level checks with static analysis. In the end, Section 4.5 demonstrates the implementation details.

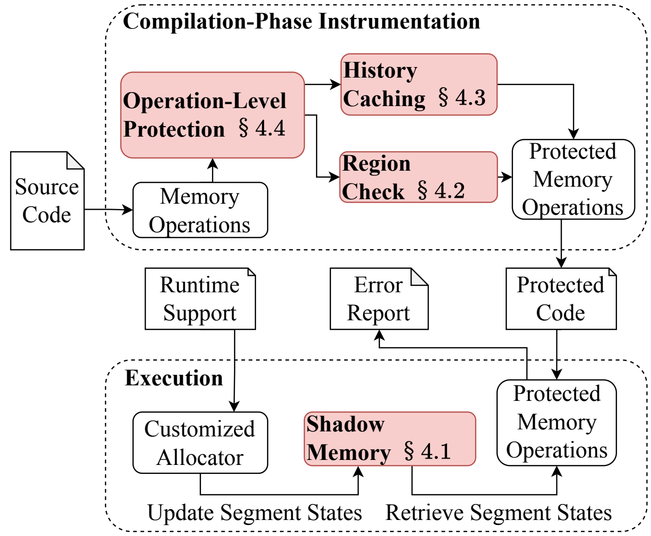


Figure 4. GiantSan's workflow

4.1 Shadow Encoding in GiantSan

In this section, we describe GiantSan's shadow memory encoding. We choose the commonly used eight-byte segment shadow memory as ASan [34]. The whole virtual memory is divided into small segments of 8 bytes, and the metadata for a segment is stored in an 8-bit data type. Same as ASan, GiantSan ensures that all objects are 8-byte aligned, which does not make a huge difference to the memory layout because, as discussed in previous work [34], most objects in modern systems are naturally 8-byte aligned.

GiantSan achieves high protection density by building summaries on "good" segments, the ones containing no non-addressable bytes. The summary strategy is *binary folding*, which locates and folds consecutive 2^x "good" segments and encodes the value x in the shadow memory. The folded segment containing 2^x "good" segments is named as an (x)-*folded segment*. As illustrated in Figure 5, an x value in the shadow memory indicates **at least** 8×2^x and **less than** $8 \times 2^{x+1}$ consecutive bytes are addressable. In modern 64-bit systems, x cannot exceed 64 because the maximum object size is less than 2^{64}.

After introducing the folded segments, three categories of segment states exist: 1) the folding degree i for (i)-folded segments, 2) the value k for k-partial segments, which has only the first k bytes addressable, and 3) error codes for non-addressable segments. There are at most 64 different i and 7 different k. We use the denotation $m[p]$ to represent the metadata stored in the p-th shadow byte, and $m[p]$ is defined as follows:

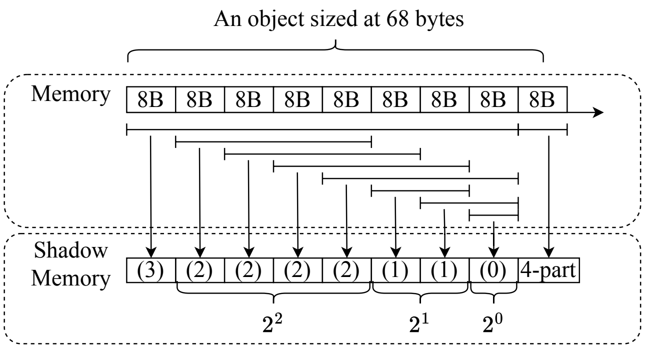


Figure 5. Shadow memory encoding for an object sized 68 bytes. "(i)" represents an (i)-folded segment. "4-part" represents a partial segment with only the first 4 bytes addressable.

Definition 1 (State Code)**.** *$m[p]$ is an 8-bit unsigned integer that can store values within $[0, 256)$.*

$$m[p] = \begin{cases} 64 - i, & \textit{the p-th segment is an (i)-folded segment} \\ 72 - k, & \textit{the p-th segment is a k-partial segment} \\ > 72, & \textit{error codes} \end{cases}$$

The monotonicity of m simplifies memory checks. A smaller $m[p]$ means more consecutive addressable bytes following the p-th segments. Suppose that we want to check whether the p-th segment is a folded segment with a folding degree equal to or higher than 3. In that case, we only need to check whether $m[p] \le 64 - 3$. Any $m[p]$ breaking the inequality indicates that there are non-addressable bytes in the memory region $[8p, 8(p+2^3))$. Checking the folding degree is the key to memory protection, which is discussed later in Section 4.2.

Though the encoding is much more complicated than existing works [2, 9, 11, 31, 34, 36, 41, 42], updating the shadow memory with the new encoding does not take extra computation. Technically, an allocated object has at most one partial segment, and all remaining segments within the allocated regions are folded. More formally, there are 2^i consecutive (i)-folded segments, e.g., there is one (0)-folded segment, two (1)-folded segments, and four (2)-folded segments. The relative positions of the folded segments follow a simple pattern illustrated in Figure 5. Based on this pattern, GiantSan efficiently updates the shadow memory in linear time, the same as existing works.

4.2 Region Checking

This section introduces how to use the new shadow memory encoding to safeguard a memory region. A memory region $[L, R)$ is safe if all except the last segment within this region are "good" segments and the first $(R \bmod 8)$ bytes in the last segment are addressable. GiantSan speeds up the "good" segment checking with folded segments. Specifically, GiantSan generates codes to safeguard a memory region $[L, R)$, denoted as CI(L, R), in two steps. Let $l = \lfloor \frac{L}{8} \rfloor$, $r = \lfloor \frac{R}{8} \rfloor$:

Algorithm 1 CI(L, R). m is the shadow memory, and L is a multiple of 8 due to the 8-byte-alignment strategy.

1: uint8_t $v = m[\frac{L}{8}]$ ▷ $L \equiv 0(\text{mod} 8)$
2: uintptr_t $u = (v \le 64) \ll (67 - v)$;
3: **if** $u < R - L$ **then** ▷ fast check
4: **if** $R - L \ge 8$ **then**
5: **if** $2 * u < R - L$ **then** ▷ check folding degree
6: ReportError() ▷ of the prefix
7: **end if**
8: **if** $m[\lfloor \frac{R-u}{8} \rfloor] \ne v$ **then** ▷ check folding degree
9: ReportError() ▷ of the suffix
10: **end if**
11: **end if**
12: **if** $m[\lfloor \frac{R-1}{8} \rfloor] > 72 - (R\&7)$ **then** ▷ check the partial
13: ReportError() ▷ segment at the
14: **end if** ▷ end
15: **end if**

- The l-th, $\cdots$, $(r-1)$-th segments must all be "good".
- The first $(R \bmod 8)$ bytes in the r-th segment are addressable.

