Eliminating Redundant Bounds Checks in Dynamic Buffer Overflow Detection Using Weakest Preconditions

Yulei Sui, Ding Ye, Yu Su, and Jingling Xue, *Senior Member, IEEE*

***Abstract*—Spatial errors (e.g., buffer overflows) continue to be one of the dominant threats to software reliability and security in C/C++ programs. Presently, the software industry typically enforces spatial memory safety by instrumentation. Due to high overheads incurred in bounds checking at runtime, many program inputs cannot be exercised, causing some input-specific spatial errors to go undetected in today's commercial software. This paper introduces a new compile-time approach for reducing bounds checking overheads based on the notion of weakest precondition (WP). The basic idea is to guard a bounds check at a pointer dereference inside a loop, where the WP-based guard is hoisted outside the loop, so that its falsehood implies the absence of out-of-bounds errors at the dereference, thereby avoiding the corresponding bounds check inside the loop. This WP-based approach is applicable to any spatial-error detection approach (in software or hardware or both). To evaluate the effectiveness of our approach, we take SOFTBOUND, a compile-time tool with an open-source implementation in low-level virtual machine (LLVM), as our baseline. SOFTBOUND adopts a pointer-based checking scheme with disjoint metadata, making it a state-of-the-art tool in providing compatible and complete spatial safety for C. Our new tool, called WPBOUND, is a refined version of SOFTBOUND, also implemented in LLVM, by incorporating our WP-based compiler approach comprising both intra and interprocedural optimizations. For a set of 20 C benchmarks selected from SPEC and MiBench,WPBOUND reduces the average runtime overhead of SOFTBOUND from 77% to 47% (by a reduction of 39%), with small code size increases.**

***Index Terms*—Programming environments, reasoning about programs, runtime, software engineering.**

ACRONYMS AND ABBREVIATIONS

WP	Weakest precondition.
LLVM	Low-level virtual machine.
GCC	GNU compiler collection.
ICC	Intel C++ Compiler.
SPEC	Standard performance evaluation corporation.
OO	Object-oriented.
CWE	Common weakness enumeration.
CVE	Common vulnerabilities and exposures.
NVD	National Vulnerability Database.
IR	Intermediate representation.
SCEV	SCalar evolution expression.
MPX	Intel memory protection extensions.
ISA	Instruction set architecture.
SSA	Static single assignment form.
MAR	Memory access region.
SCC	Strongly connected components.
Clang	Front-end for the LLVM compiler.

NOTATION

e	Expression.
c	Constant.
v	Variable.
ℓ	Loop.
$\mathcal{L}$	Loop nest forest.
F	Procedure.
W	Worklist.
S	Size of an integer.
SP	Size of a pointer.
cs	Callsite.
$\mathcal{O}$	Incomputable SCEV.

Address of a variable:

$[e_1, e_2]$	Interval range.
$\langle e_1, +, e_2 \rangle \ell$	Add recurrence for loop ℓ.
$e \Downarrow [e_1, e_2]$	Value range deduction.
s	Program point.
p	C pointer.
$p[k]$	C array with index k.
$*p = ..$	Store.
$... = *p$	Load.
$CallSites(F)$	Set of callsites calling F.
$Callers(F)$	Set of procedures calling F.
V	Set of values.
$\max(V)$	Maximum value in set V.
$\min(V)$	Minimum value in set V.
wp_p	Weakest precondition for pointer p.
$p_{\text{lb}}^{\text{mar}}$	Lower bound of memory access region mar.
$p_{\text{ub}}^{\text{mar}}$	Upper bound of memory access region mar.
`sChk`	Spatial check.
`wpChk`	Weakest precondition check.
r	Return value of weakest precondition check.

Manuscript received April 04, 2015; revised October 06, 2015 and January 07, 2016; accepted March 03, 2016. This work was supported by Australian Research Council (ARC) under Grants DP130101970 and DP150102109, and a generous gift by Oracle Labs. Associate Editor: C. Smidts.

The authors are with the School of Computer Science and Engineering, UNSW, Sydney, NSW 2052, Australia (e-mail: ysui@cse.unsw.edu.au; dye@cse.unsw.edu.au; ysu@cse.unsw.edu.au; jingling@cse.unsw.edu.au).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2016.2570538

I. INTRODUCTION

TOGETHER with its OO incarnation C++, C is widely used for implementing systems software (e.g., operating

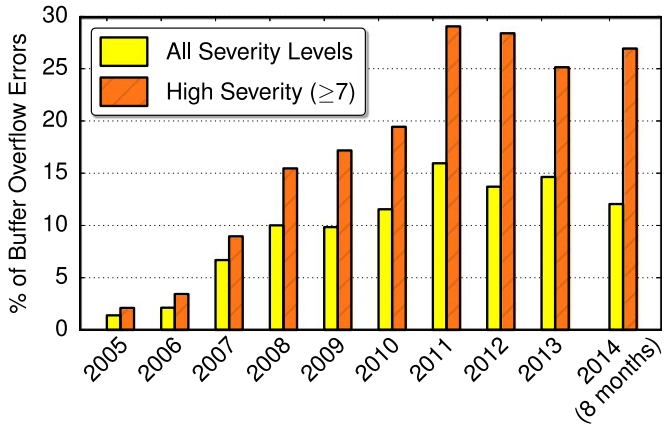


Fig. 1. Reported buffer overflow vulnerabilities in the past decade, listed as CWE-119 in the NVD database [39].

systems, language runtimes, and much of the libraries for Java and C#), embedded software (e.g., device drivers) as well as server and database applications. Due to powerful language features such as explicit memory management and pointer arithmetic, software written in C and C++ makes up the majority of performance-critical code running on most computing platforms. Unfortunately, these unsafe language features often lead to memory corruption errors, including *spatial errors* (e.g., buffer overflows) and *temporal errors* (e.g., use-after-free), undermining reliability and security.

This paper focuses on eliminating spatial errors, which directly result in out-of-bounds memory accesses of all sort and buffer overflow vulnerabilities, for C. As a long-standing problem, buffer overflows remain to be one of the highly ranked vulnerabilities, as revealed in Fig. 1, with the data taken from the National Vulnerability Database (NVD) [39]. In particular, spatial errors persist today, as demonstrated by a recently reported *Heartbleed* vulnerability in OpenSSL (CVE-2014-0160).

Several approaches exist for detecting and eliminating spatial errors for C/C++ programs at runtime: guard zone-based (by placing a guard zone of invalid memory between memory objects) [23], [24], [41], [47], [64], object-based (by maintaining per-object bounds metadata) [1], [8], [10], [13], [26], [46], pointer-based (by maintaining per-pointer metadata) either inline [2], [25], [40], [43], [60] or in a disjoint shadow space [9], [18], [34], [36]. These approaches can be implemented in software via instrumentation, at source level as in [13], [36], and [47] or binary level as in [24], and [41], accelerated in hardware [9], [34] or by a combination of both [18], [35]. As no suggested hardware support is available yet, the software industry typically employs software-only approaches to enforce spatial safety.

Detecting spatial errors at runtime via instrumentation is conceptually simple but can be computationally costly. A program is instrumented with shadow code, which records and propagates bounds metadata and performs out-of-bounds checking whenever a pointer p is used to access memory, i.e., dereferenced at a load $\cdots = *p$ or a store $*p = \cdots$. Such bounds checking can be a major source of runtime overheads, particularly when the program contains a large number of loads/stores inside loops or recursive functions.

Performing bounds checking efficiently is significant as it helps improve code coverage of a spatial-error detection tool. By being able to test against a larger set of program inputs (due to reduced runtime overheads), more input-specific spatial errors can be detected and eliminated. To this end, both software- and hardware-based optimizations have been discussed before. For example, a simple dominator-based redundant check elimination [36] enables the compiler to avoid the redundant bounds checks at any dominated memory accesses. As described in [35] and also in the recently announced MPX ISA extensions from Intel [7], new instructions are proposed to be added for the purposes of accelerating bounds checking (and propagation).

