OPTISAN: Using Multiple Spatial Error Defenses to Optimize Stack Memory Protection within a Budget

Rahul George[2], Mingming Chen[1], Kaiming Huang[1],
Zhiyun Qian[2], Thomas La Porta[1], Trent Jaeger[2]

[1]The Pennsylvania State University
[2]University of California, Riverside

Abstract

Spatial memory errors continue to be the cause of many vulnerabilities. While researchers have proposed several defenses to prevent exploitation of spatial memory errors, systems currently rely on defenses that only protect a small fraction of stack data (e.g., return addresses) and leave a window of vulnerability (e.g., by only enforcing on function returns). One proposal to address this problem is to place defenses at the lowest cost locations until a cost budget was met, but this approach only considers a single defense and does not account for the security implications of possible placements. In this paper, we propose the OPTISAN system, which is the first system to apply multiple spatial memory defenses to maximize the number of objects protected from spatial memory errors within a cost budget. OPTISAN analyzes each program to identify the stack objects that may be exploited by spatial memory errors, called *usable targets*, and estimates the overhead for individual defense operations, for both metadata management and spatial checks, to enable flexibility in placement choices. OPTISAN applies this information in a novel Mixed-Integer Non-Linear Programming formulation to generate an optimal placement. We apply OPTISAN to generate placements using a combination of identity-based (i.e., influential BaggyBounds) and location-based (i.e., widely used AddressSanitizer (ASan)) spatial memory defenses, finding that OPTISAN utilizes the more effective Baggy Bounds defense broadly, augmenting it with ASan to increase the number of memory operations with usable targets protected by 18.4% on average across a set of benchmark and server programs. OPTISAN shows that using multiple spatial memory defenses provides valuable flexibility to prevent the exploitation of many spatial memory errors within a cost budget.

1 Introduction

Since the "Anderson report" [7] in 1972, researchers have understood that spatial memory errors, such as buffer overflows [54], could be exploited to compromise process execution. A spatial memory error allows a program instruction to access memory locations outside of the memory region of the intended object. By exploiting spatial memory errors, adversaries may read unauthorized stack data (i.e., stack buffer overread or disclosure attacks) and/or modify unauthorized data (i.e., stack buffer underflow or overflow). Such attacks have been exploited in-the-wild at least since the Morris Worm [58] in 1988 to hijack process execution, and recent vulnerabilities show that such attacks remain a high priority problem even for stack memory (e.g., [9, 47–49, 53]).

Despite broad awareness of spatial memory errors and their impact, only a small fraction of stack data is protected from only a subset of spatial memory errors. First, deployed defenses only check for the modification of return addresses (e.g., due to buffer overflows), traditionally using *stack canaries* [17] and now using *shadow stacks* [2, 15]. However, current defenses do not detect spatial memory attacks that modify local variables without affecting return addresses, nor will they detect illicit reads to stack memory that could lead to *disclosure attacks* [23, 57, 63]. Second, probabilistic defenses, such as ASLR [55] make some attacks more difficult, but adversaries can still exploit objects at relative offsets to perform data-oriented attacks [29, 32]. Third, such defenses only detect illicit stack modification when the vulnerable function returns, leaving a window of opportunity for an adversary to exploit the error fully prior to detection. For example, an adversary could exploit a spatial memory error within the execution of one long-running function to achieve their goals (e.g., perform necessary system calls to modify or leak sensitive data) prior to the vulnerable function returning.

To address the limitations above, researchers have proposed many techniques to detect spatial memory errors more comprehensively. Researchers categorize such defenses [64] into *identity-based defenses* [6, 19, 21, 50, 72], which validate that every memory access is within the expected memory region, and *location-based defenses* [20, 59, 66, 73, 74], which detect memory accesses to invalid memory outside the expected memory region. Despite efforts to improve the performance of both techniques [6, 19, 21, 25, 39, 72–74], their performance overhead still remains too high for broad adoption. A question

is how we can take advantage of such defenses effectively to improve program security.

In this paper, we leverage two insights. First, we find that identity-based and location-based defenses can have significantly different overheads for protecting the same unsafe memory operations due to differences in their implementations. For some operations, location-based defenses may be much more efficient, but for other operations, identity-based defenses may be much more efficient. Thus, a method for placing multiple spatial memory defenses may be beneficial and should account for the performance implications of each defense placement, including the *residual overheads* [38, 69] (e.g., metadata management) of defenses.

Second, even a combination of strong defenses may not be able to prevent all spatial memory errors within a desired performance budget. Thus, we need a technique to determine how to place multiple defenses within a budget. The ASAP system proposed applying individual defenses to operations in increasing order of frequency until an overhead budget is consumed [69]. However, ASAP does not consider the security implications of these choices nor does it account for the residual overheads, acknowledging that the total residual overheads of some programs are greater than the budget. We want to devise a defense placement approach that accounts for performance and security to utilize a combination of location-based and identity-based defenses effectively.

In this paper, we present OPTISAN, which is a tool for generating placements of spatial memory defenses for stack objects that utilizes multiple defenses, where the goal is to maximize the protection of stack memory from spatial memory errors within a cost budget. We develop a tool, OPTISAN, that constructs and solves protection budget problems, which includes: (1) a static analysis to compute which stack objects may be exploited by unsafe memory operations, which we call *usable targets*, to estimate the security impact; (2) a performance model of each defense that captures the costs for the placement of metadata management and bounds checks for each operation to enable flexible selection of placements; and (3) a *mixed-integer non-linear program* (MINLP) formulation to generate an optimal solution for protecting usable targets within a cost budget. While (1) and (2) are customized to protect stack objects, the protection budget formulation (3) is independent of the memory region being protected as long as security impact and defense cost models can be provided.

We apply OPTISAN to SPEC CPU 2006 programs, SPEC CPU 2017 programs and five server programs[1]. First, by applying a combination of Baggy Bounds [6] (i.e., an influential identity-based defense) with AddressSanitizer [59] (ASan) (i.e., a widely used location-based defense), OPTISAN is able to generate placements that cover 18.4% more unsafe operations with usable targets on average than Baggy Bounds alone for the same cost budget. Second, we show that OPTISAN makes placement decisions that account for the number of target objects to increase the protection from each operation defended, when compared to ASAP [69]. Third, we examine nine recent CVEs, finding that these vulnerable unsafe operations are not run frequently, which is consistent with ASAP's hypothesis of most CVEs [69], but nonetheless, OPTISAN places defenses for these unsafe operations earlier than ASAP in six instances and at the same point for the other three, indicating that OPTISAN is able to robustly balance performance and security. Fourth, OPTISAN is also reasonably accurate in predicting the overhead of placements, where the average difference from each budget is 1.47%, enabling us to generate desired placements in three runs or fewer.

This work contributes the following:

- We develop a novel model of spatial error defenses that captures both metadata and check operation costs at the granularity of individual program locations, enabling accurate cost estimates.
- We propose a novel mixed-integer non-linear program (MINLP) formulation to generate solutions that use multiple defenses to maximize the protection of stack objects from spatial errors within a cost budget.
- OPTISAN generates placements for a combination of ASan and Baggy Bounds defenses, increasing the number of unsafe operations protected by 18.5% on average over Baggy Bounds alone, while biasing the protection of stack objects and providing predictable performance.

2 Motivation

2.1 Stack Spatial Memory Errors

A *spatial memory error* occurs when a memory access is performed outside of the memory region of the intended referent, which is possible in programming languages that do not enforce memory safety (C/C++). The *intended referent* [18] of a pointer is the object from whose base address the pointer was derived. Spatial memory errors may either read (e.g., buffer over-read) or write (e.g., buffer overflow) memory illicitly. While spatial memory errors have been known for over 50 years [7], they are difficult to identify, even with automated static and dynamic (e.g., fuzzing) analyses. As a result, programs written in C/C++ often have latent spatial memory errors. For example, over 200 stack buffer overflow vulnerabilities were found in the last three years, 122 of which have a severity rating of at least 7. These include vulnerabilities in Adobe Acrobat Reader (CVE-2023-21610), Adobe Animate (CVE-2023-22243), and OpenSSL (CVE-2022-3786).

Consider Figure 1, which contains code snippets for three stack memory accesses in different functions in the Apache web server (httpd 2.4.32). Researchers have proposed techniques [6, 30] to identify the memory accesses cannot violate spatial memory safety, such as by using value-range analysis [62]. Using such techniques, the stack memory access in

[1] Only five SPEC CPU 2006 and four SPEC CPU 2017 programs run more than 100 stack memory operations that may violate spatial memory safety. We evaluate our approach on these programs only.

```
1 AP_DECLARE(int) ap_parse_form_data(request_rec *r, ap_filter_t *f,
        apr_array_header_t **ptr, apr_size_t num, apr_size_t usize)
2 {
3     ...
4     char buffer[HUGE_STRING_LEN + 1];
5     ...
6     if (offset < HUGE_STRING_LEN) {
7       // Stack memory access 1
8      buffer[offset++] = c;
9     }
10 }
```

Listing 1: Stack Memory Access 1

```
1  static int check_nonce(request_rec *r, digest_header_rec *resp,
2      const digest_config_rec *conf)
3 {
4     apr_time_t dt;
5     char tmp,hash[NONCE_HASH_LEN+1];
6     time_rec nonce_time;
7     .........
8     tmp = resp->nonce[NONCE_TIME_LEN];
9     resp->nonce[NONCE_TIME_LEN] = '\0';
10    // Stack memory access 2
11    apr_base64_decode_binary(nonce_time.arr, resp->nonce);
12    .........
```

Listing 2: Stack Memory Access 2

```
1 static remoteip_parse_status_t remoteip_process_v1_header(conn_rec *c,
2 remoteip_conn_config_t *conn_conf, proxy_header *hdr, apr_size_t len,
3 apr_size_t *hdr_len)
4 {
5     //Local Target
6     char *host;
7     char buf[sizeof(hdr->v1.line)];
8     ...........
9     // Stack memory access 3
10    strcpy(buf, hdr->v1.line);
11    .........
12    // Reachable use  1
13    conn_conf->client_addr = c->client_addr;
14    ...........
15    // Reachable use 2
16    conn_conf->client_ip = apr_pstrdup(c->pool, host);
17    ...........
18 }
```

Listing 3: Stack Memory Access 3

Figure 1: Stack memory accesses in the Apache web server

Line 8 of Listing 1 will always remain within the memory region of the intended referent, buffer. However, the stack memory accesses in Listings 2 and 3 cannot be validated statically to satisfy spatial memory safety. For these operations, developers can either apply defenses proactively to prevent possible spatial memory errors or hope that the stack memory access is actually safe or not exploitable. We find that the Apache web server (httpd 2.4.32) has 2,969 memory operations (i.e., instructions in the LLVM IR) that cannot be proven to be safe from spatial memory errors (called *unsafe operations* in Table 3 in Section 6) via value-range analysis.