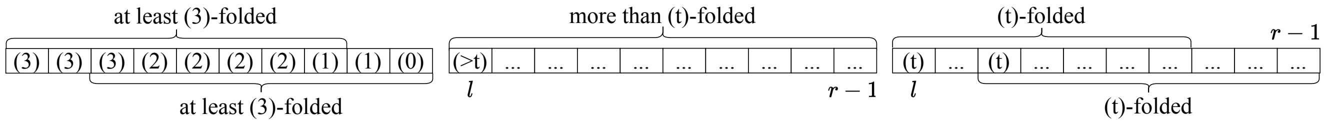
Arbitrary N consecutive "good" segments must be a union of two $(\lfloor \log_2 N \rfloor)$-folded segments. As illustrated in Figure 6a, if all 10 consecutive segments are "good", the first eight and the last eight "good" segments must be at least (3)-folded. Therefore, we only need to check if the folding degrees of a *prefix* and a *suffix* in the segment sequence are large enough. There are only two cases when all segments numbered from l to $r-1$ are "good" ($t = \lfloor \log_2 r - l \rfloor$):

- All segments are folded into one, and at least one $(t+1)$-folded segment exists, as illustrated in Figure 6b.
- All segments are divided into two (t)-folded segments, as illustrated in Figure 6c.

An important integer trick for efficient checking is that the number of addressable bytes recorded in the p-th segment is $(m[p] \le 64) \ll (67 - m[p])$, where $\ll$ is the left-shift arithmetic. The calculation result becomes 0 if $m[p]$ does not represent a folded segment (i.e., $m[p] > 64$). The trick helps avoid calculating the expensive $\log_2$ function.

Algorithm 1 shows how to safeguard the interval $[L, R)$. It contains two stages: the *fast check* (the case in Figure 6b) and the *slow check* (the case in Figure 6c). The fast check is cheap and suffices to safeguard most memory regions, while the slow check handles the remaining rare cases.

- The fast check (Lines 1~3) finds a safe region $[L, L+u)$ without non-addressable bytes based on the folded segment recorded at $m[\frac{L}{8}]$. If $[L, R)$ is within $[L, L+u)$, $[L, R)$ must be safe. According to the definition of the folded segment, u covers $>$ 50% of the addressable bytes following L; thus, u is large enough to safeguard the majority of the regions.



(a) Checking 10 segments **(b)** Case 1: one folded segment **(c)** Case 2: two folded segments

Figure 6. Checking whether the l-th, $l+1$-th, $\cdots$, $(r-1)$-th segments are all "good" based on folded segments. $t = \lfloor \log_2(r-l) \rfloor$.

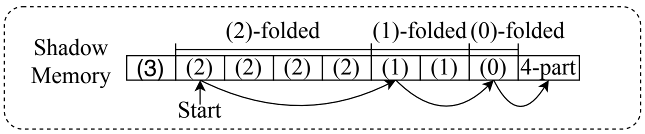


Figure 7. Locating the bound with folded segments

- The slow check (Lines 4~14) [3] verifies three parts: whether non-addressable bytes exist in 1) the prefix, 2) the suffix, and 3) the last segment of $[L, R)$. The slow check handles the case illustrated in Figure 6c, which is much more infrequent than the cases handled by the fast check.

This algorithm fully utilizes folded segments: folded segments summarize the majority (> 50%) of neighboring bytes in arbitrary safe regions, and the fast check efficiently safeguards any region within an existing summary. The region outside the fast check's scope is split into (at most) two folded segments and handled by the slow check, which is invoked only when the fast check fails. The slow check is also an O(1)-time algorithm with a better time complexity than existing location-based methods. Therefore, this algorithm can check a region with arbitrary size in constant time.

4.3 History Caching

History caching helps reduce metadata loadings on the same pointer. Intuitively, caching mainly speeds up memory protection within loops (the number of accesses outside loops is relatively limited). Thus, to better illustrate our method, we explain GiantSan's caching solution with accesses in loops.

The ideal values to be cached are the bounds of pointers since memory accesses falling within the bound do not need extra metadata. GiantSan can locate the bound by skipping over folded segments, as illustrated in Figure 7. The number of skipping is at most $\lceil \log_2 \frac{n}{8} \rceil$, where n is the size of the object, because the folding degree decreases by at least 1 after one skip and the maximum folding degree is $\lceil \log_2 \frac{n}{8} \rceil$.

Although the skipping is fast, it still takes time and is not a constant-time process. Therefore, GiantSan employs *on-demand skipping* to save time. Whenever GiantSan conducts a pointer dereference check, GiantSan caches the maximum valid address (called the *quasi-bound*) implied by the folded segment examined. In future dereference, the bound checks can use the quasi-bound until the dereference goes beyond the quasi-bound. GiantSan gets a new maximum valid address from the new folded segment visited. GiantSan reduces metadata loadings with the quasi-bound.

Figure 9 demonstrates caching logic for the memory access at Line 10 in Figure 8a. GiantSan creates a local variable, *ub*, as the quasi-bound for the buffer y. As illustrated in Figure 9, initially, the quasi-bound equals 0 because the size of the buffer is unknown. During the execution of the loop, GiantSan checks whether the offset j is beyond the quasi-bound (Line 4). If it goes beyond the bound, GiantSan checks $y[j]$ individually (Line 5) and updates *ub* (Line 7). After the quasi-bound update, *ub* is closer to the actual bound of the region, and as discussed above, the number of *ub*'s updating is at most $\lceil \log_2 \frac{n}{8} \rceil$. Further memory accesses on y that fall within the quasi-bound do not need additional metadata loadings and speed up the runtime checks.

GiantSan also detects underflow (Lines 9-11) and temporal errors (Line 14). Technically, GiantSan does not create a quasi-lower bound because it is widely reported [22, 27] that the number of accesses with negative offsets is far less frequent than positive offsets. Therefore, using a dedicated CI to check underflow results in negligible cost. Moreover, the object pointed by y can be freed during the loop execution, and a final check after the loop can capture the deallocation [42].

4.4 Check Instance Generation

This section describes enabling operation-level protection to reduce runtime overhead in GiantSan. We mainly discuss two categories of check instances supported by GiantSan, which improve the efficiency of location-based methods.

