SIMBER: Eliminating Redundant Memory Bound Checks via Statistical Inference

Hongfa Xue, Yurong Chen, Fan Yao, Yongbo Li, Tian Lan, and Guru Venkataramani(✉)

The George Washington University, Washington, DC, USA
guruv@gwu.edu

Abstract. Unsafe memory accesses in programs written using popular programming languages like C and C++ have been among the leading causes of software vulnerability. Memory safety checkers, such as Softbound, enforce memory spatial safety by checking if accesses to array elements are within the corresponding array bounds. However, such checks often result in high execution time overhead due to the cost of executing the instructions associated with the bound checks. To mitigate this problem, techniques to eliminate redundant bound checks are needed. In this paper, we propose a novel framework, SIMBER, to eliminate redundant memory bound checks via statistical inference. In contrast to the existing techniques that primarily rely on static code analysis, our solution leverages a simple, model-based inference to identify redundant bound checks based on runtime statistics from past program executions. We construct a knowledge base containing sufficient conditions using variables inside functions, which are then applied adaptively to avoid future redundant checks at a function-level granularity. Our experimental results on real-world applications show that SIMBER achieves zero false positives. Also, our approach reduces the performance overhead by up to 86.94% over Softbound, and incurs a modest 1.7% code size increase on average to circumvent the redundant bound checks inserted by Softbound.

1 Introduction

Many software bugs and vulnerabilities in applications (that are especially written using C/C++) occur due to unsafe pointer usage and out-of-bound array accesses. Security exploits, that take advantage of buffer overflows or illegal memory reads/writes, have been a major concern over the past decade. Some of the recent examples include: (i) In February 2016, a Google engineer discovered a stack overflow bug in the glibc DNS client side resolver inside getaddrinfo() function that had the potential to be exploited through attacker-controller domain names, attacker-controlled DNS servers or man-in-the-middle attack [10]; (ii) In 2016, Cisco released security patches to fix a buffer overflow vulnerability in the Internet Key Exchange (IKE) version 1 (v1) and IKE version 2 (v2) code

S. De Capitani di Vimercati and F. Martinelli (Eds.): SEC 2017, IFIP AICT 502, pp. 413–426, 2017.
DOI: 10.1007/978-3-319-58469-0_28

of Cisco ASA Software that could allow an attacker to cause a reload of the affected system or to remotely execute code [5].

In order to protect software from spatial memory/array bound violations, tools such as Softbound [12] have been developed that maintains metadata such as array boundaries along with rules for metadata propagation when loading or storing pointer values. By doing so, Softbound makes sure that pointer accesses do not violate boundaries through runtime checks. While such a tool offers protection from spatial safety violations in programs, we should also note that they often incur high performance overheads due to the following reasons. (a) Array bound checking incurs extra instructions in the form of memory loads and stores for pointer metadata and the propagation of metadata between pointers during assignments. (b) In pointer-intensive programs, such additional memory accesses can introduce memory bandwidth bottleneck, and further degrade system performance.

To mitigate runtime overheads, static techniques to remove redundant checks have been proposed, e.g., ABCD [3] builds and solves systems of linear inequalities among bound and index variables, and WPBound [14] statically computes the potential range of target pointer values inside loops to avoid Softbound-related checks. As the relationship among pointer-affecting variables (i.e., variables, whose values can influence pointers) and array bounds become more complex, static analysis is less effective and usually cannot remove a high percentage of redundant array bound checks.

In this paper, we propose SIMBER, a novel approach that verifies conditions for eliminating bound checks on the fly by harnessing runtime information instead of having to rely on discovering redundant checks solely during compile-time or using static code analysis. SIMBER is effective in removing a vast majority of redundant array checks while being simple and elegant. The key idea is to infer the *safety of a pointer dereference based on statistics from prior program executions*. If prior executions show that the access of array A with length L at index i is within bound, then it is safe to remove the checks on any future access of A with length no smaller than L and an index no larger than i.

In summary, this paper makes the following contributions:

1. Instead of solely relying on static code analysis, SIMBER utilizes runtime statistics to check whether array bound checks can be eliminated. Our experimental results show that SIMBER can discover a high number of redundant bound checks through analyzing the variables that can affect the pointer values.
2. We determine a bound check as redundant only if previous executions deem the checks to be unnecessary and current execution satisfy the condition derived from such prior execution history. This helps SIMBER to guarantee zero false positives.
3. We evaluate using applications from SPEC2006 benchmark suite [1] that have the highest performance overheads in Softbound: bzip2, lbm, sphinx3 and hmmer. In these experiments, we observe that our approach reduces the

performance overheads of spatial safety checks by over 86.94% compared to Softbound.