Consider the code snippet in Listing 3, which has been reported as a recent vulnerability (CVE-2019-10097 [52]). Listing 3 shows a function named remoteip_process_v1_header that copies a packet header into a stack buffer buf, using strcpy. The function strcpy cannot be proven to comply with the bounds allocated for either the buffer buf or the header hdr, so it may result in a buffer overflow and/or buffer over-read. In this case, adversaries can overflow the buffer buf to overwrite stack objects allocated above buf on the stack. As a result, the local pointer variable host can be modified to point to any desired payload, which can be used to corrupt important fields in the configuration object conn_conf in **line 17**. 14 stack objects may be targeted by this vulnerability.

```
1 ShadowAddr = (Addr >> 3) + Offset;
2 k = *ShadowAddr;
3 if (k != 0 && ((Addr & 7) + AccessSize > k))
4     ReportAndCrash(Addr);
5 // Memory access (store)
6 *Addr=...
```

Listing 4: ASan Check Operation

```
1 // Pointer arithmetic
2 p = baseAddr + offset;
3 k = bounds_table[baseAddr>>slot_size];
4 if ((baseAddr^p) >> k!=0)
5     p = BaggySlowPath(baseAddr,p);
6 .....
7 // Memory access (Store)
8 *p=...
```

Listing 5: Baggy Bounds Check Operation

2.2 Spatial Error Defenses

Programmers currently apply inexpensive, but incomplete and/or probabilistic defenses, such as stack canaries [17] and Address Space Layout Randomization [55] (ASLR). Stack canaries (and shadow stacks [11, 15]) check that the value of the return address at the function's return is the same as when the function was called. However, the function's local variables may be illicitly modified or read without detection. In addition, attacks that complete prior to the function returning and attacks where the adversary controls the offset will also not be detected by these defenses. ASLR randomizes the base location of the stack and code memory segment to prevent attackers from predicting the location of target code or data. However, attacks whose target is at a relative offset from a vulnerable memory region may still be accessed illicitly. For example, the vulnerability in Listing 3 above allows attackers to modify a local variable illicitly that is located at a known offset from the vulnerable buffer buf, but canaries do not prevent illicit access to local variables and ASLR does not prevent targeting objects at known offsets.

As a result, researchers have proposed defenses specifically targeted at preventing spatial memory errors. Researchers recognize two classes of spatial memory defenses, as identity-based and location-based defenses [64]. First, *identity-based defenses* rely on the intended referent of the memory access to determine validity of the access. For example, the SoftBound defense [51] tracks whether a pointer to an intended referent may ever reference memory outside of the bounds of its allocated region. Because identity-based defenses check whether a pointer value is within a memory range, these defenses instrument every pointer arithmetic operation. The Baggy Bounds defense [6] removes instrumentation for memory accesses that can be proven to satisfy spatial safety and reduces the cost of checks by aligning and padding allocations to the next power of two. The Listing 5 shows this efficient check which involves a single memory load (i.e., bounds table lookup), arithmetic operations, and a comparison. Baggy Bounds is currently the most efficient identity-based defense, so we examine its use in this paper.

Alternatively, *location-based defenses* detect spatial memory errors that access an invalid memory region. The idea is to use a metadata store, such as a *shadow memory*, to

Program	*ASan (%)*	*Baggy (%)*	*ASan is Slower (%)*	*Baggy Improves (%)*
httpd	13.15	19.66	2.43	16.76
sqlite3	0.45	0.63	0.84	12.69
sjeng	33.08	53.08	5.45	12.10
povray	5.50	6.63	30.22	49.45
gcc	8.94	14.17	1.02	17.37
perlbench	10.75	20.95	3.05	2.33

The first two columns contain the performance overhead to protect all unsafe operations using ASan and Baggy Bounds, respectively. The third and fourth columns show the percentage of cases where ASan is slower and the percentage reduction in cost if Baggy Bounds is used instead.

Table 1: Performance Trade-offs between ASan and Baggy Bounds

track state for each byte (or a portion) of the usable address space. This metadata store is consulted on each memory access to detect (illegal) accesses to invalid memory. These defenses differ in how they encode invalid memory, as either red-zones [26,27,59,61] around the object encoded in shadow memory or separate unallocated (guard) pages [45,56] outside legitimate memory regions. Address Sanitizer [59] (ASan) is a widely used location-based defense, often used in fuzz testing [46] to detect bugs. For example, ASan has detected more than 10,000 memory bugs [24,68] across various applications including over 3,000 memory bugs in Chrome [24]. It is designed to be more efficient than prior techniques [26,27,61].

ASan employs a directly mapped, compact shadow memory to detect operations that access memory outside the expected bounds using *red-zones*. For stack objects, a red-zone is 32 bytes. Listing 4 shows an ASan check, which involves a single memory load (shadow memory lookup), arithmetic operations, and a comparison. In this paper, we examine the use of the ASan defense[2]. We note that location-based defenses may be bypassed, e.g., by incrementing a pointer beyond an object's red zone , or through an unaligned access that is partially out-of-bounds [60]. Identity-based defenses cannot be bypassed, but may not be applicable to all scenarios (e.g., buffers within objects).

Neither identity-based nor location-based defenses are deployed in production environments because their performance overheads are not yet acceptable. Table 1 shows the overheads of ASan and Baggy Bounds for spatial memory error protection for all unsafe operations (i.e., stack memory accesses that cannot be proven safe by value-range analysis) for a set of programs. For benchmarking these programs, we use the test suite provided with the program such as Apache HTTP Test project [8] for Apache, the TCL test suite [65] for sqlite, and *ref* workload for the SPEC CPU 2006 programs. ASan always has a lower overall performance overhead than Baggy Bounds for these programs, but we observe that ASan may have a higher overhead than Baggy Bounds for some fraction of the unsafe operations in the program (ASan Is Slower column). In these cases, the number of memory accesses (checked by ASan) is greater than the number of pointer arithmetic operations for those objects, leading ASan to incur a higher performance overhead than Baggy Bounds. The fourth column (Baggy Improves column) shows the percent overhead reduction (i.e., relative to the ASan column) if Baggy Bounds is used instead of ASan in those cases only.

[2]ASan– [74] is an optimized version of ASan, but there are implementation issues in applying it incrementally with another defense, so we will explore using it in future work.

2.3 Managing Spatial Defense Performance

Researchers have proposed methods to apply spatial memory defenses in ways that limit their overhead. For example, the ASAP system [69] applies the Address Sanitizer [59] (ASan) spatial memory defense to as many operations as possible within a cost (i.e., performance overhead) budget. ASAP chooses which operations to defend in the order of their expected execution frequency (e.g., from an execution profile), starting with the least frequent operations, until a budget is exhausted. ASAP demonstrates that a significant fraction of memory operations can be defended (87% on average for the programs tested) within a cost budget of 5% for some programs in the SPEC CPU 2006 and Phoronix benchmarks.

While ASAP takes an important first step towards a method to utilize stronger memory defenses wisely, challenges remain. First, ASAP cannot combine multiple defenses, which may be desirable based on their performance trade-offs. As seen in Table 1, in some cases the BaggyBounds defense [6] enforces memory safety more efficiently than ASan, as explained earlier in Section 2.2. ASAP does not account for the frequencies of operations performed by defenses nor their overheads. Second, ASAP does not account for the residual costs [38, 69] of a defense (e.g., ASan's red-zone metadata management), but these costs can be significant. As a result, ASAP cannot generate placements for a desired budget for programs using ASan with high residual costs (i.e., more than the cost budget), such as gcc and perlbench. Third, ASAP does not account for whether a memory operation can actually enable any exploitation. As a result, ASAP may waste the valuable performance budget unnecessarily on operations that cannot be exploited. Fourth, for a desired budget ASAP may choose to protect operations that are inexpensive to protect (low execution frequency), but present fewer risks (e.g., fewer stack objects at risk) than other operations. While vulnerabilities along frequently executed paths appear to be less common, such vulnerabilities have been found in the wild as seen in recent work [74]. Thus, ASAP's approach may lead to such vulnerabilities being left unprotected even when they present a high risk. Instead, we propose that considering multiple defenses (i.e., performance trade-offs), the risk associated with the unsafe operations (i.e., stack objects at risk), and all their costs (i.e., check and residual costs) may enable a protection budget to be used more effectively.