4.4.1 Anchor-based Enhancement. Location-based methods insert *redzones* between objects to detect overflow. However, small redzones can be bypassed [17], while large redzones negatively impact memory performance. Our solution is to set a small redzone between objects and select an anchor point. When safeguarding memory accesses, GiantSan checks whether a redzone exists between the anchor point and the accessed location. For most memory accesses, the

[3] Codes at Lines 4, 12-14 are unnecessary if $(R-L) \bmod 8 = 0$ can be proved with static type information, e.g., reading an array of *int64_t*.

```
1  void foo(int **p, int N) {
2
3    int *x = p[0];
4
5    int *y = p[1];
6    for (int i = 0; i < N; i++) {
7
8      int j = x[i];
9
10     y[j] = i;
11   }
12
13   memset(x, 0, N * sizeof(int));
14 }
```

(a) Source Code

```
1  void foo(int **p, int N) {
2    CI(p, p + 4);
3    int *x = p[0];
4    CI(p, p + 8);
5    int *y = p[1];
6    for (int i = 0; i < N; i++) {
7      CI(x, x + 4 * i + 4);
8      int j = x[i];
9      CI(y, y + 4 * j + 4);
10     y[j] = i;
11   }
12   CI(x, x + 4 * N);
13   memset(x, 0, N * sizeof(int));
14 }
```

(b) Check Instances (before merging)

```
1  void foo(int **p, int N) {
2    CI(p, p + 8);
3    int *x = p[0];
4    int *y = p[1];
5    CI(x, x + 4 * N);
6    for (int i = 0; i < N; i++) {
7
8      int j = x[i];
9      CI(y, y + 4 * j + 4) (cached);
10     y[j] = i;
11   }
12
13   memset(x, 0, N * sizeof(int));
14 }
```

(c) Check Instances (after merging and caching)

Figure 8. Operation-level protection instrumentation that significantly reduces runtime checks and metadata loadings

```
uintptr_t ub = 0;
for (int i = 0; i < N; i++) {
  int j = x[i];
  if (4 * j >= ub)) {
    CI(y, y + 4 * j + 4);
    v = m[(y + 4 * j) >> 3];
    ub = 4 * j + (v <= 64) << (67 - v);
  }
  if (j < 0) {
    CI(y + 4 * j, y);
  }
  y[j] = i;
}
CI(y, y + ub);
```

Figure 9. Quasi-bound instrumentation for *y[j]* in Figure 8a (Line 10) to reduce metadata loadings with caching.

base pointer of a buffer is chosen as the anchor point [4]. This optimization eliminates the trade-offs on redzone sizes and protects memory efficiently and precisely.

Take the memory access *y[j]* at Line 10 in Figure 8a as an example. Existing location-based sanitizers only check the region $[y{+}4j, y{+}4j{+}4)$ because they only protect the memory region at the instruction level. It can result in a false negative if j is large enough to bypass the redzone within $[y, y+4j)$ (if it exists). Existing methods have to enlarge the redzone size to avoid this false negative. Instead, GiantSan uses the base pointer y as the anchor point and checks the region $[y, y+4j+4)$ to ensure *y[j]* is indeed a valid location within the same memory region as y. This method only requires a one-byte redzone, thus eliminating the need to use large redzones and significantly increasing runtime efficiency.

[4]Some programmers would purposely employ undefined behaviors, e.g., using an out-of-bound base pointer to simulate 1-based arrays, which we consider as bugs by default. GiantSan can use the first dereferenced address as the anchor point to turn off the warning for the undefined behaviors.

4.4.2 Operation-level Checks. Due to the capability to handle arbitrary memory regions and history caching, GiantSan uses operation-level protection, which can significantly reduce the number of runtime checks. During compilation, GiantSan first scans all instructions and intrinsic functions that manipulate the memory to generate the instruction-level checks. Later, it uses static analysis to merge and eliminate unnecessary checks to increase efficiency.

For example, there are five different codes accessing memory in Figure 8a. Figure 8b shows the checks generated in the first stage (all array accesses are anchor-based enhanced). GiantSan later merges checks with static analysis; the final result is shown in Figure 8c. After the merging, only 2 checks and N cached checks are required, much fewer than the $2+3N$ checks in existing location-based methods. We discuss the static analysis for check merging in the following.

Aliased Check Elimination. Existing efforts [9, 11, 25, 34, 42] demonstrate that sanitization tasks could be removed or merged (e.g., $p[0]$ and $p[1]$ in Figure 8a) to reduce the number of memory region safeguarding requests if the accessed pointers are must-aliased. GiantSan adopts the LLVM's intra-procedural must-alias analysis to detect aliased checks.

Check-in-Loop Promotion. Memory accesses in loops can raise multiple checks during the execution (e.g., Line 7 and Line 9 in Figure 8b). GiantSan runs SCEV analysis [28] to identify bounded loops and reduce runtime checks. For example, the N checks at Line 7 in Figure 8b are combined into one check $CI(x, x+4N)$. For unbounded loops, GiantSan employs the history caching discussed in Section 4.3.

4.5 Implementation

GiantSan is built upon the infrastructure of ASan [34] in the LLVM Project. There are two components in the LLVM project related to memory sanitization: 1) a compilation pass that inserts runtime checks and 2) a library providing the runtime environment. Specifically, GiantSan modifies the framework in two aspects: the shadow memory poisoning to build folded segments and the detection logic to construct operation-level protection.

Shadow Poisoning. GiantSan changes the way ASan poisons the shadow memory to build the folded segment summary. Specifically, instead of only marking the allocated region addressable (e.g., filling the shadow memory with zero values), GiantSan sets the folding degrees in the shadow locations of the allocated region. The other operations, e.g., redzone setting and memory unpoisoning, remain unchanged. The instrumentation is implemented on top of the ASan instrumentation pass. The compilation front end controls the location of the pass in the compilation pipeline. By default, this pass is placed at the end of the optimization pipeline.

Runtime Checking. GiantSan changes the logic of runtime checks, prompting the instruction-level protection to the operation-level protection. ASan adds runtime protection in two ways. First, ASan employs an instrumentation pass to add runtime checks during the compilation; we modify this pass to replace ASan's runtime protection with GiantSan's operation-level protection. Second, ASan provides a runtime guardian function invoked before calling standard functions (e.g., strcpy). The guardian function checks contiguous regions in linear time, and we modify its implementation into GiantSan's constant time check.

Other implementation aspects of GiantSan, including shadow memory construction, shadow memory unpoisoning after object deallocation, redzone padding, and memory quarantine, are the same as the ones of ASan. Notably, the multi-thread guarantee of GiantSan is the same as ASan, i.e., thread-local caches are utilized to avoid locking on every call of the *malloc* and *free* functions.

5 Evaluation

We experimentally evaluate GiantSan on three questions:

- **RQ1: Can GiantSan reduce runtime overhead?**
- **RQ2: What are the impacts of each optimization?**
- **RQ3: Can GiantSan effectively detect real bugs?**

We evaluate the speed of GiantSan on the latest version of the industry-standard benchmark suite, SPEC CPU 2017 [38] (**RQ 1**), and conduct an ablation study to evaluate the impact of different optimizations employed by GiantSan with the same benchmark (**RQ 2**). We then use Juliet Test Suite [32], Magma Benchmark [20], and the Linux Flaw Project [8], the widely used vulnerability databases, to evaluate GiantSan's detection ability (**RQ 3**).