In this paper, we present a WP-based compiler approach to eliminating redundant bounds checks by performing both intra and interprocedural optimizations. By reducing the high overheads incurred in bounds checking at runtime, we can significantly increase code coverage of a spatial-error detection tool so that more spatial errors can be detected. Our approach not only complements prior bounds checking optimizations but also applies to any aforementioned spatial-error detection approach (in software or hardware or both). Based on the notion of weakest precondition (WP), its novelty lies in guarding a bounds check at a pointer dereference inside a loop, where the WP-based guard is hoisted outside the loop, so that its falsehood implies the absence of out-of-bounds errors at the dereference, thereby preventing the corresponding bounds check to be performed redundantly inside the loop. In addition, a simple value-range analysis allows multiple memory accesses to share a common guard, reducing further the associated bounds checking overheads. Finally, we apply loop unswitching to a loop to trade code size for performance so that some bounds checking operations in some versions of the loop are eliminated completely.

We demonstrate the effectiveness of our WP-based optimizations by taking SoftBound [36] as the baseline. SoftBound, with an open-source implementation available in low-level virtual machine (LLVM), represents a state-of-the-art compile-time tool for detecting spatial errors. By adopting a pointer-based checking scheme with disjoint metadata, SoftBound provides source compatibility and completeness when enforcing spatial safety for C. By performing instrumentation at source level instead of binary level as in MemCheck [41], SoftBound can reduce MemCheck's overheads significantly as both the original and instrumentation code can be optimized together by the compiler. However, SoftBound can still be costly, with performance slowdowns exceeding 2X for some programs.

To boost the performance of SoftBound, we have developed a new tool, called WPBound, which is a refined version of SoftBound, also in LLVM, by incorporating our WP-based compiler approach comprising both intra and interprocedural WP-based optimizations. Our evaluation shows that WPBound is effective in reducing SoftBound's instrumentation overheads while incurring some small code size increases for the majority of the programs evaluated.

In summary, the contributions of this paper are:

1) a WP-based compiler approach comprising both intra and interprocedural optimizations for reducing bounds checking overheads for C programs;
2) a WP-based source-level instrumentation tool, WPBOUND, for enforcing spatial safety for C programs;
3) an implementation of WPBOUND in LLVM; and
4) an evaluation on 20 C programs selected from SPEC and MediaBench, showing that WPBOUND reduces SOFTBOUND's average runtime overhead from 77% to 47% (by a reduction of 39%), with small code size increases.

The rest of this paper is organized as follows. Section II provides some background for this work. Section III motivates and describes our WP-based instrumentation approach. Section IV evaluates and analyses our approach. Section V discusses additional related work and Section VI concludes.

II. BACKGROUND

In this section, we review briefly how SOFTBOUND [36] works by adopting a pointer-based checking scheme. In Section V, we will discuss some additional related work on guard zone- and object-based approaches in detail.

Fig. 2 illustrates the pointer-based metadata initialization, propagation and checking abstractly in SOFTBOUND with the instrumentation code highlighted in gray. Instead of maintaining the per-pointer metadata (i.e., base and bound) inline [2], [25], [40], [43], [60], SOFTBOUND uses a disjoint metadata space to achieve source compatibility.

The bounds metadata are associated with a pointer whenever a pointer is created, as illustrated in Fig. 2(a). The types of base and bound are typically set as `char*` so that spatial errors can be detected at the granularity of bytes. Here, the base of p, denoted p_bs, points to the first byte in a, and the bound of p, denoted p_bd, points to one past its end. For q, q_bs and q_bd are initialized appropriately. These metadata are propagated on pointer-manipulating operations such as copying and pointer arithmetic, as illustrated in Fig. 2(b).

When pointers are used to access memory, i.e., dereferenced at loads or stores, spatial checks are performed [see Fig. 2(c) and (d)] by invoking the `sChk` function shown in Fig. 2(e). The base and bound of a pointer are available in a disjoint shadow space and can be looked up in a global map `GM`. Given a pointer p pointing to another pointer, $\texttt{GM}[p] \rightarrow bs$ and $\texttt{GM}[p] \rightarrow bd$ return the base and bound of the pointer $*p$ pointed to by p, respectively. Note that during program execution, p in $\texttt{GM}[p]$ evaluates to the r-value $*p$ as desired. `GM` can be implemented in various ways, including a hash table or a trie. For each spatial check, five $\times$ 86 instructions, `cmp`, `br`, `lea`, `cmp` and `br`, are executed on $\times$86, incurring a large amount of runtime overheads, which will be significantly reduced in our WPBOUND framework.

To detect and prevent out-of-bounds errors at a load $\cdots = *p$ or a store $*p = \cdots$, two cases are distinguished depending on whether p is a pointer to a nonpointer scalar [see Fig. 2(c)] or a pointer [see Fig. 2(d)]. In the latter case, the metadata for the pointer $*p$ (i.e., the pointer pointed by p) in `GM` is retrieved for a load $\cdots = *p$ and updated for a store $*p = \cdots$.

```
int a;
int *p = &a;
char *p_bs = p, *p_bd = (char*)(p + 1);
float *q = malloc(n);
char *q_bs = q;
char *q_bd = (q == 0) ? 0 : (char*)q + n;
```
(a)

```
int *p, *q;
char *p_bs = 0, *p_bd = 0;
char *q_bs = 0, *q_bd = 0;
...
p = q;    // p = q + i or p = &q[i]
p_bs = q_bs;
p_bd = q_bd;
```
(b)

```
float *p;
char *p_bs = 0, *p_bd = 0;
...
sChk(p, p_bs, p_bd, sizeof(float));
... = *p;    // *p = ...
```
(c)

```
int **p, *q;
char *p_bs = 0, *p_bd = 0;
char *q_bs = 0, *q_bd = 0;
...
sChk(p, p_bs, p_bd, sizeof(int*));
q = *p;              // *p = q;
q_bs = GM[p]->bs;    // GM[p]->bs = q_bs
q_bd = GM[p]->bd;    // GM[p]->bd = q_bd
```
(d)

```
inline void sChk(char *p, char *p_bs,
                 char *p_bd, size_t size) {
   if (p < p_bs || p + size > p_bd) {
      ...  // issue an error message
      abort();
   }
}
```
(e)

Fig. 2. Pointer-based instrumentation with disjoint metadata. (a) Memory allocation. (b) Copying and pointer arithmetic. (c) Loads and stores for nonpointer variables. (d) Loads and Stores for pointer variables. (e) Spatial checks.

III. METHODOLOGY

As shown in Fig. 3, WPBOUND, which is implemented in the LLVM compiler infrastructure, consists of one analysis phase, one intraprocedural optimization phase and one interprocedural phase. Their functionalities are briefly described in Section III-A, illustrated by an example in Section III-B, and finally, explained further in Sections III-D–III-F.

A. *WPBOUND Framework*

The functionalities of the three phases in WPBOUND are described below. The rationale behind each phase will be further illustrated by a motivating example in Section III-B.

1) *Value Range Analysis:* This analysis phase computes conservatively the value ranges of pointers dereferenced at loads and stores, leveraging LLVM's *scalar evolution*

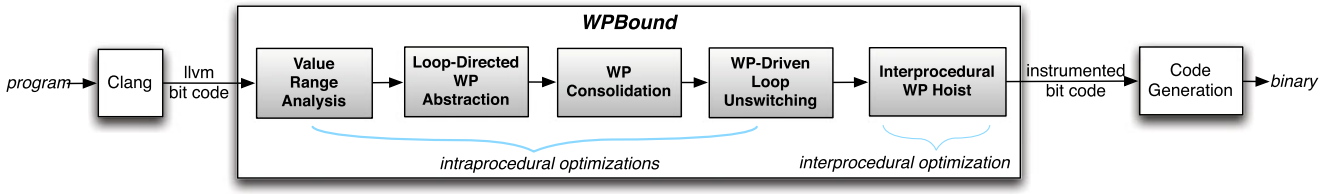


Fig. 3. Overview of the WPBOUND framework.

pass. The value range information is used for the WP computations in the intra and interprocedural transformation phases, where the instrumentation code is generated.