Ultimately, we still lack an effective approach to apply spatial memory defenses to protect the stack. Ideally, we would like to apply the complete identity-based defenses comprehensively, but they are too expensive in many cases. Given

that location-based defenses, even though they are incomplete, would be a significant improvement over the current defenses, we would like to be able to leverage them as well to maximize our protection of stack memory within a cost budget. However, we lack techniques that understand the performance implications of defenses comprehensively (e.g., including metadata) to apply defenses efficiently and predictably (i.e., incur the expected overhead overall after placement). In addition, we lack techniques that can utilize accurate performance modeling to maximize the defense we can obtain with location-based and identity-based defenses in combination. While Baggy Bounds avoids placing defenses when operations are safe from spatial errors, it does not account for whether operations are actually exploitable (i.e., any illicitly accessed target may be used). In addition, while ASAP accounts for cost [69] imprecisely, it does not consider whether a placement of multiple defenses and whether it protects the maximal amount of stack data within a cost budget.

3 System Overview

Threat Model We assume that programs may contain spatial memory errors on accesses to stack data, such as stack buffer under/overflows and buffer over-reads. We assume that adversaries can exploit any spatial memory error to access other objects illicitly in stack memory. We assume that memory accesses that are validated to satisfy memory safety are not exploitable, and that the techniques used to validate memory safety [6, 30, 62] do not misclassify any unsafe accesses as safe (i.e., are *sound*). In addition, we assume that any stack memory modified illicitly must have a *use* (i.e., a memory access to the illicitly modified stack object) after the unsafe operation to be exploitable. We assume any stack data that may be accessed by an illicit read can be exploited (e.g., by using the buffer into which the illicitly accessed data is read). We leave the problem of extending protection to heap and global objects for future work, as discussed in Section 7.

We assume the following defenses are already applied to the program. First, we assume that the program employs a defense to prevent the modification of code and the execution of data, such as Data Execution Prevention [4] (DEP). We assume that these protections cannot be disabled (e.g., no bugs exploit changing memory permissions using `mprotect`). Second, we assume that the ASan and Baggy Bounds defenses prevent illicit modification of their metadata and execute correctly. We assume that no attacks on memory accesses to the heap or globals can access stack memory or can circumvent ASan and Baggy Bounds.

OPTISAN **System** The OPTISAN system is shown in Figure 2. OPTISAN generates a placement of defenses, in this case for the ASan [59] and Baggy Bounds [6] defenses, to maximize the protection of stack objects from spatial memory errors within a cost budget. To do this, OPTISAN first identifies the operations that may violate memory safety (i.e., *unsafe operations*) using known techniques (e.g., value-range analysis [62]) in Step (0). For each unsafe operation, OPTISAN computes the stack objects that may be exploited by spatial memory errors, called *usable targets*, in Step (1), which is described in Section 4.3. Next, OPTISAN estimates the expected cost of defenses at each location where defense code may be placed, called *monitoring points*, both for spatial memory checks and for initialization and cleanup of metadata used by these defenses, using offline profiles in Step (2), as described in Section 4.1. OPTISAN then applies a novel optimization formulation to generate a defense placement that maximizes the protection of stack objects from spatial memory errors within a given performance cost budget in Step (3), as described in Section 4.2. Finally, OPTISAN instruments the program with the computed placement automatically in Step (4), including the incremental metadata support for the limited set of checks placed. The placement of instrumentation utilizes the existing ASan and Baggy Bounds methods, as described in Section 5.

4 OPTISAN Design

In this section, we describe how to develop an accurate and flexible performance model of defenses to enable maximizing the protection of stack objects within a cost budget.

4.1 Modeling Defense Performance Overheads

The goal is to apply defenses within a cost budget in a manner that accounts for the performance overhead of each defense accurately and thus the performance trade-offs between defenses. To estimate the cost of a particular defense placement accurately, we must develop a model of the performance overhead of defenses that is comprehensive (i.e., accounts for all significant costs), fine-grained (i.e., can reason about any placement), and flexible (i.e., can apply either defense to each unsafe operation). The key to achieving this goal is to accurately model the overheads of both check (i.e., prevent accesses that violate spatial memory errors) and metadata operations (i.e., setup and update the state for checks, such as red-zones and bounds). Past work [69] acknowledged that metadata operations could add significant overhead to enforcement due to *residual overheads* [38, 69], which were found to be greater than the placement budget for some programs, such as perlbench and gcc. However, modeling metadata operations presents challenges because metadata operations are associated with objects that may have multiple unsafe operations and require multiple spatial checks. In addition, since OPTISAN considers the placement of multiple defenses with different metadata representations and operations, OPTISAN must reason about the impact of redundant metadata.

OPTISAN is designed to model each check and metadata operation required to account for all defense operations (i.e., be comprehensive) at the granularity of individual defense operations per unsafe operation (i.e., be fine-grained). This enables OPTISAN to choose among multiple defenses for each

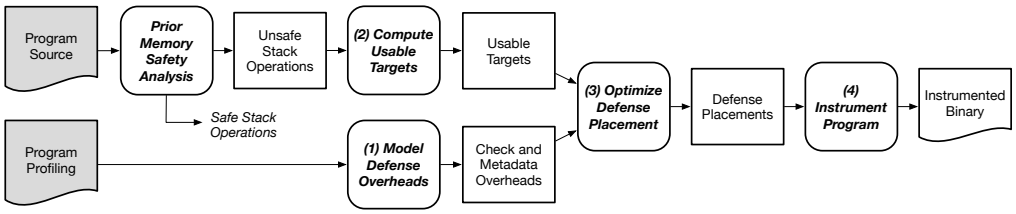

Figure 2: OPTISAN Steps: (1) model the cost of defense operations to assess the impact of their use to prevent spatial errors; (2) compute the target stack objects that may exploited by spatial memory errors in unsafe operations; (3) generate optimal defense placements to maximize prevention of the exploitation of usable targets within a cost budget; (4) instrument the program with the optimal defense placement.

unsafe operation (i.e., be flexible). The key to this approach is to model the check and metadata operations for each defense explicitly through what we call *monitoring points*: the code locations where such operations may be applied. For applying a defense to an unsafe operation, OPTISAN identifies the code locations where its check and metadata operation(s) are required to enforce spatial safety as monitoring points. This enables OPTISAN to determine the cost when a defense is chosen for an unsafe operation based on each of the monitoring points required for the check and metadata operations for that unsafe operation, the cost of each operation individually, and the frequency of execution of each monitoring point used. In addition, monitoring points allow OPTISAN to capture shared operations for each defense, i.e., metadata operations that may be amortized over multiple unsafe operations.

Cost Estimation To estimate the average cost of individual check and metadata operations for each defense, we profile programs to determine the costs attributable to individual check and metadata operations for each defense. We identify check and metadata operations for each defense by matching the IR description of each operation, similar to prior work [69]. We then execute the program instrumented in a variety of ways, similar to recent work [74], to determine the total overhead for all check and metadata operations as separate groups. We execute the program in the following ways for each defense: (1) natively (i.e., without any instrumentation); (2) with full instrumentation; (3) with all metadata operations (including heap), but no checks; and (4) with only stack metadata operations, but without checks. Note that for (2) and (3), we include both heap and stack instrumentation, which is functionally the same for both types of objects, mainly for ease of testing. The total cost of the spatial memory checks (for the heap and stack) is determined by subtracting the runtime of (3) from (2). The total cost of the metadata operations for the stack is determined by subtracting (1) from (4). The average cost for each class of operations is determined by dividing the total cost of each class by the total frequency of the relevant operations obtained using profiling (*gcov*). The costs of any initialization operations for these defenses are built into the metadata costs at present. We then associate the average check and metadata operation costs with the individual, fine-grained operations, *monitoring points*. This fine-grained association of costs with individual monitoring points enables us to estimate the cost of any placement of spatial memory checks and metadata operations of any defense.

One issue that we had to address to apply the performance modeling technique above is to enable the application of metadata operations to individual stack objects as desired. Currently, Baggy Bounds and ASan add stack metadata at the granularity of a function rather than for indivdiual stack objects. That is, should any stack object in a function require a spatial check, then metadata is allocated for all the function's stack objects for both Baggy Bounds and ASan. As we apply defenses for a smaller fraction of objects, many metadata operations may become spurious. In addition, since we may apply multiple defenses to different stack objects in the same function, redundant metadata operations may be created. Thus, we modified ASan and Baggy Bounds to enable the creation of metadata per stack object. We discuss how the metadata can be further improved in Section 7.

4.2 Optimizing Defense Placement

In this section, we formulate a *Mixed Integer Non-Linear Programming* (MINLP) problem to compute optimal defense placements to maximize the protection of the targets of attacks on spatial memory errors from exploitation within a cost budget. Designing this formulation requires us to develop a method to estimate protection from spatial memory errors to compare possible solutions. Then, we integrate the performance modeling from Section 4.1 to determine which solutions are within budget.

Figure 3 shows a view of the intuition behind our proposed method to estimate protection. Given the call graph on the left, unsafe operations are distributed among the program's functions (e.g., Unsafe 1 in Func A). Should an unsafe function have a spatial memory error, this may allow an adversary to illicitly access *targets*, i.e., stack objects other than the intended reference of the operation. Figure 3 shows a mapping between unsafe operations and targets. However, only a subset of the targets that may be illicitly accessed may be used by another instruction after the unsafe operation. We call these *usable targets*, which are associated with unsafe operations by bold edges in Figure 3. The placement of defenses may prevent spatial memory errors, indicated by the **X**s in Figure 3. However, applying a defense for one unsafe operation may

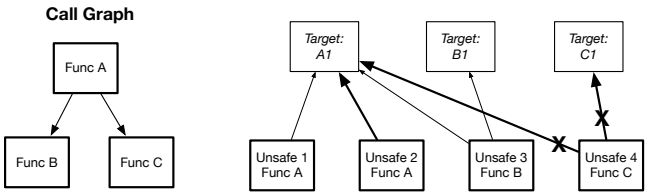

Figure 3: Shows how unsafe operations map to target stack objects in the program represented by the call graph shown. Edges map unsafe operations (e.g., Unsafe 1) to target stack objects (e.g., variable A1 in function A. Bold edges indicate that a target is used in after the unsafe operation (i.e., is a *usable target*). **X**s indicate defenses prevent spatial memory errors. However, target A1 may still be exploited by a spatial memory error from operation Unsafe 2.