2 Background

Softbound stores the pointer metadata (array base and bound) when pointers are initialized, and performs array bound checks (or validation) when pointers are dereferenced. For example, for an integer pointer *ptr* to an integer array *intArray*[100], Softbound stores *ptr_base* = & *intArray*[0] and *ptr_bound* = *ptr_base* + *size(intArray)*. When dereferencing pointers, Softbound obtains the base and bound information associated with the target pointer *ptr*, and does the following check: if the value of *ptr* is less than *ptr_base*, or, if *ptr+size* is larger than *ptr_bound*, the program terminates. A disadvantage with this approach is the high runtime performance overheads associated with metadata tracking and bound checks especially on pointers that are largely benign or safe. Figure 1 shows the runtime overhead incurred by Softbound-instrumented applications over un-instrumented application as baseline in SPEC2006 benchmarks [1].

We note that some prior works [3,14] have proposed static analysis techniques to eliminate redundant bound checks. In SIMBER, we propose a novel framework where the redundant bound check elimination is performed with the guidance of runtime statistics. Our results show that even limited amounts of runtime statistics can be a quite powerful tool to infer the safety of pointer dereferences, and eliminate unnecessary pointer bound checks.

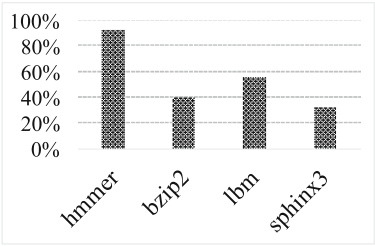

Fig. 1. Runtime performance overhead incurred by Softbound

Consider the example shown in Fig. 2, where $foo(dest, src, n)$ copies the first n characters in string *src* to *dest*, and replaces remaining characters with blocks of 4-character pattern '0000'. To guarantee safe pointer usage, Softbound checks (denoted by *CHECK_SB*) will be added before each pointer dereference, e.g., in lines 8, 9, and 20. Thus, bound checks are performed for each iteration of the *for* and *while* loops, resulting in high execution time (performance) overhead.

A static approach such as ABCD [3] relies on building constraint systems for target pointers and programs to remove redundant bound checks. In particular, it identifies that indices i and j in $foo()$ must satisfy $i \leq j$ from the conditions in line 18. Therefore, bound checks on $*(dest+i)$ in line 8 is deemed redundant given the checks performed on $*(dest + j)$ in line 20. However, such static approaches cannot be effective in eliminating other bound checks where such static inferences cannot be made (e.g., in lines 9 and 20). Further, bound information for both pointers *dest* and *src* needs to be kept and propagated inside $foo()$ at runtime.

In this paper, we show that (conditionally) removing all the bound checks in $foo()$ is indeed possible using SIMBER. Our solution stems from two key

```
1  foo_SB
2  (char *dest,char *src,int n)
3  {
4   int i, j;
5
6   for (i=0; i<n; i++)
7   {
8    CHECK_SB(dest+i);
9    CHECK_SB(src+i);
10   *(dest+i) = *(src+i);
11  }
12
13
14  int len=strlen(src);
15
16  while (i<len-4)
17  {
```

```
18    for (j=i; j<i+4; j++)
19    {
20     CHECK_SB(dest+j);
21     *(dest+j) = '0';
22    }
23    i+=4;
24   }
25  }
26
27  main()
28  {
29     char *dest, *src;
30     int n;
31     ...
32     foo_SB (dest, src, n);
33     ...
34  }
```

Fig. 2. Example code illustrating bound checks performed by SoftBound

observations. First, redundant bound checks can be effectively identified by examining different runs of $foo()$. Consider pointer dereference $*(src + i)$ in line 10 as an example. Let $i_{(k)}$ and $src\_bound_{(k)}$ denote the value of index i and the bound of array src in the kth run, which is already determined to be bound-safe, i.e., $i_{(k)} \leq src\_bound_{(k)}$. It is easy to see that any future runs of $foo()$ satisfying $i \leq i_{(k)}$ and $src\_bound \geq src\_bound_{(k)}$ will also be bound-safe, due to the following chain of inequalities $i \leq i_{(k)} \leq src\_bound_{(k)} \leq src\_bound$, implying $i \leq src\_bound$. Second, through a simple dependency analysis, we find that the value of index i is only positively affected by input variable n. Due to this positive dependency, the redundant-check condition $i \leq i_{(k)}$ is guaranteed if we have $n \leq n_{(k)}$. Thus, bound checks for $*(src+i)$ in line 9 can be determined as redundant by comparing input variables n and src_bound with that of previous runs, which entirely removes all checks and bound propagation in $foo()$ at function-level.

