Debloating Address Sanitizer

Yuchen Zhang[†] *Chengbin Pang*[†‡*] *Georgios Portokalidis*[†] *Nikos Triandopoulos*[†] *Jun Xu*[†]
[†]*Stevens Institute of Technology* [‡]*Nanjing University*

Abstract

Address Sanitizer (ASan) is a powerful memory error detector. It can detect various errors ranging from spatial issues like out-of-bound accesses to temporal issues like use-after-free. However, ASan has the major drawback of high runtime overhead. With every functionality enabled, ASan incurs an overhead of more than **1x**.

This paper first presents a study to dissect the operations of ASan and inspects the primary sources of its runtime overhead. The study unveils (or confirms) that the high overhead is mainly caused by the extensive sanitizer checks on memory accesses. Inspired by the study, the paper proposes ASan--, a tool assembling a group of optimizations to reduce (or "*debloat*") sanitizer checks and improve ASan's efficiency. Unlike existing tools that remove sanitizer checks with harm to the capability, scalability, or usability of ASan, ASan-- fully maintains those decent properties of ASan.

Our evaluation shows that ASan-- presents high promise. It reduces the overhead of ASan by 41.7% on SPEC CPU2006 and by 35.7% on Chromium. If only considering the overhead incurred by sanitizer checks, the reduction rates increase to 51.6% on SPEC CPU2006 and 69.6% on Chromium. In the context of fuzzing, ASan-- increases the execution speed of AFL by over 40% and the branch coverage by 5%. Combined with orthogonal, fuzzing-tailored optimizations, ASan-- can speed up AFL by 60% and increase the branch coverage by 9%. Running in Chromium to support our daily work for four weeks, ASan-- did not present major usability issues or significant slowdown and it detected all the bugs we reproduced from previous reports.

1 Introduction

Programs developed in low-level languages, such as C and C++, often contain an abundance of memory error bugs. When exploited, the bugs can lead to severe security issues like data breach and hijacked execution. To help detect memory errors, various runtime tools [10, 14, 27, 36, 37, 42, 44, 47, 48, 50, 53] have been created. These tools vary in many aspects such as scope and capability. In recent years, address sanitizer (ASan) [42] stands out from the rest, thanks to its superior **capability** (detection of a wide spectrum of spatial errors 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).

Technically, ASan allocates shadow memory to track the status (e.g., addressable or not) of the application memory. When the application issues an operation to a memory location, ASan checks the shadow byte mapped to the location and reports any embedded errors. The use of shadow memory and runtime checks leads to ASan's two major drawbacks: high memory overhead and runtime overhead. The drawbacks lock many potentials of ASan, such as whole system sanitization.

Past research has extended many efforts towards addressing ASan's drawbacks. In particular, hardware-assisted address sanitizer (HWASan) [25] utilizes Address Tagging [1], an AArch64 hardware feature, to reduce the use of shadow memory, decreasing the memory overhead of ASan from **1.5x-3x** to **10%-35%** [43]. However, the runtime overhead of ASan remains largely unsolved. Even HWASan still incurs similar overhead to ASan. Past research commonly blames sanitizer checks inserted by ASan for its high overhead. However, ASan has many runtime activities besides sanitizer checks, such as the preparation of metadata and logging. There has not been a systematic understanding of the overhead introduced by different activities. That means we are not even aware of how much overhead the sanitizer checks truly bring.

Study In this paper, we dissect ASan's runtime activities into five groups: *sanitizer checks*, *library function interception*, *poisoning and redzone* (i.e., metadata management), *logging*, and *heap management*. Overhead brought by each group of activities to ASan is individually measured on SPEC CPU2006. Details of the study are presented in §3.1.

Our study confirms that sanitizer checks are the dominant source of overhead. It causes about 80.8% of the overhead (86.5% out of 107.8%). The study also unveils other sources of the overhead. In particular, the customized heap management of ASan incurs an overhead of 9.6%, which is less known before. More details can be found in Figure 3.

Literature Inspired by the study, this paper proposes to reduce ASan's overhead by reducing sanitizer checks. There are many similar attempts [16, 29, 46, 49, 55] in the past. However, the techniques they proposed harm ASan's descent properties (capability, scalability, usability). GWP-ASan [16] randomly picks a sub-set of heap objects to add guard pages and ignores the others. ASAP [49] removes sanitizer checks to satisfy a performance budget. Its criterion is to remove sanitizer checks on "hot" code that is more often executed, offering no safety assurance of the removing operations. Similar to ASAP, PartiSan [29] follows a performance-driven metric to remove sanitizer checks without ensuring safety. GWP-ASan, ASAP, and PartiSan all degrade the capability of ASan. Alternatively, SANRAZOR [55] combines static patterns and dynamic patterns to identify and remove redundant sanitizer (i.e., checks

*Pang is a PhD student at Nanjing University. Pang contributed to this work while he was a Visiting Scholar at Stevens Institute of Technology.

whose safety property is covered by others). SANRAZOR is more safety-driven but still cannot ensure the removed checks are indeed redundant because the patterns used are not sound. Also, SANRAZOR needs user inputs for profiling. In short, SANRAZOR harms both the capability and usability of ASan.

Optimization In this paper, we assemble a group of optimizations to reduce ASan checks. All the optimizations are purely static. They only involve sound, lightweight, and configuration-free analysis, which maintains the capability, scalability, and usability of ASan.

- **Optimization 1** identifies stack/global accesses that can be proven in-bound and removes their sanitizer checks. The optimization has been explored by many tools [13,18,19,38,39,46]. However, these tools often rely on heavy static analysis, which may not scale. We use an approach involving only control flow traverse and basic constant propagation to pinpoint the desired memory accesses (§4.1).
- **Optimization 2** finds sanitizer checks that are dominated (or post-dominated) and meanwhile covered by another check. Such checks are redundant and removed. Previous research [42] has briefly discussed this optimization but did not provide full solutions. We develop an algorithm that combines domination analysis and basic, flow-insensitive alias analysis to achieve this optimization (§4.2).
- **Optimization 3** determines memory accesses that are neighbors in space and merge their checks into one. We propose, design, and implement the optimization (§4.3).
- **Optimization 4** focuses on memory accesses in loops. It locates memory accesses with an invariant address and moves the sanitizer checks outside the loop. It also locates memory accesses that use a constantly increasing/decreasing address and groups the sanitizer checks across iterations into one. We propose, design, and implement the optimization (§4.4).

Evaluation We have integrated the optimizations into LLVM and created a tool called ASan--. We evaluate ASan-- from multiple dimensions. First, we run ASan-- on the full Juliet Test Suite [40] and 34 vulnerabilities from the Linux Flaw project [34]. ASan-- achieves an identical error detection rate to ASan. Second, we build SPEC CPU2006 and Chromium with ASan--. ASan-- imposes a moderate increase to the compilation time and reduces the binary size by **20%**. Binaries built for the programs can support all the benchmark tests. The first and second evaluations show that ASan-- inherits the capability, scalability, and usability of ASan. Third, we compare the performance of ASan-- and ASan on both SPEC CPU2006 and Chromium. The evaluation shows that ASan-- can reduce the overhead of ASan by **41.7%** on SPEC CPU2006 and by **35.7%** on Chromium. If only considering the overhead incurred by sanitizer checks, the reduction rates increase to **51.6%** on SPEC CPU2006 and **69.6%** on Chromium. Fourth, we apply ASan-- in the application of fuzzing. ASan-- can increase the execution speed of AFL by over **40%** and the branch coverage by **5%**. Combined with fuzzing-tailored optimization techniques, ASan-- can speed up AFL by **60%** and increase the branch coverage by **9%**. Finally, we run Chromium built with ASan-- to support our daily work for four weeks. We did not experience major usability issues or significant slowdown, and ASan-- detected all the bugs we reproduced from previous reports.

Contribution We make the following main contributions.

- We present a study to dissect ASan's runtime activities and unveil the sources of overhead. The study confirms existing understandings but also brings new findings.
- We design and implement a group of optimizations to reduce sanitizer checks of ASan. To our knowledge, two of the optimizations are brought up by us for the first time.
- We implement the optimizations and integrate them into LLVM. Source code is publicly available at `https://github.com/junxzm1990/ASAN--.git`.
- We conduct an extensive evaluation on our optimizations. The results show that our optimizations can significantly reduce the overhead of ASan without compromising its properties. The results also show that our optimizations can benefit the applications of ASan.

2 Technical Background

This section covers the technical background of ASan, *based on the implementation in LLVM* [42].

2.1 Shadow Memory

ASan uses a shadow memory model illustrated in Figure 1 to facilitate efficient runtime checks. It spares one-eighth of the virtual address space as the shadow memory where each byte records the status, addressable or not, of eight bytes used by the application. Given a memory byte at address `Addr`, ASan places its shadow byte at `(Addr»3)+Offset`, where `Offset` is a constant determined at the compiling time. To ensure the region for shadow memory is always available, `Offset` needs to be chosen such that the memory area `[Offset, Offset+Max_Addr/8]` is not occupied before the shadow memory is allocated.

The value of a shadow byte encodes the addressable status of the corresponding eight application bytes. 0 means all the eight bytes are addressable, while a value `K` ranging from 1 to 7 means only the first `K` bytes are addressable. A negative value indicates all eight bytes are not addressable, and different negative values unveil different types of non-addressable memory (out-of-bound heap, out-of-bound stack, out-of-bound global, freed memory, etc.).

2.2 Redzone

ASan places a redzone before and after each data object, as shown in Figure 2. The redzones are *poisoned*. Namely,

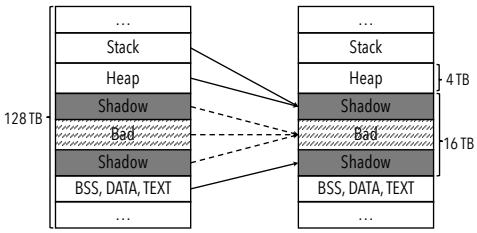

Figure 1: Shadow memory of ASan in a 64-bit application.

their shadow bytes mark them non-addressable. The object, in contrast, is unpoisoned. The creation and poisoning of redzones slightly differ when the object is in different regions.

Heap ASan replaces the standard heap management functions (e.g., `malloc`, `calloc`, `free`, etc.) with customized versions. Once a heap buffer is allocated, ASan places a redzone both before and after the buffer. Size of the redzone, a power of two, increases with the buffer size and can range from 16 to 2,048. By default, the end of the buffer is 8-byte aligned. If a gap of fewer than 8 bytes exists between the aligned end and the actual end, ASan poisons the gap via *partial poisoning*. For efficiency, ASan allocates multiple buffers of the same size as a group, positioning them next to each other. The left redzone of one buffer works as the right redzone of the previous buffer. This way ASan only needs to allocate one redzone (the left redzone) for buffers allocated from such a group. Once a buffer is freed, ASan poisons the entire buffer and places it into *quarantine* of a certain size, for the sake of detecting use-after-free.

Stack Before and after each array on the stack, ASan inserts two neighboring arrays to work as the *left* redzone and the *right* redzone, respectively. The left redzone has 32 bytes, and the right redzone has 32 bytes plus to up to 31 bytes to align the original array. Both redzones are poisoned when the host function is entered at execution time.

Global ASan replaces each global object with a new one that contains a trailing redzone. The size of the redzone is 32 or one-fourth of the size of the object, whichever is larger. The size of the object plus the redzone is rounded up to a multiple of 32 bytes and partial poisoning is used if the end of the object is not 8-byte aligned. All the redzones for global objects are poisoned at the initialization of the process.

2.3 Runtime Checks

ASan instruments every memory access, load or store, with a check on its shadow memory. Depending on the size of the memory access, the check works differently. For an 8-byte memory access, the check loads its shadow byte and inspects whether the shadow byte is zero:

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

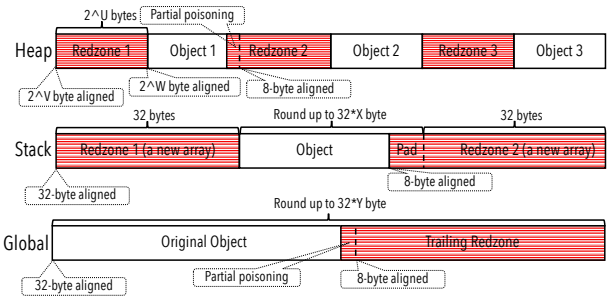

Figure 2: ASan redzones on heap, stack, and global. `U`, `V`, `W`, `X`, `Y` represent positive integers.

