arXiv:2506.05022v2 [cs.SE] 18 Jun 2025

Tech-ASan: Two-stage check for Address Sanitizer

Yixuan Cao
caoyixuan2019@email.szu.edu.cn
College of Computer Science and Software Engineering, Shenzhen University
Shenzhen, Guangdong, China

Yuhong Feng*
yuhongf@szu.edu.cn
College of Computer Science and Software Engineering, Shenzhen University
Shenzhen, Guangdong, China

Huafeng Li
Chongyi Huang
lihuafeng2020@email.szu.edu.cn
huangchongyi2020@email.szu.edu.cn
College of Computer Science and Software Engineering, Shenzhen University
Shenzhen, Guangdong, China

Fangcao Jian
jianfangcao2023@email.szu.edu.cn
College of Computer Science and Software Engineering, Shenzhen University
Shenzhen, Guangdong, China

Haoran Li
lihaoran2018@email.szu.edu.cn
College of Computer Science and Software Engineering, Shenzhen University
Shenzhen, Guangdong, China

Xu Wang
wangxu@szu.edu.cn
College of Computer Science and Software Engineering, Shenzhen University
Shenzhen, Guangdong, China

ABSTRACT

Address Sanitizer (ASan) is a sharp weapon for detecting memory safety violations, including temporal and spatial errors hidden in C/C++ programs during execution. However, ASan incurs significant runtime overhead, which limits its efficiency in testing large software. The overhead mainly comes from sanitizer checks due to the frequent and expensive shadow memory access. Over the past decade, many methods have been developed to speed up ASan by eliminating and accelerating sanitizer checks, however, they either fail to adequately eliminate redundant checks or compromise detection capabilities. To address this issue, this paper presents Tech-ASan, a two-stage check based technique to accelerate ASan with safety assurance. First, we propose a novel two-stage check algorithm for ASan, which leverages magic value comparison to reduce most of the costly shadow memory accesses. Second, we design an efficient optimizer to eliminate redundant checks, which integrates a novel algorithm for removing checks in loops. Third, we implement Tech-ASan as a memory safety tool based on the LLVM compiler infrastructure. Our evaluation using the SPEC CPU2006 benchmark shows that Tech-ASan outperforms the state-of-the-art methods with 33.70% and 17.89% less runtime overhead than ASan and ASan--, respectively. Moreover, Tech-ASan detects 56 fewer false negative cases than ASan and ASan-- when testing on the Juliet Test Suite under the same redzone setting.

CCS CONCEPTS

• **Security and privacy → Software and application security**;
• **Software and its engineering → Software defect analysis**.

KEYWORDS

Memory Safety Violation, Address Sanitizer (ASan), Two-Stage Check

ACM Reference Format:
Yixuan Cao, Yuhong Feng, Huafeng Li, Chongyi Huang, Fangcao Jian, Haoran Li, and Xu Wang. 2025. Tech-ASan: Two-stage check for Address Sanitizer. In *16th Symposium on Internetware (Internetware 2025), June 20–22, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 12 pages. https://doi.org/xx.xxx/xx.xxx

1 INTRODUCTION

Programs written in memory-unsafe languages such as C and C++ often contain memory safety violations, which can be categorized into *temporal errors* and *spatial errors*. A temporal error occurs when code attempts to access a memory object after it has been freed, whereas a spatial error happens when the code reads or writes beyond the bounds of a valid memory object.

Memory safety violations are the root cause of many of today's most severe vulnerabilities [28], which may lead to severe issues such as system crash, data breach, and hijacked execution when exploited [27]. As reported in the 2023 CWE Top 10 KEV Weaknesses, use-after-free, heap-based buffer overflow, and out-of-bounds write rank 1st, 2nd, and 3rd among all weaknesses, respectively.[1] In recent years, sanitizing for memory safety has become a widely adopted method for identifying vulnerabilities in software testing, especially within the realm of fuzzing [7, 22]. Motivated by numerous security incidents caused by memory safety violations in system software, various sanitizers have been created to detect memory safety violations at runtime to help developers fix them, where Address Sanitizer (ASan) is the most popular tool due to its outstanding *capability* (detection of a wide spectrum of spatial errors

*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Internetware 2025, June 20–22, 2025, Trondheim, Norway
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN xxx-xxx/25/06
https://doi.org/xx.xxx/xx.xxx

[1] https://cwe.mitre.org/top25/archive/2023/2023_kev_list.html

and temporal errors), *scalability* (ability to support industry-grade programs like operating system kernels and web browsers), and *usability* (nearly zero configuration and seamless integration into mainstream compilers such as clang and gcc) [24, 34].

Technically, ASan allocates additional shadow memory, poisons/unpoisons the shadow memory at runtime to record the *addressable* state of the memory, and checks the shadow memory before memory access to determine whether it is a memory safety violation. However, ASan brings approximately 1× runtime overhead to the tested program, which limits the efficiency of ASan in testing large software. Over the past decade, reducing the runtime overhead of ASan has attracted the interest of many researchers. Existing studies have shown that more than 80% of the ASan's runtime overhead is introduced by sanitizer checks [34]. To address this issue, the existing optimization solutions mainly focus on two aspects: *check elimination* [13, 17, 25, 30–32, 34] and *check acceleration* [7, 16].

Check elimination. As the official ASan documentation[2] states, finding all bugs does not require to instrument all memory accesses, sanitizer checks for memory with safety assurance can be eliminated. Existing check-eliminating methods can be categorized into *performance-driven* and *security-driven* methods. Performance-driven methods [13, 17, 25, 30] eliminate sanitizer checks to meet performance constraints but do not guarantee safety. DoubleTake [17] uses canaries to overwrite unaddressable memory and divides the program runtime into several epochs, checking whether the canary has been tampered with at the end of each epoch. However, this design can only detect write vulnerabilities, but not read vulnerabilities. In addition, DoubleTake only partially overwrites the freed heap memory with canaries, so it will miss use-after-free in unprotected areas. GWP-ASan [25] randomly protects heap objects with guard pages and ignores most of the others. ASAP [30] removes sanitizer checks on "hot" code that is more frequently executed. PartiSan [13] follows a performance-driven metric to remove sanitizer checks.

Security-driven methods [32, 34] are designed to eliminate redundant sanitizer checks to reduce runtime overhead with safety assurance. SANRAZOR [32] combines static patterns and dynamic patterns to identify and remove redundant sanitizer checks, but it still cannot ensure the removed checks are indeed redundant due to its unsound patterns. Also, SANRAZOR needs user input for profiling. Therefore, SANRAZOR degrades ASan's capability and usability. Finally, ASan-- [34] uses static analysis to eliminate redundant checks while ensuring safety at compile time, which is still the state-of-the-art (SOTA) method to accelerate ASan. However, ASan-- still has room for further optimization: First, it fails to speed up the sanitizer check. Second, it fails to fully remove checks within loops. Although checks in loops account for about 45% of the overhead introduced by all ASan checks [34], eliminating redundant checks in loops remains unsolved. In summary, existing check elimination methods either eliminate checks without safety assurance or fail to adequately eliminate checks on security access.

Check acceleration. In recent years, some works have attempted to accelerate checks to reduce the overall runtime overhead of the sanitizer, but they compromise the capabilities of ASan. FloatZone [7] replaces comparison-based checks with floating-point underflow exception-based checks to enable higher instruction-level parallelism, and cancels shadow memory to reduce cache miss rate. But FloatZone introduces false positives because metadata is no longer isolated in shadow memory. GiantSan [16] introduces history caching and region checking to achieve fast operation-level protection. However, our experimental evaluation shows that it still misses memory safety violations in the Juliet Test Suite [1] and in the real world that can be detected by ASan. In summary, both FloatZone and GiantSan reduce ASan's detection capability.