3 Overview of System Design

SIMBER consists of five modules: Dependency Graph, Statistical-guided Inference, Knowledge Base, Redundant checks removal and Check-HotSpot Identification. Figure 3 presents our system diagram. Given a target pointer, SIMBER aims to determine if the pointer dereference needs to be checked. First, SIMBER collects values of pointer-affecting variables which can affect the target pointer. It constructs multi-dimensional safe regions where the values of such pointer-affecting variables do not result in bad program behavior (e.g., program crash, buffer overflow). In the current program execution, if the data point representing pointer-affecting variables is inside the safe region, then this pointer dereference is determined to be safe.

3.1 Dependency Graph Construction

Dependency Graph (DG) is a bi-directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ that represents program variables as vertices in $\mathcal{V}$, and models the dependency between the variables

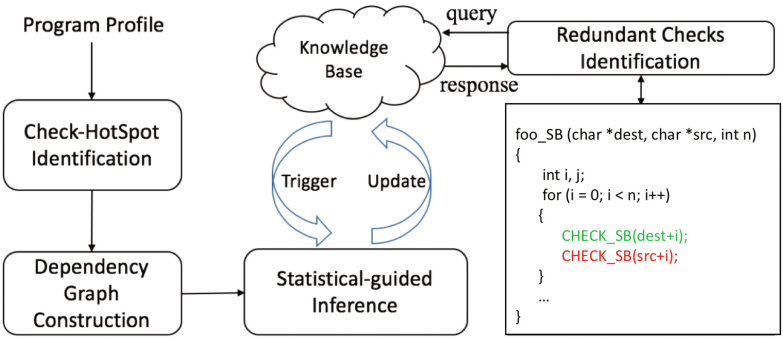

Fig. 3. SIMBER overview and key modules

and array indices/bounds through edges in $\mathcal{E}$. We construct a DG for each function by including all if its pointers and the pointer-affecting variables that could affect the value of pointer. We add trip count (number of times a branch is taken) as auxiliary variables to assist the analysis of loops.

Definition 1 (DG-Node). *The nodes in dependency graphs are the variables that can affect the pointers such as (a) the variables that determine the base of pointers through pointer initialization, assignment or casting; (b) variables that affect the offset and bound of pointers like array index, pointer increment and variables affecting memory allocation size; (c) Trip Count (TC): the number of times a branch (in which a target pointer value changes) is taken.*

Definition 2 (DG-Edge). *DG-Node v_1 will have an out-edge to DG-Node v_2 if v_1 can affect v_2.*

Abstract Syntax Tree (AST) is commonly used by compilers to represent the structure of program code, and to analyze the dependencies between variables and statements. We use Joern tool [18] to generate AST for each function.

Algorithm 1. Dependency graph construction for a given function $foo()$

```
Input: source code of function foo()
Construct AST of function foo()
Initialize V = φ, E = φ
for each variable v in AST do
    V = V + {v}
for each statement s in AST do
    for each pair of variables j, k in s do
        add edge e(j, k) to E according to Remark 1
Output: Dependency-Graph G = (V, E)
```

Algorithm 1 shows the pseudocode of Dependency Graph (DG) construction for a given function, $foo()$. First, we obtain all of the pointers and their corresponding pointer-affecting variables, and represent them as DG-Nodes. Second,

we traverse dependency graph and identify adjacent DG-Nodes that represent the pointer-affecting variables associated with each target pointer. Each target pointer will have an entry in the form of $(func : ptr, var_1, var_2, ..., var_n)$ where $func$ and ptr are the names of the function and target pointer respectively, with var_i being the name of pointer-affecting variables associated with ptr. Through logging the values of these variables during program executions, we build conditions to determine safe regions that help eliminate redundant bound checks.