For a `N`-byte (`N` = 1, 2, or 4) memory access, the check examines whether its shadow byte indicates that the first `N` bytes are addressable:

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

Both checks only need one memory read and a few arithmetic/-comparison instructions, presenting high efficiency. ASan also intercepts C library functions that often cause memory errors (e.g., `memcpy`). When such a function is entered, ASan checks whether the source buffer and the destination buffer are poisoned, using an optimized implementation.

3 Motivating Study

A major drawback of ASan is its high runtime overhead. As reported in [42], the implementation of ASan in LLVM incurs an average overhead of 73%. To gain insights into reducing the overhead of ASan, we run a study to understand the origins of the overhead.

3.1 Study Methodology

Following the setup of [42], our study focuses on ASan implemented in LLVM and considers SPEC CPU2006 as the benchmark. SPEC CPU2006 [6] is an industry-standardized, CPU-intensive benchmark suite, consisting of 12 integer programs and 7 floating point programs. It comes with three workloads called *train*, *test*, and *reference*. We use the reference workload in our study as it represents a real dataset and activates long-duration execution.

The first step of our study measures the overhead of ASan with default settings. LLVM (or Clang) fails to compile `omnetpp`. Hence, we skip this program. Problems also occur on `h264ref` and `perlbench` because the two programs trigger ASan errors given the reference inputs. To continue their execution at the errors, we compile them with `-fsanitize-recover=address` and run them after setting the `ASAN_OPTIONS` environment variable to "`halt_on_error=0`". The follow-up steps in turn make the

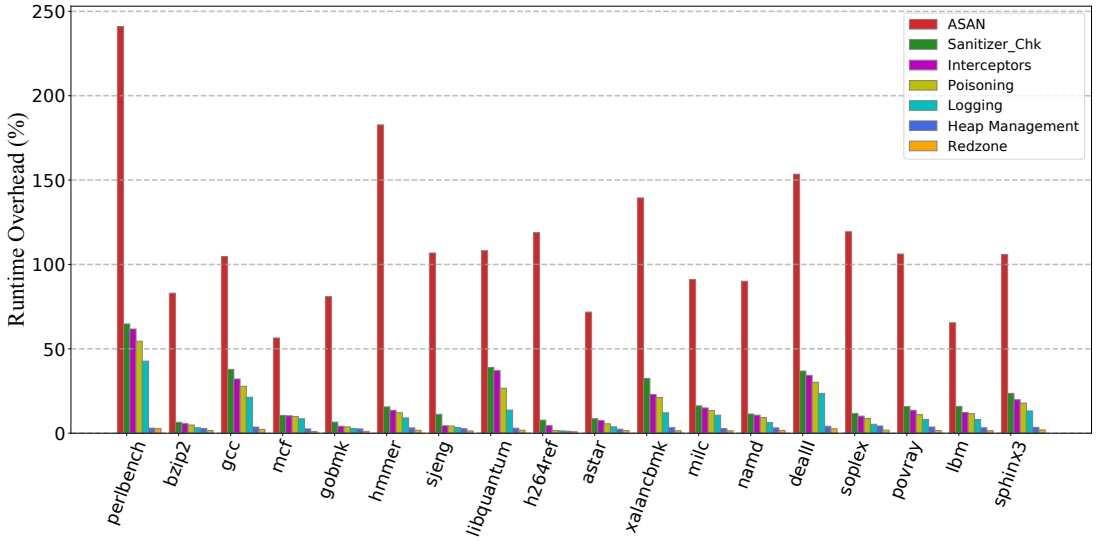

Figure 3: Breakdown of runtime overhead of ASan. `"ASAN"` shows the overhead of stock ASan. From left to right, the other bars show the overhead after we disable the corresponding activities, one after another. For instance, `"Sanitizer_Chk"` shows the overhead without checks on individual memory accesses.

following modifications to ASan and separately measure the overhead after each.

- **Sanitizer Checks:** disabling the sanitizer checks on memory accesses (both loads and stores).
- **Interceptor:** disabling the interception of standard C libraries (`memcpy`, `memset`, and `memmove`, etc.).
- **Poisoning:** skipping the poisoning of redzones.
- **Logging:** removing intermediate logging, in particular the logging of the stack trace during `malloc` and `free`.
- **Heap Management:** revoking ASan's heap management to the one from the standard C library.
- **Redzone:** removing the remaining redzones and the related operations.

3.2 Result Analysis

Figure 3 shows the average results of 30 repeated tests. Stock ASan brings an overhead of 107.8%. This is higher than what is reported in [42], but it is expected since the evaluation in [42] disabled all the logging and removed the quarantine for freed objects. Checks on individual memory accesses are the primary source of ASan overhead, which is attributed to an overhead of 86.5% (80.8% of all the overhead). This matches the common understanding [49, 55] since memory accesses are prevalent. A further breakdown unveils that the checks in sequential code take about 55% of the 80.8% overhead, and the checks in loops take the remaining. Each other activity of ASan also brings overhead, leading to a total overhead of 20.5%. A previously less-known finding is that ASan's heap management averages an overhead of 9.6%. On the programs of `perlbench`, it incurs an overhead of over 39%. The remaining activities are less significant, mostly leading to an overhead of 2%-3%.

3.3 An Extension

Past research [24] has done a related study to understand the time spent by ASan. The study brings a unique finding that complements ours. They find that modern computers broadly use a 64-bit address space, where ASan uses a 4TB heap and needs a 16TB shadow memory, as shown in Figure 1. When the application's memory is scattered in the address space, so will be the shadow memory in the 16TB space. This can result in sparse page table entries (PTEs) and increased page faults during page table management, bringing extra memory management costs. By restricting the application to run in a "smaller" address space, ASan can also use a smaller shadow memory. Both help compact the page table and reduce page faults, thus benefiting various ASan activities (e.g., redzone poisoning and shadow memory checks) that access the application memory and the shadow memory.

The original study in [24] focuses on smaller programs under the context of fuzzing. We replicate the study on SPEC CPU2006 in normal execution environments. As we will explain in §5.2, the public version of the tool presented in [24] fails on many CPU2006 programs due to its strict restriction of memory usage. Hence, we use an approximating approach. The idea is to only allocate one shadow memory page and map all the application memory to the same memory page. Technically, we replace `(Addr » 3) + Shadowoffset` with `(Addr » 36) + Shadowoffset` when mapping `Addr` to its shadow byte. We also set all the shadow bytes to zero when poisoning redzones, preserving the operations but disabling all the errors. This way, we simulate the execution with a 4KB shadow memory. Table 6 in the Appendix shows the results averaged on 30 runs. The results match the observation in [24]. Using a 4-KB shadow memory can reduce about 28.6% of

the page faults, bringing the overhead of ASan from 107% down to 95.5%.

4 Debloating ASan

To reduce the overhead of ASan, one strategy is to shrink the shadow memory [24]. It has demonstrated effectiveness but still leaves behind high overhead (95.5% per our study). To further cut down ASan overhead, an orthogonal, promising strategy is to remove sanitizer checks [16, 46, 49, 55], since they constitute the primary source of overhead.

When removing ASan checks, a desirable principle is to maintain the decent characteristics of ASan, instead of sacrificing them for better efficiency. The principle can break down into the following properties.

- **Capability:** ASan offers the capability of detecting most cases of both spatial and temporal memory errors. Removal of checks should not reduce the detecting capability. Formally, the removal of a check should only proceed after it is proven safe to do so.
- **Scalability:** ASan can scale to support large-size programs like Chromium and Firefox browsers, enabling high practicality and broad adoption. Preferably, the removal of checks should not degrade the scalability of ASan.
- **Usability:** ASan has been integrated into mainstream compilers like LLVM and GCC. Enabling ASan simply requires a compiler flag (e.g., `-fsanitize=address` on LLVM). Hurting this high usability will hinder the adoption of ASan.

In this work, we assemble a set of techniques satisfying the three properties to reduce (or "debloat") ASan checks, with a focus on the ASan in LLVM. Our contribution to different techniques varies, which is indicated by the number of ✧. Three ✧ mean that we are the first ones proposing, designing, and implementing the technique. In contrast, two ✧ indicate that others have informally mentioned the technique while we deepen, design, and implement the technique. Finally, a single ✧ suggests that the technique has been explored, and we implement the technique in LLVM or improve the existing implementation. Our efforts on the single ✧ techniques are less scientific but are deemed necessary as they will enable an understanding of the overall improvement by all the techniques. In the rest of this section, we elaborate on the designs of our techniques. *Properties of the techniques are described and validated in Append A, due to the limited space.*

4.1 Removing Unsatisfiable Checks ✧

The first technique focuses on removing ASan checks on global/stack access. The key insight is to remove checks where the error condition can never be satisfied. Below shows such an example on the stack (code in red is inserted by ASan).

```
1 int foo(){
2   char buf[20];
3   unsigned int i = 10;
4   i++;//i = 11
5   ASan(buf+i);//ASan check 1; unsatisfiable
6   buf[i] = 0;
7 }
```

The access to `buf`, an array on the stack, has a constant index of `11`, which cannot overflow. Thus, no ASan check is needed there. Formally, given an array `buf` of `N` bytes on the stack or in the global, a `K`-byte access `buf[offset]` needs no ASan check if on any execution path:

Condition-1: `offset` $\geq$ `0 && N - offset` $\geq$ `K`

Past research has discussed similar issues several times and has explored various solutions [13, 18, 19, 39, 46] to identify stack/global accesses that meet the above condition. A systematic approach [52] is to symbolically interpret the execution paths reaching a target access and verify whether every path satisfies the condition. However, the approach may not scale as it involves complex constraint solving.

Our Choice We opt to adopt a lightweight analysis shipped with LLVM (disabled by default). The analysis focuses on *direct* stack/global accesses whose target object is propagated from an `alloca` instruction or a global object. Given a direct stack/global access, the analysis backward traces whether the index propagates from constants. Consider `buf[i]` at Line 6 in the above code as an example. It traces `i` backward from Line 6 to Line 4 where it represents `i@L6` as `i@L4 + 1`, followed by another step jumping to Line 3 to further represent `i@L4 + 1` as `10 + 1`. This way, it learns the constant value, `11`, of `i` at Line 6 and can compare `i` with the array size. The backward tracing stops wherever uncertainty appears (e.g., an intermediate value is loaded from memory or a store happens to an intermediate value) and returns an unknown result, thus offering soundness.

We slightly extend the above analysis. When backward tracing the index of an access, we concurrently collect comparisons that dominate the access and operate between the index and a constant. Doing so enables us to accumulate loose boundaries of the index and compare the index with the object size even the index is not a constant. Our extension helps identify removable accesses like the one below. Our extension inherits the soundness from the basic version. Further, whenever there is a store on the index, we abandon the comparisons affected by the store.

```
1 int foo(){
2   char buf[20];
3   unsigned int i = input();//i is not a constant
4   if(i < = 10) //the comparison will be captured
5     ASan(buf+i);//ASan check 1; still unsatisfiable
6     buf[i] = 0;
7 }
```

Algorithm 1: IDENTIFYING RECURRING ASAN CHECKS.

```
1  Procedure find_mem_group(M_grp, m):
       /* Find the group m belongs to                    */
2      for m_i in M_grp do
3          for m_j in M_grp[m_i] do
4              if m == m_j || must_alias(m,m_j) then
5                  return m_i;
6              end
7          end
8      end
9      return NULL;
10 Algorithm
       Input  : A function F
       Output : Recurring ASan checks R_c = {C_1, C_2 ..., C_n}
11     Initialization: R_c = ∅; M_grp = dict();
       /* M_grp is a dictionary where the key is a
          unique memory location and the value is a
          list of memory accesses to that location   */
12     for each memory access m_i in F do
13         m_j = find_mem_group(M_grp, m_i);
14         if m_j == NULL then
15             M_grp[m_i] = list();
16             m_j = m_i;
17         end
18         M_grp[m_j].add(m_i);
19     end
20     for m in M_grp do
21         for m_i in M_grp[m] do
22             for m_j in M_grp[m] do
23                 if m_i ≠ m_j && dominate(m_i, m_j) &&
                     sizeof(m_i) ≥ sizeof(m_j) then
24                     R_c.add(ASan(m_j));
25                     M_grp[m].remove(m_j)
26                 end
27             end
28         end
29         for m_i in M_grp[m] do
30             for m_j in M_grp[m] do
31                 if m_i ≠ m_j && post_dominate(m_i, m_j)
                     && sizeof(m_i) ≥ sizeof(m_j) then
32                     R_c.add(ASan(m_j));
33                     M_grp[m].remove(m_j)
34                 end
35             end
36         end
37     end
38 return R_c;
```