From the above analysis, we state the problem as: How to further reduce the runtime overhead while maintaining the capability, scalability, and usability of ASan? To address it, we propose Tech-ASan, a novel ASan optimization method. First, Tech-ASan introduces a two-stage checker: when checking a memory access instruction, the fast check stage determines whether the accessing location contains a magic value. If so, the slow check stage checks the metadata of shadow memory to determine whether the memory is addressable. Since the probability of triggering the slow checker is very low and the slow check stage accesses shadow memory infrequently, the check is accelerated. Second, Tech-ASan proposes an effective optimizer to identify and eliminate redundant checks, which integrates an original algorithm for removing checks in loops.

The contribution of this paper can be summarized as follows:

- We design a novel and fast two-stage sanitizer check algorithm for ASan, which does not rely on loading shadow bytes in most cases.
- We propose an optimizer to eliminate redundant sanitizer checks with safety assurance, which integrates a novel algorithm for removing checks within the loop.
- We implement Tech-ASan as a memory safety tool based on LLVM compiler infrastructure [12].
- Our comprehensive evaluation experiments show that Tech-ASan outperforms the SOTA methods with 33.70% and 17.89% less runtime overhead than ASan and ASan--, respectively, while maintaining the advantages of ASan.

The rest of the paper is organized as follows. Section 2 reviews the technical background of ASan. Section 3 presents our original optimization approach for ASan. Section 4 describes the performance evaluation. Section 5 reviews the related work, and finally Section 6 concludes the paper.

2 BACKGROUND

This section briefly reviews the technical background of ASan from three aspects: *shadow memory*, *redzone*, and *runtime check*.

2.1 Shadow Memory

As shown in Figure 1, ASan utilizes a shadow memory model to support sanitizer checks. By default, ASan spares one-eighth of the virtual address space as shadow memory, where each shadow byte records the status of eight bytes used by the application. Given the application memory address `Addr`, which can be located in stack, heap, or global regions, the corresponding address of the shadow byte is computed as `(Addr >> 3) + Offset`, where the `Offset` is a constant that must be chosen statically at the compiling time for every platform. With the same address computation way, addresses

[2] https://github.com/google/sanitizers/wiki/AddressSanitizerCompileTimeOptimizations

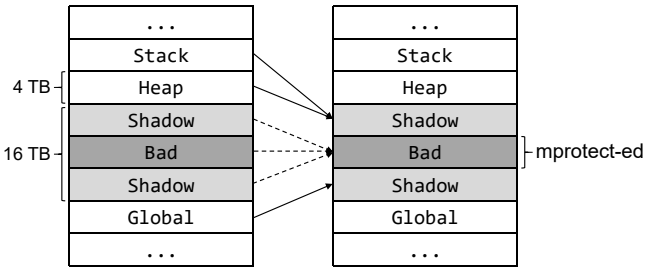

Figure 1: Shadow memory model in a 64-bit address space

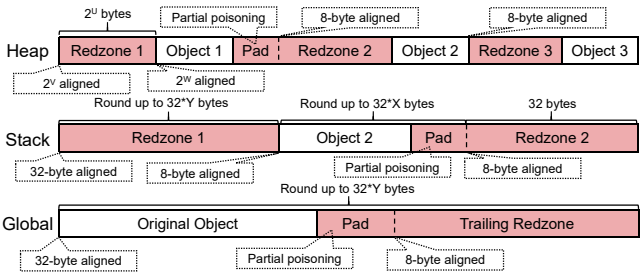

Figure 2: ASan's redzones in heap, stack, and global regions, where `U`, `V`, `W`, `X`, and `Y` represent positive integers.

in the shadow memory can be mapped into the *Bad* region, which is protected via page protection.

In shadow memory, each byte is encoded with the following regular: 0 means that the entire 8-byte corresponding application memory region is addressable; k ($1 \le k \le 8$) means that the first k bytes are addressable and the last (8 - k) bytes are not; any negative value means that the entire 8-byte word is unaddressable. In the last scenario, different negative values are used to indicate different types of memory safety violations, which help developers to locate and fix bugs.

2.2 Redzone

Figure 2 shows that ASan places redzones before and after each memory object in interesting memory regions, including stack, heap, and global [24, 34]. When allocating a memory object, ASan sets the object itself as addressable and poisons its redzones as unaddressable to detect spatial errors, e.g., heap-buffer-overflow and stack-buffer-overflow. When freeing a memory object, ASan poisons the freed memory object as unaddressable to detect temporal errors, e.g., heap-use-after-free and double-free. Note that poisoning operators are to change the shadow byte but not the application memory.

Heap. There are several heap allocation functions and operators provided by the standard C and C++ languages, i.e., `malloc()`, `calloc()`, `mmap()`, `new()`, and `new[]()`. ASan replaces them with customized functions. When a heap object is allocated, ASan places a redzone both before and after the buffer. The size of the redzone is a power of two, which can range from 16 to 2,048 according to the object size. When an object is freed by the functions or the operators provided by the standard C and C++ languages, i.e., `free()`, `munmap()`, `delete`, and `delete[]`, the freed object is poisoned and quarantined into a 256MB queue to detect heap-use-after-free and double-free.

Stack. The Stack objects can be allocated statically or allocated dynamically by the C standard function `alloca()`. For each stack object, ASan allocates and poisons the left redzone (32 bytes) and the right redzone (32 bytes plus up to 31 bytes for alignment) when entering a function.

Global. The global region manages global variables, static variables, and constants in programs written in the C and C++ languages, which corresponds to `.data` and `.bss` segments. ASan places a trailing redzone at the right of each global object. The redzone size is the larger of 32 bytes or one-fourth of the object size, plus a padding size if the object size is not 8-byte aligned. Finally, the size of the object plus the redzone is rounded up to a multiple of 32 bytes. All the redzones for global objects are poisoned when initializing the process. Since global objects are never to be freed, no temporal error occurs in the global region, which provides an optimization opportunity for eliminating unnecessary sanitizer checks.

2.3 Runtime Check

ASan instruments every memory access instructions, including load and store instructions in Intermediate Representation (IR) code, to check whether an access is addressable. There are two ways to implement this: inserting an inline instruction sequence or a function call. In order to compromise between runtime performance and code size, ASan inserts inline instructions by default to reduce runtime overhead, but when the number of instrumentation reaches a threshold, the check function is called.

Depending on the size of the memory access, the check works differently. When instrumenting an 8-byte memory access, ASan computes the address of the corresponding shadow byte, loads that byte, and checks whether it is 0:

```
1 ShadowAddr = (Addr >> 3) + Offset;
2 if (*ShadowAddr != 0)
3     ReportAndCrash(Addr);
```

where `Offset` is a large platform-dependent constant. When instrumenting an N-byte memory access, where N = 1, 2, or 4, ASan checks if the first k bytes in the 8-byte word are addressable:

```
1 ShadowAddr = (Addr >> 3) + Offset;
2 k = *ShadowAddr;
3 if (k != 0 && ((Addr & 7) + AccessSize > k))
4     ReportAndCrash(Addr);
```

ASan also intercepts standard C/C++ library functions that may cause memory errors (e.g., `memset()` and `strcpy()`). When such a function is called, ASan checks the shadow memory using an optimized function `__asan_region_is_poisoned()` to determine whether the accessed memory region is entirely addressable.

ASan's sanitizer checks rely on obtaining the corresponding shadow address through shift and addition calculations, loading the shadow byte, and determining whether it is a memory safety violation via comparison that may be more than once. Frequent access to shadow memory destroys the locality of the program,

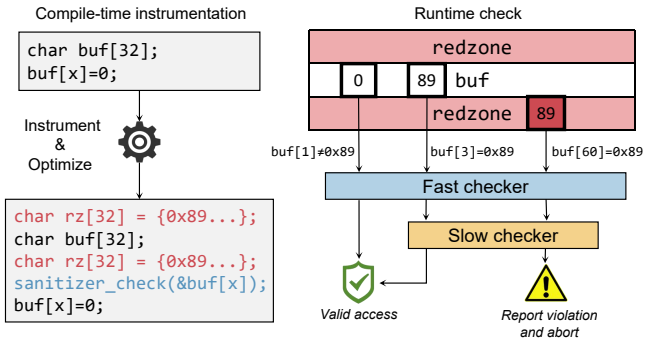

Figure 3: The framework of Tech-ASan

which increases runtime overhead. Our design focuses on how to complete the sanitizer check with only a fast comparison in most cases, without introducing expensive load shadow byte operations.