Remark 1. ***Edges added into Dependency Graph:***

E1 Assignment statements	A := B	$B \rightarrow A$
E2 Function parameters	Func(A,B)	$B \leftrightarrow A$
E3 Loops	for.../while...	*Add TC to Loops*
(1) Assignment inside Loops	A := B	$TC \rightarrow A$
E4 Array Indexing	A[i]	$i \rightarrow A$

3.2 Statistical-Guided Inference

This module builds safe regions based on the pointer-affecting variables identified by DGs, and updates the safe region through statistical inference from previous execution. Once the pointer-affecting variables for the target pointer are determined, SIMBER collects the values of pointer-affecting variables from runtime profile, and produces a **data point** (or a vector) in Euclidean space with the coordinates of data point being the actual values of pointer-affecting variables. The dimension of the Euclidean space is the number of pointer-affecting variables for the target pointer.

The inference about pointer safety (for pointer-affecting variables that are positively-correlated with the array bound) can be derived as follows: Let us say that a data point $p = (vp_1, vp_2, \ldots, vp_d)$ from prior execution is checked and deemed as safe. Consider another data point $q = (vq_1, vq_2, \ldots, vq_d)$ for the same target pointer from current execution. If each element of q is no larger than that of p, i.e., $vq_1 \leq vp_1$, $vq_2 \leq vp_2$, ..., $vq_d \leq vp_d$, then the bound checks on the target pointer can be removed in the current execution. To extend this inference to pointer-affecting variables that are negatively-correlated with array bound, we unify the representation by converting the variable *bound* into $C - bound$ for sufficiently large constant C (such as the maximum value of an unsigned 32-bit integer). Thus $C - bound$ is also positively correlated, and $C - bound_q \leq C - bound_p$ implies $bound_q \geq bound_p$.

Definition 3 (False Positive). *A false positive occurs if a bound check, that is identified as redundant, is indeed necessary and should have not been removed.*

Definition 4 (Safe Region (SR)). *Safe region is an area that is inferred and built from given data points, such that for any point within the region, the corresponding target pointer is guaranteed to have only safe memory access, e.g., all*

bound checks related to the pointer can be removed with zero false positive, under the assumption that point-affecting variables have monotonic linear relationships with pointer bound.

Thus, the Safe Region derived from a single data point p is the enclosed area by projecting it to each axis. In other words, it includes all vectors that have smaller (pointer-affecting variable) values and are dominated by p. For example, the safe region of a point $(3, 2)$ is all of the points in the euclidian space with the first coordinate smaller than 3 and the second coordinate smaller than 2 in $\mathbb{R}^2$, namely $q = (q_1, q_2) : q_1 \leq 3, q_2 \leq 2$. We can obtain the Safe Region of multiple data points by taking the union of the safe regions generated by each data point.

Given a set $\mathcal{S}$ which consists of N data points in $\mathbb{R}^D$, where D is the dimension of data points, we first project point $p_i, i = 1, 2, \ldots, N$, to each axis and build N-surface enclosed area in $\mathbb{R}^D$, e.g., building a safe region for each data point. The union of these N safe regions is the safe region of $\mathcal{S}$, denoted by $SR(\mathcal{S})$. *Thus, if a new data point q falls inside $SR(\mathcal{S})$, we can find at least one existing point $p \in \mathcal{S}$ that dominates q, i.e., $q \leq p$. That is to say, the enclosed projection area of p covers that of q, which means for every pointer-affecting variable in p is larger than that of q.*

There are data points that cannot be determined as safe based on existing (current) safe region when $q \notin SR(\mathcal{S})$. In this case, SIMBER performs bound checks to determine memory safety of such data points and adaptively updates the safe region based on the outcome. More precisely, given current safe region $SR(\mathcal{S})$ and the new coming data point $q \notin SR(\mathcal{S})$, $SR(\mathcal{S})$ will expand to $SR(\mathcal{S})'$ by:

$$SR' = SR(\mathcal{S} \cup q) = SR(\mathcal{S}) \cup SR(q) = SR(\mathcal{S}) \cup \{x : x \leq q\}, \tag{1}$$

where $\{x : x \leq q\}$ is the set of safe points dominated by vector q. It expands the safe region if (i) there are pointer-affecting variables in the new input q that have a larger value than all points in current safe region $SR(S)$, or (ii) there are array lengths or negatively-correlated variables that have smaller values than all points in $SR(S)$, allowing higher degree of redundant bound check elimination in future executions.

3.3 Knowledge Base

SIMBER stores the safe regions for target pointers in a disjoint memory space - Knowledge Base. The data in Knowledge Base, in the format of (*key, value*), represents the position and the sufficient conditions for removing the redundant bound checks for each target pointer. Statistical Inference is *triggered* to compute the Safe Region whenever the Knowledge Base is updated with newer data points and new execution logs.