Table 2: Definitions and Parameters

$x_{j,p}$	Indicator of defense p at unsafe operation j.
$y_{k,p}$	Indicator of monitoring point k of defense p.
$a_{j,p}$	Accuracy of defense p at unsafe operation j.
T_i	Set of unsafe operations for target i.
$M_{j,p}$	Monitor points of defense p for unsafe operation j.
D_i	Minimum detection accuracy threshold for target i.
f_k	Frequency of monitoring point k.
c_p	Cost of defense p.
g_i	Gain of target i.
B	Monitoring budget.
S_k	Set of unsafe operations for object i.
l	Number of stack objects to be monitored.
n	Number of unsafe operations.
m	Number of targets.
o	Number of sanitizer types, $o \geq 2$.
q	Number of monitor points.

not fully protect a target. While target C1 is fully protected by enforcing spatial memory safety on unsafe operation Unsafe 4, target A1 may still be exploited by an attack on unsafe operation Unsafe 2 followed by its use (i.e., A1 is a usable target of Unsafe 2). In general, for a target to be protected from spatial memory errors a complete defense must guard every unsafe operation in which that target is usable.

We formalize the model shown in Figure 3, based on the parameters definitions in Table 2, as follows. First, any improvement in attack prevention depends on how comprehensively each target is protected from illicit access via unsafe operations. We refer to this as the *gain*, g_i, of target object i. We use the equation below to estimate the gain g_i for each target object, as Constraint (3) in the formulation.

$$g_i = \frac{\sum_{j \in T_i} \sum_{p=1}^{o} g_{i,j,p}}{|T_i|},$$

where: (1) T_i is the set of unsafe operations for target i; o is the number of defense types considered; and $g_{i,j,p}$ is the gain attributable to defending each unsafe operation j with defense p for target object i. For a specific defense p applied to an unsafe operation j for which target i is a usable target, the gain of the said unsafe operation is $g_{i,j,p} = a_{j,p} \cdot x_{j,p}$, where $a_{j,p}$ is the accuracy of the defense p at unsafe operation j and $x_{j,p}$ is a boolean indicating whether defense p is applied at unsafe operation j [3].

A question is how to estimate the impact of defending each unsafe operation individually for each target. For example, the effect of defenses for a target could be the minimum defense applied to any unsafe operation that could illicitly access this target. However, using the minimum defense means that any individual unsafe operations left unprotected by only one unsafe operation may undermine the defense rating overall. Since we have only a limited budget, we assume that some unsafe operations may be unguarded. As a result, we aggregate impact of each unsafe operation by averaging the defense provided, as shown above. Should any or all targets require defense, the formulation includes a threshold constraint to specify the minimum required defense accuracy for each target in a solution, as Constraint (4) in the formulation below.

To model the performance overhead, we utilize the estimates of the overheads of check and metadata operations determined using the methods described in Section 4.1. For each defense, we determine where to place check and metadata operations by computing its *monitoring points* per unsafe operation. We define $y_{k,p}$ to represent whether a monitoring point k of defense p is used.

$$y_{k,p} = \begin{cases} 1, & \text{if monitoring point } k \text{ active for defense } p \\ 0, & \text{otherwise} \end{cases}$$

Constraint (5) of the formulation below ensures that when a defense p is applied for unsafe operation j, all the monitoring points $M_{j,p}$ for that defense are activated.

The overall formulation is shown below. The objective is to maximize the sum of the gain for all of the target objects in Equation 1. The objective value represents the protection of all *usable targets*, aggregating the protection at each relevant unsafe operation for each usable target.

$$\max \sum_{i=1}^{m} g_i \tag{1}$$

Subject to :

$$\sum_{p=1}^{o} x_{j,p} \leq 1, \quad \forall j \in \{1,...,n\} \tag{2}$$

$$\forall i \in \{1,...,m\}, \quad g_i - \frac{\sum_{j \in T_i} \sum_{p=1}^{o} a_{j,p} \cdot x_{j,p}}{|T_i|} = 0 \tag{3}$$

$$g_i \geq D_i, \quad \forall i \in \{1,...,m\}, \tag{4}$$

[3] The accuracy of a defense applied to an unsafe operation is the same for all targets of that operation and dependent on the defense technique chosen, as discussed in Section 6.

Algorithm 1: findUsableTargets(op, OBJs, PDG)

```
1: {op:an unsafe operation}
2: {PDG: inter-procedural program dependence graph}
3: U ← ϕ
4: Funcs ← ϕ
5: {Collect Allocating Functions of OBJs to Funcs}
6: Funcs ← FuncsAllocObjs(OBJs)
7: while Funcs do
8:     for each function F ∈ Funcs do
9:         { Collect Stack Objects from F as Target Objects}
10:        T ← StackObjects(F)
11:        { Usable Targets Have a Use Reachable without a Kill}
12:        for each target t ∈ T do
13:            for each use u ∈ Uses(t) do
14:                K ← Kills(t)
15:                if !GraphCut(op,u,K,PDG) then
16:                    U ← U ∪ t
17:                end if
18:            end for
19:        end for
20:    end for
21:    Funcs ← Funcs ∪ FindCaller(F,PDG)
22: end while
23: return U
```

$$x_{j,p} = 1 \Rightarrow y_{k,p} = 1, \quad \forall y_{k,p} \in M_{j,p} \tag{5}$$

$$\sum_{p=1}^{o} c_p \sum_{k=1}^{q} f_k \cdot y_{k,p} \leq B \tag{6}$$

$$x_{j,p} \in \{0,1\} \tag{7}$$

$$y_{k,p} \in \{0,1\} \tag{8}$$

Each unsafe operation can select at most one defense, Constraint (2). Constraint (6) is the budget constraint, which restricts the placement selected to the cost budget.

One issue of the formulation above is that we place defenses at the unsafe operation granularity, rather than stack object granularity. This allows the solver to employ one defense at one unsafe operation and another defense at a different unsafe operation for the same stack object. The use of multiple defenses for the same stack object may be wasteful because the system has to maintain multiple copies of metadata for the same object. We propose an optional constraint to select defenses at the object granularity instead to prevent the creation of redundant metadata. In this case, the unsafe operations of the same object, S_k, are limited to use the same defense by adding the Constraint (9).

$$\sum_{p,q \in \{1,\ldots,o\}, p \neq q} \left(\sum_{j \in S_k} x_{j,p} \cdot \sum_{j \in S_k} x_{j,q} \right) = 0, \quad \forall k \in \{1,\ldots,l\} \tag{9}$$

However, we find that allowing the solver flexibility to choose among defenses, even when accounting for the creation of redundant metadata, results in better solutions in many cases, as discussed in Section 6.

4.3 Computing Usable Targets

In this section, we describe a method to compute the usable targets of an unsafe operation in the stack memory region. Solving this problem consists of two main tasks: (1) collecting the target objects of an unsafe operation and (2) determining whether each target object is usable in an exploit. Algorithm 1 shows this approach, where lines 6-10 collect the targets of an unsafe operation and lines 12-16 determine whether the target is usable. As research in triaging vulnerabilities to determine what exploits may be possible is a nascent and emerging field [14, 32], we propose preliminary, conservative solutions to these two problems. We hope that future work will develop more precise techniques to solve these problems.

Algorithm 1 uses a program dependence graph [34] (PDG), which represents the program's control flows and data flows in one model. We use the PtrSplit [40] PDG representation, which provides a sound alias analysis to ensure an over-approximation of all control and data flows. We apply value-range analysis [62] to the program's PDG representation to identify unsafe operations based on a prior, open-source implementation [30]. Value-range analysis over-approximates unsafe behaviors, classifying an operation as unsafe if it cannot be proven safe. Thus, some operations classified as unsafe operations may not actually be unsafe.

Next, the task is identify all target objects for each unsafe operation. Currently, we take a naive, but conservative, position that any object that could be present on the stack at the time of an unsafe operation may be illicitly accessed by the unsafe operation. Thus, we use the call graph to identify all functions that may be present on the stack for each unsafe operation. The set of stack objects for each of these functions (including the unsafe operation's function) forms the set of target objects.

Recall Figure 3 from Section 4.2. In the given call graph, an unsafe operation in Function A can only illicitly access objects in that function. However, unsafe operations in Functions B and C can both access stack objects of Function A, although not each other's stack objects.

OPTISAN determines whether a target object is a usable target in two ways. For the targets of read operations, we assume that they all have uses (i.e., are all usable targets) since read operations typically aim to use the data read. This assumption is an overapproximation as some data may be read and found not to match a criteria for use, but these criteria are often ad hoc. We will explore more refined interpretations of uses of read operations in future work.

For the targets of write operations, OPTISAN determines whether there are any operations that use the target after it may be illicitly modified. OPTISAN first identifies operations that may perform a memory access to the target. We compute these operations using the intra-procedural data flows (i.e., based on the alias analysis) for the function in which the target object was allocated. We prune operations prior to the unsafe operation using the control flow. In addition, we prune operations that are dominated by a kill operation for the target. We stop the computation at uses that pass a target object to a callee or escape the function through global or heap memory,

declaring the target as usable in those cases.