4.2 Removing Recurring Checks ✧✧

The second technique aims to remove recurring checks or *checks guaranteed to have been done*. The code below is an example of recurring checks.

```
1  int *p;
2  ASan(p); //ASan check 1
3  if(*p == 0){
4    ASan(p); //ASan check 2 (a recurring check)
5    *p = 1;
6  }
```

In the example, ASan places two checks on the dereferences at Line 1 and Line 3. However, the two dereferences access the same location with the same size, meaning that the check at Line 4 is a replica of that at Line 2 and thus, can be removed. Past research [42] has demoed recurring checks but did not provide full solutions. We first formalize recurring checks. Given two ASan checks, denoted as `ASan(ptr1)` and `ASan(ptr2)`, on memory accesses `*ptr1` and `*ptr2`. `ASan(ptr2)` is a recurring check of `ASan(ptr1)` if:

Condition-2:
`(ptr1 == ptr2 || alias(ptr1, ptr2)) && sizeof(*ptr1) ≥ sizeof(*ptr2) && (dominate(*ptr1,*ptr2) || post-dominate(*ptr1,*ptr2));`

Following the above formalization, Algorithm 1 is designed to identify recurring ASan checks in individual functions. The first step aggregates memory accesses to the same location into one group (Line 12-19). When determining whether two memory accesses have the same location, aliases are also considered but in a sound way. Specifically, we reuse the basic, flow-insensitive alias analysis in LLVM and only accept the must-alias results. The internal algorithm of LLVM ensures that two memory objects with must-alias results always start at exactly the same location [3], protecting the safety of our optimization. The second step processes each memory group separately. Given a group, the algorithm inspects each pair of memory accesses therein and identifies recurring ASan checks according to the above conditions (Line 20-37). In this step, we reuse the analysis offered by LLVM to understand the domination/post-domination relation [4]

Discussion: The soundness of Algorithm 1 can fail in certain cases. For instance, `*ptr1` dominates `*ptr2` and satisfies the above condition. Thus, the ASan check on `ptr2` will be removed. However, if the memory is freed between `*ptr1` and `*ptr2`, `*ptr2` will be use-after-free, and we will miss it. We use a conservative approach to avoid such issues: we skip the removing if any function call exists on any execution path from `*ptr1` to `*ptr2`. We do not do this if `*ptr1` post-dominate `*ptr2` as `*ptr1` shall always execute after `*ptr2`.

4.3 Optimizing Neighbor Checks ✧✧✧

This technique explores optimizing ASan checks on neighboring memory accesses. Code below illustrates the ideas.

```
1  struct{
2    int a; int b; int c;
3  }test;
4  int foo(){
5    struct test *ptr1, *ptr2;
6    ASan(ptr1->a); //ASan check 1
```

```
7   *ptr1->a = 1;
8   ASan(ptr1->b); //ASan check 2; can be merged into
        ASan check 1
9   *ptr1->b = 2;
10  ASan(ptr2->a); //ASan check 3
11  *ptr2->a = 1;
12  ASan(ptr2->b); //ASan check 4; can be removed
13  *ptr2->b = 2;
14  ASan(ptr2->c); //ASan check 5
15  *ptr2->c = 3;
16 }
```

In the above code, ASan places a check on each access in Line 7-15. The checks can be optimized in two ways.

First, `ptr1->a` and `ptr1->b` are neighbors in memory, and they fall into an 8-byte region, mapping to at most two shadow bytes.This enables merging the two checks on `ptr1->a` and `ptr1->b` into a single check of the following form.

```
1 short *Shadow //2 bytes
2 Shadow = (ptr1->a >> 3) + Offset;
3 if (*Shadow != 0){//load and check shadow bytes for
      both ptr1->a and ptr1->b in a single operation
4    //slow path
5    second_chk(ptr1->a);
6    //check whether ptr1->a is addressable
7    second_chk(ptr1->b);
8    //check whether ptr1->b is addressable
9 }
```

Since the majority of the accesses should be error-free, the check will likely exit at line 3 instead of routing to the slow path. Therefore, the efficiency of the check should approximate a single ASan check. We call the checks on `ptr1->a` and `ptr1->b` *mergeable neighbor checks*.

Second, `ptr2->b` locates in the gap between `ptr2->a` and `ptr2->c`, and the gap is 4 bytes in size. If `ptr2->b` is in a redzone, then at least one of `ptr2->a` and `ptr2->c` must also be in the redzone (maybe partially) since a redzone has at least 16 bytes. As such, any error on `ptr2->b` will always be captured by the checks on `ptr2->a` (Line 14) and `ptr2->c` (Line 18). Thus, the check on `ptr2->b` (Line 16) can be removed. We call such cases *removable neighbor checks*.

Merging Neighbor Checks Formally, given two memory accesses `*ptr1` and `*ptr2`, `ASAN(ptr2)` can be merged to `ASAN(ptr1)` if

```
Given:
base1 = base(ptr1); base2 = base(ptr2);
/*base(p) gets the base address of the object that
p points to*/
offset1 = offset(ptr1); offset2 = offset(ptr2);
/*offset(p) gets the offset of p in the object*/
offset2 > offset1;
MaxBitRead = the maximal number of bits a single
memory-read can load;

Then Condition-3:
base1 == base2 &&
RoundUpTo(offset2 - offset1 + sizeof(*ptr2), 8)
<= MaxBitRead &&
(dominate(*ptr1,*ptr2) &&
post-dominate(*ptr2,*ptr1)) ||
(dominate(*ptr2, *ptr1) &&
post-dominate(*ptr1, *ptr2))
```

At the high level, the condition ensures that (i) `ASan(ptr1)` and `ASan(ptr2)` either both happen or both not happen and (ii) the shadow bytes for the region between the first byte of `*ptr1` and the last byte of `*ptr2` can be loaded in one memory-read. These ensure that `ASan(ptr1)` and `ASan(ptr2)` can be merged to the above form, after adjusting the type of `Shadow` based on the number of shadow bytes.

We use an intra-procedural analysis to locate mergeable neighbor checks. Given function `F`, memory accesses therein are grouped based on their base addresses and sorted by their offsets. The challenge here is to identify the base address and offset of a given access. In LLVM IR, a memory access is either a `load` or a `store`, and the address can be extracted. We backward trace the propagation of the address until a `getelementptr` instruction [5], where the base address and the offset can be separately obtained. In the backward tracing, we only allow `bitcast` instructions because other instructions may change the address. A `getelementptr` instruction often includes multiple-dimensional offsets, all following the format of constants or variables. If all the offsets are constants, we flatten them into a single one based on the scale of each dimension. Otherwise, we mark the offset unknown.

Given a group of memory accesses with the same base address (denoted $\vec{M} = \{m_1, m_2 ..., m_n\}$), all the mergeable pairs are gathered and then reorganized as $\vec{m_i} \rightarrow \{m_{i1}, m_{i2} ..., m_{in_i}\}$ $(1 \leq i \leq n)$, meaning that the checks on $m_{i1}, m_{i2} ..., m_{in_i}$ can be merged to the check on $\vec{m_i}$. Algorithm 2, a greedy algorithm, is then used to pick pairs to merge. The algorithm, by itself, is straightforward to understand, but its properties are worth mentioning. First, the algorithm prioritizes the handling of $\vec{m_i}$ with a larger size, potentially helping identify more mergeable pairs. Second, the algorithm supports recursive merging. For instance, after m_i is merged to m_j, m_j can be later merged to m_k, if m_i can also be merged to m_k.

For a pair of mergeable memory accesses (m_1, m_2), where `ASan(`m_2`)` can be merged to `ASan(`m_1`)`, a new check will be created and inserted under the post-dominating access. Both the original checks will be removed. All checks that are previously merged to m_2 will be migrated to the new check.

Removing Neighbor Checks Removable neighbor checks can be formalized as follows. Given three memory accesses `*ptr1`, `*ptr2`, and `*ptr3`, `ASAN(ptr2)` can be removed if

```
Given:
base1 = base(ptr1); base2 = base(ptr2);
base3 = base(ptr3;)
off1 = offset(ptr1); off2 = offset(ptr2);
off3 = offset(ptr3);
```

Algorithm 2: PICKING MERGEABLE NEIGHBOR CHECKS.

Input : A list of mergeable relations:
$\vec{M}_r = \{\vec{m}_1, \vec{m}_2, ...\vec{m}_n\}$
/* $\vec{m}_i = \{m_{i1}, m_{i2}..., m_{in_i}\}$ $(1 \le i \le n)$ represents the list of memory accesses that can be merged to $\vec{m}_i$ */
Output : A list of mergeable memory access pairs
$\vec{P} = \{(m_{i1}, m_{j1}), (m_{i2}, m_{j2})..., (m_{ik}, m_{jk})\}$

Initialization: $\vec{P} = \emptyset$;
$\vec{M}_r$ = sorted($\vec{M}_r$);
/* sort by $|\vec{m}_i|$ $(1 \le i \le n)$ in descending order */
for *each list* $\vec{m}_i$ *in* $\vec{M}_r$ **do**
  for *each member* m *in* $\vec{m}_i$ **do**
    if $\vec{M}_r$.*merged_to_others(m)* **then**
      /* m has merged to another memory access */
      Continue;
      /* avoid repeated merging */
    end
    $t\vec{m}p = \vec{M}_r.find\_merged\_by(m)$;
    /* find accesses that are merged to m */
    if $t\vec{m}p \not\subset \vec{m}_i$ **then**
      Continue; /* not all accesses merged to m can be merged to m_i, skip */
    end
    $\vec{P}.add(m_i, m)$;
    /* merge m to m_i */
  end
end
return $\vec{P}$;

```
off1 < off2 < off3;
size1 = sizeof(*ptr1); size2 = sizeof(*ptr2);
size3 = sizeof(*ptr3);
MinRdSz = the minimal size of a redzone;
```

Then Condition-4:

```
base1 == base2 == base3 && off3 - off1 < MinRdSz
&& off2 + size2 ≤ off3 + size3 &&
(dominate(*ptr1, *ptr2) ||
post-dominate(*ptr1, *ptr2)) &&
(dominate(*ptr3, *ptr2) ||
post-dominate(*ptr3, *ptr2))
```

In essence, the above condition guarantees two properties. First, `ptr2` being in a redzone implies `ptr1` and/or `ptr2` are in the same redzone, fully or partially. Second, `ASAN(ptr2)` being reached means both `ASAN(ptr1)` and `ASAN(ptr3)` are reached. As such, if the condition holds, any errors on `*ptr2` will be captured by `ASAN(ptr1)` or `ASAN(ptr3)`. Thus, it is safe to remove `ASAN(ptr2)`.

We also use an intra-procedural analysis to collect removable neighbor checks. Memory accesses in each function are first grouped based on their base addresses. From each group, tuples satisfying `Condition-4` are then gathered. The tuples follow the format of `[(`m_i`, `m_j`), `m_k`]`, meaning that `ASAN(`m_k`)` can be removed because it is covered by `ASAN(`m_i`)` and `ASAN(`m_j`)`. In the follow-up step, the analysis aggregates the tuples and produces a removable list for each `(`m_i`, `m_j`)`: $(\vec{m_i}, \vec{m_j}) = \{m_{ij1}, m_{ij2}..., m_{ij_{nij}}\}$. The final step is to go over each list and collect removable memory accesses. Simply including all the members in each list will cause mistakes. Consider the example where $(\vec{m_a}, \vec{m_b}) = \{m_c\}$ and $(\vec{m_c}, \vec{m_d}) = \{m_e\}$. After handling $(\vec{m_a}, \vec{m_b})$, we remove m_c. If we continue with $(\vec{m_c}, \vec{m_d})$ to remove m_e, mistakes will arise because m_c has been removed and cannot help cover m_e. To avoid such mistakes, after including m_i to the removable set, we abandon all the lists matching $(\vec{m_i}, *)$ and $(*, \vec{m_i})$. To maximize the number of removable cases, we sort all the lists based on their size in descending order and then handle each of them in turn.