3 APPROACH

This section elaborates on how Tech-ASan is designed to accelerate ASan. Figure 3 overviews Tech-ASan, which consists of a compile-time instrumentation phase and a runtime check phase. In the compile-time instrumentation phase, Tech-ASan takes the program source code as its input, and outputs an instrumented executable program. Specifically, the program source code is first compiled into LLVM IR [12], and then instrumented to manage redzones and inject a magic value, as well as check violations (cf. Section 3.1). Unlike ASan, we not only poison and unpoison the shadow memory, but also poison and unpoison the redzones and the freed memory regions using the magic value, i.e., `0x89`. Since not all checks are necessary, Tech-ASan utilizes an optimizer to reduce runtime overhead by eliminating redundant checks (cf. Section 3.3).

In the runtime check phase, Tech-ASan executes normally if no memory safety violation is detected (e.g., `buf[1]` in Figure 3), otherwise it reports a violation and aborts the process immediately. Specifically, we design a two-stage checker where every sanitizer check is divided into two stages (cf. Section 3.2). In the fast check stage, Tech-ASan checks whether there is a magic value in the access location, serving as a fast filtering mechanism. When a magic value is detected during the fast check stage, the slow check stage is started for a more precise verification: If the slow checker does not detect an invalid memory access, the program continues execution (e.g., `buf[3]` in Figure 3), otherwise a violation is reported and the program is terminated (`buf[60]` in Figure 3).

3.1 Instrumentation

This section provides details on how Tech-ASan instruments the program under test. Tech-ASan first compiles the program under test from source code into LLVM IR and then proceeds with instrumentation. Similar to post compiler-based methods [7, 9, 16, 24, 34], Tech-ASan's instrumentation operations are performed at the IR level, therefore, the discussions in this paper do not involve memory objects managed by third-party libraries with inaccessible source code. Tech-ASan reuses the ASan's instrumentation infrastructure as reviewed in Section 2, and we focus on the two main differences between Tech-ASan and ASan, namely *magic value injection* and *memory safety check*.

Magic value injection. As illustrated in Figure 2, ASan places redzones on both sides of heap, stack, and global memory objects to detect spatial errors. When a memory object is allocated, ASan poisons shadow bytes corresponding to the redzones, and unpoisons shadow bytes corresponding to the memory object regions to detect spatial errors. When a memory object is freed, ASan poisons the shadow bytes corresponding to the memory region of that object to detect temporal errors. Tech-ASan adopts the shadow memory management approach of ASan, and goes a step further by filling the redzones and the freed memory regions with a magic value, as shown in Figure 4. Considering the allocated memory objects in ASan (including reused ones) inherently come with initial values, unlike the management of shadow memory, the magic value injection process in Tech-ASan only requires poisoning and does not involve explicit unpoisoning. As a result, the additional runtime overhead introduced by magic value injection is minimal.

Memory safety check. Similar to [7, 16, 24, 34], Tech-ASan instruments memory access instructions and functions at compile time to check for memory safety violations. First, for all functions in the program, Tech-ASan sequentially traverses all instructions to identify *the interesting memory access instructions*, such as Load instructions and Store instructions. For fair comparison, we ensure the coverage of the interesting memory access instructions exactly matches both ASan and ASan--. Then, for each identified memory access instruction, we insert a two-stage checking logic (cf. Section 3.2). Additionally, we design an optimizer to eliminate redundant checks (cf. Section 3.3).

Second, Tech-ASan replaces standard library functions in C/C++ that may trigger memory safety violations, such as `memcpy()` and `strcat()`, with customized versions. Notably, existing methods [16, 24, 34], including ASan in the latest version of LLVM (LLVM 20.1.0)[3], overlook functions that operate on `wchar_t` strings, such as `wcscpy()`. Considering that memory errors can also occur in these functions [1], Tech-ASan supports detecting them to improve detection capability. In addition, Tech-ASan replaces functions and operators such as `free()`, `munmap()`, `delete`, and `delete[]` with the corresponding customized functions to detect double-free. The checking logic within the aforementioned customized functions remains consistent with that of ASan and ASan--.

3.2 Two-stage Check

Tech-ASan utilizes a two-stage checker to implement the fast check for all interesting memory accesses, which consists of a fast check stage and a slow check stage. In the fast check stage, Tech-ASan checks whether the value of the accessed memory is equal to a predefined `N`-byte constant magic value `MAGIC_VALUE_N`, where `N` is 1, 2, 4, or 8. If not, the access is considered valid, and program execution continues as normal. If they are equal, the access is potentially located in a freed memory region or a redzone. However, since the program itself may legitimately use the same magic value, the access cannot be immediately classified as an invalid access. In such cases, Tech-ASan proceeds to the slow check stage for further verification to eliminate false positives. In the slow check stage,

[3]https://github.com/llvm/llvm-project/releases/tag/llvmorg-20.1.0

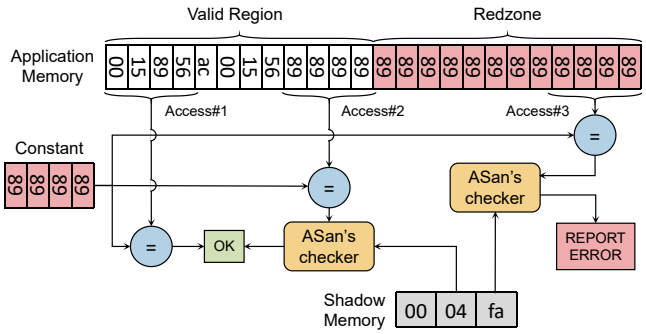

Figure 4: Examples of two-stage checks

Tech-ASan adopts the same check logic as ASan, verifying the validity of the access by checking the corresponding shadow byte. Figure 4 presents three cases when using the two-stage checking of Tech-ASan:

- Access#1 (valid): The value at the accessed location is not a magic value, so the fast check stage determines it as a valid access.
- Access#2 (valid): The value at the accessed location is a magic value, causing the fast check stage to flag it as potentially invalid and triggering the slow check stage. The slow stage, which uses ASan's native checker, reads the shadow byte (`0x04`) and confirms it as a valid access.
- Access#3 (invalid): The fast stage first detects a magic number at the accessed memory location, and the slow check stage further identifies a negative shadow byte, confirming it as an invalid memory access.

This two-stage checker ensures efficient detection capability of memory violations while efficiently reducing runtime overhead.

To further reduce the runtime overhead of Tech-ASan, we adopt different strategies for checking store and load instructions. For store instructions accessing 1-, 2-, 4-, or 8-byte memory regions, Tech-ASan inserts the following checking logic **before** the instruction:

```
1 if (*p == MAGIC_VALUE_N)          // Fast check
2     asan_check(p);                // Slow check
3 *p = 1;                    // A store instruction
```

Only by a combination of load and compare instructions (Line 1) can Tech-ASan check a store instruction (Line 3) in most cases, i.e., when the slow check is not triggered. Since the load instruction (Line 1) and the store instruction (Line 3) access the same address, no additional cache miss is introduced. Our statistical analysis of SPEC CPU2006 benchmarks confirms that 85% of Tech-ASan's interesting memory access instructions target load instructions. Although the above check logic is efficient, the `*p` appears twice, resulting in one additional memory access compared to the original program. For load instructions accessing 1-, 2-, 4-, or 8-byte memory regions, Tech-ASan inserts the checking logic **after** the instruction:

```
1 foo = *p;                  // A load instruction
2 if (foo == MAGIC_VALUE_N)         // Fast check
3     asan_check(p);                // Slow check
```

In Line 2, we reuse the result of `*p` to avoid introducing additional memory access overhead. Note that if we do not explicitly reuse the result of `*p` when inserting check logic, LLVM's backend does not automatically reuse it. Therefore, unlike recent methods [7, 34] that require inserting an extra load instruction to read metadata from memory before the origin load instruction, we directly reuse the value from the origin load instruction, saving a load instruction.