Configuration. GiantSan is built on the LLVM-12, and the experiments are conducted on a workstation with Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz CPU, 128G memory (OS: ubuntu 18.04, Kernel version: 4.15.0-117-generic).

As for the sanitizer configuration, we use the default settings listed in the ASan documentation [14] for all ASan-based implementations: ASan [34], ASan-- [34], and our tool GiantSan, except setting *halt_on_error=false* to prevent early termination of the evaluation due to the widely-reported memory errors existing in the SPEC benchmark.

5.1 Performance Study

Setting. We use the latest version of the industry-standard CPU-intensive benchmark suite, SPEC CPU 2017 [38], to evaluate the performance improvement of GiantSan thoroughly. This benchmark consists of two testing modes: speed test and rate test. The speed test runs one copy of the target program to evaluate the execution time under the intensive CPU computation environment. The rate test runs multiple concurrent programs simultaneously to evaluate the throughput and performance in multi-threaded environments.

Not all programs in the benchmark are selected due to compilation issues (e.g., requiring Fortran instead of C/C++). We test projects on which at least one sanitizer can work and choose the *ref* workloads for all projects.

We choose ASan [34] (the most widely adopted location-based sanitizers) and ASan-- [42] (the state-of-the-art redundant check eliminating solution based on static analysis) as the baseline of location-based methods. We plan to use BBC [2] as the baseline of rounded-up allocation size methods, but it is not publicly available. Instead, we choose LFP [9, 11], an improved version of BBC with more variety of allocation sizes for object allocation.

Results. The overall performance is shown in Table 2. LFP fails to build four projects *perlbench*, *gcc*, *parest*, and *imagick*. On average, GiantSan introduces 46.04% execution overhead on the native execution, with 59.10%, 38.52%, and 25.45% improvements over ASan, ASan--, and LFP, respectively. GiantSan outperforms ASan and ASan-- on all projects and is only slower than LFP on 5 out of the 24 projects. The result shows GiantSan has the best average performance, indicating the effectiveness of the new shadow encoding with the segment folding algorithm.

5.2 Ablation Study

This section breaks down the contributions of the two optimizations introduced in Section 4.2 and Section 4.3: large region checks help eliminate unnecessary checks, and history caching reduces unnecessary metadata loading.

Figure 10 demonstrates the ratio of optimized check codes in GiantSan by our optimizations. On average, 52.56% of the checks are optimized (30.76% eliminated and 21.80% cached). In the projects *mcf*, *namd*, and *lbm*, more than 80% of the checks introduced by ASan are eliminated or cached. Most of the checks in these projects are within simple loops and structure accesses with constant offsets, which our optimizations can efficiently handle. The remaining unoptimized codes include the ones that employ the fast check only and those that require the full check (i.e., fast check + slow check). GiantSan can remove some slow checks because memory regions of specific constant sizes (e.g., a power of 2) do not

Table 2. Runtime Overhead (seconds). *R* is the ratio compared to the native execution (RE: Runtime Error, CE: Compile Error). *CacheOnly* is the GiantSan version with history caching optimization only, and *EliminationOnly* is the one with check elimination only. The redzone sizes for location-based methods (GiantSan, ASan, and ASan--) are the default value (16 bytes).

	Performance Study									Ablation Study	
Programs	Native	GiantSan	*R*	ASan	*R*	ASan--	*R*	LFP	*R*	CacheOnly	EliminationOnly
500.perlbench_r	358	718	200.56%	822	229.61%	780	217.88%	CE	-	219.83%	221.23%
502.gcc_r	256	714	278.91%	847	330.86%	729	284.77%	CE	-	296.88%	284.77%
505.mcf_r	399	510	127.82%	667	167.17%	551	138.10%	602	150.88%	148.87%	142.11%
508.namd_r	295	317	107.46%	665	225.42%	479	162.37%	675	228.81%	194.92%	173.90%
510.parest_r	430	585	136.05%	1314	305.58%	886	206.05%	CE	-	218.37%	174.19%
511.povray_r	426	1068	250.70%	1604	376.53%	1235	289.91%	1227	288.03%	262.68%	277.23%
519.lbm_r	275	278	101.09%	431	156.73%	347	126.18%	554	201.45%	126.55%	124.36%
520.omnetpp_r	343	675	196.79%	1010	294.46%	872	254.23%	532	155.10%	232.36%	238.19%
523.xalancbmk_r	408	560	137.25%	739	181.13%	600	147.06%	418	102.45%	150.25%	150.98%
531.deepsjeng_r	289	408	141.18%	587	203.11%	442	152.94%	595	205.88%	173.36%	175.43%
538.imagick_r	499	681	136.47%	930	186.37%	863	172.95%	CE	-	140.68%	138.28%
541.leela_r	456	664	145.61%	933	204.61%	808	177.19%	906	198.68%	171.05%	171.49%
557.xz_r	362	415	114.64%	554	153.04%	488	134.81%	574	158.56%	187.29%	149.72%
600.perlbench_s	349	722	206.88%	1113	318.91%	806	230.95%	CE	-	236.10%	235.53%
602.gcc_s	476	604	126.89%	1341	281.72%	729	153.15%	RE	-	135.08%	128.15%
605.mcf_s	788	1062	134.77%	1276	161.93%	1205	152.92%	1113	141.24%	143.53%	142.51%
619.lbm_s	551	582	105.63%	676	122.69%	608	110.34%	535	97.10%	131.94%	133.03%
620.omnetpp_s	323	686	212.38%	1042	322.60%	871	269.66%	518	160.37%	243.03%	251.70%
623.xalancbmk_s	396	536	135.35%	714	180.30%	618	156.06%	417	105.30%	152.78%	156.06%
631.deepsjeng_s	347	498	143.52%	750	216.14%	540	155.62%	705	203.17%	178.10%	177.23%
638.imagick_s	2119	2635	124.35%	3751	177.02%	4271	201.56%	3604	170.08%	123.78%	116.61%
641.leela_s	452	669	148.01%	1041	230.31%	816	180.53%	904	200.00%	173.01%	171.02%
644.nab_s	1198	1355	113.11%	1915	159.85%	1480	123.54%	1464	122.20%	139.73%	138.48%
657.xz_s	871	1045	119.98%	1323	151.89%	1342	154.08%	1240	142.37%	165.56%	152.35%
Geometric Means.			**146.04%**		**212.58%**		**174.89%**		**161.76%**	**175.63%**	**170.24%**

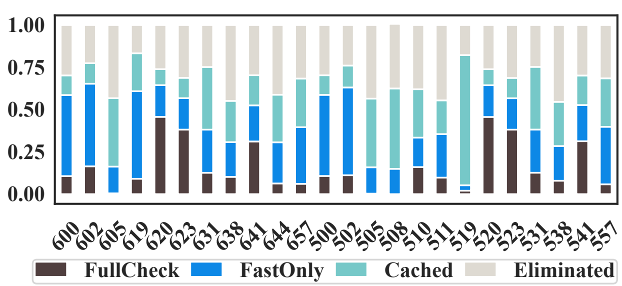


Figure 10. The proportion of memory instructions handled by different optimizations in GiantSan with ASan as the baseline. The x-labels are the project IDs. *Eliminated* are codes removed due to the check merging, and *Cached* are the ones optimized by the caching. *FastOnly* are the codes where the fast check suffices, and *FullCheck* are the ones that require both fast check and slow check.

require the slow check to tackle the corner cases outside the fast check's scope. The data shows that 49.22% of the remaining unoptimized tasks only use fast checks. The result indicates that the optimizations significantly reduce runtime checks and metadata loadings to help GiantSan gain high efficiency, and the fast check suffices to cover the majority of protection tasks.