2) *Intraprocedural Bounds-Checking Elimination:* In this WP-based intraprocedural bounds-checking elimination phase, we proceed in three stages:
   a) *Loop-Directed WP Abstraction:* In this first stage, we insert spatial checks for memory accesses (at loads and stores). For each access in a loop, we reduce its bounds checking overhead by exploiting but not actually computing exactly the WP that verifies the assertion that an out-of-bounds error definitely occurs at the access during some program execution. As value-range analysis is imprecise, a WP is estimated conservatively, i.e., weakened. For convenience, such WP estimates are still referred to as WPs. For each access in a loop, its bounds check is guarded by its WP, with its evaluation hoisted outside the loop, so that its falsehood implies the absence of out-of-bounds errors at the access, preventing its check to be made redundantly inside the loop.
   b) *WP Consolidation:* During this second stage, we consolidate the WPs for multiple accesses, which are always made to the same object, into a single one, in order to reduce the number of WPs used.
   c) *WP-Driven Loop Unswitching:* During the last stage, we trade code size for performance by applying loop unswitching to a loop so that the instrumentation in its frequently executed versions is effectively eliminated.
3) *Interprocedural WP Hoisting:* This phase allows our WP-based approach to be applied across the procedural boundaries. Our WP-based interprocedural optimization performs a context-sensitive whole-program analysis to hoist recursively some WP checks along call chains to further reduce their runtime instrumentation overhead incurred.

B. Motivating Example

We explain how WPBOUND works with a program in C (rather than in its LLVM low-level code) given in Fig. 4. In the program shown in Fig. 4(a), there are a total of five memory accesses in function `bar`, four loads (lines 21, 24, 29, and 32) and one store (line 34), with the last three contained in a `for` loop. With the unoptimized instrumentation (as obtained by SOFTBOUND), each memory access triggers a spatial check (highlighted in gray). To avoid cluttering, we do not show the medadata initialization and propagation, which are irrelevant to our WP-based optimizations.

1) *Value Range Analysis:* We compute conservatively the value ranges of all pointers dereferenced for memory accesses in the program, by using LLVM's scalar evolution pass. For the five dereferenced pointers, we obtain their value ranges as follows:

$$\begin{aligned} \&\mathtt{p}[\mathtt{k}] &: [\mathtt{p}+\mathtt{k}\times\mathtt{SP},\ \mathtt{p}+\mathtt{k}\times\mathtt{SP}] \\ \&\mathtt{p}[\mathtt{k}+1] &: [\mathtt{p}+(\mathtt{k}+1)\times\mathtt{SP},\ \mathtt{p}+(\mathtt{k}+1)\times\mathtt{SP}] \\ \&\mathtt{a}[\mathtt{i}-1] &: [\mathtt{a},\ \mathtt{a}+(\mathtt{L}-1)\times\mathtt{S}] \\ \&\mathtt{b}[\mathtt{i}] &: [\mathtt{b}+\mathtt{S},\ \mathtt{b}+\mathtt{L}\times\mathtt{S}] \\ \&\mathtt{a}[\mathtt{i}] &: [\mathtt{a}+\mathtt{S},\ \mathtt{a}+\mathtt{L}\times\mathtt{S}] \end{aligned}$$

where the two constants, `S` and `SP`, are defined at the beginning of the program in Fig. 4(a), and `L` is the upper bound of the `for` loop spanning lines 27–36. The values of `a`, `b` and `L` (used in `bar`) are obtained from their corresponding formal parameters `x`, `y`, and `N` at the callsite in line 9 of `foo`.

2) *Loop-Directed WP Abstraction:* According to the value ranges computed above, the WPs for all memory accesses at loads and stores are computed (weakened if necessary). The WPs for the three memory accesses in the `for` loop are found conservatively and hoisted outside the loop to perform a *WP check* by calling `wpChk` given in Fig. 4(a), as shown in Fig. 4(b). The three spatial check calls to `sChk` at `a[i-1]`, `b[i]` and `a[i]` that are previously unconditional (in SOFTBOUND) are now guarded by their WPs, `wp_a1`, `wp_b` and `wp_a2`, respectively.
   Note that `wp_a1` is exact since its guarded access `a[i-1]` will be out-of-bounds when `wp_a1` holds. However, `wp_b` and `wp_a2` are not since their guarded accesses `b[i]` and `a[i]` will never be executed if expression `t < ...` in line 30 always evaluates to false. In general, a WP for an access is constructed so that its falsehood implies the absence of out-of-bounds errors at the access, thereby causing its spatial check to be elided. The WPs for the other two accesses `p[k]` and `p[k + 1]` are computed similarly but omitted in Fig. 4(b).
3) *WP Consolidation:* The WPs for accesses to the same object are considered for consolidation. The code in Fig. 4(b) is further optimized into the one in Fig. 4(c), where the two WPs for `p[k]` and `p[k + 1]` are merged as `cwp_p` and the two WPs for `a[i - 1]` and `a[i]` as `cwp_a`.

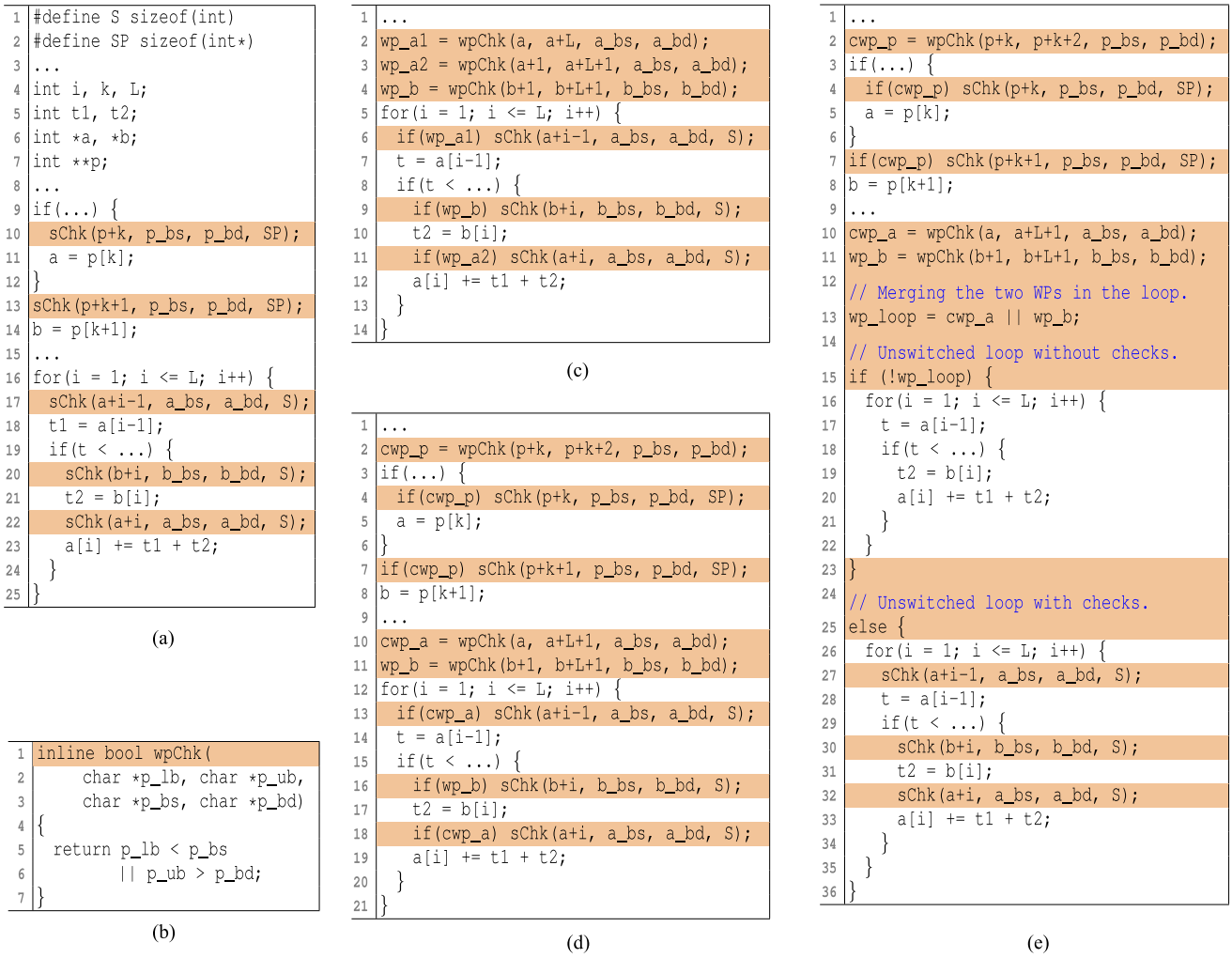


Fig. 4. Motivating example. (a) Unoptimized instrumentation. (b) Loop-directed WP abstraction. (c) WP consolidation. (d) WP-Driven Loop unswitching. (e) Interprocedural WP hoisting.