5 Implementation

We develop a tool, OPTISAN, to automate the proposed approach. It comprises of four key components: (1) LLVM-based [37] static analyses to compute usable target objects for unsafe operations in Step 1 of Figure 2; (2) an LLVM-based static analysis to estimate defense overheads from profiling information, as described in Section 4.1, in Step 2 of Figure 2; (3) a mixed-integer, non-linear program solver [1] to compute the optimal placement within a performance overhead budget for Step 3 of Figure 2; (4) LLVM-based static instrumentation pipeline to instrument the program in Step 4 of Figure 2 using ASan and Baggy Bounds as per the computed placement. OPTISAN's static analyses, performance utility pass, and instrumentation passes are built using LLVM-10.0 in around 5*K* lines of code (LOC). We decouple the LLVM analyses for Step 1 (i.e., computing usable targets) and Step 4 (i.e., placing instrumentation) by using a graph database, Neo4j [3] to store the results of Step 1 to apply the computed placement in Step 4. This enables us to perform the safety analysis [6, 30] and compute usable targets once, and apply the solver (Step 3) to produce solutions that are instrumented (Step 4) independently. Our tool is publicly accessible at `https://github.com/rahultgeorge/OptiSan`.

Modeling Defense Overheads To obtain the defense overheads in Step 1 in Figure 2, we compute the execution profile to enable OPTISAN to estimate the overheads of check and metadata operations for ASan and Baggy Bounds defenses as described in Section 4.1. To obtain the execution profile of the program it is instrumented using *gcov*. The execution profile is stored in a Neo4j graph database [3]. To assign the defense overheads per monitoring point using the profile results, we use an LLVM-based static analysis pass.

Computing Usable Targets In Step 2 of Figure 2, OPTISAN computes the usable targets for unsafe operations. To compute the unsafe operations, OPTISAN leverages the well-known value-range analysis [62] on the program dependence graph (PDG) constructed from prior work [40]. We leverage the SVF [67] points-to analysis to compute the aliases used in the PDG to identify all stack objects that may be referenced by these unsafe operations. To compute the usable targets, we implement the Algorithm 1 described in Section 4.3 using LLVM-based static analyses [37] in around 2K LOC. We rely on LLVM IR to identify stack objects. The passes save the results in a Neo4j graph database.

Optimizing Defense Placements In Step 3 of Figure 2, OPTISAN computes optimal placements using the MNLIP formulation proposed in Section 4.2 implemented using the Gurobi solver [1]. The solver stores the selected placement in the graph database by annotating the specific unsafe operations. The static analysis performed in the Step 1 computes the necessary information needed to instrument any of the unsafe operations with the selected defense.

Instrumenting Defense Placements To instrument a program as per the computed placement, Step 4 of Figure 2, OPTISAN applies the results from the prior steps stored in the Neo4j graph database in LLVM-based static instrumentation passes [37]. The specific unsafe operations and necessary functions are fetched from the database. The pass uses debug information to identify the operations and relevant functions. The functions and unsafe operations are annotated in the IR. This avoids instrumenting unnecessary functions.

The program is instrumented using Baggy Bounds first and then Address Sanitizer as per the computed placement to prevent conflicts. However, there may be redundant checks as each defense is unaware of the specific unsafe operations and instruments each annotated function completely. Subsequently, another instrumentation pass checks for any such redundant checks and removes them using a technique from prior work [69]. The compiler then proceeds as normal and generates the instrumented binary.

6 Evaluation

To demonstrate the efficacy of the proposed placement approach, we attempt to answer the following questions:

- Does OPTISAN's ability to consider multiple defenses protect more unsafe operations? How are the multiple defenses used together?
- Does the security impact of unsafe operations improve the choice of which operations are protected?
- Does considering security impact with cost enable OPTISAN to apply protection for the unsafe operations exploited in real CVEs earlier than prior work?
- Can OPTISAN produce an optimal placement with the desired overhead budget?

We evaluate OPTISAN on the SPEC CPU 2006 programs, SPEC CPU 2017 programs, and seven real-world programs, including five programs from an open-source fuzzing benchmark[4]. We analyze the sixteen C/C++ programs in the SPEC CPU 2017 benchmark (i.e., all the non-Fortran programs). For profiling these programs, we use the test suites provided, such as Apache HTTP Test project [8] for Apache, the TCL test suite [65] for sqlite and the *ref* workloads for the SPEC CPU (2006, 2017) benchmarks [5].

The counts of unsafe operations and targets per program are shown in Table 3. Notably, we find by leveraging value-range analysis [62] that only five of the SPEC CPU 2006 programs, four of the SPEC CPU 2017 programs, and five of the real-world programs have a significant number of unsafe stack

[4]We select all the programs from the Magma benchmark [28] that have a large number of non-free unsafe operations and a test suite.

[5]We modify the Apache HTTP Test to ignore modules without any unsafe operations.

	Name	Unsafe Operations				Usable Targets			
		Read	*Write*	***Non-Free***	*Total*	*Objects*	*Pointers*	***Non-Free***	*Total*
SPEC CPU 2006	mcf	0	0	0	0	-	-	-	-
	libquantum	1	0	0	1	7	0	0	7
	bzip2	4	5	0	9	34	6	0	40
	hmmer	2	6	4	8	16	5	9	21
	h264ref	43	51	17	94	86	8	13	94
	astar	4	2	6	6	42	11	53	53
	milc	30	21	49	51	83	18	93	101
	sphinx3	12	3	14	15	19	8	3	27
	sjeng	247	54	220	301	878	104	803	982
	gobmk	1,465	932	2,397	2,397	9,965	548	10,513	10,513
	povray	281	201	139	482	675	255	497	930
	gcc	3,893	1,894	4,438	5,787	14,630	7,243	15,881	21,873
	perlbench	1,721	648	2,129	2,369	4,302	6,427	6,647	10,729
SPEC CPU 2017	**511.povray**	1,714	505	2,147	2,219	4,505	2,887	7,250	7,392
	600.perlbench	136	522	658	658	7,982	10,717	18,699	18,699
	602.gcc	69	162	189	231	31,209	57,453	73,752	88,662
	625.x264_s	217	732	829	949	895	256	980	1,151
Real programs	**httpd**	1,834	1,135	1,565	2,969	1,208	900	1,452	2,108
	sqlite3	130	47	119	177	461	392	573	853
	redis-cli	307	144	358	451	1,316	1,332	2,561	2,648
	libxml2	1,885	729	2,607	2,614	2,402	5,609	8,004	8,011
	openssl	50	48	98	98	14,529	23,585	38,114	38,114
	libtiff	2	18	20	20	16	23	39	39

The columns contain the number of unsafe operations and usable targets for various SPEC CPU2006 programs, SPEC CPU2017 programs, and **six** server programs.

Table 3: Counts of Unsafe Operations (relative to spatial memory safety) and Their Usable Target Objects

operations (Table 3 under the column Unsafe Operations, Total). For SPEC CPU 2006 and server programs, Table 3 shows that many have under 100 non-free, unsafe operations. For the SPEC CPU 2017 programs, seven of the sixteen programs do not have any unsafe stack operations and five of the remaining nine programs can be protected with a negligible overhead of 0.30% or less, on average. Furthermore, as identified in prior work [69], a significant number of unsafe operations are never executed in the provided workloads, which can be covered for free (i.e., relative to the workload). For the programs in Table 3, 28% of the unsafe operations on average can be protected for free, leaving the remaining unsafe operations to be consider for protection by OPTISAN, as listed in Table 3 under the column Unsafe Operations, Non-Free.

As described in Section 4.2, the potential security impact of unsafe operations can be characterized through the *usable targets*, also shown in Table 3. Examining the impact of the workload on usable targets, 33% of the usable targets can be protected for free on average (i.e., they may only be accessed by unsafe operations that are not run in the workloads). In the following experiments, we only consider the non-free, unsafe operations and the usable targets accessed by those operations, in the Usable Targets, Non-Free column in Table 3.

ASan Accuracy Recall that unlike Baggy Bounds, in some cases the ASan checks may be circumvented, as described in Section 2.2, so ASan may not prevent all attacks on spatial errors. To account for this, we want to estimate ASan's accuracy of enforcement between 0 (i.e., defense can be bypassed) and 1 (i.e., defense cannot be bypassed). Recent work [74] used the Juliet test suite to validate the correctness of their proposed ASan optimizations, so we use the NIST Juliet C/C++ *v*1.3 test suite [10] to estimate the accuracy of the ASan defense. As shown in Table 4, there are four CWEs [5] relevant to stack spatial memory errors in the test suite. For CWE 126 and CWE 127, we only examine the stack test cases. To estimate the accuracy of ASan, we identify the test cases (i.e., one unsafe operation per test case) for each of these CWEs in the Juliet test suite [10] where ASan can be bypassed. Similar to recent work [74], we estimate accuracy by finding the test cases where ASan may be bypassed because an offset used in the memory access is variable and may be larger than the red-zone size. For example, in several cases offsets are only restricted to be a positive value. We estimate ASan accuracy for each CWE as the fraction of the remaining Juliet test cases that do not exhibit possible bypass as shown in Table 4. We take the average across these four CWEs to obtain an average accuracy of 0.80 for ASan, which we use in this evaluation.

CWE	***Description***	***ASan Accuracy***
CWE 121	Stack Buffer Overflow	0.95
CWE 124	Stack Under-write	0.77
CWE 126	Buffer Overread	0.70
CWE 127	Buffer Underread	0.76

Table 4: ASan accuracy estimates for the NIST Juliet test suite using its bad tests. Bad tests check cases that may bypass red-zones.

Baggy Bounds Accuracy While Baggy Bounds is a complete defense in many cases, it cannot prevent spatial errors in some specific cases identified in Section 2.2. We do not apply Baggy Bounds in cases where it cannot provide an accurate defense (e.g., scalar field within a structure).