The ablation study column in Table 2 shows the runtime overhead of GiantSan with solely caching enabled and check elimination enabled, respectively. On average, compared to ASan, GiantSan-CacheOnly and GiantSan-EliminationOnly show 32.82% and 37.61% improvements, respectively. Meanwhile, with either optimization enabled, GiantSan has comparable efficiency to ASan-- and LFP with about 70% overhead, and combining both optimizations achieves the best performance among all test configurations. GiantSan is faster than ASan because it supports operation-level protection with constant time region checks and history caching. Though ASan-- also uses static analysis to reduce redundant checks (it has a similar efficiency with GiantSan-Elimination-Only), it does not support the history cache that can further reduce runtime overhead. GiantSan is faster than LFP because LFP has to use extra instructions to simulate the stack due to the incomplete stack protection caused by the high memory alignment requirement. This result shows that both optimizations in GiantSan have significantly contributed to reducing the number of checks, and the fast check covers most of the memory protection tasks, allowing us to achieve a notable performance improvement.

Table 3. Detection capability on the Juliet Test Suite. All test cases have two versions: buggy and non-buggy versions. All tested tools have no false-positive issues under the C/C++ standard and pass all the non-buggy tests. Therefore, only the results for the buggy versions are presented to illustrate the false-negative issue.

CWE ID & Type	GiantSan	ASan	ASan--	LFP	Total
121: Stack Buffer Overflow	1435	1435	1435	49	1439
122: Heap Buffer Overflow	1504	1504	1504	4	1504
124: Buffer Underwrite	767	767	767	767	767
126: Buffer Overread	441	441	441	352	449
127: Buffer Underread	916	916	916	916	916
416: Use After Free	393	393	393	393	393
476: NULL Pointer Dereference	288	288	288	288	288
761: Free Pointer Not at Start of Buffer	192	192	192	192	192
Total	5063	5063	5063	2088	5075

5.3 Detectability Study

On top of the performance improvement, we also evaluate the practicalness of GiantSan in detecting memory errors.

Setting. We evaluate the bug detection ability on Juliet Test Suite (version 1.3) [32], Magma [20], and Linux Flaw Project [8], which are error collections widely used to evaluate the effectiveness of software assurance tools.

Juliet Test Suite contains cases that wait for an external signal (e.g., sockets), and some test cases include a randomized version (triggered with probability). We remove these cases to avoid infinitely waiting and non-deterministic results. Linux Flaw Project contains CVEs related to real-world programs, and we pick the memory-related ones, including 28 vulnerabilities from 8 programs written in C/C++. Magma [20] provides 58,969 test cases collected from its fuzzing campaign. We evaluate ASan, ASan-- and GiantSan on Magma to examine the effectiveness of GiantSan's anchor-based enhancement.

Results. Table 3 and Table 4 show the results on Juliet Test Suite and Linux Flaw Project, respectively. GiantSan, ASan, and ASan-- have the same results in all cases, while LFP has a significant number of false negatives in both benchmarks. LFP has many false negatives because it allocates objects with more spaces than the program requires, similar to BBC [2] discussed in Section 2.1. The cases missed by GiantSan, ASan, and ASan-- are potential overflow errors caused by uninitialized values. However, the uninitialized values loaded do not really trigger an overflow; thus, these tools do not generate bug reports since no overflow occurs.

For the redzone setting test, we evaluate GiantSan, ASan, and ASan-- on Magma, and the result is listed in Table 5. As we can see, GiantSan and ASan perform similarly in most projects. However, for large-scale project *PHP*, GiantSan reports 463 more cases than ASan and ASan-- (redzone=16) and 57 more cases than ASan and ASan-- (redzone=512). These false negatives are the POCs for *CVE-2018-14883* and are caused by the small redzone size. The result supports our

Table 4. Detection capability for CVEs in Linux Flaw Project.

Program	CVE ID	GiantSan	ASan	ASan--	LFP
libzip	CVE-2017-12858	✓	✓	✓	
autotrace	CVE-2017-9164	✓	✓	✓	✓
	CVE-2017-9165	✓	✓	✓	
	CVE-2017-9166~9173	✓	✓	✓	✓
imageworsener	CVE-2017-9204~9207	✓	✓	✓	✓
lame	CVE-2015-9101	✓	✓	✓	✓
zziplib	CVE-2017-5976~5977	✓	✓	✓	✓
libtiff	CVE-2016-10270~10271	✓	✓	✓	✓
	CVE-2016-10095	✓	✓	✓	✓
potrace	CVE-2017-7263	✓	✓	✓	✓
mp3gain	CVE-2017-14407~14408	✓	✓	✓	✓
	CVE-2017-14409	✓	✓	✓	

Table 5. Detection capability in real-world projects from Magma Test Suite. *rz* is short for *redzone*.

Project (LoC)	ASan-- (rz=16)	ASan-- (rz=512)	ASan (rz=16)	ASan (rz=512)	GiantSan (rz=16)	Total
php (1.3M)	**1556**	**1962**	**1556**	**1962**	**2019**	3072
libpng (86K)	1881	1881	1881	1881	1881	1881
libtiff (91K)	9858	9858	9858	9858	9858	9858
libxml2 (284K)	30566	30566	30566	30566	30566	30574
openssl (535K)	46	46	46	46	46	1509
sqlite3 (367K)	1528	1528	1528	1528	1528	1528
poppler (43K)	10201	10201	10201	10201	10201	10547

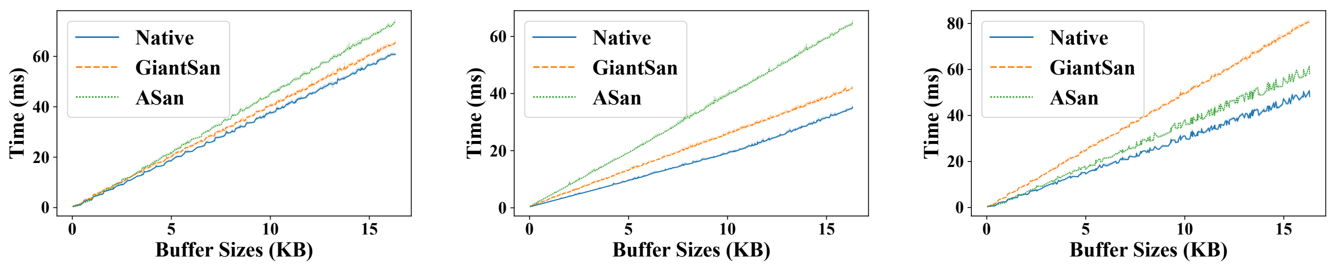
conclusion in Section 4.4.1: insufficient redzone size leads to a false negative because of redzone bypassing, and GiantSan solves this with anchor-based enhancement.