We use SQLite [2] to store our Knowledge Base. We create a table to store conditions derived from pointer values and the corresponding pointer-affecting variables.

3.4 Redundant Checks Identification

SIMBER instruments functions within the program with a call to *SIMBER*(), that collects *pointer-affecting input parameters* inside a target function, and queries the knowledge base to obtain the conditions for eliminating array bound checks. In particular, if the all of the data points (formed using function parameters) are within the safe region, the propagation of bound information and the array bound checks can be safely removed from this target function entirely.

We maintain two versions of Check-Hotspot functions: the original version (which contains no bound checks) and the Softbound-instrumented version (that has bound checks and bound meta-data propagation). Based on the result of *SIMBER*() outcome, we can either skip all bound checks inside the function (if the condition holds) or proceed to call the Softbound-instrumented function (if the condition is not satisfied) where bound checks would be performed as shown in Fig. 2. The instrumentation of SIMBER() condition verification inside functions leads to a small increase in code size (by about 1.7%), and we note that such extra code is added only to a small subset of functions with highest runtime overhead for Softbound (see Sect. 3.5 for details).

3.5 Check-HotSpot Identification

To minimize the effect of runtime bound checks, we choose Check-Hotspots functions that have high levels of pointer activity. We identify Check-HotSpots as follows: (a) We use Perf profiling tool [6] to profile two versions of programs: non-instrumented version and softbound-instrumented source code. (b) We compute the difference in absolute execution time spent on different functions between non-instrumented source programs and softbound-instrumented programs to capture the extra time spent on softbound-related code. For every function, we calculate the function-level overhead as the ratio of the time spent on softbound-related code to the total execution time spent in the original version. (c) We list all of the functions with function-level overhead of at least 5% as the Check-HotSpots.

```
1  //original foo() function
2  foo
3  (char *dest,char *src,int n)
4  {...}
5
6  //softbound instrumented foo()
7  foo_SB
8  (char *dest,char *src,int n)
9  {...}
10 main()
11 {
12  char *dest, *src;
13  int n;
```

```
14   ...
15  /*determine whether it is
16   inside the safe region*/
17  if(SIMBER(dest,src,n))
18  {
19   foo(dest, src, n);
20  }
21  else
22  {
23   foo_SB (dest, src, n)
24  }
25  ...
26 }
```

Fig. 4. SIMBER optimized code that determines if bound checks can be removed at function-level granularity

3.6 SIMBER-Optimized Softbound Code

SIMBER instruments the program by adding two branches as shown in Fig. 4. Function *SIMBER()* verifies whether the input variables of $foo()$ satisfy the condition to eliminate array bound check, and chooses one of the two possible branches accordingly. Recall the Softbound-instrumented *foo*() function in Fig. 2. The dependency graph contains edges from n to i (due to the *for* loop in line 6), from *src_len* to *len* (due to the assignment in line 14), from *len* to i (due to the *while* loop in line 16), from i to j (due to the second *for* loop in line 18), and from i, j to pointers *src* and *dest* (due to pointer dereference in lines 10 and 21).

We focus on bound checks for $*(dest + j)$ in line 20 to illustrate SIMBER. From the dependency graph, we find bound-affecting variables *len* and n, and form a 3-dimensional vector $(len, n, C - dest\_bound)$ (for large enough, constant C) to represent the safe region corresponding to bound checks for $*(dest + j)$. Assume that C is 1024 and that three previous data points are available: $P_1 = (200, 160, 1024-256)$, $P_2 = (180, 120, 1024-256)$ and $P_3 = (150, 140, 1024-512)$, respectively. Per our discussion in Sect. 3.2, a safe region can be derived from the three data point vectors in a $\mathbb{R}^3$ space, i.e., $SR = \{x : x \leq P_i, \forall i = 1, 2, 3\}$, where inequality $x \leq P_i$ between two vectors is component-wise.

In future executions, new input variables $y = (len, n, C - dest\_bound)$ are verified by $SIMBER()$ to determine if vector x is inside this safe region, i.e., $y \in SR$. As long as we can find one vector from P_1, P_2 and P_3 that dominates y, then the memory access of $*(dest + j)$ in line 20 is safe, allowing us to remove all bound checks and propagation.