4.4 Optimizing Checks in Loops ✧✧✧

ASan checks in loops are expensive. According to our study in §3.2, checks in loops account for 45% of the overhead introduced by all ASan checks. Modern compilers run many optimizations to reduce operations in loops, which, however, still cannot handle many optimizable ASan checks. The code below shows two types of optimizable ASan checks in loops.

```
1  char *ptr1;
2  char *ptr2;
3  for(int index = 0; index < limit; index += 2){
4    ASan(ptr1+index);
5    /*ASan check 1; can be merged*/
6    ptr1[index] = getchar();
7    ASan(ptr2);
8    /*ASan check 2; can be moved out of loop*/
9    *ptr2 = getchar();
10 }
```

First, `ptr2` is dereferenced (Line 9) and ASan-checked (Line 7) in each loop iteration. However, `ptr2` never changes. Thus, one ASan check on `ptr2` out of the loop is sufficient. The compiler cannot optimize the memory access because it is a write. We call checks like `ASan(ptr2)` *invariant checks*.

Second, `ptr1` is accessed (Line 6) with an offset increased by 2 in each iteration (Line 3). Thus, any two consecutive accesses to `ptr1` are 2-byte apart, and the checks on them resemble the mergeable neighbor checks we discussed in §4.3 (`Condition-3`). Merging the checks together will reduce the cost. Considering that the address of the access (e.g., `index` at Line 6) monotonically increases/decreases, we call the check on it (e.g., Line 6) a *monotonic check*.

Relocating Invariant Checks Formally, given a memory access `*ptr` in a loop, `ASan(ptr)` is an invariant check if

Condition-5: `ptr is a loop invariant`

To identify invariant checks, we gather all the loops from a function then process each loop in turn. Given a loop, each memory access inside is visited to inspect whether the address

is a loop invariant. If so, we mark the ASan check as an invariant check. LLVM offers a built-in interface to determine loop invariant. It is, however, very preliminary. Only when the address is a value created outside the loop, the interface considers it a loop variant. This misses invariant addresses (e.g., an address composed of two other invariant values). We design a new algorithm, Algorithm 3, to more systematically identify loop invariant. Due to the space limit, Algorithm 3 is presented in the Appendix. The algorithm backward traces the generation of an address until outside of the loop. If all the ingredients involved in generating the address are invariants, the algorithm reports the address also as an invariant.

To optimize an invariant check, an intuitive idea is to move the check after the exit of the loop. This works when the invariant check dominates the loop exit. However, many invariant checks are guarded by conditional statements inside a loop. Their execution is not guaranteed when the loop is entered. Consider the code below as an example.

```
1  bool asan_ptr = false;
2  for(condition1){//outer loop
3    for(condition2){//inner loop
4      if(condition3){//guardian condition
5        asan_ptr = true;
6        //original ASan(ptr) is here
7        *ptr = getchar(); //ptr is an invariant
8      }
9    }
10   //simply putting ASan(ptr) here is erroneous
11   if(asan_ptr){
12     asan_ptr = false; //if not outermost loop
13     ASan(ptr);
14   }
15 }
```

At Line 7, `ptr` is a loop invariant and is dereferenced. An ASan check on it originally exists at Line 6. The dereference and the ASan-check are guarded by `condition3` at Line 4. Moving the ASan check to the exit of the loop (Line 10) can cause mistakes. The dereference and the original ASan check may not always execute when the loop (Line 3-9) is entered, while the new check at Line 10 always does. That means we may introduce checks that should not happen.

To optimize a conditional invariant check, we introduce a supportive local variable to track whether the memory access is ever executed. In the code above, `asan_ptr` is introduced to trace the execution of Line 8. `Asan_ptr` is initialized as `false` and is set to `true` at the memory access. At the loop exit, `asan_ptr` being `true` will trigger the relocated ASan check. Otherwise, the ASan check is skipped. Further, if the memory access is not in an outermost loop, we reset the local variable to `false` at the loop exit (Line 12-15 in the above code) because, otherwise, the `true` status will carry over when the loop is re-entered in the next iteration of the outer loop.

Discussion: The optimization of invariant checks can run into another problem. The code from an invariant check to the loop exit can free the memory. A check at the loop exit may, thus, report false use-after-free. Our backward analysis of invariant already considers the problem. If any value involved in the generation of the address is passed to a function, the analysis marks the address as a variant (see Algorithm 3).

Grouping Monotonic Checks Given a memory access `*ptr` in a loop, `ASan(ptr)` is a monotonic check if

`Condition-6`: `ptr in any two consecutive iterations has a constant distance.`

To identify monotonic checks, a systematic approach is to use the scalar evolution (SCEV) [7], which is an analysis to represent variables with complicated behavior in a more straightforward way. In particular, SCEV can represent monotonically increasing/decreasing loop variables as *add recurrences* [11]. An add recurrence has the format of `{Init, +/-, Step}` (`Step` is a constant), meaning that the variable has an initial value of `Init` and increases by `Step` in an iteration.

Consider `ptr1[index]` at Line 8 in the code presented at the beginning of this subsection as an example. The original ASan check, executed in each loop iteration, is:

```
1 V = * ((ptr1 + index >> 3) + Offset);
2 if (V != 0 && ((ptr1 + index & 7) + 1 > V))
3     ReportAndCrash(Addr);
```

In our optimization, we first run SCEV to represent the address as `{Init, +/-, Step}` (i.e., `{ptr1, +, 2}`) and then confirm that `Init` (i.e., `ptr1`) is an invariant. Finally we replace the ASan check in the loop with a new one of the following format (the address is represented as `Addr`):

```
1 intN  *Shadow; //hold shadow memory for MinRdSz
       application bytes (MinRdSz represents the minimal
       size of a redzone); N >= MinRdSz
2 if((Addr - Init) % MinRdSz < Step){
3     Shadow = (Addr >> 3) + Offset;
4     if (*Shadow != 0) //check the shadow memory for
          MinRdSz bytes
5         ASan(Addr);
6 }
```

At the loop exit, we further insert another piece of code. The code inspects whether the final `Addr` differs from `Init`. If so, it performs an ASan check on `Addr`.

The idea of our optimization is to check the shadow memory of a chunk of `MinRdSz` bytes at once, using only one operation. If the shadow memory is all zero, we skip the checks on follow-up access to the same chunk. Otherwise a regular ASan check is performed. This way we group multiple checks into one. The number of checks grouped depends on `Step` (`MinRdSz/Step` checks will be grouped into one). By default, the maximal `Step` we allow is `MinRdSz/4`.

Discussion: A memory access may cross the boundary of a MinRdSz chunk, but its beginning address does not meet the condition of Line 6. Thus, we will miss checking the second half of the access. This is OK as the second half will be checked in the next chunk thanks to the alignment in the mapping to shadow memory.

Table 1: Detection capability of ASan and ASan-- on the Juliet Test Suite. Good tests measure false positives. Bad tests measure false negatives. ASan and ASan-- achieve **identical** results shown in this table.

CWD (number)	Good Test (Pass/Total)	Bad Test (Pass/Toal)
Stack-based Buffer Overflow (121)	2348/2348	2111/2348
Heap-based Buffer Overflow (122)	1677/1677	1595/1677
Buffer Under-write (124)	584/584	571/584
Buffer Over-read (126)	445/445	420/445
Buffer Under-read (127)	590/590	565/590
Total	5644/5644	5262/5644

5 Implementation and Evaluation

We have implemented the optimization techniques in a tool called ASan--. ASan-- is built on top LLVM-4.0, with around 2.5K lines of C++ added. We have ported a preliminary version of ASan-- to LLVM-12.0. ASan-- on both LLVM-4.0 and LLVM-12.0 is publicly available at https://github.com/junxzm1990/ASAN--.git. The use of ASan-- is identical to ASan. The optimizations are enabled by default but can be disabled through a customized environment variable. The rest of this section presents our evaluation of ASan--, centering around three questions:

- *Can ASan-- maintain the capability, scalability, and usability of ASan?*
- *Can ASan-- reduce the runtime overhead of ASan?*
- *Can ASan-- benefit the applications of ASan?*

5.1 Capability, Scalability, and Usability

Capability To measure the detection capability of ASan--, we run two experiments. In the first experiment, we run both ASan-- and ASan on the memory errors from the Juliet Test Suite (version 1.3) [40]. As shown in Table 1, ASan-- and ASan achieve identical results on both good and bad tests. For comparison, we further run SANRAZOR and ASAP on the Juliet Test Suite. SANRAZOR failed to run because of an error in the profiling component. The error has been confirmed by the developers and is pending fix. ASAP runs with LLVM-3.7 which does not support continuous execution at ASan errors. Thus, ASAP could not finish the test as Juliet triggers all the bugs in a single execution. To address this issue, we extend LLVM-3.7 to avoid halting at ASan errors. When running ASAP, we use the good tests of Juliet for profiling and the bad tests for evaluation. Further, we configure ASAP to use ASan--'s overhead as the performance budget (i.e., setting the cost level to 51.6%). At the end, we successfully run ASAP on 1,524 Juliet bugs. ASAP detects 985 of them, missing about 35% of the bugs. This shows that ASan-- detects more bugs than ASAP when presenting similar performance.

In the second experiment, we collect vulnerabilities from the Linux Flaw Project [34]. After visiting all the vulnerabilities in the database, we reproduce 34 of them, including 23 used in the evaluation of SANRAZOR [55]. Details of the vulnerabilities are presented in Table 2. Both ASan-- and ASan can detect all the 34 vulnerabilities. In contrast, SANRAZOR, removing ASan checks based on static/dynamic patterns, misses two vulnerabilities (CVE-2017-12858 and CVE-2017-14406). The experiment demonstrates the soundness of ASan-- and also its advantages over SANRAZOR.

Table 2: Detection capability of ASan and ASan-- on vulnerabilities from the Linux Flaw project. The first 23 cases are also used in the measurement of SANRAZOR [55]. The cases highlighted are missed by SANRAZOR in L1 and L2 modes.

Software	CVE	Type	ASan	ASan--	SANRAZOR
LAME	CVE-2015-9101	heap-buffer-overflow	✓	✓	✓
LIBTIFF	CVE-2016-10095	stack-buffer-overflow	✓	✓	✓
LIBTIFF	CVE-2016-10270	heap-buffer-overflow	✓	✓	✓
LIBTIFF	CVE-2016-10271	heap-buffer-overflow	✓	✓	✓
ZZIPLIB	CVE-2017-5976	heap-use-after-free	✓	✓	✓
ZZIPLIB	CVE-2017-5977	heap-use-after-free	✓	✓	✓
POTRACE	CVE-2017-7263	heap-buffer-overflow	✓	✓	✓
AUTOTRACE	2017-9167∼9173	heap-buffer-overflow	✓	✓	✓
AUTOTRACE	2017-9164∼9166	heap-buffer-overflow	✓	✓	✓
LIBZIP	CVE-2017-12858	heap-use-after-free	✓	✓	☒
G.MAGICK	CVE-2017-12937	heap-use-after-free	✓	✓	✓
MP3GAIN	CVE-2017-14406	null-pointer-dereference	✓	✓	☒
MP3GAIN	CVE-2017-14407	stack-buffer-overflow	✓	✓	✓
MP3GAIN	CVE-2017-14408	stack-buffer-overflow	✓	✓	✓
MP3GAIN	CVE-2017-14409	global-buffer-overflow	✓	✓	✓
PROFTPD	CVE-2006-6563	heap-buffer-overflow	✓	✓	-
CTORRENT	CVE-2009-1759	stack-overflow	✓	✓	-
LIBTIFF	CVE-2009-2285	heap-buffer-overflow	✓	✓	-
LIBTIFF	CVE-2010-2481	out-of-order	✓	✓	-
LIBTIFF	CVE-2010-2482	null-pointer-dereference	✓	✓	-
LIBTIFF	CVE-2013-4243	heap-buffer-overflow	✓	✓	-
POPPLER	CVE-2013-4473	stack-smashing	✓	✓	-
POPPLER	CVE-2013-4474	stack-buffer-overflow	✓	✓	-
PYTHON	CVE-2014-1912	heap-buffer-overflow	✓	✓	-
LIBTIFF	CVE-2015-8668	heap-buffer-overflow	✓	✓	-
BINUTILS	CVE-2018-9138	stack-overflow	✓	✓	-

Scalability ASan-- introduces extra analyses to ASan. Although the analyses are not complex, they may still affect the scalability of ASan. To this end, we apply ASan-- to SPEC CPU2006 and Chromium (58.0.3003.0). Chromium and many programs in CPU2006 (e.g., `perl` and `gcc`) should be large enough to stress test the scalability. ASan-- can successfully compile and build CPU2006 and Chromium. The binaries can pass all the benchmark tests (see §5.2). We have also been using the Chromium built with ASan-- for four weeks and have not encountered major issues (see §5.3).