5.4 Limitation

Because GiantSan only provides a single-sided summary, i.e., it summarizes segments from lower addresses to higher addresses, GiantSan may not effectively safeguard lower addresses given only higher addresses, causing potential efficiency deterioration in reverse traversals with unbounded loops when anchor-based enhancement is enabled.

To study this potential limitation, we conducted an additional study on Perlbench, which is a project in the SPEC CPU 2017 we used in Section 5.1. It is a program interpreter that intensively iterates the input buffer and contains different buffer iteration patterns, e.g., forward / reverse / random traversals. We evaluated the execution time to complete a traversal of the input buffer to compare the performance of GiantSan's history caching and ASan in different buffer traversal patterns. Each run is repeated 100 times to reduce variations, and the geometric mean is presented.

The results in Figure 11 show that GiantSan is 1.48x and 1.07x faster than ASan in random and forward traversals, respectively. However, due to the extra instructions to perform anchor-enhanced checks, GiantSan is 1.39x slower than ASan in reverse traversals. The reason is that GiantSan has one-sided complexity guarantees with history caching, i.e., quasi-bound converges to the upper bound of the allocated



(a) Forward Traversal: iterate over the buffer from the lowest address to the highest address

(b) Random Traversal: iterate over the buffer in random order

(c) Reverse Traversal: iterate over the buffer from the highest address to the lowest address

Figure 11. The time cost of GIANTSAN and ASan in three buffer traversal patterns: Forward, Random, and Reverse. The baseline *Native* is the execution time without sanitization.

region in $\lceil \log_2 \frac{n}{8} \rceil$ time; however, it does not provide time guarantees for the lower bound. Therefore, GIANTSAN is able to save time by predicting the addressability of higher addresses from lower addresses, but not vice versa.

The experimental data empirically evidence the performance difference of our approach in handling different traversal patterns, which is consistent with our theoretical justification. Fortunately, the number of reverse traversals in real-world programs is relatively limited. For example, in the real-world programs collected by the SPEC CPU 2017, only 0.39% of the buffer traversals are in reverse order. Past studies [15, 22] show that the impact of underflow is comparatively less severe than overflow. Furthermore, the SCEV optimization could eliminate the runtime checks by inferring the loop bounds, if possible.

For programs that heavily use reverse traversals, several alternatives can mitigate the efficiency deterioration. One is to remove the anchor-based enhancement in underflow detection so that GIANTSAN's detection degrades to ASan's mode (i.e., only checking the location of the access and ignoring the anchor); however, this would eliminate the superiority of GIANTSAN over ASan in terms of underflow detection accuracy. The second solution is to locate the lower bound before buffer reverse traversals by enumerating the folding degrees and checking whether corresponding folded segments exist.

Also, though GIANTSAN improves the efficiency of location-based methods, it still shares some common limitations with existing works.

Sub-object Overflow Insensitivity: GIANTSAN detects memory accesses outside objects' allocated regions but cannot detect memory safety violations related to sub-objects, which is an open question in the existing literature. The best practices in detecting sub-object overflow are pointer-based methods like Softbound+CETS [29, 30] and EffctiveSan [10]. However, they all suffer from high runtime overhead and require precise type information, which might not be available in real-world programs.

Quarantine Bypassing: GIANTSAN detects temporal errors based on memory quarantine, but the memory quarantine can be bypassed with a small probability. It is a common issue for memory quarantine-based solutions [34, 41, 42]. In practice, the probability of bypassing the quarantine queue is low, and few related false negative reports exist.

6 Related Work

Researchers have proposed various dynamic error detectors. We further discuss existing works targeting memory errors.

Token Authentication. HWASAN [35] uses *address tagging* to replace the redzone with token authentication. A random token is attached to pointers with the *Top-Byte-Ignore* hardware support, and the token is stored in the shadow memory for memory regions. The token mismatch between pointers and memory regions results in memory errors. Like GIANTSAN's anchor-based enhancement, it mitigates the redzone dilemma. Specifically, HWASAN solves the problem that traditional location-based methods are unable to distinguish between different allocated memory regions by assigning an 8-bit identifier to each region. It propagates the identifier in a pointer-based manner and removes the need for redzones with the token-matching model.

However, it does not improve the detection efficiency of the location-based methods, where a single check only safeguards a small region (e.g., 16 bytes). Therefore, it suffers from the low protection density issue that requires excessive runtime checks to safeguard a large region, decreasing its efficiency. This efficiency issue is GIANTSAN's key motivation.

Redzone Enhancement. Location-based solutions divide the memory into separated regions using redzones to detect sophisticated bugs. Some methods that focus on redzone enhancement aim to reduce runtime overhead with redzone poisoning or improve accuracy with adaptive redzone sizes.

For example, in-band redzone methods [16, 18] fill the redzone with a random pattern and compare the loaded data with that pattern. If they are different, the memory access is not in the redzone and is safe. These methods reduce dedicated data structure inquiries (e.g., shadow memory), thus promoting memory locality. However, this method protects only a small region with one check and faces the same low protection density issue as other location-based methods. Similarly, it suffers from small redzone size, e.g., FloatZone [16] cannot detect CVE-2017-7263 with 16-byte in-band redzones. These two issues are what GiantSan addresses.

Some approaches reduce the impact of redzone sizes with adaptive settings. LBC [18] selects different redzones based on the allocated region sizes. MEDS [17] spreads the objects evenly in the address space to increase the distance between objects as much as possible. To minimize memory consumption, MEDS uses page aliasing to allow multiple virtual pages to share the same physical page.

GiantSan is compatible with all these redzone enhancement techniques because GiantSan does not impose any extra requirements on redzone settings and the contents in the redzone areas. GiantSan only modifies the shadow memory encoding for non-redzone areas and reduces the dependency on redzone size by modifying the runtime check logic with the selected anchors.