Thus, the number of `wpChk` calls has dropped from 5 to 3 (appearing in lines 3, 11, and 12).

4) *WP-Driven Loop Unswitching:* By applying loop unswitching, we obtain the code given in Fig. 4(d). The two WPs in the loop, `cwp_a` and `wp_b`, are merged as `cwp_a || wp_b`, enabling the loop to be unswitched. The `if` branch at lines 6–14 is instrumentation-free. The `else` branch at lines 16–27 proceeds as before with the usual spatial checks performed. The key insight for trading code size for performance this way is that the instrumentation-free loop version is often executed more frequently at runtime than its instrumented counterpart in real programs.
5) *Interprocedural WP Hoisting:* In Fig. 4(d), the function `bar` contains the two WP checks made by calling `wpChk` in lines 3 and 4. Performing WP checks this way can be costly when `bar` is called frequently by `foo`. We can significantly reduce such instrumentation overhead by hoisting the two `wpChk` calls from `bar` interprocedurally to the point just above the `for` loop in `foo`, as shown in lines 3 and 4 of Fig. 4(e). Of course, hoisting WPs this way requires an appropriate parameter mapping to be performed, with `a` mapped to `x`, `b` to `y` and `L` to `N`. In addition, two global boolean shadow variables, `gcwp_a` and `cwp_b`, have been introduced to enable the hoisted WPs to be used interprocedurally.

C. Low-Level Virtual Machine Intermediate Representation

WPBOUND, as shown in Fig. 3, works directly on the LLVM-IR, LLVM's intermediate representation (IR). As illustrated in Fig. 5, all program variables are partitioned into a set of *top-level variables* (e.g., `a`, `x` and `y`) that are not referenced by pointers, and a set of *address-taken variables* (e.g., `b` and `c`) that can be referenced by pointers. In particular, top-level variables are maintained in SSA (Static Single Assignment) form so that each variable use has a unique definition, but address-taken variables are not in SSA form.

All address-taken variables are kept in memory and can only be accessed (indirectly) via loads ($q = *p$ in pseudocode) and stores ($*p = q$ in pseudocode), which take only top-level pointer variables as arguments. Furthermore, an address-taken variable

```
int **a, *b;          a = &b;
int c, i;             x = &c;
a = &b;               *a = x;
b = &c;               y = 10;
c = 10;               *x = y;
i = c;                i = *x;
```

(a) (b)

Fig. 5. LLVM-IR (in pseudocode) for a C program (where x and y are top-level temporaries introduced). (a) C. (b) LLVM-IR (in pseudocode).

can only appear in a statement where its address is taken. All the other variables referred to are top-level.

In the rest of this paper, we will focus on memory accesses made at the pointer dereferences $*p$ via loads $\cdots = *p$ and stores $*p = \cdots$, where pointers p are always top-level pointers in the IR. These are the points where the spatial checks are performed as illustrated in Fig. 2(c) and (d).

Given a pointer p (top-level or address-taken), its bounds metadata, base (lower bound) and bound (upper bound), are denoted by $p_{\rm bs}$ and $p_{\rm bd}$, respectively, as shown in Fig. 2.

D. Value Range Analysis

We describe this analysis phase for estimating conservatively the range of values accessed at a pointer dereference, where a spatial check is performed. We conduct our analysis based on LLVM's scalar evolution pass (see Fig. 3), which calculates closed-form expressions for all top-level scalar integer variables (including top-level pointers) in the way described in [55]. This pass, inspired by the concept of *chains of recurrences* [4], is capable of handling any value taken by an induction variable at any iteration of its enclosing loops.

A scalar integer expression in the program can be represented as a SCEV (SCalar EVolution expression):

$$e := c \mid v \mid \mathcal{O} \mid e_1 + e_2 \mid e_1 \times e_2 \mid \langle e_1, +, e_2 \rangle \ell.$$

Therefore, a SCEV can be a constant c, a variable v that cannot be represented by other SCEVs, or a binary operation (involving $+$ and $\times$ as considered in this paper). In addition, when loop induction variables are involved, an add recurrence $\langle e_1, +, e_2 \rangle \ell$ is used, where e_1 and e_2 represent, respectively, the initial value (i.e., the value for the first iteration) and the stride per iteration for the containing loop ℓ. For example, in Fig. 4(a), the SCEV for the pointer `&a[i]` contained in the `for` loop in line 27 is $\langle \texttt{a}, +, \texttt{sizeof(int)} \rangle \ell_{27}$, where the subscript ℓ_{27} stands for the loop at line 27. Finally, the notation $\mathcal{O}$ is used to represent any value that is neither expressible nor computable in the SCEV framework.

When performing our range analysis, we ensure that the functional equivalence between a program and its transformed program is preserved in the following sense. According to the range analysis rules given in Fig. 6, the WP-generating SCEV expressions involving $+$ and $*$ will not throw any runtime exceptions. For example, the division operator/is not considered. In addition, we handle unsigned and signed integers in the standard manner.

$$\text{[Termi]}\ \frac{}{e \Downarrow [e,\ e] \quad (e = c \mid v \mid \mathcal{O})}$$

$$\text{[Add]}\ \frac{e_1 \Downarrow [e_1^l,\ e_1^u] \qquad e_2 \Downarrow [e_2^l,\ e_2^u]}{e_1 + e_2 \Downarrow [e_1^l + e_2^l,\ e_1^u + e_2^u]}$$

$$\text{[Mul]}\ \frac{e_1 \Downarrow [e_1^l,\ e_1^u] \qquad e_2 \Downarrow [e_2^l,\ e_2^u] \qquad V = \{e_1^l \times e_2^l,\ e_1^l \times e_2^u,\ e_1^u \times e_2^l,\ e_1^u \times e_2^u\}}{e_1 \times e_2 \Downarrow [\min(V),\ \max(V)]}$$

$$\text{[AddRec]}\ \frac{e_1 \Downarrow [e_1^l,\ e_1^u] \quad e_2 \Downarrow [e_2^l,\ e_2^u] \quad tc(\ell) \Downarrow [\_,\ \ell^u] \quad V = \{e_1^l,\ e_1^u + e_2^l \times (\ell^u - 1),\ e_1^u + e_2^u \times (\ell^u - 1)\}}{<e_1, +, e_2>_\ell \Downarrow [\min(V),\ \max(V)]}$$

Fig. 6. Range analysis rules.

The range of every scalar variable will be expressed in the form of an interval $[e_1, e_2]$. We handle unsigned and signed values differently due to possible integer overflows. According to the C standard, unsigned integer overflow wraps around but signed integer overflow leads to undefined behavior. To avoid potential overflows, we consider conservatively the range of an unsigned integer variable as $[\mathcal{O}, \mathcal{O}]$. For operations on signed integers, we assume that overflow never occurs. This assumption is common in compiler optimizations. For example, the following function (with x being a signed int):

```
bool foo(int x) { return x + 1 < x; }
```

is optimized by LLVM, GCC, and ICC to return false.

The rules used for computing the value ranges of signed integer and pointer variables are given in Fig. 6. [TERMI] suggests that both the lower and upper bounds of a SCEV, which is c, v or $\mathcal{O}$, are the SCEV itself. [ADD] asserts that the lower (upper) bound of an addition SCEV $e_1 + e_2$ is simply the lower (upper) bounds of its two operands added together. When it comes to a multiplication SCEV, the usual $\min$ and $\max$ functions are called for, as indicated in [MUL]. If $\min(V)$ and $\max(V)$ cannot be solved statically at compile time, then $[\mathcal{O}, \mathcal{O}]$ is assumed. For example, $[i, i + 10] \times [2, 2] \Downarrow [2i, 2i + 20]$ but $[i, 10] \times [j, 10] \Downarrow [\mathcal{O}, \mathcal{O}]$, where i and j are scalar variables (which can contain either positive or negative values). In the latter case, the compiler cannot statically resolve $\min(V)$ and $\max(V)$, where $V = \{10i, 10j, ij, 100\}$.