We also measure the time cost of compilation. We believe it is also scalability-related since extremely high compilation time will make the tool impractical for large programs. Table 7 shows the results. On average, ASan-- increases the compilation time by `1x+`, but the overall compilation time should still be acceptable. Precisely, ASan-- can finish the compilation of most programs in seconds or minutes. Even in the case of Chromium, ASan-- only needs several hours. The length of the time is tolerable in practice.

Usability ASan-- is implemented to share the same assumption, same requirement, and same interfaces with ASan. Usage-wise, it is identical to ASan. ASan-- also helps reduce the binary size because it removes many checks. On average, ASan-- can shrink the binary size by 20.4% (see Table 7 in Appendix), which helps improve the usability of ASan, in particular in contexts where the storage space is limited.

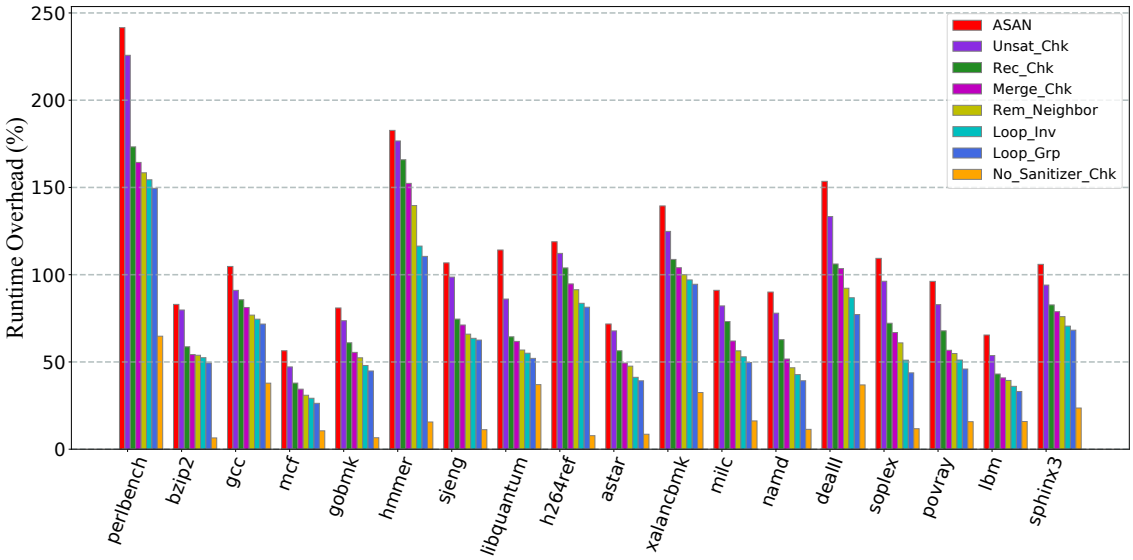

Figure 4: Runtime overhead on CPU2006. `"ASAN"` shows the overhead of stock ASan. From left to right, the other bars show the overhead after we enable the corresponding optimization in ASan--, one after another.

Table 3: Performance comparison with SANRAZOR. The **Average** data in this table differs from Figure 3 and Figure 4 because only 11 programs from SPEC CPU2006 are included.

Benchmark	Performance Overhead				
	ASan	SanRazor_L0	SanRazor_L1	SanRazor_L2	ASan--
401.bzip2	81.90%	58.55%	45.06%	50.76%	49.39%
429.mcf	54.10%	35.11%	9.03%	2.67%	26.30%
445.gobmk	76.90%	62.81%	47.57%	30.65%	44.83%
456.hmmer	179.0%	133.8%	37.38%	39.52%	110.5%
458.sjeng	102.0%	105.3%	86.77%	97.74%	62.54%
462.libquantum	108.0%	48.31%	53.51%	58.96%	52.05%
433.milc	89.10%	44.97%	30.19%	43.25%	49.72%
444.namd	86.10%	41.92%	34.19%	26.00%	39.22%
453.povray	101.0%	64.81%	54.94%	50.62%	45.95%
470.lbm	62.40%	31.23%	6.62%	18.61%	33.08%
482.sphinx3	103.0%	80.57%	43.59%	50.19%	68.21%
Average	94.86%	64.31%	40.81%	42.63%	52.89%

5.2 Runtime Overhead

SPEC CPU2006 We run both ASan-- and ASan on SPEC CPU2006, using the reference workload as testing data. In the experiment, both SANRAZOR [54] and FuZZan [24] are included as baselines. Following the paper presenting SANRAZOR [55], we perform the profiling for SANRAZOR using the training workload and measure its performance using the reference workload. Currently, the public version of SANRAZOR only supports 11 programs of CPU2006, as listed in Table 3. FuZZan offers different heap sizes (1G, 4G, 8G, and 16G). For each program, we start with the minimal heap and increase it until the test can succeed. In total, FuZZan can run 10 programs of CPU2006, as shown in Table 9. All the tests are repeated 30 times, and average results are collected.

Figure 4 compares the overhead of ASan and ASan--. On average, ASan-- reduces the overhead of ASan from 107.8% to 63.3%, producing a reduction rate of 41.7%. Only considering the overhead of ASan checks, the reduction rate is 51.6%. Figure 4 also unveils that every of our optimizations

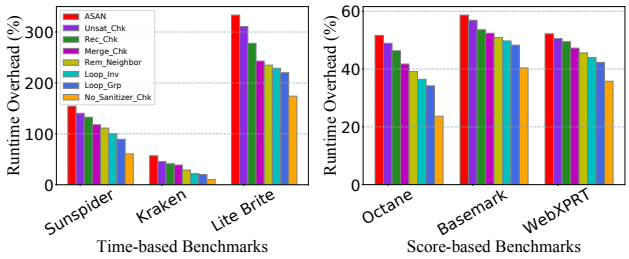

Figure 5: Runtime overhead on Chromium. The bars follow the same setting as Figure 4.

contributes. The amount of overhead reduced by most optimizations is over 5%. In particular, removing recurring checks reduces an overhead of 16.7%. We further measure the number of ASan checks reduced by ASan--. The results, shown in Table 8, are consistent with ASan--'s impacts on the overhead. Averagely, ASan-- removes 38.7% of the ASan checks and every optimization brings a meaningful contribution.

Table 3 presents the comparison between ASan-- and SANRAZOR. Both ASan-- and SANRAZOR can bring the overhead down. When SANRAZOR runs in L1 and L2 modes, it more aggressively removes ASan checks. Thus, it incurs less overhead (40.81% and 42.63%) than ASan-- (52.89%). However, the two modes can miss detecting memory errors, such as the two reported in Table 2. In contrast, when SANRAZOR runs in the more conservative L0 mode, it removes fewer checks and brings an overhead higher than ASan-- (64.31% *v.s.* 52.89%), although it still cannot offer the soundness of ASan--.

Table 9 in the Appendix shows the improvement of FuZZan to ASan. Regardless of which program and what heap size we test, FuZZan does not bring evident improvement to ASan. In summary, ASan-- and FuZZan provide similar bug detection but ASan-- is more scalable and runs faster.

Table 4: Results of fuzzing evaluation. All the numbers are increase rates using ASan as the baseline.

Benchmark	Branch Coverage Increase						Total Execution Increase					
	FuZZan		ASan--		FuZZan +ASan--		FuZZan		ASan--		FuZZan +ASan--	
	AFL_Seeds	FuZZan_Seeds	AFL_Seeds	FuZZan_Seeds	AFL_Seeds	FuZZan_Seeds	AFL_Seeds	FuZZan_Seeds	AFL_Seeds	FuZZan_Seeds	AFL_Seeds	FuZZan_Seeds
OBJDUMP	3.94%	4.33%	3.41%	2.14%	11.30%	10.1%	27.5%	18.6%	25.1%	14.4%	40.3%	37.9%
SIZE	2.81%	3.05%	3.56%	7.42%	9.89%	10.3%	54.2%	48.5%	60.7%	61.9%	79.9%	79.7%
C++FILT	4.24%	3.49%	3.94%	2.15%	8.12%	9.07%	45.9%	39.6%	34.9%	33.2%	71.0%	61.1%
NM	3.67%	2.79%	5.91%	5.69%	6.79%	7.21%	74.6%	52.2%	62.2%	61.4%	86.4%	78.9%
TCPDUMP	17.4%	20.7%	5.82%	5.69%	7.96%	8.37%	63.1%	83.4%	35.8%	34.8%	42.2%	54.2%
PNGFIX	6.01%	3.18%	6.58%	7.48%	10.30%	11.2%	67.5%	55.1%	58.5%	60.5%	75.1%	77.4%
FILE	5.62%	8.56%	5.90%	5.34%	8.60%	11.9%	18.7%	22.4%	17.1%	12.1%	20.9%	25.7%
Average	6.24%	6.58%	5.02%	5.13%	8.99%	9.73%	50.2%	45.7%	42.1%	39.8%	59.4%	59.3%

An Extension: Considering that SPEC CPU2006 was years old, we repeated the evaluation of ASan-- on the new version of CPU2017 [2]. CPU2017 includes two modes, SPECrate and SPECspeed, to separately measure time-based performance and throughput performance. Due to issues of compiler compatibility, we could only run 10 SPECrate programs and 5 SPECspeed programs SPECspeed. The results, summarized in Table 11 in the Appendix, show that ASan-- similarly benefits CPU2017. The reduction rates of overhead are 40.6% and 39.1% respectively on SPECrate and SPECspeed.

Chromium In this experiment, we measure the performance of Chromium under ASan and ASan--. Six popular web-browser benchmarks are used, including both Time-based ones (Sunspider [9], Kraken [33], Lite Brite [32]) and Score-based ones (Octane [21], Basemark [12], WebXPRT [41]). We repeat the experiment 30 times and report the average results in Figure 5. Other tools were omitted as they could not run Chromium. On average, ASan-- reduces the overhead of ASan from 117.8% to 75.8%, presenting a reduction rate of 35.7%. If only counting the overhead of ASan checks, the reduction rate is 69.6%. Considering the scale of Chromium, such reduction rates are meaningful in general.

5.3 Applications

Fuzzing One of the most popular applications of ASan is to help bug detection in fuzzing. Our evaluation of this application focuses on how much ASan-- can improve the efficiency of ASan-enhanced fuzzing. We also include FuZZan (https://github.com/HexHive/FuZZan) as a baseline. To properly compare with FuZZan, we consider AFL-2.52b as the fuzzing tool and reuse the configurations presented in [24] (see Table 10 in the Appendix). For consistency, we run all the tests on Amazon EC2 instances with Intel XeonE5 Broadwell 16 cores, 64GB RAM, and Ubuntu 18.04 LTS. To avoid interference, different tests are run separately for 24 hours. All the tests are repeated 10 times with average results gathered. We also perform the evaluation twice, separately using seeds shipped with AFL and seeds shipped with FuZZan.

Table 4 shows the evaluation results. On average, ASan-- increases the execution speed of AFL by 42.1% and 39.8%, on AFL's seeds and FuZZan's seeds, respectively. The increase of execution speed leads to a growth rate of 5.0% and 5.1% in branch coverage. In comparison to FuZZan, ASan-- presents a comparable, but slightly smaller, improvement. Execution speed wise, FuZZan brings an increase of 50.2% and 45.7%, given AFL's seeds and FuZZan's seeds. Branch coverage wise, FuZZan brings a growth of 6.24% and 6.58%, depending on the seeds. The results are understandable, because FuZZan is tailored for fuzzing while ASan-- is more general.

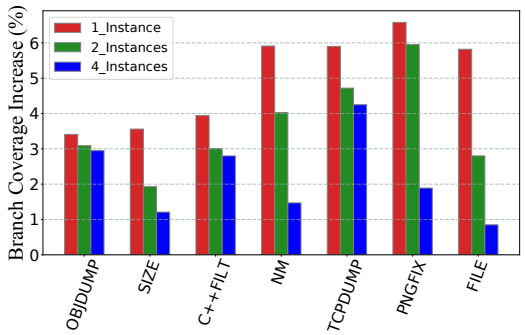

Figure 6: Increase of branch coverage brought by ASan-- with ASan as the baseline. 1/2/4 instance represents fuzzing with one sanitized instance and 0/1/3 non-sanitized instances.