4 Evaluation

We use Softbound as the baseline to evaluate the effectiveness of SIMBER in removing redundant bound checks. All measurements are made on a 2.54 GHz Intel Xeon(R) CPU E5540 8-core server with 12 GByte of main memory. The operating system is ubuntu 14.04 LTS.

We select several applications from SPEC 2006 benchmark suite [1] with high performance overheads, including *bzip2*, *hmmer* from SPECint and *lbm*, *sphinx3* from SPECfp. In the evaluation, we first instrument the applications using Softbound, and use Perf [6] to identify the Check-HotSpot functions. Similar to ABCD [3], we consider the optimization of upper- and lower-bound checks as two separate problems. In the following, we focus on eliminating redundant upper-bound checks, and we note that this approach can be adapted to the dual problem of lower-bound checks. We use *reference* inputs provided with SPEC benchmarks. For applications that do not provide developer-supplied representative test cases, we note that fuzzing techniques [11,16] can be used to generate test cases. The policies considered in our evaluation are (a) Softbound-instrumented version (denoted as **Softbound**). (b) SIMBER-Optimized Softbound (denoted as **S.O.S**), where redundant bounds check are removed.

Based on our Check-HotSpot identification results, we study 8 functions shown in Table 1. We note that some Check-HotSpot functions may contribute to high runtime overhead mainly because they are executed frequently, e.g., *bzip2::mainGtU* is called more than 8 million times, despite having small code footprint.

4.1 Redundant Checks

To illustrate SIMBER's efficiency in eliminating redundant bounds checks, Table 1 shows the number of array bound checks required by Softbound, and the number of redundant checks removed by SIMBER along with rate of false positives reported under S.O.S. Our results show that Softbound-related checks can be completely eliminated by S.O.S in three out of eight cases.

Table 2 shows the execution time incurred by Check-Hotspot functions in Softbound and S.O.S. Our experiments show that upto 86.94% improvement in execution time overhead can be achieved by S.O.S through eliminating redundant array bound checks inserted by Softbound. In a few functions, despite totally

Table 1. Redundant array bound checks in Check-HotSpot functions

Benchmark::Function name	# Bounds checks	# Redundant checks	False positives
bzip2::generateMTFValues	2,928,640	1,440,891 (49.2%)	0 (0.0%)
bzip2::mainGtU	81,143,646	81,136,304 (99.9%)	0 (0.0%)
bzip2::BZ2_decompress	265,215	196,259 (74.0%)	0 (0.0%)
hmmer::P7Viterbi	176,000,379	124,960,267 (71.0%)	0 (0.0%)
lbm::LBM_performStreamCollide	128277886	128277886 (100.0%)	0 (0.0%)
sphinx3::vector_gautbl_eval_logs3	2,779,295	2,779,295 (100.0%)	0 (0.0%)
sphinx3::mgau_eval	725,899,332	725,899,332 (100.0%)	0 (0.0%)
sphinx3::subvq_mgau_shortlist	24,704	4,471 (18.1%)	0 (0.0%)

Table 2. Execution time improvement for Check-HotSpot functions

Function name	Time spent in		Execution time reduction
	Softbound	S.O.S	
bzip2::generateMTFValues	77.21 s	39.46 s	48.89%
bzip2::mainGtU	47.94 s	6.26 s	86.94%
bzip2::BZ2_decompress	35.58 s	9.10 s	74.42%
hmmer::P7Viterbi	3701.11 s	812.91 s	78.04%
lbm::LBM_performStreamCollide	1201.79 s	407.06 s	66.13%
sphinx3::vector_gautbl_eval_logs3	1580.03 s	318.10 s	79.87%
sphinx3::mgau_eval	1582.68 s	473.10 s	70.11%
sphinx3::subvq_mgau_shortlist	270.84 s	221.81 s	18.1%

eliminating Softbound-instrumented array bound checks, a small runtime overhead is still incurred due to the extra code added by SIMBER to circumvent redundant bound checks at the function-level.

4.2 Memory Overhead and Code Increase

We note that SIMBER's memory overhead for storing Knowledge Base and additional code instrumentation are modest. Our experiments show that the worst memory overhead is only 20 KB, and the maximum code size increase is less than 5%. Across all applications, SIMBER has an average 5.28 KB memory overhead with an average 1.7% code increase. Overall, we reduce memory overhead by roughly 50% compared to that of Softbound.