Pointer Tracking. Pointer-based techniques provide a memory safety guarantee by tracking the lifetime of pointers. As discussed in Section 2.1, pointer-based methods require the pointer type information to propagate tags and avoid tag misuse. The complete memory safety guarantee in pointer-based methods requires instrumenting the source codes of the whole runtime environment, which is usually expensive and unavailable and thus makes these methods less portable.

Traditional pointer-based solutions [4, 29, 30] require extra instructions to propagate metadata (e.g., bound) along pointer arithmetics; in contrast, location-based solutions only check pointer dereference operations, which is much fewer than pointer arithmetics. The propagation is the primary source of the pointer-based solutions' runtime overhead [37]. *Pointer tagging* is a popular solution to mitigate the overhead issue in propagation. With the proliferation of large bit-width systems (e.g., 64-bit), a single pointer structure can now represent far larger address space than a program needs, resulting in some upper spare bits in pointers. Consequently, many pointer-based methods [15, 22, 23, 25] propagate metadata with the upper spare bits so that the metadata associated with pointers can be propagated automatically.

Though pointer tagging solves the efficiency problem of data propagation, it faces a new problem related to the bit width: the upper spare bits are not enough to hold the metadata. One solution is reducing the address space. For example, Delta Pointers [22] and SGXBound [23] use 32-bit address space in a 64-bit platform and record the metadata with the other 32 bits. The narrowing down of the address space makes them less suitable for programs with large memory footprints. Delta Pointers mitigate this issue by providing a trade-off between the maximum object size and the address space size. Another solution [25] is to store the metadata in a key-value database, and the pointer tag only serves as the key. Compared with the shadow memory inquiry used in location-based solutions, the key-value store takes more time to retrieve the metadata.

GiantSan also suffers from a bit-width limitation, i.e., a single shadow byte can only hold 256 different states. GiantSan solves this limitation with the on-demand inquiry. The segment folding technique in GiantSan can be considered as a key-value store that takes logarithmic time to index an object's bound. However, one of our key observations is that the program does not always traverse the entire allocated region, and in most cases, we only need to safeguard a subregion. This observation allows us to reduce the number of queries by looking up folding degrees on demand.

The spirit of on-demand inquiry is orthogonal to the pointer-based solutions and could mitigate the bit width requirement faced by the pointer tagging technique. Integrating the on-demand inquiry spirit into pointer-based solutions is a future research direction we are going to address.

Rounded-Up Bound. Works like LFP [9, 11] and BBC [2] obtain the object bound by directly fetching the bound from shadow memory. However, to enable compact shadow memory, they only support a limited set of allocation sizes to reduce the bit width for recording the bound. As a result, they overapproximate the object sizes required by the programs, leading to significant false negative issues.

BBC [2] uses the power-of-two strategy similar to GiantSan from a particular perspective. However, BBC uses the power-of-two spirit to *approximate* the real object bound, while GiantSan uses the power-of-two spirit to build *precise summaries* of addressable regions. Therefore, GiantSan is more precise than BBC. LFP enhances BBC by introducing more variety of allocation sizes but still has numerous false negatives, as shown in our experiments.

7 Conclusions

We present GiantSan, a location-based sanitizer optimizing runtime checks with segment folding. GiantSan summarizes segments without non-addressable bytes to increase protection density. It largely reduces 59.10% and 38.52% of the overhead introduced by ASan and ASan-- on the SPEC CPU 2017 benchmark, respectively. Furthermore, the evaluation on the PHP project demonstrates that GiantSan can minimize the dependence on the redzone, thus resulting in a more effective detection ability than ASan and ASan--.

8 Acknowledgements

We thank the anonymous reviewers for their valuable comments and opinions for improving this work. This work is supported by the ITS/440/18FP grant from the Hong Kong Innovation and Technology Commission and research grants from Huawei, Microsoft, and TCL. Heqing Huang is the corresponding author.

A Artifact Appendix

A.1 Abstract

This section describes how to reproduce the main results in this paper. The artifact includes a C/C++ compiler equipped with GiantSan's memory sanitization technique, the testing benchmarks, and the shell scripts to reproduce the results. We also prepare a docker image to simplify the workflow.

A.2 Artifact check-list (meta-information)

- **Program:** Python; ShellScript; C;C++.
- **Binary:** LLVM-GiantSan/bin/clang++.
- **Data set:** Juliet; LinuxFlaw; Magma.
- **Hardware:** x86_64 platform.
- **Run-time environment:** Ubuntu 18.04.
- **Metrics:** The number of buggy test cases detected.
- **Output:** Numbers similar to the results in Tables 3, 4, and 5.
- **Experiments:** Scripts and a docker image are provided to run the experiment automatically.
- **How much disk space required (approximately)?:** 60G.
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes.
- **How much time is needed to complete experiments (approximately)?:** 1 hour.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** LGPL-v2.1
- **Archived (provide DOI)?:** 10.5281/zenodo.10608291

A.3 Description

A.3.1 How to access.

The artifact can be downloaded from the following repository.
https://doi.org/10.5281/zenodo.10608291

A.3.2 Data sets.

The artifact includes Juliet Test Suite, Magma Benchmark, and the Linux Flaw Project we used in this work. Since SPEC CPU 2017 is a commercial program, we cannot share its copy.

There are two LLVM distributions included in the artifact, which are equipped with GiantSan and ASan, respectively. Scripts are provided to compile, run, and fetch the sanitization results of all benchmarks. The artifact also provides a docker image to help automate the workflow.

A.4 Installation

Download the artifact and use the following command in the directory to build the docker image.

```
docker build -t "artifact" .
```

A.5 Experiment workflow

Use the following commands to start the experiments with GiantSan and ASan, respectively.

```
docker run artifact /bin/bash -c \
    "source /artifact/activeGiantSan.sh; \
    python3 /artifact/run.py"
```

```
docker run artifact /bin/bash -c \
    "source /artifact/activeASan.sh; \
    python3 /artifact/run.py"
```

Use the following scripts instead of `run.py` to run the experiments for Juliet, LinuxFlaw, and Magma separately, instead of running the whole experiment: `juliet/run.py`, `linuxflaw/run.py`, `magma/run.py`.

A.6 Evaluation and expected results

The results are automatically printed by the docker image, including the number of memory error-related test cases detected in different benchmarks, corresponding to our evaluation results in Tables 3, 4, and 5 for the benchmark Juliet, LinuxFlaw, and Magma, respectively. Since the results of memory sanitization would be affected by the runtime environment, we recommend running the two experiments (GiantSan and ASan) in the same environment and comparing their relative performance.

A.7 Experiment customization

The usage of GiantSan is the same as ASan. Set the environment variable `PATH` to `LLVM-GiantSan/bin/` and use the compiler frontend `clang/clang++` with the compilation flag `-fsanitize=address`.