As we pointed out in §3, ASan-- and FuZZan use orthogonal techniques. By intuition, combining techniques of both tools can produce better improvement. We, thus, migrate FuZZan's approach of reducing memory into ASan--. We limit the heap size to 16GB and relocate the heap plus stack [26] to be below the shadow memory. This way, we shrink the address space to 19GB (16GB heap + 1GB stack + 2GB BSS/DATA/-TEXT) and shadow memory to 19/8GB. Similar to FuZZan, we also disable the logging and relocate the poisoning of global data to run before AFL's fork server. As shown in Table 4, combining ASan-- and FuZZan outperforms both tools alone. It increases the execution speed of AFL by 59.4% and 59.3% on AFL's seeds and FuZZan's seeds. This leads to 8.99% and 9.73% more branch coverage. An outlier is `tcpdump`, where FuZZan runs the fastest. The primary reason is FuZZan also uses a red-black-tree based check to replace ASan check in certain cases, bringing extra optimization.

Our evaluation above runs only one fuzzing instance. In practice, we often need to run multiple instances in parallel and the community suggests using only one sanitized instance [8]. By intuition, the improvement of ASan-- will decrease with the number of non-sanitized instances. Accordingly, we perform an extra experiment where we increase the number of non-sanitized instances from 0 to 1 and 3, using AFL's seeds. Figure 6 shows the results. Not surprisingly, the improvement of ASan-- reduces when more non-sanitized in-

stances run. Nonetheless, ASan-- consistently benefits ASan.

Selective Deployment In general, ASan-- can still be considered unsuitable for deployed software because of its remaining runtime overhead and high memory cost. However, modern computers usually provide high computation power and large memory. Thus, it makes sense to selectively run ASan-- on critical deployed software (e.g., web browsers). Accordingly, we have been using the Chromium browser compiled with ASan-- for four weeks in our daily work. None of us experienced usability issues or significant slowdown.

To test ASan--'s capability of bug detection in the deployed environment, we collected 89 memory bugs from `bugs.chromium.org` and the CVE database. Replaying the bugs in the Chromium (same version as ours) pre-built with ASan by Google, we reproduced 4 bugs as listed in Table 5. Running the bugs in our Chromium, ASan-- detected all of them. We also tested the bugs with SANRAZOR and ASAP. However, both tools could not run Chromium. Alternatively, we performed static reasoning and observed that SANRAZOR and ASAP may miss 1 and 3 of the bugs (see Appendix C).

6 Related Work

6.1 Memory Error Detection

Spatial Error Detectors StackGuard [48] and P-SSP [50] insert random values before the `return` address and detect whether the random value is overwritten before the function returns. They introduce nearly zero runtime overhead but could only detect stack buffer overflows. Safe-C [10], CCured [37], SoftBound [36], Low-fat Pointers [27], and Gregory Jet al. [17] encode pointers with boundary information. At runtime, the information is used to check if memory accesses are within the boundaries. They can detect various spatial errors, but they have many problems (e.g., high runtime overhead and ABI incompatibility) that hinder their adoption.

Temporal Error Detectors DangNull [28], FreeSentry [53], and DangSan [47] keep track of pointers of allocated objects and invalidate pointers once the object is freed. CETS [35] maintains a unique identifier with each object and assigns a corresponding key to its pointers. When the object is freed, the lock is reset. Any access after `free` can be detected by checking if the key encoded in the pointer matches the lock of the object. Undangle [14] taints pointers at the `malloc` site and propagates the pointers dynamically. It detects temporal errors by checking if the source of a pointer has been freed. These methods usually have high runtime overhead.

Sanitizers ASan [42] and memory sanitizer (MSan) [44] use shadow memory to record the sanity of application memory. They further place checks on memory accesses to inspect the shadow bytes and detect errors at runtime. ASan can detect both spatial ones and temporal ones. MSan complements ASan to detect uninitialized memory read. Compared to other memory error detectors, ASan and MSan offer better detection, higher efficiency, and broader generality.

6.2 Address Sanitizer Optimization

Reducing Address Space Techniques in the line reduce the address space used by the application and the shadow memory. This reduces cost of memory management and the overall overhead. A technique is FuZZan [24]. As we have discussed FuZZan, we omit the details here. According to our study in 3, using the approach of FuZZan helps reduce runtime overhead but still cannot handle 90%+ of it.

Reducing Sanitizer Checks Techniques in this category adopt various strategies to reduce sanitizer checks. ASAP [49], PartiSan [29], and Bunshin [51] enforce sanitizer checks to a subset of the code. ASAP profiles the program to identify "hot" code that is more often executed. It removes checks in the hot code and retains the remaining. As such, fewer, cheaper checks are inserted. PartiSan creates different versions of the same code segment, making some versions more sanitized and the others less. At runtime, PartiSan adjusts the versions per the performance budget. Both ASAP and PartiSan are performance-driven, thus missing memory errors. In contrast, Bunshin runs multiple variants of the same software. It distributes the sanitizer checks into different variants. Bunshin does not lose checks, but it incurs several times more resource consumption. Similar to ASAP, SANRAZOR [55] also profiles the program, but the goal is to identify and remove redundant checks. In principle, SANRAZOR offers higher safety than ASAP but still no guarantee.

7 Conclusion

This paper first presents a study on dissecting the overhead of ASan, bringing knowledge about the primary sources of its overhead. Inspired by the study, the paper develops ASan--, a tool assembling a set of four optimizations to reduce the runtime overhead of ASan. Unlike existing techniques that negatively hurt the capability, scalability, or usability of ASan, ASan--'s techniques well maintain these decent properties. Our evaluation shows high utility of ASan--. It can reduce about 40% and 36% of the overhead introduced by ASan into SPEC CPU2006 and Chromium. Applying to fuzzing, ASan-- can meaningfully increase the execution speed and benefit the branch coverage. Deployed in Chromium for daily activities, ASan-- can detect bugs discovered by the community.

Acknowledgments

We thank our shepherd Wenke Lee and the anonymous reviewers for their feedback. This project was supported by National Science Foundation (Grant#: CNS-2031377; CNS-1718782), Office of Naval Research (Grant#: N00014-17-1-2787;

N00014-17-1-2788), and DARPA (Grant#: D21AP10116-00). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agency.

References

[1] "ARM Cortex-A Series Programmer's Guide for ARMv8-A," https://developer.arm.com/documentation/den0024/a/The-Memory-Management-Unit/Translation-table-configuration/Virtual-Address-tagging.

[2] "CPU 2017 Overview," https://www.spec.org/cpu2017/Docs/overview.html#metrics.

[3] "LLVM Alias Analysis Infrastructure," https://llvm.org/docs/AliasAnalysis.html#must-may-or-no.

[4] "LLVM's Analysis and Transform Passes," https://releases.llvm.org/8.0.0/docs/Passes.html#domtree-dominator-tree-construction.

[5] "The Often Misunderstood GEP Instruction," https://llvm.org/docs/GetElementPtr.html.

[6] "Spec cpu2006 documentation," https://www.spec.org/cpu2006/Docs/, 2011.

[7] J. Absar, "Scalar evolution-demystified," in *European LLVM Developers Meeting*, 2018.

[8] AFL++, "American Fuzzy Lop plus plus," https://github.com/AFLplusplus/AFLplusplus/blob/stable/README.md/#c-sanitizers.

[9] Apple, "Sunspider: a javascript benchmark," https://webkit.org/perf/sunspider-1.0.2/sunspider-1.0.2/driver.html.

[10] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, 1994, pp. 290–301.

[11] O. Bachmann, "Chains of recurrences," Ph.D. dissertation, Citeseer, 1997.

[12] Basemark, "Basemark: a comprehensive web browser performance benchmark," https://web.basemark.com/.

[13] R. Bodik, R. Gupta, and V. Sarkar, "Abcd: eliminating array bounds checks on demand," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, 2000, pp. 321–333.

[14] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 133–143.

[15] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "Savior: Towards bug-driven hybrid testing," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1580–1596.

[16] A. Developers, "Gwp-asan," https://developer.android.com/ndk/guides/gwp-asan, 2021.

[17] G. J. Duck, R. H. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers." in *NDSS*, vol. 17, 2017, pp. 1–15.

[18] A. Gampe, J. von Ronne, D. Niedzielski, and K. Psarris, "Speculative improvements to verifiable bounds check elimination," in *Proceedings of the 6th international symposium on Principles and practice of programming in Java*, 2008, pp. 85–94.

[19] A. Gampe, J. von Ronne, D. Niedzielski, J. Vasek, and K. Psarris, "Safe, multiphase bounds check elimination in java," *Software: Practice and Experience*, vol. 41, no. 7, pp. 753–788, 2011.

[20] GNU, "Binutils: a set of programming tools for creating and managing binary programs," https://ftp.gnu.org/gnu/binutils/.

[21] Google, "Octane: the javascript benchmark for the modern web," http://chromium.github.io/octane/.

[22] S. Guy, Eric, D. Andreas, and R.-P. Glenn, "Libpng: the official portable network graphics reference library," https://github.com/glennrp/libpng.

[23] D. Ian, "File is used to determine the type of a file," https://github.com/file/file.

[24] Y. Jeon, W. Han, N. Burow, and M. Payer, "Fuzzan: Efficient sanitizer metadata design for fuzzing," in *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 2020, pp. 249–263.

[25] S. Kostya, S. Evgenii, S. Aleksey, T. Vlad, and V. Dmitry, "Hwasan: An aarch64-specific compiler-based tool," https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html, 2018.

[26] T. Kroes, K. Koning, C. Giuffrida, H. Bos, and E. van der Kouwe, "Fast and generic metadata management with mid-fat pointers," in *Proceedings of the 10th European Workshop on Systems Security*, 2017, pp. 1–6.

[27] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight Jr, and A. DeHon, "Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 721–732.

[28] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing use-after-free with dangling pointers nullification." in *NDSS*. Citeseer, 2015.

[29] J. Lettner, D. Song, T. Park, P. Larsen, S. Volckaert, and M. Franz, "Partisan: fast and flexible sanitization via run-time partitioning," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018, pp. 403–422.

[30] C. Liu, Y. Chen, and L. Lu, "Kubo: Precise and scalable detection of user-triggerable undefined behavior bugs in os kernel," in *NDSS*, 2021.

[31] LLVM, "Undefined Behavior Sanitizer," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[32] Microsoft, "Litebrite: designed to measure the performance of your browser," https://testdrive-archive.azurewebsites.net/Performance/LiteBrite/.

[33] Mozilla, "Kraken javascript benchmark," https://krakenbenchmark.mozilla.org/kraken-1.1/driver.html.

[34] D. Mu, "Linux flaw project," https://github.com/VulnReproduction/LinuxFlaw, 2019.

[35] S. Nagarakatte, J. Zhao, M. Martin, and S. Zdancewic, "Cets: compiler enforced temporal safety for c," in *Proceedings of the 2010 International Symposium on Memory Management*, 2010, pp. 31–40.

[36] S. Nagarakatte, J. Zhao, M. M. Martin, and S. 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*, 2009, pp. 245–258.

[37] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 477–526, 2005.

[38] G. C. Necula, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy code," in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2002, pp. 128–139.

[39] D. Niedzielski, J. von Ronne, A. Gampe, and K. Psarris, "A verifiable, control flow aware constraint analyzer for bounds check elimination," in *International Static Analysis Symposium*. Springer, 2009, pp. 137–153.

[40] Nist, "Software assurance reference dataset," https://samate.nist.gov/SRD/testsuite.php, 2017.

[41] T. Principled, "Webxprt3: a browser benchmar," https://www.principledtechnologies.com/benchmarkxprt/webxprt.

[42] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012, pp. 309–318.

[43] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory Tagging and how it improves C/C++ memory safety," https://arxiv.org/pdf/1802.09517.pdf, 2018.

[44] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 46–55.

[45] V. J. Steve McCanne, Craig Leres, "Tcpdump: a data-network packet analyzer," http://www.tcpdump.org/release/.

[46] Y. Sui, D. Ye, Y. Su, and J. Xue, "Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions," *IEEE Transactions on Reliability*, vol. 65, no. 4, pp. 1682–1699, 2016.

[47] E. Van Der Kouwe, V. Nigade, and C. Giuffrida, "Dangsan: Scalable use-after-free detection," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 405–419.

[48] P. Wagle, C. Cowan *et al.*, "Stackguard: Simple stack smash protection for gcc," in *Proceedings of the GCC Developers Summit*. Citeseer, 2003, pp. 243–255.

[49] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder, "High system-code security with low overhead," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 866–879.