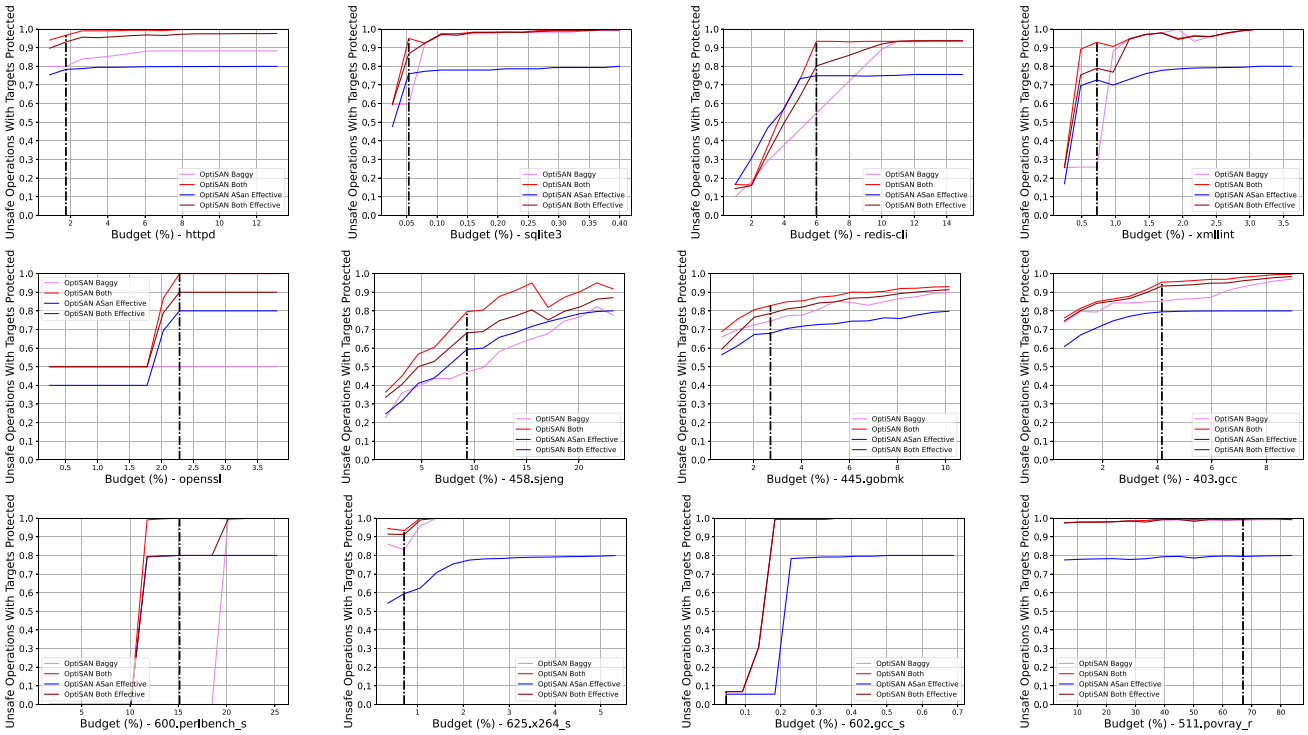

Figure 4: OPTISAN's defense placement solutions: Compare the number of unsafe operations protected by Baggy Bounds alone (OPTISAN Baggy, Pink) to both Baggy and ASan together (OPTISAN Both, Red). The dashed vertical lines show the cost budget where the maximum difference in unsafe operations protected. Also shown are the effective number of unsafe operations protected when accounting for the limited accuracy of ASan for the combination of Baggy Bounds and ASan (OPTISAN Both Effective, Brown) and ASan alone (OPTISAN ASan, Blue).

6.1 Applying Multiple Defenses

The key benefit of applying OPTISAN is to leverage the capabilities of multiple defenses to maximize protection of usable targets. The plots in Figure 5 a) show the measured impact of applying Baggy Bounds and ASan together to protect non-free unsafe operations with usable targets[6]. The OPTISAN Baggy line (Pink) shows the number of unsafe operations that are protected by Baggy Bounds for a budget in the optimal placement using the formulation in Section 4.2. We compare this to the number of unsafe operations that can be protected when both Baggy Bounds and ASan are available in the OPTISAN Both line (Red). In many cases, the combination enables many more unsafe operations to be protected at particular budget levels. The dotted vertical line shows the budget with the greatest improvement by using both defenses over Baggy alone. Note that in some cases, the number of unsafe operations protected decreases as the budget increases, but this is because the formulation is maximizing the gain in protection for usable targets, not the number of unsafe operations.

Figure 4 also accounts for the limited accuracy of ASan in the OPTISAN Both Effective line (Brown). Because ASan may be bypassed in some cases, the effective number of unsafe operations protected is lower than the number in which it and Baggy Bounds are applied. However, even the OPTISAN Both Effective line shows a significant improvement over Baggy Bounds alone in many cases for particular budgets, showing that OPTISAN provides an opportunity to improve security within a budget by combining defenses. Finally, the curve for OPTISAN ASan Effective (Blue) shows the effective protection provided by ASan alone for the unsafe operations. Although ASan is more efficient (performance-wise) overall, its limited accuracy reduces its impact below that of the other options in most cases, which is the common assumption.

We find that OPTISAN Both (Red) protects 18.4% more non-free unsafe operations on average across all fourteen programs for all budgets evaluated, ranging from a minimum of 0% to a maximum of 52%, compared to OPTISAN Baggy (Pink). We note that Baggy Bounds is the more cost efficient choice for all the unsafe stack operations for three programs (i.e., 625.x264_s, 602.gcc_s, and libxml2) [7]. This along with frequency distribution of the unsafe operations for 602.gcc_s leads to only Baggy Bounds being used. When accounting for accuracy, we find that OPTISAN Both Effective (Brown) effectively protects 14% more non-free unsafe operations

[6] Twelve of the fourteen test programs from Table 3 are shown in Figure 4. The remaining two programs, povray and perlbench in SPEC CPU 2006, do not exhibit a significant improvement compared to OPTISAN Baggy alone when applying both defenses.

[7] We note that for 511.povray_r the OPTISAN Both and OPTISAN Baggy lines do not overlap completely i.e., there is minor improvement compared to OPTISAN Baggy alone. ASan is only selected for a maximum of 4.7% of the unsafe operations covered.

across all fourteen programs for all budgets evaluated, ranging from a minimum of 0% to a maximum of 41%, compared to OPTISAN Baggy Effective. Further, we focus on the maximum difference at any budget shown in Figure 4 by the dotted vertical lines. The average maximum difference shown is 51.27% with a minimum of 0% and a maximum of 256%.

Finally, we also evaluate OPTISAN by limiting each unsafe stack object to only one defense to remove all redundant metadata. This restriction causes fewer unsafe operations, 3% on average, to be protected effectively across all programs compared to OPTISAN Both. By allowing the formulation to consider redundant metadata, OPTISAN is able to find more optimal solutions that use multiple defenses for objects used in multiple unsafe operations.

Usage of Defenses We find that when both defenses are considered Baggy Bounds is selected for 86% of the unsafe operations on average across programs for all budgets. In addition to the greater enforcement accuracy, we find two factors that affect Baggy Bound's usage when considering both defenses. First, most programs have a skewed frequency distribution (positively skewed) for the unsafe operations [69], i.e., only a small fraction of operations are really hot and incur majority of the cost to defend, as shown for Apache in Figure 5 a). We find that on average across all the programs evaluated 19% of the non-free, unsafe operations account for 80% of the cost. Intuitively, Baggy Bounds tends to be applied to unsafe operations that have a lower frequency (cost), which is the common case. Second, as shown in Section 2.2 quantitatively, we find that Baggy Bounds can be a more efficient option than ASan in some cases. As shown in Table 1, there is an average overhead reduction of 18% if Baggy Bounds is used in all cases where it is more efficient than ASan. We also note that the Baggy's stack metadata operations can be more cost efficient than ASan's for some programs, such as for 511.povray_r.

6.2 Protecting Usable Targets

Not all unsafe operations have the same security impact. Thus, we want to evaluate whether using multiple defenses can help maximize the security impact within a cost budget. In this paper, we have proposed that the security impact should be measured in terms of usable targets, as explained in Section 4.2.

To better understand the benefit of OPTISAN, we analyze the placements produced by OPTISAN and ASAP for the *Apache* program with a 7% budget in Figure 5b). On the x-axis, we show the check costs (i.e., the operation frequency multiplied by the cost per check) of each non-free unsafe operation for simplicity. However, metadata cost is accounted for when computing the placement. On the y-axis, Figure 5b) shows the effective usable targets protected by covering each unsafe operation. We find that OPTISAN covers at least 7 unsafe operations (i.e., shown as green X's), not covered by ASAP. We find 6 of them to be very important, as they protect more than 10 usable targets each. In addition, we find ASAP covers multiple unsafe operations that have low check costs and protects close to zero usable targets, as can be seen by the orange star in the bottom-left corner. Thus, the OPTISAN formulation biases the choice of unsafe operations to those with more usable targets to increase the stack memory protection.

Program	*CVE (severity)*	*Freq Rank*	*Target Rank*	*Place Rank*	*Total Unsafe*
libtiff	2022-1355 (6.1)	2	4	2	20
openssl	2016-2176 (8.2)	33	5	29	98
	2022-3602 (7.5)	2	2	2	98
	2022-3786 (7.5)	1	1	1	98
libxml2	2017-9047 (7.5)	155	39	149	2,614
	2017-9048 (7.5)	116	39	110	2,614
httpd	2019-10097 (7.2)	1,448	2,869	1,406	2,969
	2020-35452 (7.3)	1,900	2,598	1,831	2,969
readelf	2021-20294 (7.8)	348	356	322	361

The ranks shown are from lowest to highest in the set of all unsafe operations.