For an add recurrence, the LLVM scalar evolution pass computes the trip count of its containing loop ℓ, which is also represented as a SCEV $tc(\ell)$. A trip count can be $\mathcal{O}$ since it may neither be expressible nor computable in the SCEV formulation. In the case of a loop with multiple exits, the worst-case trip count is picked. Here, we assume that a trip count is always positive. However, this will not affect the correctness of our overall approach, since the possibly incorrect range information is never used inside a nonexecuted loop.

In addition to some simple scenarios demonstrated in our motivating example, our value range analysis is capable of handling more complex ones, as long as LLVM's scalar evolution

is. Consider the following double loop:

```
for (int i = 0; i < N; ++i) // L1
    for (int j = 0; j <= i; ++j) // L2
        a[2*i+j] = ...; // a declared as int*.
```

The SCEV of `&a[2 * i+j]`, i.e., `a+2 * i+j` is given as $\langle\langle \mathtt{a}, +, 2 \times \mathtt{sizeof(int)}\rangle L1, +, \mathtt{sizeof(int)}\rangle L2$ by scalar evolution, and $tc(L1)$ and $tc(L2)$ are `N` and $\langle 0, +, 1\rangle L1 + 1$, (i.e., `i` +1), respectively. The value range of `&a[2 * i + j]` is then deducted via the rules in Fig. 6 as:

$$[\mathtt{a},\ \mathtt{a} + 3 \times (\mathtt{N} - 1) \times \mathtt{sizeof(int)}].$$

Since the maximum values of `i` and `j` are both `N - 1`, the upper bound for `a + (2 * i + j) × sizeof(int)` is therefore `a + (2 * (N - 1) + (N - 1)) × sizeof(int)`, which simplifies to `a + 3 * (N - 1) × sizeof(int)`.

E. Intraprocedural Bounds-Checking Elimination

We describe how WPBOUND generates the instrumentation code for a program during its intraprocedural optimization phase, based on the results of value range analysis. We only discuss how bounds checking operations are inserted in its three stages since WPBOUND handles metadata initialization and propagation exactly as in SOFTBOUND, as illustrated in Fig. 2.

1) Loop-Directed WP Abstraction: During this first stage, we compute the WPs for all dereferenced pointers and inserts guarded or unguarded spatial checks for them. As shown in our motivating example, we do so by reasoning about the WP for a pointer p at a load $\cdots = *p$ or a store $*p = \cdots$. Based on the results of value range analysis, we estimate the WP for p according to its *Memory Access Region* (MAR), denoted $[p_{\mathrm{lb}}^{\mathrm{mar}}, p_{\mathrm{ub}}^{\mathrm{mar}})$. Let the value range of p be $[p_{\mathrm{l}}, p_{\mathrm{u}}]$. There are two cases:

a) $p_{\mathrm{l}} \neq \mathcal{O} \wedge p_{\mathrm{u}} \neq \mathcal{O}$: $[p_{\mathrm{lb}}^{\mathrm{mar}}, p_{\mathrm{ub}}^{\mathrm{mar}}) = [p_{\mathrm{l}},\ p_{\mathrm{u}} + \mathtt{sizeof}(*p))$. As a result, its WP is estimated to be

$$p_{\mathrm{lb}}^{\mathrm{mar}} < p_{\mathrm{bs}} \ \vee\ p_{\mathrm{ub}}^{\mathrm{mar}} > p_{\mathrm{bd}}$$

where p_{bs} and p_{bd} are the base and bound of p (Section III-C). The result of evaluating this WP, called a *WP check*, can be obtained by a call to `wpChk` $(p_{\mathrm{lb}}^{\mathrm{mar}}, p_{\mathrm{ub}}^{\mathrm{mar}}, p_{\mathrm{bs}}, p_{\mathrm{bd}})$ in Fig. 4(a).

b) $p_{\mathrm{l}} = \mathcal{O} \vee p_{\mathrm{u}} = \mathcal{O}$: The MAR of p is $[p_{\mathrm{lb}}^{\mathrm{mar}}, p_{\mathrm{ub}}^{\mathrm{mar}}) = [\mathcal{O}, \mathcal{O})$ conservatively. The WP is set as true.

In general, the WP thus constructed for p is not the weakest one, i.e., the one ensuring that if it holds during program execution, then some accesses via $*p$ must be out-of-bounds. There are two reasons for being conservative. First, value range analysis is imprecise. Second, all branch conditions [e.g., the one in line 30 in Fig. 4(a)] affecting the execution of $*p$ are ignored during this analysis, as explained in Section III-B.

However, by construction, the falsehood of the WP for p always implies the absence of out-of-bounds errors at $*p$, in which case the spatial check at $*p$ can be elided. However, the converse may not hold, implying that some bounds checking operations performed when the WP holds are redundant.

Algorithm 1: Loop-Directed WP Abstraction.

```
Procedure INSTRUMENT(F)
begin
1   foreach pointer dereference *p in function F do
2       Let SIZE be sizeof(*p);
3       s ← POSITIONINGWP("*p", p);
4       if [p_lb^mar, p_ub^mar] ≠ [O, O) ∧ s ≠ p then
5           Insert a wpChk call for *p at point s:
              wp_p = wpChk(p_lb^mar, p_ub^mar, p_bs, p_bd);
6           Insert a guarded spatial check before *p:
              if (wp_p) sChk(p, p_bs, p_bd, SIZE);
        else
7           Insert an unguarded spatial check before *p:
              sChk(p, p_bs, p_bd, SIZE);

Procedure POSITIONINGWP(x, p)
begin
8   s ← x; // x denotes a point where *p is
9   while s is inside a loop do
10      Let ℓ be the innermost loop containing s;
11      if p_lb^mar and p_ub^mar are invariants in ℓ then
12          s ← the point just before ℓ;
13      else break;
14  return s;
```

After the WPs for all dereferenced pointers in a program are available, INSTRUMENT (F) in Algorithm 1 is called for each function F in the program to guard the spatial check at each pointer dereference $*p$ by its WP when its MAR is neither $[\mathcal{O}, \mathcal{O})$ (in which case, its WP is true) nor loop-variant. In this case (lines 4–6), the guard for p, which is loop-invariant at point s, is hoisted to the point identified by POSITIONINGWP(), where it is evaluated. Note that the first parameter of POSITIONINGWP() expects a program point to be specified. When calling it in line 3, "$*p$" stands for the program point where p is accessed. The spatial check at the pointer dereference $*p$ becomes conditional on the guard. Otherwise (line 7), the spatial check at the dereference $*p$ is unconditional as is the case in SOFTBOUND.

Note that an access $*p$ may appear in a set of nested loops. POSITIONINGWP returns the point just before the loop at the highest depth for which the WP for p is loop-invariant and p (representing the point where $*p$ occurs) otherwise.

Let us return to Fig. 4(b). The MAR of `b[i]` in line 11 is $[\mathtt{b} + \mathtt{SZ}, \mathtt{b} + (\mathtt{L} + 1) \times \mathtt{SZ})$, whose lower and upper bounds are invariants of the `for` loop in line 6. With the WP check, `wp_b`, evaluated in line 5, the spatial check for `b[i]` inserted in line 10 is performed only when `wp_b` is true.

Compared to SOFTBOUND that produces the unguarded instrumentation code as explained in Section II, our WP-based instrumentation may increase code size slightly. However, many

Algorithm 2: WP Consolidation.