The two-stage checker design of Tech-ASan is based on two fundamental observations. First, the majority of memory accesses are valid in real-world software testing. Second, although ASan's per-access shadow memory validation is accurate, it can reduce cache hit rates and degrade overall performance. Guided by the principle of locality in program execution, Tech-ASan's design philosophy argues that it is unnecessary to perform slow checks for every memory access as ASan does. For most valid accesses, executing only the fast-stage check is sufficient to efficiently filter out valid accesses and significantly accelerate checks. For suspicious accesses, Tech-ASan relies on slow-stage shadow memory checks to eliminate false positives, thus achieving a balance between runtime performance and detection capability.

Due to the low probability of triggering the slow check, Tech-ASan implements the slow checker as a function call rather than an inline instruction version. This is because the overhead of a function call is acceptable in this context. Additionally, this approach helps reduce the size of the binary file, further improving the usability of Tech-ASan.

3.3 Optimization

Since there is no need to instrument on all memory accesses, Tech-ASan's optimizer is designed to eliminate redundant checks. Among the most advanced redundant check elimination techniques, the most basic and important idea is *removing unsatisfiable checks* on stack and global memory objects. Let us take the code in Listing 1 as an example. ASan's native code has a basic implementation. For the memory access in Line 4, ASan determines that the offset of the accessed location is less than the length of the memory object, ensuring that a buffer overflow cannot occur. As a result, it removes the check in Line 3. Based on this philosophy, ASan-- has further optimizations: For the memory access in Line 7, ASan-- uses data flow analysis to determine that the access in Line 7 is only triggered when the condition in Line 5, i.e., `i < 20`, is satisfied. Since the condition in Line 5 already ensures that the access in Line 7 cannot cause a buffer overflow, the check in Line 6 is removed.

Listing 1: An example of redundancy elimination

```
1 int buf[20];
2 unsigned int i = input();
3 sanitizer_check(buf+10);      // Removed by ASan
4 buf[10] = 0;
5 if (i < 20) {
6     sanitizer_check(buf+i);// Removed by ASan--
7     buf[i] = 0;
8 }
9 for (unsigned int j = 0; j < 20; j++) {
10    sanitizer_check(buf+j);    // Removed by us
```

```
11      buf[j] = input();
12      sanitizer_check(buf);       // Removed by us
13      buf[0] = 1;
14  }
```

However, removing unsatisfiable checks of ASan-- becomes ineffective in loops (Lines 11 and 13) because its data flow analysis struggles to handle PHI nodes in LLVM IR adequately. The PHI node is an IR instruction that selects incoming values from the different predecessor basic blocks in the form of static single assignment (SSA) rule. However, PHI nodes will also be obstacles for data flow analysis. Although ASan-- introduces specific extra optimizations targeting loop scenarios to mitigate this issue, it inevitably adds new operations. As a result, in our tests, the benefits are not significant. As highlighted in [34], checks in loops account for about 45% of the overhead introduced by all ASan checks. Therefore, safely eliminating redundant checks inside loops remains an important and unresolved problem.

Tech-ASan first integrates the unsatisfiable check removal techniques from both ASan and ASan--. Then, for removing Lines 10 and 12 in Listing 1, Algorithm 1 is proposed to eliminate redundant checks inside loop bodies via static analysis, i.e., checks for $a[index]$, where *index* is a variable or a constant and a is the base address of a stack or global object. Here we define the size of the memory object as *size* and a constant as c. A memory access m_i is safe (thus satisfying check elimination) if one of the following conditions holds (Lines 1-15):

(1) If the index is a constant, the analysis finds *index* is always within a safe range $[0, c]$ where $0 \leq c < size$.
(2) If the index is a variable, not only does *index* need to satisfy the safe range as above, but also the memory access instruction m_i dominates the compare instruction *cmp* or *cmp* dominates m_i.

Due to loop semantics, *index* remains constrained by the initial index value and the compare instructions served as the loop termination condition. If the initial index value is in bounds, even if the access instruction m_i dominates the compare instruction *cmp* in control flow, in the subsequent iteration, the access instruction m_{i+1} is still bounded by the compare instruction *cmp* of the previous iteration (Line 13).

Algorithm 1 scans each memory access instruction m_i in the function $\mathcal{F}$ to find its corresponding getelementptr instruction *gep* (Lines 18-22). Each getelementptr instruction *gep* consists of a base address and offset indexes, where each index g_j needs to be found whether it is in bounds (Lines 24-39). For each g_j:

(1) If g_j comes from a PHI node p, we recognize it as an index that changes during loop iterations. For such cases, we must check all incoming values v_k from different predecessor basic blocks. The access is only considered safe when every incoming value v_k stays within the permitted range (Lines 26-34).
(2) If g_j does not come from a PHI node (Lines 35-37), i.e., g_j is not relevant to loop iterations. In this case, we still need to confirm that g_j remains within the safe range throughout execution.

Algorithm 1: Removing redundant checks in loop

```
1  Procedure is_safe_access(m_i, index, size)
2      if index is a constant then
3          if ¬(0 ≤ index < size) then
4              return false;
5          return true;
6      for each u_q ∈ index.users() do
7          if ¬(u_q is a compare instruction) then
8              continue;
9          cmp ← dyncast_to_compare_instruction(u_q);
10         c ← get_constant(cmp);
11         if ¬(cmp indicates "index < c") ∨¬(0 ≤ c < size) then
12             continue;
13         if m_i dominates cmp ∨ cmp dominates m_i then
14             return true;
15     return false
16 Algorithm
       Input: An LLVM IR function F
       Output: A safe memory access instruction set R
17     R ← φ;
18     M ← find_all_interesting_memory_access(F);
19     for each memory access instruction m_i ∈ M do
20         if get_loop_depth(m_i) ≠ 1 then
21             continue;
22         gep ← find_getelementptr_instruction(m_i);
23         size ← get_access_object_size(gep);
24         all_g_j_is_safe ← true ;
25         for each g_j ∈ gep.indexes() do
26             if g_j comes from phi node p then
27                 all_v_k_is_safe ← true ;
28                 for each v_k ∈ p.incoming_values() do
29                     if ¬ is_safe_access (m_i, v_k, size) then
30                         all_v_k_is_safe ← false ;
31                         break;
32                 if ¬all_v_k_is_safe then
33                     all_g_j_is_safe ← false ;
34                     break;
35             else if ¬is_safe_access (m_i, g_j, size) then
36                 all_g_j_is_safe ← false ;
37                 break;
38         if all_g_j_is_safe then
39             R ∪ {m_i};
40     return R;
```

Due to the trade-off between compilation performance and runtime performance, Algorithm 1 only removes redundant checks in single-level loops (Line 20). Despite this limitation, our evaluation on a loop-intensive benchmark gcc-loops[4] shows our optimizer with Algorithm 1 eliminates 18% of checks in loops. When the two-stage check is disabled, Tech-ASan using only the optimizer introduces 102% runtime overhead on gcc-loops, compared to 144% for ASan--. This demonstrates the efficiency of Algorithm 1.