Table 5: CVE Analysis - Frequency (Freq) rank and target rank with associated placement (Place) rank for OPTISAN. ASAP places at the frequency rank.

We also measure the number of usable targets protected from all unsafe operations that may access the target for the defense placements produced. Figure 5c) shows the fraction of usable targets protected from all unsafe operations (i.e., using Baggy or ASan) in the placement computed (y-axis) for the three budgets amounting to 25%, 50% and 75% of the budgets necessary to fully protect all unsafe memory operations[8] (x-axis). As Figure 5c) shows, the fraction of usable targets protected can vary significantly at lower budgets (25%), but are approaching full coverage at 75% budget. However, we find that on average only a small fraction of the usable targets (i.e., about 20%) are fully protected for the *povray* program, even at a 75% budget. After investigation, we find that 3% of the unsafe stack operations in povray may access over 70% of the total usable targets. Hence, if only one of these unsafe operations remains unprotected at least (70%) of the usable targets would not be fully protected. Furthermore, we find that 6.56% of these high-impact, unsafe operations account for 99% of the total cost to cover all of them. These high-cost, high-impact unsafe operations remain unprotected even at the 75% budget evaluated. OPTISAN can help programmers identify these high-cost, high-impact operations, providing motivation for code changes to prevent spatial errors.

6.3 Protecting Unsafe Operations for CVEs

To consider the impact of OPTISAN in preventing the exploitation real CVEs, we examine nine CVEs, consisting of six known stack vulnerabilities from an open-source fuzzing dataset [28] and three stack vulnerabilities in other open-source programs - two in Apache and one in the binutils program, readelf. As shown in Table 5, we evaluate whether considering performance (Freq Rank for frequency) and usable targets (Target Rank) enables OPTISAN to prioritize the

[8]The specific budgets at these levels are shown in Table 6.

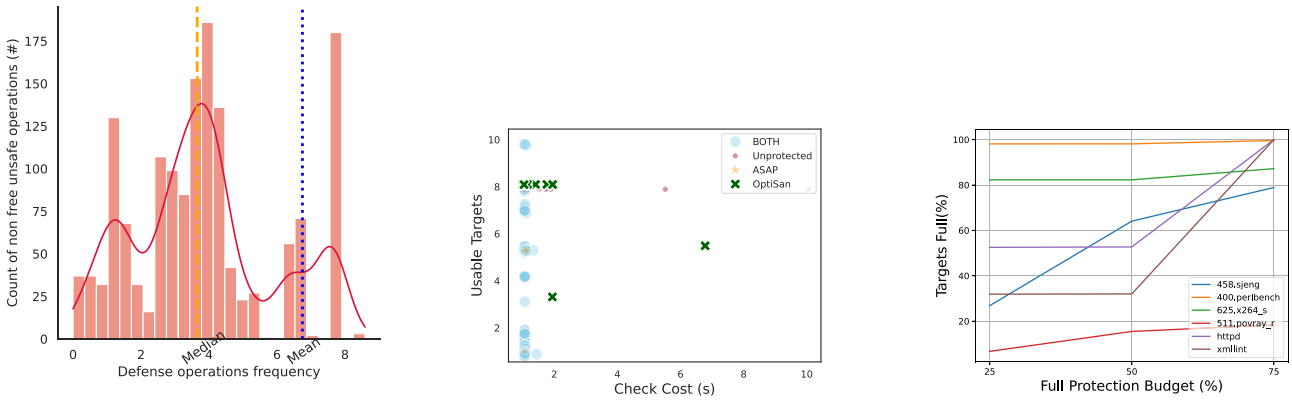

Figure 5: a) Frequency distribution for unsafe operations in Apache b) Distribution of the number of effective usable targets vs. the cost of defending for unsafe operations protected by ASAP, OPTISAN, and both for Apache using ASan. c) Fraction of usable targets protected fully vs. the fraction of the budget to protect all operations.

Program	***Baseline (s)***	***Full Protection Budget (s)***	***25 (%)***			***50 (%)***			***75 (%)***			***Mem (%)***
			Exp (%)	Obs (%)	It	Exp (%)	Obs (%)	It	Exp (%)	Obs (%)	It	
sjeng	400.76	133.71	8.34	6.85	2	16.68	15.32	2	25.02	20.34	3	1.00
perlbench*	243.97	26.52	2.26	1.85	1	3.81	1.87	1	10.27	10.98	2	0.25
httpd	104.44	14.83	3.54	2.63	2	7.09	6.07	1	10.65	7.54	3	0.30
libxml2	74.10	2.53	0.85	0.39	2	1.71	1.78	1	2.70	2.79	1	0.44
625.x264_s	240.65	2.44	0.25	0.40	2	0.50	0.40	1	0.75	0.80	2	0.51
511.povray_r	452.55	383.15	21.17	17.94	1	42.33	42.22	2	63.50	59.61	1	0.13
511.povray_r	452.55	383.15	4.23	2.75	2	8.47	5.01	5	12.70	12.73	1	0.13

Table 6: Performance metrics for programs instrumented with OPTISAN placements and iterations to achieve desired budgets. The columns contain the expected overhead *Exp* , the final observed runtime overhead *Obs* and the number of iterations performed *It* for each budget.
* - For *perlbench*, the budgets at 25, 50, and 75% did not generate distinct placements (see Figure 4), so we chose lower budgets (2.26,3.81 and 10.27 %).

placement of defenses for unsafe operations (Place Rank), before a cost-based heuristic ASAP [70], which only considers performance (Freq Rank).

We find that OPTISAN places defenses for these CVEs prior to ASAP for six of the nine unsafe operations and at the same rank for the other three. The other three cases have low frequencies, so they are prioritized for placement early using the frequency rank alone. While this is not a statistically significant sample and OPTISAN does not guarantees placement prior to ASAP for all CVEs, these results illustrate how at certain budgets ASAP may choose not to protect these vulnerabilities, as it decides solely based on the frequency rank. By considering the usable targets (Target Rank) as well as performance (Freq Rank), OPTISAN protects these vulnerabilities earlier (Place Rank) in these cases.

6.4 Instrumentation Overheads

The practicality of OPTISAN relies on it producing defense placements close to the desired budget. To measure this, we instrument the two SPEC CPU 2006 programs, two SPEC CPU 2017 programs, and two real-world programs as shown in Table 6. We specify three budgets for each program - 25, 50, and 75% of the estimated cost to cover all unsafe operations (*Full Protection Budget*) using *ASan* [59], or Baggy Bounds in cases where it is cheaper (*libxml2 and 625.x264_s*). Since *povray* has a high overhead, we also show that we can generate placements on tighter budgets for programs that are expensive to defend. We also measure the average memory overhead, by measuring the percentage increase in *max RSS (kbytes)*.

On average, the final observed run-time overhead deviates from the specified overhead by 1.47% of the program execution time, or 3.40 seconds, as seen in Table 6. There are 9 cases where we apply OPTISAN iteratively. We change the budget each iteration based on the difference between the expected and observed run-time overhead for each budget evaluated for a program. We find that within two iterations, on average, OPTISAN finds solutions with an average run-time overhead close to the desired budget for each of these programs. There are 5 cases that are slightly over the specified budget on average. Three of the cases are within budget in 3 out of 5 executions and within one standard deviation. The other two cases may be caused by: (1) folding initialization costs in metadata costs which may impact placement at low overheads (*625.x264_s* at 25%) and (2) the step-function effect of perlbench placements (see Figure 4). On average the memory overhead of the defense placement is 0.44%.

7 Discussion

In this section we discuss some current limitations of our approach and possible extensions.

One issue is that OPTISAN considers spatial memory errors with respect to stack memory only. Our approach can be extended to consider global memory, as one can identify *usable targets*, similar to the stack. To extend OPTISAN to the heap requires a reasonable technique to compute usable targets. This is challenging because heap objects may be reallocated to different locations, commonly have many more aliases, and have much larger lifetimes than stack objects.

OPTISAN's use of value range analysis to prune safe operations along with the application of ASan and Baggy Bounds defenses still leaves some operations unprotected at desired budgets, as shown in Figure 4. Ideally, we could integrate other defenses and proposed optimizations [25, 39, 74] to enable the coverage of all unsafe operations using cheaper defenses. In addition, we could use more comprehensive analyses to identify operations that must satisfy spatial safety to reduce checks and isolate usable targets from attack. One possible approach is to apply the recent DataGuard work [30] to identify safe objects using a combination of static analysis and symbolic execution, and isolate the safe stack objects using Safe Stack [16]. We leave this to future work.

OPTISAN currently supports placement using multiple metadata schemes, but this may lead to redundant metadata. However, a single unified and compact metadata scheme could be applied [20] to improve performance [33].

OPTISAN's *Mixed Integer Non-Linear Programming (MINLP)* formulation can be used to include additional defenses such as CFI [44], DFI [13]. They could be modelled as defense options for a group of unsafe operations such as enforcing CFI for a function. We leave this to future work.

8 Related Work

Sanitizer Placement ASAP [69] proposed a greedy approach to sanitize programs. For a given cost level, the most expensive checks are removed until a user-provided cost budget is met. The idea being that a few hot checks account for most of the overhead and that most bugs are along cold paths. Subsequent work [38] extended this idea to runtime partitioning wherein bugs along hot paths are detected probabilistically based on when the sanitized variants of functions are invoked.