Procedure CONSOLIDATEWP(F)
begin
1 $W \leftarrow$ set of pointers dereferenced in function F;
2 **while** $W \neq \varnothing$ **do**
3 $p \leftarrow$ a pointer from W;
4 $G \leftarrow \{p\}$;
5 $s_p \leftarrow$ POSITIONINGWP(p);
6 **foreach** $q \in W$ such that $q \neq p$ **do**
7 $s_q \leftarrow$ POSITIONINGWP(q);
8 $p'_{\rm lb} \leftarrow \min(\{p^{\rm mar}_{\rm lb}, q^{\rm mar}_{\rm lb}\})$;
9 $p'_{\rm ub} \leftarrow \max(\{p^{\rm mar}_{\rm ub}, q^{\rm mar}_{\rm ub}\})$;
10 $s'_p \leftarrow$ DOMINATOR($F, s_p, s_q, p'_{\rm lb}, p'_{\rm ub}$);
11 **if** $s'_p \neq \epsilon$ **then**
12 $G \leftarrow G \cup \{q\}$;
13 $p^{\rm mar}_{\rm lb} \leftarrow p'_{\rm lb}$;
14 $p^{\rm mar}_{\rm ub} \leftarrow p'_{\rm ub}$;
15 $s_p \leftarrow s'_p$;
16 **if** $G \neq \{p\}$ **then**
17 Insert a `wpChk` call for $*p$ at point s_p:
$cwp_G = \texttt{wpChk}(p^{\rm mar}_{\rm lb}, p^{\rm mar}_{\rm ub}, p_{\rm bs}, p_{\rm bd})$;
18 **foreach** $q \in G$ **do**
19 Let $SIZE$ be $sizeof(*q)$;
20 Replace the spatial check for $*q$ by:
`if` (cwp_G) `sChk`$(q, q_{\rm bs}, q_{\rm bd}, SIZE)$;
21 $W \leftarrow W - G$;

Procedure DOMINATOR($F, s_1, s_2, p_{\rm l}, p_{\rm u}$)
begin
22 **if** $p_{\rm l} = \mathcal{O} \ \vee \ p_{\rm u} = \mathcal{O}$ **then return** ϵ;
23 $V \leftarrow \{v \mid$ variable v occurs in SCEV $p_{\rm l}$ or SCEV $p_{\rm u}\}$;
24 $S \leftarrow$ set of (program) points in the CFG of F;
25 **if** $\exists\ s \in S$: (s dominates s_1 and s_2 in F's CFG) $\wedge$ ($\forall\ v \in V$: the def of v dominates s in F's CFG) **then**
26 **return** s;
else
27 **return** ϵ;

Algorithm 3: WP-Driven Loop Unswitching.

Procedure LOOPUNSWITCHING(F)
begin
1 $\mathcal{L} \leftarrow$ a loop nest forest obtained in function F;
2 **foreach** loop ℓ in reverse topological order in $\mathcal{L}$ **do**
3 $S \leftarrow \{wp \mid (1)$ "`if (wp) sChk(...)`" is inside $\ell \ \wedge\ (2)\ wp$ is an invariant in $\ell \ \wedge\ (3)\ (\nexists\ \ell' \in \mathcal{L}$: ℓ' contains $\ell \ \wedge\ wp$ is an invariant in $\ell')\}$;
4 **if** $S = \varnothing$ **then continue**;
5 $\Pi \leftarrow$ a partition of S into groups;
6 **foreach** group $\pi \in \Pi$ **do**
7 Insert $wp_\pi \leftarrow \bigvee_{wp \in \pi} wp$ just outside ℓ;
8 **foreach** $wp \in \pi$ **do**
9 Replace each wp inside ℓ by wp_π;
10 Unswitch ℓ for every wp_π, where $\pi \in \Pi$;

WPs are expected to be true in real programs. Instead of the five instructions, `cmp`, `br`, `lea`, `cmp`, and `br`, required for performing a spatial check, `sChk`, two instructions, `cmp` and `br`, are usually executed to test its guard only.

2) WP Consolidation: During this second stage, we conduct an intraprocedural analysis to combine the WPs corresponding to a set of memory accesses to *the same object* (e.g., the same array) into a single one to be shared [e.g., `cwp_p` and `cwp_a` in Fig. 4(c)]. If a pointer dereference is not in a loop, its spatial check is not guarded according to Algorithm 1 (since $s = p$ in line 3). By combining its WP with others, we will also make such a check guarded as well [e.g., `cwp_p` in Fig. 4(c)].

Algorithm 2 is simple. Given a function F, where W initially contains all pointers dereferenced at loads and stores in F (line 1), we start with $G = \{p\}$ (line 4). We then add iteratively all other pointers $q_1, \ldots, q_n$ in F (lines 6–15) to $G = \{p, q_1, \ldots, q_n\}$, so that the following properties hold:

1) *Proposition 1:* All these pointers point to the same object. If q selected in line 6 does not point to the same object as p, $p'_{\rm lb}$ or $p'_{\rm ub}$ will be $\mathcal{O}$, causing $s'_p = \epsilon$ (due to line 22). In this case, q will not be added to G (line 11).
2) *Proposition 2:* The WPs for these pointers are invariants with respect to point s_p found at the end of the *foreach* loop in line 6 (due to lines 23–27). As all variables in V (line 23) are in SSA form, the definition of v in line 25 is unique.

When $|G| > 1$ (line 16), we can combine the WPs in G into a single one, cwp_G (line 17), where $[p^{\rm mar}_{\rm lb}, p^{\rm mar}_{\rm ub})$ is constructed to be the union of the MARs of all the pointers in G. Note that `wpChk` is called only once since $\forall q \in G : q_{\rm bs} = p_{\rm bs} \wedge q_{\rm bd} = p_{\rm bd}$ by construction. In lines 18–20, the spatial checks for all pointers in G are modified to use cwp_G instead.

Consider Fig. 4(c) again. The MARs for `a[i-1]` in line 15 and `a[i]` in line 20 are $[\texttt{a},\ \texttt{a} + \texttt{L} \times \texttt{SZ})$ and $[\texttt{a} + \texttt{SZ},\ \texttt{a} + (\texttt{L} + 1) \times \texttt{SZ})$, respectively. The consolidated MAR is $[\texttt{a},\ \texttt{a} + (\texttt{L} + 1) \times \texttt{SZ})$, yielding a WP `cwp_a` weaker than the WPs, `wp_a1` and `wp_a2`, for `a[i-1]` and `a[i]`, respectively. The WP check `cwp_a` is inserted in line 11, which dominates `a[i-1]` and `a[i]` in the CFG. The spatial checks for `a[i-1]` and `a[i]` are now guarded by `cwp_a`.

3) WP-Driven Loop Unswitching: During this last stage of our intraprocedural optimization phase, we apply loop unswitching, a standard loop transformation, to a loop, as illustrated in Fig. 4(d), to unswitch some guarded spatial checks, so that its guards are hoisted outside the loop, resulting in their repeated tests inside the loop being effectively removed in some versions of the loop. However, unswitching all branches in a loop may lead to code growth exponential in its number of branches.

To avoid code explosion, we apply Algorithm 3 to a function F to process its loops inside out. For a loop ℓ (line 2), we first partition a set S of its guarding WPs selected in line 3 into a few groups (discussed below in more detail) (line 5). We then insert a disjunction wp_π built from the WPs in each group π just before ℓ (line 7). As wp_π is weaker than each constituent wp, we can replace each wp by wp_π at the expense of more spatial checks (lines 8–9). Finally, we unswitch loop ℓ so that each spatial check guarded by wp_π is either performed unconditionally (in its true version) or removed (in its false version). As these "unswitched" checks will not be considered again (line 3), our algorithm will eventually terminate.

Let us discuss the three conditions used in determining a set S of guarding WPs to unswitch in line 3. Condition (1) instructs us to consider only guarded special checks. Condition (2) avoids any guarding WP that is loop-variant since it may be introduced by Algorithm 2. Condition (3) allows us to exploit a sweet-spot to make a tradeoff between code size and performance for real code. Without (3), S tends to be larger, leading to weaker wp_π's than otherwise. As a result, we tend to generate fewer loop versions, by trading performance for code size. With (3), the opposite tradeoff is made.

In line 5, there can be a number of ways to partition S. In general, a fine-grained partitioning eliminates more redundant bounds checks than a coarse-grained partitioning, but results in more code versions representing different combinations of instrumented and uninstrumented memory accesses. Note that the space complexity (i.e., code expansion) of loop unswitching is exponential to $|\Pi|$, i.e., the number of partitions.

To keep code sizes manageable in our implementation of this algorithm, we have adopted a simple partitioning strategy by setting $\Pi = \{S\}$. Together with Conditions (1)–(3) in line 3, this partitioning strategy is effective in practice.

Let us apply our algorithm to Fig. 4(c) to unswitch the `for` loop, which contains two WP guards, `cwp_a` and `wp_b`. Merging the two WP guards and then unswitching the loop yields the final code in Fig. 4(d). There are two versions for the loop: the instrumentation-free version appears in lines 6–14 and the instrumented one in lines 16–27.