4.3 Case Studies

bzip2. *bzip2* is a compression program to compress and decompress inputs files, such as TIFF image and source tar file. We use the function *bzip2::mainGtU* as an example to illustrate how SIMBER removes redundant bound checks. Using Dependency Graph, we first identify *nblock*, i_1, and i_2 as the pointer-affecting variables for the target buffer pointer. For each execution, the Statistical Inference module computes and updates the Safe Region, which results in the following (sufficient) conditions for identifying redundant bounds checks in *bzip2::mainGtU*:

$$nblock > i_1 + 20 \text{ or } nblock > i_2 + 20 \tag{2}$$

Therefore, every time this check-hotspot function is called, SIMBER will eliminate bound checks if the inputs variables' values: *nblock*, i_1, and i_2 satisfy the conditions above. Because its safe region is one-dimensional, the condition checks have low runtime overhead. If satisfied, the conditions guarantee a complete removal of bounds checks in *bzip2::mainGtU* function.

As a second example in *bzip2::generateMTFValue*, we study the conditions to remove bound checks on five different target pointers inside of the function. We observed that three out of five target pointers, with constant array length, are relatively safe from out-of-bound accesses that may also be handled through static (pre-runtime) methods. The array bounds for the other two target pointers are not constant, and eliminating redundant checks on these pointer require a more careful consideration of runtime statistics and conditions formed using pointer-affecting variables. We note that *bzip2::BZ2_decompress* also has similar behavior.

hmmer. *hmmer* is a program for searching DNA gene sequences, and involves many double pointer operations. There is only one Check-HotSpot function, *P7Viterbi*, which contributes over 98% of the performance overhead.

Inside of the *hmmer::P7Viterbi* function, there are four double pointers: *xmx, mmx, imx* and *dmx*. To cope with double pointers in this function, we consider the row and column array bounds separately, and construct safe regions for

each dimension. Besides the four double pointers, we also identify conditions for identifying redundant bound checks for another 14 one-dimensional arrays and pointers. In this case, SIMBER is able to eliminate most of the redundant checks for these 14 one-dimensional arrays with relatively simple conditions for bound check removal. However, for the four double pointers, SIMBER is slightly more conservative due to higher number of dimensions in the conditions.

lbm. lbm is developed to simulate incompressible fluids in 3D, and has only one Check-HotSpot function: *lbm::LBM_performStreamCollide*. The function has two pointers (as input variables) with pointer assignments and dereferencing inside of a *for* loop. Using SIMBER, we obtain the bound conditions for each pointer dereferencing. Using runtime profile, we observed that the pointer dereferences to the same set of memory addresses repeatedly, providing an opportunity to remove all of the bound checks after successfully verifying bound conditions in the first iteration.

sphinx3. Sphinx3 is a well-known speech recognition system. For the first Check-HotSpot function *sphinx3::vector_gautbl_eval_logs3*, there are four target pointers inside this function. Due to the identical access pattern, once we derive the bound check removal conditions for one single pointer, it can also be used for all others, allowing for the redundant checks to be eliminated simultaneously in this function. We observed a similar behavior for a second Check-HotSpot function *sphinx3::mgau_eval*.

The last function *sphinx3::subvq_mgau_shortlist* also has four target pointers. For this function, SIMBER only removed 18.1% redundant checks. On further investigation, we found that a pointer, named *vqdist*, inside of this function had indirect memory access with its index value derived from another pointer: *map*. To handle such situations, we note that our DGs can be extended to include dependencies resulting from such indirect pointer references. Since we do not handle indirect memory accesses in the current version, we are unable to eliminate any redundant bound check that Softbound may perform for this pointer.

5 Related Work

Static code analysis and tools has been widely studied for discovering program vulnerabilities and bugs [8,20]. Nurit et al. [7] have studied techniques that target string-related bugs in C programs with conservative pointer analysis using abstract constraint expressions for pointer operations. Such static approaches require extensive program modeling and analysis (e.g., by constructing constraint solver systems) and may offer limited scope in dealing with certain vulnerabilities that occur only at runtime (e.g., due to user input-related bugs). Wurthinger et al. [17] use dominator tree to maintain the conditions for code blocks in Java-based programs. CCured [13] is a type-safe system that classifies pointers to three types: safe, sequence, dynamic, and then applies different rules to check

them. Different from these prior works, SIMBER leverages runtime profile to determine safe pointer accesses.