For checks in loops that access contiguous memory regions and can not be optimized by Algorithm 1, we explore an alternative optimization approach. We first leverage LLVM's built-in Scalar

[4]https://github.com/llvm/llvm-test-suite/blob/main/SingleSource/UnitTests/Vectorizer/gcc-loops.cpp

Evolution (SCEV) analysis to infer the start addresses and the sizes of memory accesses within the loops. Then we insert ASan's optimized region-checking function `__asan_region_is_poisoned()` at the exit block of the loop. This technique demonstrates significant speedups in benchmarks where large arrays are accessed iteratively. However, in SPEC CPU2006 [10], measurable performance gains are observed only in a small subset of programs due to two key limitations: First, `__asan_region_is_poisoned()` incurs expensive calling costs, which outweighs its benefits when the accessed contiguous memory regions are short. Second, for loops with variable iterations, the length of continuous accesses cannot be statically determined at compile time. Since many loops in SPEC CPU2006 operate on short memory regions, this approach introduces overhead rather than optimization. Given these trade-offs, we remove this optimization in our final implementation.

In addition, we successfully integrate the other SOTA redundant check elimination techniques into the optimizer of Tech-ASan, including *removing recurring checks* and *optimizing neighbor checks* [7, 34]. First, removing recurring checks identifies checks sharing the same memory location and access size in each function using LLVM built-in alias-analysis, if they either all happen or all do not happen, we only need to retain a single check. Second, optimizing neighbor checks is to merge or remove adjacent checks. When checks either both happen or both do not happen, there are two situations that can be optimized: (1) If two memory accesses can fall into an 8-byte region, their checks can be merged into a single check. (2) Given three memory accesses, i.e., (`addr1`, `size1`), (`addr2`, `size2`), and (`addr3`, `size3`), where `addr1 < addr2 < addr3`. Tech-ASan's check for the second access, i.e., (`addr2`, `size2`), can be safely eliminated if `addr3 - addr1 < MinRdSz` and `addr2 + size2 ≤ addr3 + size3`, where `MinRdSz` is the minimal size of a redzone.

3.4 Implementation

We have implemented Tech-ASan on top of the ASan's infrastructure in the LLVM compiler [12]. Compared to the native ASan, our implementation introduces a total of 2.4K additional lines of code. The use of Tech-ASan is identical to ASan because Tech-ASan is implemented to share the same assumptions, requirements, and interfaces with ASan. Our design is fully generalizable to other compilers. We have not added any architecture-specific code, allowing Tech-ASan to run on any machine with instruction set architectures supported by LLVM's backend. Therefore, Tech-ASan maintains ASan's usability.

As demonstrated in our experiments, Tech-ASan does not compromise the detection capabilities of ASan while reducing the size of the binaries compiled with ASan. The additional compilation time introduced by Tech-ASan is also within an acceptable range.

4 EVALUATION

We experimentally evaluate Tech-ASan on the following research questions (RQs):

- **RQ1:** Can Tech-ASan maintain the detection capability of ASan?
- **RQ2:** Can Tech-ASan reduce the runtime overhead of ASan?
- **RQ3:** How does Tech-ASan impact compilation time and binary size compared to ASan?

By inheriting ASan's shadow memory model, Tech-ASan incurs no additional memory overhead, therefore, comparative memory analysis is not needed. When evaluating the detection capability of Tech-ASan, all compiler optimizations are disabled to prevent vulnerabilities from being suppressed by optimization passes. For other measurements, the optimization option `O3` is enabled by default. Unless otherwise specified, all the experiments are conducted on an x86-64 server running Ubuntu 18.04, equipped with an AMD EPYC 7702 192-core 2.0GHz CPU and 128GB RAM.

4.1 RQ1: Detection Capability

To measure the detection capability of Tech-ASan, we conduct two experiments. The first experiment is conducted on the latest version of the Juliet Test Suite (version 1.3) [1], where all CWEs have both buggy and non-buggy testcases, which are used to test the false negatives and false positives of the sanitizers, respectively. However, there are 4 types of testcases that are not suitable for evaluation:

(1) Testcases that wait for an external signal, e.g., sockets. We just remove them to avoid waiting infinitely.
(2) Testcases with violations that are triggered depend on a random number. We change them to the non-random versions for a deterministic result.
(3) Testcases that print a string without the terminating character, which produces random overflow. We remove them for a deterministic result.
(4) Testcases with violations that are only triggered on 32-bit systems but are not triggered on 64-bit systems.

After adjustments, the number of remaining testcases in each CWE type is still much more than that used in recent works [7, 16, 34] for the more comprehensive evaluation. The adjusted Juliet Test Suite has been published in the community for future science.[5]

For comparison, we further run recent available methods ASan-- [34], GiantSan [16], and FloatZone [7] on the Juliet Test Suite. Note that FloatZone is executed on a 64-bit server equipped with an Intel CPU since its public version can be run on Intel CPUs only. Table 1 shows that only Tech-ASan has no false positive and false negative issues. On buggy testcases, ASan, ASan--, GiantSan, and FloatZone can not detect 56 memory safety violations that occur in `wcscpy()`, which remains unsolved even in the latest version of ASan. GiantSan generates an additional 913 false negatives, due to the insufficient check on the stack arrays, failing to report an error when an out-of-bounds write occurs to a stack array within a loop. FloatZone generates 283 false negatives due to the following reasons: (1) In testcases containing both partial and full out-of-bounds (OOB) accesses, partial OOB operations corrupt the redzone headers. This corruption invalidates FloatZone's check logic that depends on a complete redzone, consequently failing to detect subsequent full OOB accesses. (2) In some testcases, FloatZone fails to insert redzones for memory objects when allocating via C standard library functions, e.g., `malloc()`.

[5] https://github.com/Hufffman/Adjusted-Juliet-Test-Suite

Table 1: Detection capability on the adjusted Juliet Test Suite with buggy and non-buggy testcases

CWE & Type	Total	Buggy testcases					Non-buggy testcases				
		Tech-ASan	ASan	ASan--	GiantSan	FloatZone	Tech-ASan	ASan	ASan--	GiantSan	FloatZone
121: stack-buffer-overflow	2956	2956	2948	2948	2611	2822	2956	2956	2956	2956	2908
122: heap-buffer-overflow	3438	3438	3390	3390	3008	3337	3438	3438	3438	3438	3366
124: buffer-underwrite	1024	1024	1024	1024	1021	976	1024	1024	1024	1024	1024
126: buffer-overread	672	672	672	672	493	672	672	672	672	672	672
415: double-free	818	818	818	818	818	818	818	818	818	818	818
416: use-after-free	393	393	393	393	381	393	393	393	393	393	393
Total	**9301**	**9301**	**9245**	**9245**	**8332**	**9018**	**9301**	**9301**	**9301**	**9301**	**9181**

Table 2: Detection capability for real-world memory safety violations from CVE. ✓, ×, and - represent that a violation is detected, a violation is not detected, the program can not be instrumented or executed normally, respectively.