F. Interprocedural WP Hoisting

In our interprocedural phase, we apply our WP-based approach across the procedural boundaries to reduce instrumentation overhead further. Given a function F called from inside a loop l, the basic idea is to hoist some WP checks made in the callee function F outside the loop l so that these WP checks are performed just once before the loop begins rather than repeatedly every time when F is called. This optimization is illustrated earlier in Fig. 4(e), with the two WP checks in `bar` being hoisted outside the `for` loop at line 5. Presently, our optimization provides benefits for a WP check only if it can be moved backwards across the boundaries of some loops.

Given a function F, INTERWP in Algorithm 4 is applied recursively to hoist some WP checks from F to its callers interprocedurally along its call chains. Our algorithm is context sensitive when hoisting a WP check out of a callee function into its different calling contexts. For a function pointer used to call a function at a callsite, we apply a flow-sensitive pointer analysis introduced in [63] to find its callee functions, i.e., its indirect call graph edges soundly and precisely. As is standard, recursion cycles discovered in the call graph of the program are collapsed into Strongly Connected Components. A WP check is not hoisted from a callee function F into its callers if F is involved in a recursion cycle (line 1).

Every WP check $r = wpChk(p_{\rm lb}^{\rm mar}, p_{\rm ub}^{\rm mar}, \ldots)$ in the callee F is potentially a candidate to be hoisted (line 4). Let $F_{\rm cs}$ be a caller function containing a callsite cs at which F is called. Note that the base and bounds metadata are propagated across the functions using a shadow stack mechanism, as discussed in Section II. However, the actual parameters in WP checks (i.e., $p_{\rm lb}^{\rm mar}$ and $p_{\rm ub}^{\rm mar}$) that are hoisted from a callee are replaced explicitly by the pointers in the caller scope by an interprocedural parameter mapping performed at compile time (lines 6–10). To ensure that $r = wpChk(p_{\rm lb}^{\rm mar}, p_{\rm ub}^{\rm mar}, \ldots)$ is well-defined in $F_{\rm cs}$, the WP check can be mapped from F to $F_{\rm cs}$ if the bounds $p_{\rm lb}^{\rm mar}$ and $p_{\rm ub}^{\rm mar}$ (inferred by our intraprocedural SCEV anal-

Algorithm 4: Inter-Procedural WP Hoisting.

Procedure INTERWP(F)
begin
1 if F appears in a recursion cycle in the call graph of the program being optimized **then return**;
2 $CallSites(F) \leftarrow$ set of callsites calling F;
3 $S_F \leftarrow$ set of WP checks contained in F;
4 **foreach** $(r = wpChk(p_{\rm lb}^{\rm mar}, p_{\rm ub}^{\rm mar}, \ldots)) \in S_F$ **do**
5 **if** $p_{\rm lb}^{\rm mar}$ *and* $p_{\rm ub}^{\rm mar}$ involve only constants, globals or formal parameters of F **then**
6 Let r' be a global boolean variable;
7 **foreach** $cs \in CallSites(F)$ **do**
8 Let $r' = wpChk'_{cs}(q_{\rm lb}^{\rm mar}, q_{\rm ub}^{\rm mar}, \ldots)$ be the new WP check obtained from $r = wpChk(p_{\rm lb}^{\rm mar}, p_{\rm ub}^{\rm mar}, \ldots)$ for the callsite cs after all formal parameters of F in $wpChk'_{cs}$ have been replaced by their corresponding actual parameters at cs;
9 PLACEWP($cs, r' = wpChk'_{cs}(q_{\rm lb}^{\rm mar}, q_{\rm ub}^{\rm mar}, \ldots)$);
10 Replace all uses of r with r';
11 Remove $r = wpChk(p_{\rm lb}^{\rm mar}, p_{\rm ub}^{\rm mar}, \ldots)$;
12 $Callers(F) \leftarrow$ set of caller functions calling F;
13 **foreach** $G \in Callers(F)$ **do**
INTERWP(G);

Procedure PLACEWP($cs, r = wpChk'_{cs}(q_{\rm lb}^{\rm mar}, q_{\rm ub}^{\rm mar}, \ldots)$)
begin
14 $s \leftarrow$ POSITIONINGWP ("cs", q)
15 **if** $s \neq$ "cs'' **then**
16 insert $r = wpChk'_{cs}(q_{\rm lb}^{\rm mar}, q_{\rm ub}^{\rm mar}, \ldots)$ at point s;
17 **else**
18 insert $r = wpChk'_{cs}(q_{\rm lb}^{\rm mar}, q_{\rm ub}^{\rm mar}, \ldots)$ before cs;

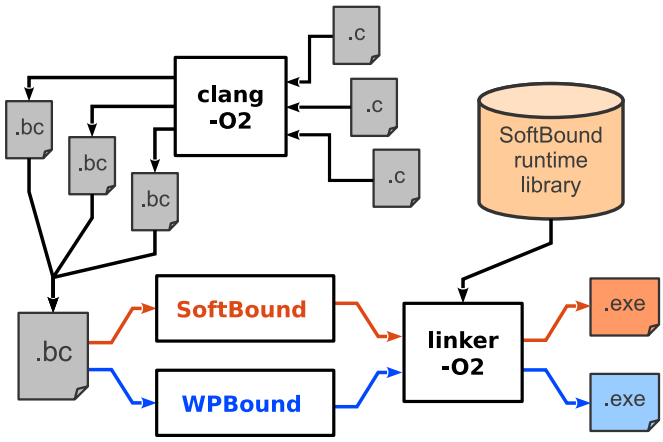


Fig. 7. Interprocedural WP hoisting in `hmmer`.

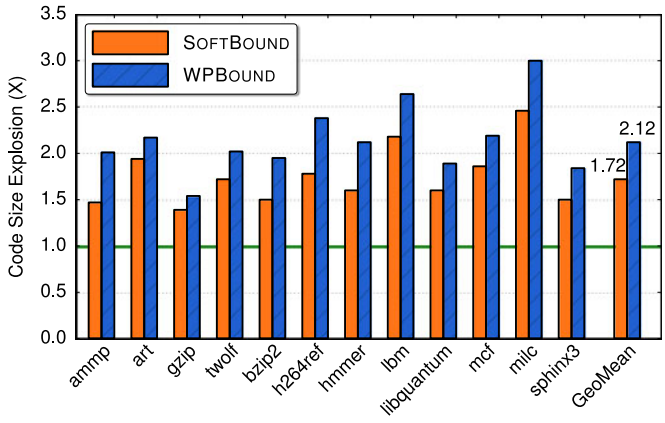


Fig. 8. Compilation workflow.

ysis) involve only constants, globals or formal parameters of F (line 5). In this case, any formal parameter of F that appears in $wpChk(p_{\text{lb}}^{\text{mar}}, p_{\text{ub}}^{\text{mar}}, \ldots)$ can be replaced by its corresponding actual parameter q at the callsite cs in F_{cs} to obtain $wpChk'_{\text{cs}}(q_{\text{lb}}^{\text{mar}}, q_{\text{ub}}^{\text{mar}}, \ldots)$.

After the interprocedural parameter mapping has been performed, the hoisted WP check becomes $r' = wpChk'_{\text{cs}}(q_{\text{lb}}^{\text{mar}}, q_{\text{ub}}^{\text{mar}}, \ldots)$, where r' is a newly created global variable. PLACEWP is called to insert it in the caller F_{cs} (line 9), at either a point outside a loop identified by POSITIONINGWP (line 16) or the point just before callsite cs, represented by "cs" (line 18). All the uses of r for the original WP check (line 4) are replaced by r' (line 10). The old WP check (line 4) in F can be safely removed (line 11).

However, hoisting a WP check from a callee into its multiple calling contexts may cause it to be replicated multiple times. To strike a balance between code size and performance, our INTERWP algorithm can be applied in a demand-driven manner to some user-specified functions, e.g., top few frequently executed functions in a program.

In Fig. 7, we apply our interprocedural optimization to a code fragment extracted from a hot function `FChosse()` in `hmmer`, a real program used in our experimental evaluation. An array pointed by p is read in line 309 in `FChoose()`, which is a frequently executed function in `hmmer` (see Fig. 9). `FChoose()` is called in line 336 from inside a loop contained in another function `SampleCountvector`. Our interprocedural phase is able to hoist the WP check for `p` in `FChoose()` outside the loop in `SampleCountvector` so that the WP check is no longer performed redundantly in `FChoose()`.