[50] Z. Wang, X. Ding, C. Pang, J. Guo, J. Zhu, and B. Mao, "To detect stack buffer overflow with polymorphic canaries," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 243–254.

[51] M. Xu, K. Lu, T. Kim, and W. Lee, "Bunshin: compositing security mechanisms through diversification," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 271–283.

[52] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in os kernels," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 661–678.

[53] Y. Younan, "Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers." in *NDSS*, 2015.

[54] J. Zhang, S. Wang, M. Rigger, P. He, and Z. Su, "Sanrazor artifact." https://github.com/SanRazor-repo/SanRazor, 2021.

[55] J. Zhang, S. Wang, M. Rigger, P. He, and Z. Su, "{SANRAZOR}: Reducing redundant sanitizer checks in c/c++ programs," in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021, pp. 479–494.

A Properties of Optimizations and Validation

This section aims to describe the properties of our optimizations regarding false positives and false negatives, followed by a validation of the properties. For simplicity, we will separately discuss spatial errors and temporal errors.

A.1 Removing Unsatisfiable Checks

This optimization only removes ASan checks. Hence, it will not introduce false positives. In the following, we focus on the discussion of false negatives.

Spatial Errors Our optimization ensures that the access falls into the legitimate range of the target object, thus avoiding false negatives.

`Proof by Contradiction:` Given a data object `buf` of `N` bytes on the stack or in the global, we assume the ASan check on a `K`-byte access `buf[offset]` is removed but the access goes out of bound. That means at least one path exists where `offset < 0 || offset+K > N`. This conflicts with `Condition-1` presented in §4.1, and, thus, is infeasible.

Temporal Errors Global objects are always alive. As such, it has no temporal errors. Stack objects can have *use-after-return* and *use-of-out-scope* issues. However, we only target direct stack access, where allocation of the object can be identified by LLVM. If the access is either use-after-return or use-of-out-scope, LLVM will throw a compilation error. Thus, false negatives are impossible.

A.2 Removing Recurring Checks

This optimization also only removes ASan checks, so there are no false positives. Therefore, we again focus on the discussion of false negatives.

Spatial Errors When removing ASan checks on `*ptr1`, we ensure there is another access, `*ptr2`, to the same memory location, and `*ptr2` exists in any path visiting `*ptr1`. This guarantees that any spatial error happening to `*ptr1` will also happen to `*ptr2`. Thus, the error will be captured by the check on `*ptr2`.

`Proof by Contradiction`: There are two situations of false negatives. First, `*ptr1` incurs a spatial error but `*ptr2` does not. That means `*ptr1` is in a redzone but `*ptr2` is not. This conflicts with that `*ptr1` and `*ptr2` are in the same memory location. Second, `*ptr1` is executed but `*ptr2` is not. This conflicts with that `*ptr2` exists in any path visiting `*ptr1`. In summary, false negatives are impossible.

Temporal Errors False negatives of temporal errors can only happen when the lifetime of the target memory changes. Specifically, given that `*ptr1` dominates `*ptr2` and the check on `*ptr2` is removed, false negative of use-after-free can happen when the memory is freed between `*ptr1` and `*ptr2`. This can happen in two scenarios. First, the memory is freed by the code between `*ptr1` and `*ptr2` through a heap deallocation interface. This is impossible because we will avoid removing the check in such a case (recall §4.2). Second, the memory is freed by another concurrent thread. This is possible, but it shall be a race condition issue instead of memory error, which in principle goes out of ASan's scope. We also want to note that, when `*ptr1` post-dominates `*ptr2` and the check on `*ptr2` is removed, no action is needed. The reason is that `*ptr1` always runs later than `*ptr2` and, hence, the use-after-free will be safely captured.

A.3 Merging Neighbor Checks

To simplify our discussion, we assume that the checks on `*ptr1` and `*ptr2` are merged into one by our optimization.

Spatial Errors False positives can never happen because whenever the merged check indicates an error, the original ASan checks will be performed again. Next we validate that false negatives cannot happen neither.

`Proof by Contradiction`: False negatives can happen only when `*ptr1` and/or `*ptr2` fall into a redzone but our merged check does not capture it. Formally, assuming that `*ptr1` `*ptr2` respectively fall into the memory ranges of `[addr_1l, addr_1h]` and `[addr_2l, addr_2h]`. False negatives mean our merged check does not cover the range of `[addr_1l, addr_2h]` (assuming `addr_1 < addr_2h`). This will conflict with `Condition-3` in §4.3. `Condition-3` ensures that our check covers the range `[addr_1l, addr_1l + MaxBitRead]` and `MaxBitRead > addr_2h - addr_1l`. False negatives are, thus, excluded.

Temporal Errors False positives are impossible. Our optimization ensures that `*ptr1` and `*ptr2` access the same data object, and modern heap allocator mandates that an data object cannot be partially freed. Thus, whenever use-after-free is detected, that certainly happens to `*ptr1` and/or `*ptr2`. False negatives can only happen when the object is freed between `*ptr1` and `*ptr2`. But this is OK since we always insert the merged check at the post-dominating access. The free will be safely captured.

A.4 Removing Neighbor Checks

This optimization again only removes ASan checks, so no false positives will be introduced. Below we discuss the case of false negatives. For simplicity, we assume `*ptr2` falls into the range between `*ptr1` and `*ptr3` and the check on `*ptr2` is removed. Further, we assume (i) `*ptr1`, `*ptr2`, and `*ptr3` fall into the memory ranges of `[addr_1l, addr_1h]`, `[addr_2l, addr_2h]`, and `[addr_3l, addr_3h]` and (ii) the location of `*ptr1`, `*ptr2`, and `*ptr3` ranges from low memory address to high memory address.

Spatial Errors False negatives are impossible.

`Proof by Contradiction`: False negatives can only happen when `*ptr2` is in a redzone but both `*ptr1` and `*ptr3` are not. That means `(addr_2l - addr_1h) + (addr_2h - addr_2l) + (addr_3l - addr_2h) >= MinRdSz`. This will conflict with `Condition-4`.

Temporal Errors False negatives of temporal errors are similar to the optimization of removing recurring checks. To rule out false negatives, we can reuse the approach applied to removing recurring checks. We avoid doing so because `*ptr1` and/or `*ptr3` post-dominate `*ptr2` in nearly every case and thus, use-after-free is typically captured.

A.5 Relocating Invariant Checks

Spatial Errors Since the address of invariant checks never changes, no false negative or false positive will be introduced.

Temporal Errors This optimization relocates a check to be executed later than the original one. Thus, no false negatives will be introduced because a freed object will not be reallocated. False positives can only happen when the object is freed in the code between the original access and the loop exit. This is possible, but our algorithm to identify invariant address (Algorithm 3) has considered such cases. If the target is possibly freed inside the loop, we will skill relocating the ASan check. Note that, similar to removing recurring checks, this optimization may report false temporal errors caused by de-allocation in another thread.

Table 5: Memory errors replayed with the deployed Chromium. Both ASan and ASan-- can detect all of them.

Module	Type	ASan	ASan--	Issue (with PoC)	Note
PDFium	heap-buffer-underflow	✓	✓	https://bugs.chromium.org/p/chromium/issues/detail?id=1116869	Compiling with `pdf_use_skia=true`
PDFium	heap-buffer-overflow	✓	✓	https://bugs.chromium.org/p/chromium/issues/detail?id=1099446	Compiling with `pdf_use_skia=true`
FreeType	heap-buffer-overflow	✓	✓	https://bugs.chromium.org/p/chromium/issues/detail?id=1139963	Linking `lib_freetype_harfbuzz`
GPU	heap-buffer-overflow	✓	✓	https://bugs.chromium.org/p/chromium/issues/detail?id=848914	Running with `-disable-gpu` option

A.6 Grouping Monotonic Checks

This optimization only reduces checks, so it will not introduce false positives. We discuss false negatives in the following.

Spatial Errors Given a chunk of memory ranging from `addr_l` to `addr_h` (`addr_h - addr_l < MinRdSz`), our optimization only checks the beginning bytes. False negatives may happen when the beginning bytes fall out of a redzone but the remaining bytes fall in. This is possible but will not bring false negatives. In general, there are two situations. First, the loop continues iterating and enters the next chunk. In this case, the beginning bytes of the new chunk will overlap with the redzone and the error will be detected. Second, the loop exits without entering the next chunk. In this case, the last access will touch the redzone and our check at the exit of the loop will detect the redzone.

Temporal Errors If the object is free when the loop iterate over the same chunk, false negative can happen. But it can only happen when the chunk is the last chunk, because otherwise the free will be detected when the next chunk is accessed. Similar to spatial errors, such cases shall be captured by our check at the exit of the loop.

B Other Applications of Optimizations

Our optimizations are designed to improve ASan as it draws more attention. However, they can also be applied to both Undefined Behavior Sanitizer (UBSan) [31] and Memory Sanitizer (MSan) [44]. Specifically, *removing unsatisfiable checks* and *relocating invariant checks* can be directly applied to UBSan. In fact, removing unsatisfiable checks has been applied on UBSan before and presented promising results [15, 30]. *Removing recurring checks* is also applicable, and it can be even further extended by defining recurring checks on non-memory objects (e.g., integers). Other optimizations may not be applied as UBSan uses no shadow memory.

All our optimizations, except *removing neighbor checks*, are applicable to MSan. Removing neighbor checks cannot be applied because MSan uses no redzones. When applied to MSan, our optimizations need the following customization:

- **Removing unsatisfiable checks:** In the context of MSan, the unsatisfiable condition should be redefined as that a memory read is dominated by a write to the same location.
- **Removing recurring checks:** This optimization is directly applicable. One thing worth noting is that when `*ptr2` post-dominates `*ptr1`, we need to make sure there is no write to the same location between `*ptr1` and `*ptr2`, before we remove the check on `*ptr1`.
- **Merging/removing neighbor checks:** The two optimizations are also applicable. When merging the checks on `*ptr1` and `*ptr2` (assuming `*ptr1` dominates `*ptr2`), we will place the merged check on `*ptr1` under the requirement that no write can happen to `*ptr2` after `*ptr1`. When removing neighbor checks, we also need to ensure no write happens between the removed one and dominating others.
- **Optimization loop checks:** The two optimizations for loops are in principle applicable. But for safety, we will have to make sure the target memory location is not initialized inside the loop. Statically guaranteeing this can lead to skipping of too many cases, and thus, the optimizations can present limited effectiveness.

C Discussion of Chromium Bugs

This section discusses unsound tools, ASAP and SANRAZOR, can detect the Chromium bugs and identify the reasons if not.

Issue 116869 (SANRAZOR ✓; ASAP ✗): The bug, shown in the code below, happens because the index `m_clipIndex` gets excessively decreased to become negative (line 4) and thus, the access to `m_commands` at line 5 goes out of bound. Since no memory access shares the same data dependency as `m_commands[m_clipIndex]`, SANRAZOR will not remove the check and can detect the bug. In contrast, ASAP will consider `m_commands[m_clipIndex]`, which happens in a loop, as hot code. If the `while` loop is executed in profiling with a low overhead budget, ASAP may remove its sanitizer check, and in result, miss detecting the bug.

```
1 void AdjustClip(int limit) {
2     while (m_clipIndex > limit) {
3         do {
4             --m_clipIndex;
5         } while (m_commands[m_clipIndex] !=
              Clip::kSave); // m_clipIndex < 0 and thus
              m_commands[m_clipIndex] underflows
6         m_pDriver->SkiaCanvas()->restore();
7     }
8 }
```

Issue 1099446 (SANRAZOR ✓; ASAP ✓): This bugs happens due to unsafe use of `memcpy` at line 5. The number of bytes operated is larger than size of the source buffer. Both SANRAZOR and ASAP should be able to detect this bug as they do not optimize the use of library functions.

```
sk_sp<SkData> SkData::PrivateNewWithCopy(...) {
    ...
    if (srcOrNull) {
        // heap-buffer-overflow here; The size of
             "srcOrNull" is 68 bytes, but "length" is 256
        memcpy(data->writable_data(), srcOrNull, length);
    }
    return data;
}
```

Issue 1139963 (SANRAZOR ✗; ASAP ✗): This bug is caused by an integer truncation. At line 4 and line 6, `imgWidth` and `imgHeight` are truncated from 32 bits to 16 bits. The truncated values are then used to calculate the size of a bitmap at line 9. However, the bitmap is still accessed using a 32-bit index at line 20. When `imgWidth` and/or `imgHeight` is larger than 65535 ($2^{16}-1$), the allocated buffer cannot fit the bitmap and thus, overflow can happen at line 22.