Program	Version	LoC (k)	Vulnerabilities	Vulnerability Type	Tech-ASan	ASan	ASan--	GiantSan
binutils	2.15	1456.9	CVE-2006-2362	stack-buffer-overflow	✓	✓	✓	✓
binutils	2.29	3888.6	CVE-2018-9138	stack-overflow	✓	✓	✓	✓
fcron	3.0.0	35.7	CVE-2006-0539	heap-buffer-overflow	✓	✓	✓	✓
fig2dev	3.2.7b	49.7	CVE-2020-21675	stack-buffer-overflow	✓	✓	✓	-
GraphicsMagick	1.3.26	476.8	CVE-2017-12937	heap-buffer-overflow	✓	✓	✓	✓
GoHttp		0.5	CVE-2019-12160	heap-use-after-free	✓	✓	✓	✓
libpng	1.6.37	94.1	CVE-2021-4214	heap-buffer-overflow	✓	✓	✓	×
libtiff	3.8.0	113.5	CVE-2006-2025	Integer-overflow	✓	✓	✓	✓
libtiff	3.8.2	112.4	CVE-2009-2285	heap-buffer-overflow	✓	✓	✓	✓
libtiff	4.0.1	126.8	CVE-2013-4243	heap-buffer-overflow	✓	✓	✓	×
			CVE-2015-8668	heap-buffer-overflow	✓	✓	✓	✓
libzip	1.2.0	48.1	CVE-2017-12858	heap-use-after-free	✓	✓	✓	✓
lua	5.4.3	31.7	CVE-2021-44964	heap-use-after-free	✓	✓	✓	✓
mp3gain	1.5.2	9.1	CVE-2017-14407	stack-buffer-overflow	✓	✓	✓	×
			CVE-2017-14408	stack-buffer-overflow	✓	✓	✓	✓
			CVE-2017-14409	global-buffer-overflow	✓	✓	✓	×
mxml	2.12	27.8	CVE-2018-20004	stack-buffer-overflow	✓	✓	✓	✓
nasm	2.15.04rc3	155.0	CVE-2020-24978	double-free	✓	✓	✓	✓
python	3.1.5	692.0	CVE-2014-1912	heap-use-after-free	✓	✓	✓	✓
yasm	1.3.0	164.9	CVE-2021-33468	heap-use-after-free	✓	✓	✓	✓
Total		**7483.6**	**20**		**20**	**20**	**20**	**15**

On non-buggy testcases, Tech-ASan, ASan, ASan--, and GiantSan have no false positive issue. However, FloatZone produces 120 false positives in buffer overflow detection due to erroneous reports when programs access residual redzones in reused stack memory regions.

In the second experiment, we select real-world memory-related memory safety violations from common vulnerabilities and exposures (CVEs) [29], including 20 CVEs with known proof-of-concepts (PoCs) from 14 projects written in C/C++. Since Table 1 shows that FloatZone may produce false positives, we exclude it to avoid uncertainty in confirming whether a vulnerability is actually detected. Table 2 shows that ASan, ASan--, and Tech-ASan detect all the CVEs. However, GiantSan fails to detect four known CVEs, and triggers an assertion failure, leading to a program crash during the reproduction of an additional CVE.

In summary, Tech-ASan preserves ASan's detection capability while accelerating and eliminating sanitizer checks.

4.2 RQ2: Runtime Overhead

Following the recent studies [6, 7, 33, 34], we use the classic version of the industry-standard CPU-intensive runtime benchmark suite, SPEC CPU2006 [10], to evaluate the performance improvement of Tech-ASan thoroughly. Since some of the programs in SPEC CPU2006 contain memory safety violations, all programs are compiled with `-fsanitize-recover=address` and run after setting the environment variable `ASAN_OPTIONS` to `"halt_on_error=0"`, which ensures that sanitizers continue execution after detecting a violation, rather than halting the program. By analyzing the logs, we confirm that Tech-ASan correctly detects violations without any false positives. To mitigate accidental errors, we calculate the median of the 10 running times.

Table 3 presents the runtime performance of the SPEC CPU2006 benchmark. Tech-ASan, ASan, ASan--, and GiantSan introduce 64.0%, 97.7%, 81.89%, and 62.64% runtime overhead compared to the vanilla clang. Tech-ASan reduces ASan's runtime overhead by

Table 3: Runtime overhead on SPEC CPU2006 Benchmark (seconds). Ratio *R* is computed as the execution time of sanitizer-enabled binaries divided by that of non-instrumented vanilla clang binaries. The "-" indicates abnormal program termination during testing.

Programs	Performance Study									Ablation Study	
	Vanilla Clang	Tech-ASan	*R*	ASan	*R*	ASan--	*R*	GiantSan	*R*	Tech-ASan^{ac}	*R*
400.perlbench	262	1415	540.08%	1600	610.69%	1390	530.53%	1440	549.62%	1390	530.53%
401.bzip2	393	502	127.61%	623	158.52%	588	149.62%	552	140.46%	524	133.21%
403.gcc	242	1145	473.14%	959	396.28%	929	383.88%	857	354.13%	1180	487.60%
429.mcf	390	553	141.67%	738	189.23%	725	185.90%	662	169.74%	568	145.51%
445.gobmk	358	493	137.71%	625	174.58%	547	152.79%	664	185.47%	497	138.83%
456.hmmer	348	429	123.28%	646	185.63%	568	163.22%	400	114.94%	475	136.35%
458.sjeng	489	707	144.48%	898	183.64%	742	151.74%	-	-	721	147.34%
462.libquantum	513	622	121.25%	672	130.99%	682	132.94%	603	117.54%	631	122.90%
473.astar	448	559	124.67%	664	148.21%	595	132.81%	487	108.71%	569	127.01%
483.xalancbmk	823	1525	185.30%	1740	211.42%	1530	185.91%	1380	167.68%	1545	187.73%
433.milc	537	807	150.19%	1125	209.50%	976	181.75%	798	148.60%	835	155.40%
444.namd	320	379	118.44%	518	161.88%	530	165.63%	374	116.88%	394	122.97%
447.dealII	1100	1520	138.18%	1770	160.91%	1720	156.36%	1520	138.18%	1540	140.00%
450.soplex	318	466	146.38%	517	162.58%	491	154.40%	457	143.71%	479	150.63%
453.povray	179	451	251.96%	544	303.91%	453	253.07%	452	252.51%	452	252.23%
470.lbm	242	264	109.09%	322	133.06%	289	119.42%	257	106.20%	279	115.29%
482.sphinx3	364	521	143.13%	616	169.23%	615	168.96%	445	122.25%	508	139.56%
geomean			**164.00%**		**197.70%**		**181.89%**		**162.64%**		**167.91%**

33.7%, and outperforms ASan-- by 17.89%, demonstrating the effectiveness of Tech-ASan's two-stage check and redundant check elimination techniques. GiantSan introduces the lowest runtime overhead, due to its operation-level instrumentation and history caching mechanism. Although Tech-ASan has 1.36% higher overhead than GiantSan, it remains within an acceptable range due to its detection capability. Specifically, Tech-ASan outperforms GiantSan on 429.mcf and 445.gobmk, because GiantSan's operation-level check merging provides less effectiveness due to non-continuous memory access such as tree and graph. In addition, Tech-ASan exhibits higher runtime overhead than ASan on 403.gcc, resulting from the frequent memory allocation and free, which increases the runtime overhead of magic value injection.

An ablation study is conducted to further clarify the runtime benefits brought by Tech-ASan's check acceleration technique. Let Tech-ASan^{ac} represents that Tech-ASan only enables check acceleration. Table 3 presents that Tech-ASan^{ac} reduces ASan's overhead by 29.79%, which demonstrates that our design of two-stage check is effective to speed up sanitizer checks and reduce the runtime overhead. Tech-ASan achieves 3.91% lower overhead than Tech-ASan^{ac} due to the optimizer. Furthermore, as the overhead of individual sanitizer checks is significantly reduced by the two-stage check mechanism, the efficacy of the optimizer becomes less pronounced in this context.

4.3 RQ3: Usability Analysis

This section evaluates Tech-ASan's binary size expansion and compilation time overhead compared to ASan.

The binary size is a critical factor in the deployment of memory safety tools, especially in environments with limited storage capacity, such as embedded systems, Internet of Things (IoT) devices, and mobile applications. Figure 5 presents the binary sizes of programs in the SPEC CPU2006 benchmark. The average sizes of binary code compiled with Tech-ASan, ASan, ASan--, and GiantSan are 11.64×, 11.78×, 11.52×, and 38.94× those of vanilla clang, respectively. ASan-- eliminates partial redundant checks and introduces minimal additional code for its "loop check optimization" feature, thereby reducing binary size, which is consistent with observations in [34]. Compared to ASan and ASan--, although Tech-ASan introduces additional code for redzone management and replaces ASan's native check logic with a two-stage check logic, it still reduces the average binary size by 14.79% comparable to ASan. This improvement is due to two reasons: First, Tech-ASan implements second stage checks through function calls rather than inline instruction sequences. Second, Tech-ASan's optimizer incorporates a novel algorithm for eliminating redundant checks within loops. GiantSan's implementation of complicated operation-level protection and shadow memory encoding mechanisms significantly increases instrumented code size.