Similarly, SanRazor [73] proposes identifying likely redundant checks to improve performance. This approach aims to leverage the execution profile and data flow facts of the checks to eliminate likely redundant checks. However, these approaches aim to improve performance based on heuristics that do not consider the security impact of the elided checks.

Recent work [74] developed multiple optimizations to improve Address Sanitizer [59]'s performance. The first optimization improves the safety analysis employed by identifying safe accesses related to stack and global objects. They remove redundant checks that are dominated by or post-dominated by the other accesses that refer to the same object (including aliases). They merge neighboring checks (memory accesses that access neighboring memory locations). They also optimize checks in loops i.e., loop invariant addresses and memory accesses which monotonically increase or decrease with the loop (change by a constant value).

Optimizing Defense Placements Researchers have long argued for applying systematic techniques based on optimization to make decisions rather than ad hoc heuristics [35]. However, few works consider the problem of optimizing defense placements for security. ProgramMandering [41] (PM) provides a method to generate privilege-separated domains that enables the programmer to optimize one metric while constraining others. The PM system defines four metrics covering security and performance properties that may be impacted by the choice of protection domain boundaries for a privilege-separated domain. PM uses a binary Integer Programming model for the partitioning problem given the metrics, proposing a semi-automated, iterative approach for programmers to determine the code to include in a domain. OPTISAN applies a Mixed-Integer Non-Linear Programming (MINLP) approach to solving a different problem.

Automatic Patch Generation Several approaches [22, 31, 36, 42, 43, 71] have been proposed to automate patch generation for vulnerabilities in programs. These include approaches which generate patches based on fixed patterns [36], such as adding a null check or a memory de-allocation statement to fix memory leaks [22]. Such approaches are not robust as they depend on these fixed patterns. Additional, approaches based on program mutation [71], symbolic execution [43] and machine learning [42] have been proposed.

Recent work [31] proposed using safety properties to automate program repair. The safety properties are manually specified and program-independent. This approach relies on concolic execution [12] to identify the safety property violated. These approaches are limited by their reliance on the knowledge of the vulnerability (i.e., *proof-of-vulnerability*).

9 Conclusions

We present OPTISAN, which is the first system to produce spatial memory defense placements that optimize security and performance within a cost budget. OPTISAN estimates the expected costs of spatial memory defenses, accounting for check and metadata operations and identifies the usable targets that may be exploited. OPTISAN provides an optimization formulation to compute a placement that maximizes the defense of such targets within a prescribed cost budget. Our evaluation shows that OPTISAN can produce placements that protect 18.4% more operations with usable targets using a combination of Baggy Bounds and ASan than Baggy Bounds alone at the same cost budget across fourteen programs. OPTISAN is shown to predict overheads accurately by modeling both metadata costs and bounds checks, making it a practical tool to place defenses.

10 Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. We would also like to thank our shepherd for guiding us through the revision. This research was sponsored by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2- 0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory of the U.S. government. The U.S. government is authorized to reproduce and distribute reprints for government purposes notwithstanding any copyright notation here on.

References

[1] Gurobi. http://gurobi.com.

[2] Intel CET. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[3] Neo4j. https://neo4j.com/.

[4] Linux 2.6.7. NX (no execute) support for x86. https://lkml.org/lkml/2004/6/2/228, 2004.

[5] MITRE CWE. https://cwe.mitre.org/, 2019.

[6] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy Bounds Checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[7] J. P. Anderson. Computer Security Technology Planning Study, Volume II. Technical Report ESD-TR-73-51, Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), October 1972.

[8] Apache. Apache HTTPD Test Framework. https://github.com/apache/httpd-tests.

[9] AttackerKB. CVE-2021-20038. https://attackerkb.com/topics/QyXRC1wbvC/cve-2021-20038/rapid7-analysis, 2019.

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

[11] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining Light on Shadow Stacks. *2019 IEEE Symposium on Security and Privacy (S&P)*, pages 985–999, 2019.

[12] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 2008 Conference on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

[13] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[14] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. KOOBE: towards facilitating exploit generation of kernel Out-Of-Bounds write vulnerabilities. In *Proceedings of the 29th USENIX Security Symposium*, pages 1093–1110, 2020.

[15] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A Compile-time Solution to Buffer Overflow Attacks. In *Proceedings 21st International Conference on Distributed Computing Systems (ICDCS)*, 2001.

[16] Clang Documentation - SafeStack. Clang document at https://clang.llvm.org/docs/SafeStack.html, 2020.

[17] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium (USENIX Security)*, 1998.

[18] Dinakar Dhurjati and Vikram Adve. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 28th International Conference on Software Engineering*, 2006.

[19] Gregory J. Duck and Roland H.C. Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 132–142, 2016.

[20] Gregory J. Duck and Roland H.C. Yap. Effectivesan: type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–195, 2018.

[21] Gregory J. Duck, Roland H.C. Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In *Proceedings of the 2017 Network and Distributed Systems Security Symposium*, pages 1–15, 2017.

[22] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe memory-leak fixing for C programs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 459–470. IEEE, 2015.

[23] E. Goktas, B. Kollenda, P. Koppe, E. Bosman, G. Portokalidis, T. Holz, H. Bos, and C. Giuffrida. Position-independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure. In *Proceedings of 3rd IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018.

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

[25] Floris Gorter, Enrico Barberis, Raphael Isemann, Erik Van Der Kouwe, Cristiano Giuffrida, and Herbert Bos. {FloatZone}: Accelerating memory error detection using the floating point unit. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 805–822, 2023.

[26] Niranjan Hasabnis, Ashish Misra, and R Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 135–144, 2012.

[27] Reed Hastings. Purify: Fast detection of memory leaks and access errors. In *Proc. 1992 Winter USENIX Conference*, pages 125–136, 1992.

[28] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), December 2020.

[29] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[30] Kaiming Huang, Yongzhe Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. The taming of the stack: Isolating stack data from memory errors. In *Proceedings of the 2022 Network and Distributed System Security Symposium (NDSS)*, 2022.

[31] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. Using safety properties to generate vulnerability patches. In *Proceeding of the 2019 IEEE Symposium on Security and Privacy (SP)*, pages 539–554. IEEE, 2019.

[32] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[33] Yuseok Jeon, WookHyun Han, Nathan Burow, and Mathias Payer. FuZZan: Efficient sanitizer metadata design for fuzzing. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC)*, pages 249–263, 2020.

[34] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. Exploring and enforcing security guarantees via program dependence graphs. In *Proceedings of the 2015 Conference on Programming Language Design and Implementation*, pages 291–302, June 2015.

[35] Kimberly Keeton, Terence Kelly, Arif Merchant, Cipriano Santos, Janet Wiener, Xiaoyun Zhu, and Dirk Breyer. Don't settle for less than the best: Use optimizationg to make decisions. In *11th Workshop on Hot Topics in Operating Systems*, 2007.

[36] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 802–811. IEEE, 2013.

[37] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, pages 75–86. IEEE, 2004.

[38] Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz. Partisan: fast and flexible sanitization via run-time partitioning. In *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings 21*, pages 403–422. Springer, 2018.

[39] Hao Ling, Heqing Huang, Chengpeng Wang, Yuandao Cai, and Charles Zhang. Giantsan: Efficient memory sanitization with segment folding. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 433–449, 2024.

[40] Shen Liu, Gang Tan, and Trent Jaeger. PtrSplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 2017.

[41] Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. Program-mandering: Quantitative privilege separation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1023–1040, 2019.

[42] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312, 2016.

[43] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 691–701, 2016.

[44] Control Flow Guard. https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx.

[45] Microsoft. BWorld Robot Control Software. https://codeantenna.com/a/dp00Qd37wy, 2000.

[46] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[47] MITRE. Nginx CVE-2020-14147. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-14147, 2020.

[48] MITRE. Nginx CVE-2020-25624. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-25624, 2020.

[49] MITRE. Nginx CVE-2021-3444. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3444, 2021.

[50] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[51] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[52] NIST. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0226.

[53] NVD. HTTPD CVE. https://nvd.nist.gov/vuln/detail/CVE-2019-10097, 2019.

[54] A. One. Smashing the Stack for Fun and Profit. *Phrack*, 7(49), 1997. Available at http://www.phrack.org/issues.html?id=14&issue=49.

[55] PAX. Address Space Layout Randomization. https://pax.grsecurity.net/docs/aslr.txt., 1993.

[56] Bruce Perens. Electric fence malloc debugger. https://elinux.org/Electric_Fence, 1993.

[57] Roman Rogowski, Micah Morton, Forrest Li, Fabian Monrose, Kevin Z. Snow, and Michalis Polychronakis. Revisiting Browser Security in the Modern Era: New Data-Only Attacks and Defenses. In *Proceedings of the 2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[58] Donn Seeley. A Tour of the Worm. https://www.cs.unc.edu/~jeffay/courses/nidsS05/attacks/seely-RTMworm-89.html.

[59] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, pages 309–318, 2012.

[60] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *ATC '12*, 2012.

[61] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[62] Axel Simon. *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[63] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.

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

[65] Sqlite. How SQLite Is Tested. https://www.sqlite.org/testing.html.

[66] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: fast detector of uninitialized memory use in c++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55. IEEE, 2015.

[67] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.

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

[69] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *2015 IEEE Symposium on Security and Privacy*, pages 866–879. IEEE, 2015.

[70] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *SP '15*, 2015.

[71] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*, pages 364–374. IEEE, 2009.

[72] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. AriCheck: An Efficient Pointer Arithmetic Checker for C Programs. *Proceedings of the 5th International Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2010.

[73] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. SanRazor: Reducing redundant sanitizer checks in C/C++ programs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 479–494, 2021.

[74] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. Debloating address sanitizer. In *Proceedings of the 2022 Usenix Security Symposium*, 2022.