According to SANRAZOR's definition, `dp` at line 16 and `dp32` at line 20 share similar data dependency as they "flow" from the same source. Further, SANRAZOR deems line 17 as a "user-check". As such, SANRAZOR considers that ASan check on `*dp32` at 20 is a redundancy of line 17 and will remove it. This way SANRAZOR shall miss the memory error. Similar to Issue 116869, `*dp32` at 22 happens in a loop, and ASAP may miss it when a low overhead budget is given.

```

FT_LOCAL_DEF(FT_Error) Load_SBit_Png(...){
    ...
    metrics->width = (FT_UShort)imgWidth;
    //truncate 32-bit to 16-bit
    metrics->height = (FT_UShort)imgHeight;
    //truncate 32-bit to 16-bit
    map->pitch    = (int)( map->width * 4 );
    FT_ULong size = map->rows * (FT_ULong)map->pitch;
    error = ft_glyphslot_alloc_bitmap( slot, size );
    // allocate a buffer; when width / height is over
         65535, the buffer cannot fit the bitmap
    ...
}
void png_combine_row(png_const_structrp png_ptr,
     png_bytep dp, int display){
    ...
    if (dp && sp && ... ){
        png_uint_32p dp32 = dp, sp32 = sp;
        do{
            do{
                *dp32++ = *sp32++;//dp32 points to the
                     buffer and overflows
                ...
            }while (c > 0);
        }while(bytes_to_copy <= row_width)
}
```

Issue 848914 (SANRAZOR ✓; ASAP ✗): Code below demonstrates the bug. In this issue, `id_states_` is a C++ standard vector. When an `id` object is created, it will be inserted to `id_states_` and the index of the object will be saved in `ids`. However, when objects are popped out from `id_states_`, the index saved to `ids` is not updated. As a consequence, heap overflow can happen when `id_states_` is accessed using the old index (`id_states_[id - 1]` at line 5). This case is similar to Issue 116869. SANRAZOR will not remove the check and can detect the overflow. However, if the loop is executed in profiling with a low overhead budget, ASAP may remove the check and miss detecting the bug.

```
bool FreeIds(...) override {
    ...
    for (GLsizei ii = 0; ii < n; ++ii) {
        GLuint id = ids[ii];
        if (id != 0) id_states_[id - 1] = kIdPendingFree;
    }
    ...
}
```

D Supplementary Algorithms and Data

Table 6: Page fault and runtime overhead before and after we shrink the shadow memory of ASan.

Benchmark	# of Page Faults (K)		Runtime Overhead	
	Before	After	Before	After
400.perlbench	187,718	138,214	273%	230%
401.bzip2	1,159	933	81.9%	67.9%
403.gcc	136,262	91,466	101%	60.1%
429.mcf	626	465	54.1%	44.3%
445.gobmk	432	390	76.9%	63.8%
456.hmmer	373	333	179%	176%
458.sjeng	89	78	102%	87.7%
462.libquantum	555	413	108%	102%
464.h264ref	390	315	117%	89.4%
473.astar	607	526	70.8%	62.6%
483.xalancbmk	648	593	130%	99.7%
433.milc	29,400	21,380	89.1%	83.7%
444.namd	50	46	86.1%	82.3%
447.dealII	1,694	1,424	138%	135%
450.soplex	12,863	9,294	103%	95.1%
453.povray	124	117	101%	82.5%
470.lbm	117	138	62.4%	57.4%
482.sphinx3	12,863	9,293	103%	99.4%
Average	21,442	15,301	107%	95.5%

Table 7: Comparison of compilation time and binary size

Benchmark	Compilation Time (Sec)			Binary Size (MB)		
	ASan	ASan--	Overhead	ASan	ASan--	Reduced
400.perlbench	65.2	397	6.1x	8.7	7.9	9.20%
401.bzip2	12.1	21.5	1.8x	2.7	2.0	25.9%
403.gcc	25.2	85.7	3.4x	22	18	18.2%
429.mcf	7.03	11.1	1.5x	2.4	2.1	12.5%
445.gobmk	20.3	32.5	1.6x	13	11	15.4%
456.hmmer	18.8	35.7	1.9x	3.8	3.3	13.2%
458.sjeng	7.14	10.7	1.5x	2.3	1.8	21.7%
462.libquantum	7.12	14.1	1.9x	2.5	1.7	32.0%
464.h264ref	31.4	113	3.6x	6.2	5.4	12.9%
473.astar	10.8	15.3	1.4x	2.7	1.9	29.6%
483.xalancbmk	50.3	412	8.2x	52	48	7.69%
433.milc	12.1	21.9	1.8x	2.9	2.3	20.7%
444.namd	17.3	57.1	3.3x	3.8	3.4	10.5%
447.dealII	44.7	125	2.8x	47	39	17.0%
450.soplex	6.81	19.7	2.9x	7.2	6.8	5.56%
453.povray	9.81	45.1	4.6x	8.8	8.1	7.95%
470.lbm	4.57	7.76	1.7x	2.4	1.6	33.3%
482.sphinx3	6.17	9.34	1.5x	3.3	2.8	15.2%
Average	19.8	79.7	2.8x	10.7	9.2	17.1%
Chromium	3.46h	7.28h	2.1x	352	280	20.4%

Table 8: The amount of ASan checks after the adoption of ASan--. Baseline is ASan with the default setting. From left to right, the columns show the amount of ASan checks left after enabling different optimizations in ASan--, one after another.

Benchmark	Ratio of ASan Checks					
	Unsat_Chk	Rec_Chk	Merge_Chk	Rem_Neighbor	Loop_Inv	Loop_Grp
400.perlbench	86.64%	64.74%	61.39%	58.71%	55.53%	53.83%
401.bzip2	88.29%	73.48%	71.38%	71.19%	70.02%	68.90%
403.gcc	85.99%	67.85%	60.49%	56.74%	54.86%	52.93%
429.mcf	82.02%	63.79%	60.53%	57.58%	51.74%	51.51%
445.gobmk	92.54%	80.72%	72.85%	68.32%	65.39%	60.25%
456.hmmer	92.69%	79.14%	75.11%	73.38%	66.42%	63.54%
458.sjeng	88.07%	78.44%	77.18%	76.21%	74.80%	74.02%
462.libquantum	64.82%	58.29%	57.84%	56.33%	55.96%	55.44%
464.h264ref	87.59%	78.17%	69.71%	64.46%	58.76%	56.36%
473.astar	92.76%	74.38%	73.84%	73.31%	71.63%	71.09%
483.xalancbmk	92.04%	75.62%	69.79%	63.59%	60.01%	56.94%
433.milc	89.91%	77.48%	75.42%	73.61%	72.24%	70.95%
444.namd	90.55%	78.19%	76.31%	71.01%	66.21%	63.90%
447.dealII	85.87%	68.36%	64.26%	57.65%	48.57%	39.89%
450.soplex	92.83%	80.47%	77.55%	76.21%	70.50%	66.60%
453.povray	84.93%	68.72%	63.41%	57.13%	52.37%	47.14%
470.lbm	93.19%	86.54%	80.48%	79.36%	77.35%	75.92%
482.sphinx3	92.90%	82.07%	79.69%	78.75%	76.91%	73.96%
Average	87.98%	74.25%	70.40%	67.42%	63.85%	61.29%

Table 9: Performance improvement brought by FuZZan to ASan on SPEC CPU2006. Cells with "–" indicate no results.

Benchmark	Performance Increase			
	1G	4G	8G	16G
401.bzip2	–	–	0.96%	0.19%
429.mcf	–	–	2.44%	1.63%
456.hmmer	–	–	–	1.78%
458.sjeng	–	0.33%	-0.66%	0.50%
462.libquantum	–	–	3.86%	4.18%
433.milc	–	–	0.68%	0.45%
444.namd	1.89%	1.26%	1.89%	1.26%
453.povray	–	–	2.62%	4.12%
470.lbm	–	–	–	2.05%
482.sphinx3	–	–	–	0.71%

Table 10: Setup of fuzzing evaluation.

Programs				Settings
Name	Version	Driver	Source	Options
BINUTILS	2.31	OBJDUMP	[20]	-a @@
BINUTILS	2.31	SIZE	[20]	@@
BINUTILS	2.31	C++FILT	[20]	-n
BINUTILS	2.31	NM	[20]	@@
LIBPCAP	5.0.0	TCPDUMP	[45]	-n -e -r @@
LIBPNG	1.6.38	PNGFIX	[22]	@@
FILE	1.62	FILE	[23]	-m magic.mgc @@

Table 11: Runtime overhead on SPEC CPU2017. `**_r` represents SPECrate and `**_s` represents SPECspeed.

Benchmark	Performance Overhead							
	ASAN	Unsat_Chk	Rec_Chk	Merge_Chk	Rem_Neighbor	Loop_Inv	Loop_Grp	No_Sanitizer_Chk
505.mcf_r	62.78%	57.28%	49.84%	45.31%	40.78%	38.83%	35.60%	10.68%
508.namd_r	143.8%	128.9%	112.8%	98.93%	90.37%	82.89%	73.80%	48.66%
510.parest_r	144.8%	135.2%	111.4%	101.1%	91.73%	76.27%	72.13%	25.86%
511.povray_r	262.0%	245.3%	229.2%	209.1%	199.4%	187.2%	180.8%	102.7%
519.lbm_r	71.12%	67.38%	61.50%	55.61%	52.41%	48.13%	42.25%	13.36%
520.omnetpp_r	130.6%	123.1%	120.7%	118.5%	117.2%	111.2%	106.4%	64.51%
531.deepsjeng_r	98.81%	89.68%	79.76%	67.06%	63.10%	55.95%	46.83%	20.23%
538.imagick_r	143.3%	134.9%	124.6%	113.5%	100.1%	86.70%	76.85%	39.65%
541.leela_r	85.37%	80.98%	78.29%	76.10%	67.07%	65.12%	59.51%	21.46%
544.nab_r	77.78%	73.12%	65.59%	58.06%	48.75%	37.28%	31.18%	16.84%
Average_r	122.05%	113.59%	103.38%	94.34%	87.08%	78.97%	72.55%	36.40%
605.mcf_s	63.88%	58.66%	44.33%	39.85%	34.03%	28.81%	25.60%	15.07%
619.lbm_s	68.19%	61.97%	53.47%	47.67%	41.87%	34.92%	28.19%	10.26%
620.omnetpp_s	145.3%	140.9%	137.9%	134.1%	127.6%	123.5%	118.8%	66.67%
631.deepsjeng_s	112.0%	103.3%	92.33%	86.33%	76.33%	66.00%	60.00%	31.00%
641.leela_s	120.7%	112.8%	103.7%	92.84%	88.89%	82.72%	78.02%	29.63%
Average_s	102.03%	95.56%	86.36%	80.17%	73.74%	67.19%	62.13%	30.53%

Algorithm 3: IDENTIFYING INVARIANT IN LOOP

```
1  Procedure backward_slicing_inLoop(A, L, V):
2      if ¬ L.contains(A) || V.find(A) then
3          return;
4      end
5      V.push_back(A);
6      if TypeInst P = dyn_cast<TypeInst>(A) then
7          for each op in P.getOperands() do
8              backward_slicing_inLoop(op, L, V);
9          end
10     end
11 return;
12 Procedure check_addr_type(A, V, P):
13     if P.find(A) then
14         return NON_INV;
           /* A has been visited, so it is recursively
              involved. Consider it as a variant for
              safety.                                  */
15     end
16     for each user in A.users() do
17         if isa<CallInst>(user) ||
            isa<LifeTimeEndIntrinsic>(user) then
18             return NON_INV;
               /* the memory may have been freed       */
19         end
20         if isa<StoreInst>(user) then
21             return NON_INV;
               /* the value may have been modified     */
22         end
23     end
24     if ¬V.find(A) then
25         return INV;
           /* value comes from outside of loop, so
              mark it an invariant                     */
26     end
27     P.push_back(A);
28     if TypeInst K = dyn_cast<TypeInst>(A) then
29         for each op in K.getOperands() do
30             opTy = check_addr_type(op, V, P);
31             if opTy == NON_INV then
32                 return NON_INV;
33             end
34         end
35     end
36     return INV;
37 Algorithm
       Input  : Address A, Loop L
       Output : INV or NON_INV
38     Initialize: P = vector();
39     backward_slicing_inLoop(A, L, V);
40 return check_addr_type(A, V, P);
```