Statsym [9] proposes a novel framework to combine statistical and formal methods to discover for vulnerable paths in program, and can dramatically reduce the search space for vulnerable paths compared to symbolic executors such as KLEE [4]. Additionally, some works employ machine learning to improve the efficiency of static code analysis, and use the similarity of code patterns to facilitate discovery of bugs and errors [19,20]. We note that the accuracy of such methods rely on the choice of machine learning algorithms. Hardware support to identify malicious information outflows [15] and code reuse-based attacks [21] through buffer overflows have also been studied by prior works. SIMBER can work synergistically with these approaches to improve the security of applications.

6 Conclusions and Future Work

In this paper, we propose SIMBER, a framework integrating with statistics-guided inference to remove redundant array bound checks based on runtime profile. Its statistical inference adaptively builds a knowledge base using program execution logs containing variables that affect pointer values, and then uses this information to remove redundant array bound checks inserted by popular array bound checkers such as Softbound. SIMBER reduces performance overhead of Softbound by up to 86.94%, and incurs a modest 1.7% code size increase on average to circumvent redundant bound checks inserted by Softbound. Currently, SIMBER works at function-level granularity. For future work, we will study ways to deploy SIMBER at a finer granularity to remove redundant bound checks.

Acknowledgments. This work was supported by the US Office of Naval Research (ONR) under Award N00014-15-1-2210. Any opinions, findings, conclusions, or recommendations expressed in this article are those of the authors, and do not necessarily reflect those of ONR.

References

1. SPEC CPU (2006). https://www.spec.org/cpu2006/
2. SQLite. https://www.sqlite.org
3. Bodík, R., Gupta, R., Sarkar, V.: ABCD: eliminating array bounds checks on demand. ACM SIGPLAN Not. **35**, 321–333 (2000). ACM
4. Cadar, C., Dunbar, D., Engler, D.R., et al.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, vol. 8, pp. 209–224 (2008)
5. Cisco. CVE-2016-1287: Cisco ASA software IKEv1 and IKEv2 buffer overflow vulnerability (2016). https://goo.gl/QCPvut
6. de Melo, A.C.: The new linux 'perf' tools. In: Slides from Linux Kongress, vol. 18 (2010)
7. Dor, N., Rodeh, M., Sagiv, M.: CSSV: towards a realistic tool for statically detecting all buffer overflows in C. ACM SIGPLAN Not. **38**, 155–167 (2003). ACM

8. Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. IEEE Softw. **19**(1), 42–51 (2002)
9. Fan, Y., Yongbo, L., Yurong, C., Hongfa, X., Venkataramani, G., Tian, L.: StatSym: vulnerable path discovery through statistics-guided symbolic execution. In: Proceedings of 2017 47th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE (2017)
10. Google: CVE-2015-7547: glibc getaddrinfo stack-based buffer overflow (2015). https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html
11. McNally, R., Yiu, K., Grove, D., Gerhardy, D.: Fuzzing: the state of the art. Technical report, DTIC Document (2012)
12. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: SoftBound: highly compatible and complete spatial memory safety for C. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 245–258. ACM (2009)
13. Necula, G.C., McPeak, S., Weimer, W.: CCured: type-safe retrofitting of legacy code. ACM SIGPLAN Not. **37**, 128–139 (2002). ACM
14. Sui, Y., Ye, D., Su, Y., Xue, J.: Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. IEEE Trans. Reliab. **65**(4), 1682–1699 (2016)
15. Venkataramani, G., Chen, J., Doroslovacki, M.: Detecting hardware covert timing channels. IEEE Micro **36**(5), 17–27 (2016)
16. Woo, M., Cha, S.K., Gottlieb, S., Brumley, D.: Scheduling black-box mutational fuzzing. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 511–522. ACM (2013)
17. Würthinger, T., Wimmer, C., Mössenböck, H.: Array bounds check elimination for the Java HotSpot client compiler. In: 5th International Symposium on Principles and Practice of Programming in Java, pp. 125–133. ACM (2007)
18. Yamaguchi, F.: Joern: a robust code analysis platform for C/C++ (2016). http://www.mlsec.org/joern/
19. Yamaguchi, F., Lottmann, M., Rieck, K.: Generalized vulnerability extrapolation using abstract syntax trees. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 359–368. ACM (2012)
20. Yamaguchi, F., Wressnegger, C., Gascon, H., Rieck, K.: Chucky: exposing missing checks in source code for vulnerability discovery. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 499–510. ACM (2013)
21. Yao, F., Chen, J., Venkataramani, G.: Jop-alarm: detecting jump-oriented programming-based anomalies in applications. In: Proceedings of IEEE 31st International Conference on Computer Design (ICCD), pp. 467–470 (2013)