Compilation efficiency is also a critical factor in the adoption of memory safety tools, especially in large-scale projects where frequent recompilation is required. To ensure statistically accurate compilation time measurements, we employ single-threaded compilation throughout the experiments. Table 4 presents the compile time of programs in the SPEC CPU2006 benchmark. Tech-ASan, ASan, ASan--, and GiantSan increase compile time by 29.52%, 23.50%, 33.46%, and 120.93% compared to vanilla clang, respectively. Compared to vanilla clang, the additional compilation time mainly comes from two sources: instrumentation and analysis. Compared to ASan, ASan-- employs extensive static analysis for which checks

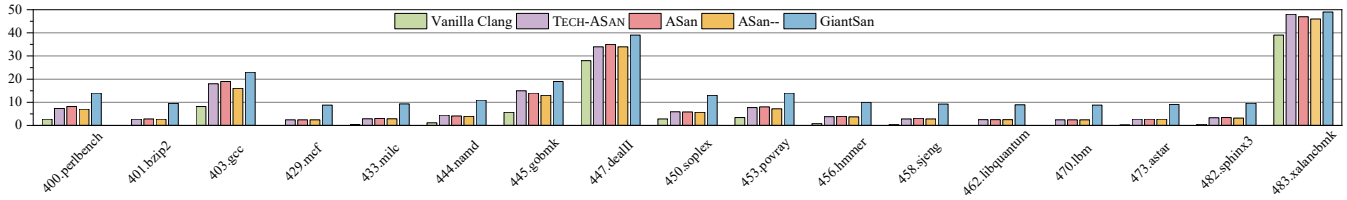

Figure 5: The binary sizes of programs in SPEC CPU2006 benchmark (MB)

Table 4: Compile-time overhead on SPEC CPU2006, each number is the ratio compared to the native compilation.

Programs	TECH-ASAN	ASan	ASan--	GiantSan
400.perlbench	286.11%	159.15%	287.12%	657.81%
401.bzip2	157.87%	130.64%	135.96%	3640.85%
403.gcc	169.70%	162.17%	189.15%	318.48%
429.mcf	105.43%	103.70%	107.16%	118.27%
433.milc	113.63%	114.24%	114.09%	133.69%
444.namd	162.85%	136.83%	182.06%	224.34%
445.gobmk	133.10%	148.38%	152.37%	227.93%
447.dealII	120.48%	118.04%	121.78%	205.34%
450.soplex	112.97%	111.77%	117.97%	172.03%
453.povray	125.92%	134.08%	133.41%	206.49%
456.hmmer	128.27%	132.82%	134.81%	177.72%
458.sjeng	108.56%	122.46%	114.62%	142.07%
462.libquantum	110.26%	104.77%	105.25%	127.92%
470.lbm	112.67%	101.38%	108.54%	125.07%
473.astar	108.05%	109.32%	113.35%	125.64%
482.sphinx3	112.75%	117.00%	121.53%	174.79%
483.xalancbmk	112.95%	114.09%	115.89%	178.31%
geomean	**129.52%**	**123.50%**	**133.46%**	**220.93%**

can be safely eliminated, consequently introducing additional compilation overhead. In contrast, TECH-ASAN utilizes a more lightweight loop analysis approach, resulting in a 3.94% reduction in compilation time relative to ASan--. GiantSan incurs significant analysis overhead to merge multiple instruction-level instrumentations into an operation-level instrumentation, resulting in a longer compilation time compared to others. Notably, check elimination can reduce compilation time in certain cases. For instance, both TECH-ASAN and ASan-- compile 458.sjeng faster than ASan. This occurs because the analysis overhead is offset by reduced instrumentation costs and decreased pressure on the compiler backend. Overall, TECH-ASAN introduces only 29.52% compile-time overhead on SPEC CPU2006, which represents an acceptable cost for large-scale software compilation.

5 RELATED WORK

This section reviews the related work on mitigating and detecting memory safety violations in C/C++ programs at runtime. Existing solutions can be categorized into *location-based* methods [3, 7, 16, 17, 20, 24, 34] and *pointer-based* methods [4, 5, 8, 9, 11, 14, 18, 19, 23, 26].

5.1 Location-based Methods

Location-based methods model the memory with a focus on memory bytes by recording which byte is addressable at runtime. In general, these methods use canary values or shadow bytes to mark allocated memory as addressable and unallocated or freed memory as unaddressable. Location-based methods are widely deployed for their high compatibility.

Memcheck [20] and Dr. Memory [3] utilize shadow memory to track the state of each memory byte. Both tools incur more than a 10× runtime overhead due to the use of a dynamic binary instrumentation framework [2, 21]. ASan [24], the SOTA location-based method, instruments the tested programs at compile-time and leverages a compact state encoding in shadow memory to record which byte is addressable at runtime. DoubleTake [17] uses canary values to mark unaddressable memory locations and divides program execution into multiple epochs. It checks whether the canary value has been modified at the end of each epoch to determine whether a memory safety violation has occurred. However, DoubleTake does not provide adequate protection for freed memory objects, and the canary-based mechanism can only detect write vulnerabilities, but not read vulnerabilities. ASan-- [34] reduces ASan's redundant checks at compile time through static analysis, provided safety is guaranteed. FloatZone [7] leverages the floating-point unit (FPU) in the CPU to speed up sanitizer checks, but inevitably introduces false positives. GiantSan [16] proposes a shadow encoding with segment folding to increase the protection density and introduces operation-level protection to accelerate the sanitizer checks.

However, a prior study [9] shows that location-based methods may miss use-after-free vulnerabilities. To mitigate this, a commonly adopted solution is to quarantine freed heap objects in a queue. Once the queue is full, the oldest objects are popped and reallocated to the program, which can still lead to false negatives. Fortunately, few reports related to false negatives exist in practice due to the low probability of bypassing the quarantine queue [16].

5.2 Pointer-based Methods

Pointer-based methods model memory with a focus on pointers by tracking which memory regions are safe for each pointer to access.

SoftBound+CETS [18, 19] ensures spatial and temporal safety through pointer-based bounds checking and identifier metadata associated with pointers. However, SoftBound+CETS introduces high runtime overheads due to expensive metadata propagation and complex logic for checking. EffectiveSan [5] enforces type and memory safety using a combination of low-fat pointers, type metadata, and type/bounds check instrumentation. EffectiveSan can detect type and sub-object bounds errors through dynamic type

checking, but its temporal safety protection is not as comprehensive as that of CETS. CAMP [15] only protects heap memory, validating the memory access of each pointer with boundary checking as well as escape tracking. It also neutralizes dangling pointers with a customized seglist allocator that tracks memory ranges for each allocation. Safe Sulong [23] compiles C/C++ programs into Java byte code, leveraging the Java virtual machine to take over memory safety detection. Oscar [4] utilizes page aliases and page-level permissions to achieve heap use-after-free protection. Building upon the philosophy of Oscar, DangZero [8] utilizes a privileged backend to mark reserved bits in page tables to implement heap use-after-free detection. HWASan [26] uses ARM's Top Byte Ignore (TBI) feature to embed an address tag into the top byte of each pointer to identify a memory region. This tag is implicitly propagated to subsequent pointers during pointer assignments. Every load and store instruction raises an exception on a mismatch between the address tag and the memory tag. HWASanIO [11] adds support for identifying sub-objects based on HWASan, but it incurs significant additional runtime and memory overhead. Similarly, to avoid the extra overhead of tracking pointers, PACMem [14] seals the metadata into pointers with ARM's Pointer Authentication (PA) feature.