A.8 Notes

Once SPEC CPU 2017 is installed, use the following commands to run SPEC CPU 2017-related experiments. Suppose that the directory of the artifact is /artifact and SPEC CPU 2017 is installed in /spec2017.

```
cp /artifact/GiantSan.cfg /spec2017/config/
source /spec2017/shrc
export PATH=/artifact/LLVM-GiantSan/bin/;$PATH
export ASAN_OPTIONS=halt_on_error=false
runcpu --config=GiantSan --rebuild all
```

References

[1] Sam Ainsworth and Timothy M. Jones. MarkUs: Drop-in use-after-free prevention for low-level languages. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 578–591, 2020. https://doi.org/10.1109/SP40000.2020.00058.

[2] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy Bounds Checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, page 51–66, USA, 2009. USENIX Association. https://www.usenix.org/legacy/events/sec09/tech/full_papers/akritidis.pdf.

[3] Android. Hwasan, asan and kasan. https://source.android.com/docs/security/test/memory-safety/hwasan-asan-kasan.

[4] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. Cup: Comprehensive user-space protection for c/c++. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, page 381–392, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3196494.3196540.

[5] Microsoft Corporation. How to use pageheap.exe in windows xp, windows 2000, and windows server 2003. 2000. https://mskb.pkisolutions.com/kb/286470.

[6] C. Cowan. Software security for open-source systems. *IEEE Security Privacy*, 1(1):38–45, 2003. https://doi.org/10.1109/MSECP.2003.1176994.

[7] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A practical Page-Permissions-Based scheme for thwarting dangling pointers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 815–832, Vancouver, BC, August 2017. USENIX Association. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/dang.

[8] Dongliang Mu. Linux flaw project. https://github.com/mudongliang/LinuxFlaw, 2017.

[9] Gregory J. Duck and Roland H. C. Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 132–142, New York, NY, USA, 2016. Association for Computing Machinery. https://doi.org/10.1145/2892208.2892212.

[10] Gregory J. Duck and Roland H. C. Yap. Effectivesan: Type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 181–195, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3192366.3192388.

[11] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. 01 2017. https://doi.org/10.14722/ndss.2017.23287.

[12] Reza Mirzazade farkhani, Mansour Ahmadi, and Long Lu. PTAuth: Temporal memory safety via robust points-to authentication. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1037–1054. USENIX Association, August 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/mirzazade.

[13] GCC. The gnu compiler collection. https://gcc.gnu.org/.

[14] Google. Addresssanitizier wiki. https://github.com/google/sanitizers/wiki/AddressSanitizerFlags.

[15] Amogha Udupa Shankaranarayana Gopal, Raveendra Soori, Michael Ferdman, and Dongyoon Lee. TAILCHECK: A lightweight heap overflow detection mechanism with page protection and tagged pointers. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 535–552, Boston, MA, July 2023. USENIX Association. https://www.usenix.org/conference/osdi23/presentation/gopal.

[16] Floris Gorter, Enrico Barberis, Raphael Isemann, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos. FloatZone: Accelerating memory error detection using the floating point unit. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 805–822, Anaheim, CA, August 2023. USENIX Association. https://www.usenix.org/conference/usenixsecurity23/presentation/gorter.

[17] Wookhyun Han, Byunggill Joe, Byoungyoung Lee, Chengyu Song, and Insik Shin. Enhancing memory error detection for large-scale applications and fuzz testing. 01 2018. https://doi.org/10.14722/ndss.2018.23312.

[18] Niranjan Hasabnis, Ashish Misra, and R. Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, page 135–144, New York, NY, USA, 2012. Association for Computing Machinery. https://doi.org/10.1145/2259016.2259034.

[19] Reed Hastings. Purify: Fast detection of memory leaks and access errors. In *Proc. 1992 Winter USENIX Conference*, pages 125–136, 1992. https://web.stanford.edu/class/cs343/resources/purify.pdf.

[20] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), jun 2021. https://doi.org/10.1145/3428334.

[21] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*, 1997. https://www.doc.ic.ac.uk/~phjk/Publications/BoundsCheckingForC.pdf.

[22] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta Pointers: Buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3190508.3190553.

[23] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 205–221, New York, NY, USA, 2017. Association for Computing Machinery. https://doi.org/10.1145/3064176.3064192.

[24] Chris Arthur Lattner. LLVM: An infrastructure for multi-stage optimization. 2002. http://llvm.org.

[25] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1901–1915, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3548606.3560598.

[26] Linux Kernel. The kernel address sanitizer. https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html.

[27] David Litchfield. Buffer underruns, dep, aslr and improving the exploitation prevention mechanisms (xpms) on the windows platform. *Next Generation Security Software*, 2005. https://research.nccgroup.com/wp-content/uploads/episerver-images/assets/854f87540884465e8c6930b1b2fabf9b/854f87540884465e8c6930b1b2fabf9b.pdf.

[28] LLVM. Scalar evolution and loop optimization. https://llvm.org/devmtg/2009-10/ScalarEvolutionAndLoopOptimization.pdf.

[29] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 245–258, New York, NY, USA, 2009. Association for Computing Machinery. https://doi.org/10.1145/1542476.1542504.

[30] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for c. *SIGPLAN Not.*, 45(8):31–40, jun 2010. https://doi.org/10.1145/1837855.1806657.

[31] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89–100, New York, NY, USA, 2007. Association for Computing Machinery. https://doi.org/10.1145/1250734.1250746.

[32] NIST. Software assurance reference dataset. https://samate.nist.gov/SARD/test-suites, 2017.

[33] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Network and Distributed System Security Symposium*, 2004. https://www.ndss-symposium.org/ndss2004/practical-dynamic-buffer-overflow-detector/.

[34] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.

https://dl.acm.org/doi/10.5555/2342821.2342849.

[35] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory tagging and how it improves c/c++ memory safety. *arXiv preprint arXiv:1802.09517*, 2018. https://arxiv.org/abs/1802.09517.

[36] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with Bit-Precision. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association. https://www.usenix.org/conference/2005-usenix-annual-technical-conference/using-valgrind-detect-undefined-value-errors-bit.

[37] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295, 2019. https://doi.org/10.1109/SP.2019.00010.

[38] Standard Performance Evaluation Corporation. Spec cpu® 2017. https://www.spec.org/cpu2017/, 2022.

[39] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013. https://doi.org/10.1109/SP.2013.13.

[40] The 2022 CWE Top 25 Team. 2022 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html, 2022.

[41] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. SANRAZOR: Reducing redundant sanitizer checks in C/C++ programs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 479–494. USENIX Association, July 2021. https://www.usenix.org/conference/osdi21/presentation/zhang.

[42] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. Debloating address sanitizer. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4345–4363, Boston, MA, August 2022. USENIX Association. https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-yuchen.