6 CONCLUSIONS

ASan has become the most popular solution for detecting memory safety violations in C/C++ programs during execution, but it imposes significant runtime overhead. Existing methods for speeding up ASan either fail to adequately eliminate redundant checks or compromise detection capability. To address this issue, this paper presents Tech-ASan, which leverages a novel two-stage check mechanism to effectively reduce ASan's runtime overhead. We also design an efficient optimizer to eliminate redundant checks, which integrates a novel algorithm for removing checks in loops. Evaluation on the SPEC CPU2006 benchmark demonstrates that Tech-ASan achieves remarkable improvements, reducing runtime overhead by 33.70% and 17.89% compared to ASan and ASan--, respectively. Moreover, under the same redzone setting, Tech-ASan detects 56 fewer false negative cases than ASan and ASan-- when testing on the Juliet Test Suite.

ACKNOWLEDGMENTS

The authors would like to appreciate the anonymous reviewers for their feedback, and we thank the constructive suggestions from Yuchen Zhang, Alexander Potapenko, Xinyue Zhou, Cheng Wen, and Jiadong Peng. This project is supported by the Shenzhen Science and Technology Foundation (General Program, JCYJ20210324093212034), 2022 Guangdong Province Undergraduate University Quality Engineering Project (Shenzhen University Academic Affairs [2022] No. 7), Science and Technology R&D Program of Shenzhen (20220810135520002), Guangdong Province Key Laboratory of Popular High Performance Computers 2017B030314073, and Guangdong Province Engineering Center of China-made High Performance Data Computing System.

REFERENCES

[1] Paul E Black. 2018. *Juliet 1.3 test suite: Changes from 1.2.* US Department of Commerce, National Institute of Standards and Technology.

[2] Derek Bruening and Saman Amarasinghe. 2004. Efficient, transparent, and comprehensive runtime code manipulation. (2004).

[3] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *International Symposium on Code Generation and Optimization (CGO 2011)*. 213–223. https://doi.org/10.1109/CGO.2011.5764689

[4] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2017. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 815–832. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/dang

[5] Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: type and memory error detection using dynamically typed C/C++. *SIGPLAN Not.* 53, 4 (jun 2018), 181–195. https://doi.org/10.1145/3296979.3192388

[6] Rahul George, Mingming Chen, Kaiming Huang, Zhiyun Qian, Thomas La Porta, and Trent Jaeger. 2024. OPTISAN: Using Multiple Spatial Error Defenses to Optimize Stack Memory Protection within a Budget. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 7195–7212. https://www.usenix.org/conference/usenixsecurity24/presentation/george

[7] Floris Gorter, Enrico Barberis, Raphael Isemann, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos. 2023. FloatZone: Accelerating Memory Error Detection using the Floating Point Unit. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 805–822. https://www.usenix.org/conference/usenixsecurity23/presentation/gorter

[8] Floris Gorter, Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2022. DangZero: Efficient Use-After-Free Detection via Direct Page Table Access. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1307–1322. https://doi.org/10.1145/3548606.3560625

[9] Binfa Gui, Wei Song, and Jeff Huang. 2021. UAFSan: an object-identifier-based dynamic approach for detecting use-after-free vulnerabilities. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 309–321. https://doi.org/10.1145/3460319.3464835

[10] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[11] Konrad Hohentanner, Florian Kasten, and Lukas Auer. 2023. HWASanIO: Detecting C/C++ Intra-object Overflows with Memory Shading. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (Orlando, FL, USA) *(SOAP 2023)*. Association for Computing Machinery, New York, NY, USA, 27–33. https://doi.org/10.1145/3589250.3596139

[12] Chris Lattner and Vikram Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[13] Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz. 2018. PartiSan: Fast and Flexible Sanitization via Run-Time Partitioning. In *Research in Attacks, Intrusions, and Defenses*, Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis (Eds.). Springer International Publishing, Cham, 403–422.

[14] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. 2022. PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1901–1915. https://doi.org/10.1145/3548606.3560598

[15] Zhenpeng Lin, Zheng Yu, Ziyi Guo, Simone Campanoni, Peter Dinda, and Xinyu Xing. 2024. CAMP: Compiler and Allocator-based Heap Memory Protection. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 4015–4032. https://www.usenix.org/conference/usenixsecurity24/presentation/lin-zhenpeng

[16] Hao Ling, Heqing Huang, Chengpeng Wang, Yuandao Cai, and Charles Zhang. 2024. 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* (, La Jolla, CA, USA,) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 433–449. https://doi.org/10.1145/3620665.3640391

[17] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2016. DoubleTake: fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 911–922. https://doi.org/10.1145/2884781.2884784

[18] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for c. *SIGPLAN Not.* 44, 6 (June 2009), 245–258. https://doi.org/10.1145/1543135.1542504

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

[20] Nicholas Nethercote and Julian Seward. 2007. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments* (San Diego, California, USA) *(VEE '07)*. Association for Computing Machinery, New York, NY, USA, 65–74. https://doi.org/10.1145/1254810.1254820

[21] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *SIGPLAN Not.* 42, 6 (jun 2007), 89–100. https://doi.org/10.1145/1273442.1250746

[22] Arvind S Raj, Wil Gibbs, Fangzhou Dong, Jayakrishna Menon Vadayath, Michael Tompkins, Steven Wirsz, Yibo Liu, Zhenghao Hu, Chang Zhu, Gokulkrishna Praveen Menon, Brendan Dolan-Gavitt, Adam Doupé, Ruoyu Wang, Yan Shoshitaishvili, and Tiffany Bao. 2024. Fuzz to the Future: Uncovering Occluded Future Vulnerabilities via Robust Fuzzing. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 3719–3733. https://doi.org/10.1145/3658644.3690278

[23] Manuel Rigger, Roland Schatz, René Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. 2018. Sulong, and Thanks for All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model. *SIGPLAN Not.* 53, 2 (mar 2018), 377–391. https://doi.org/10.1145/3296957.3173174

[24] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[25] Kostya Serebryany, Chris Kennelly, Mitch Phillips, Matt Denton, Marco Elver, Alexander Potapenko, Matt Morehouse, Vlad Tsyrklevich, Christian Holler, Julian Lettner, David Kilzer, and Lander Brandt. 2024. GWP-ASan: Sampling-Based Detection of Memory-Safety Bugs in Production. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice* (Lisbon, Portugal) *(ICSE-SEIP '24)*. Association for Computing Machinery, New York, NY, USA, 168–177. https://doi.org/10.1145/3639477.3640328

[26] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and how it improves C/C++ memory safety. *CoRR* abs/1802.09517 (2018). arXiv:1802.09517 http://arxiv.org/abs/1802.09517

[27] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*. 48–62. https://doi.org/10.1109/SP.2013.13

[28] Emanuel Vintila, Philipp Zieris, and Julian Horsch. 2025. Evaluating the Effectiveness of Memory Safety Sanitizers . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 88–88. https://doi.org/10.1109/SP61157.2025.00088

[29] Common Vulnerabilities. 2005. Common vulnerabilities and exposures. *The MITRE Corporation,[online] Available: https://cve. mitre. org/index. html* (2005).

[30] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. 2015. High System-Code Security with Low Overhead. In *2015 IEEE Symposium on Security and Privacy*. 866–879. https://doi.org/10.1109/SP.2015.58

[31] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. 2017. Bunshin: Compositing Security Mechanisms through Diversification. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 271–283. https://www.usenix.org/conference/atc17/technical-sessions/presentation/xu-meng

[32] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. 2021. SanRazor: Reducing Redundant Sanitizer Checks in C/C++ Programs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 479–494. https://www.usenix.org/conference/osdi21/presentation/zhang

[33] Yiyu Zhang, Tianyi Liu, Zewen Sun, Zhe Chen, Xuandong Li, and Zhiqiang Zuo. 2023. Catamaran: Low-Overhead Memory Safety Enforcement via Parallel Acceleration. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 816–828. https://doi.org/10.1145/3597926.3598098

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