IV. EVALUATION

The goal of this evaluation is to demonstrate that our WP-based tool, WPBOUND, can significantly reduce the runtime overhead of SOFTBOUND, a state-of-the-art instrumentation tool for enforcing spatial memory safety of C programs.

A. Implementation Considerations

Based on the open-source code of SOFTBOUND, we have implemented WPBOUND also in LLVM (version 3.3). In both cases, the bounds metadata are maintained in a separate shadow space. Like SOFTBOUND, WPBOUND handles a number of issues identically as follows. Array indexing (also for multiple-dimensional arrays) is handled equivalently as pointer arithmetic. The metadata for global pointers are initialized, by using the same hooks that C++ uses for constructing global objects. For external function uses in uninstrumented libraries, we resort to SOFTBOUND 's library function wrappers (see Fig. 8), which enforce the spatial safety and summarize the side effects on the metadata. For a function pointer, its bound equals to its base, describing a zero-sized object that is not used by data objects. This prevents data pointers or nonpointer data from being interpreted as function pointers. For pointer type conversions via either explicit casts or implicit unions, the bounds information simply propagates across due to the disjoint metadata space used. Finally, we do not yet enforce the spatial safety for variable argument functions.

B. Experimental Setup

All experiments are conducted on a machine equipped with a 3.00 GHz quad-core Intel Core2 Extreme X9650 CPU and 8GB DDR2 RAM, running on a 64-bit Ubuntu 10.10. The SOFTBOUND tool is taken from the *SoftBoundCETS* open-source project (version 1.3) [36], [37], configured to enforce spatial memory safety only.

Table I lists a set of 20 benchmarks, including 14 SPEC benchmarks and six MediaBench benchmarks, used in our evaluation. We have selected 10 from the 12 C benchmarks in the SPEC2006 suite, by excluding `gcc` and `perlbench` since both cannot be processed correctly under SOFTBOUND (as described in its README). In addition to SPEC2006, we have included four loop-oriented SPEC2000 benchmarks, `ammp`, `art`, `gzip` and `twolf`, in our evaluation. These benchmarks are frequently used in the literature on detecting spatial errors [1], [23], [35], [36], [47]. To broaden the scope of our evaluation, we have also selected six loop-oriented and array-intensive

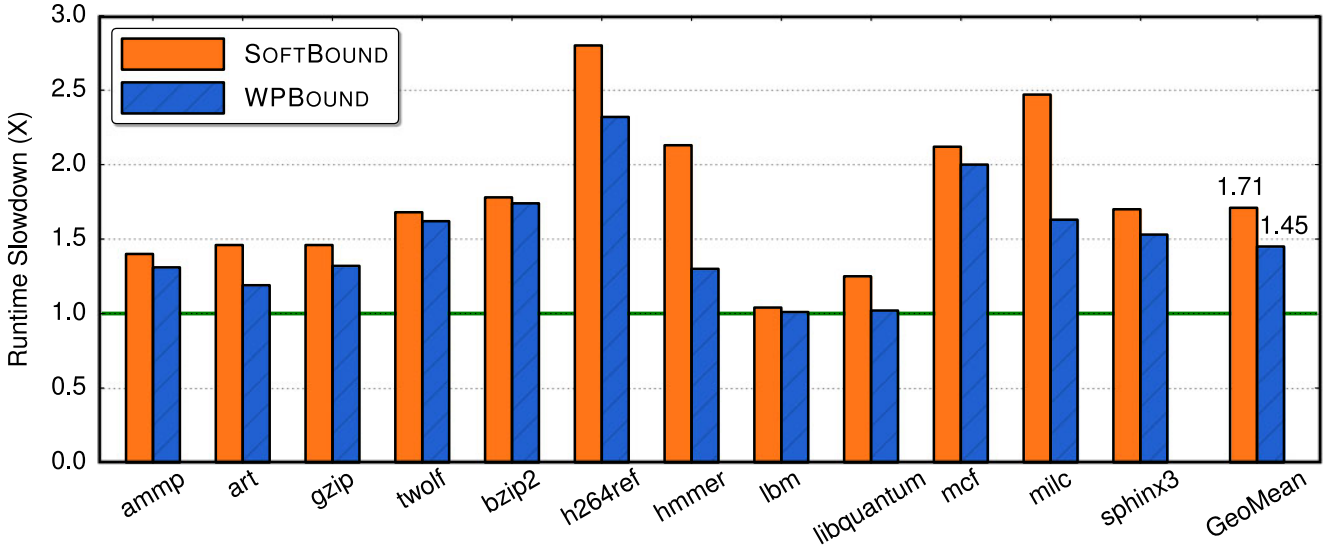


Fig. 9. Top three most frequently executed functions in a program (with their execution times given as percentages of the program's total execution time).

TABLE I
BENCHMARK STATISTICS (IN ABSOLUTE TERMS)

						WP-Based Instrumentation					
Benchmark	#Functions	#Loads	#Stores	#Loops	$\#SC_{SB}$	$\#wp_a$	$\#wp_c$	$\#wp_l$	$\#wp_i$	$\overline{\lvert wp_l \rvert}$	$\max_{\lvert wp_l \rvert}$
ammp	180	3705	1187	650	3962	516	2673	150	2	4.2	54
art	27	471	182	158	461	84	34	46	0	2.0	6
bzip2	68	2570	1680	545	2414	324	1114	116	1	3.2	59
djpeg	339	6434	3715	1109	8339	1232	4936	330	9	4	81
h264ref	517	20 984	8277	2698	25 626	3820	10 668	743	9	5.9	235
hmmer	472	8345	3608	1667	8644	1586	3434	502	2	3.7	48
gobmk	2477	16 598	4458	2470	15 875	2079	6280	606	0	3.8	38
gsmencode	60	775	434	99	961	102	672	34	6	3.2	40
gsmdecode	60	776	434	99	960	102	672	34	8	3.2	40
gzip	72	936	711	257	1096	83	118	56	1	1.5	7
lbm	18	244	114	32	319	278	282	10	0	27.8	76
libquantum	96	604	317	144	572	140	358	34	0	4.1	35
mcf	26	347	224	76	472	37	216	13	0	2.6	9
milc	236	3443	1094	544	3266	571	1556	97	19	7.7	49
mpeg2enc	71	1753	507	294	1628	366	367	74	24	5.1	108
mpeg2dec	58	1204	565	203	1215	210	323	54	18	4.5	39
mesamipmap	931	12 474	7748	1755	18 432	2620	13 073	642	7	4.9	87
sjeng	133	3318	1950	430	3618	170	1208	49	0	3.5	30
sphinx3	320	4628	1359	1240	4260	654	1735	343	14	2.2	41
twolf	188	9781	3304	1253	9328	532	2683	195	0	2.8	32
Mean	317	4970	2093	786	5572	775	2620	206	6	5.0	56

$\#SC_{SB}$ denotes the number of spatial checks inserted by SoftBound. $\#wp_a$ is the number of `wpChk` calls inserted (i.e., the number of wp_p in line 5 of Algorithm 1). $\#wp_c$ represents the number of unconditional checks reduced by WP consolidation. $\#wp_l$ is the number of merged WPs by loop unswitching (i.e., the number of non-empty S at line 3 of Algorithm 3). $\#wp_i$ is the total number of hoisted WP checks from the top three hot functions in a program (see Fig. 9) by applying our WP-based inter-procedural optimization. $\overline{\lvert wp_l \rvert}$ and $\max_{\lvert wp_l \rvert}$, respectively, stand for the average and maximum numbers of the WPs used to build a disjunction (i.e., the average and maximum sizes of nonempty S at line 3 of Algorithm 3).

embedded benchmarks from Mediabench, `djpeg`, which are `gsmencode`, `gsmdecode`, `mesamipmap`, `mpeg2enc`, and `mpeg2dec`, covering a number of application domains including image processing, speech processing, video compression, and 3-D graphics.

C. Methodology

Fig. 8 shows the compilation workflow for both SoftBound and WPBound in our experiments. All source files of a program are compiled under the "-O2" flag and then merged into one bitcode file using LLVM-link. The instrumentation code is inserted into the merged bitcode file by a SoftBound or WPBound pass. Then the bitcode file with instrumentation code is linked to the SoftBound runtime library to generate binary code, with the link-time optimization flag "-O2" used to further optimize the instrumentation code inserted.

To analyze the runtime overheads introduced by both tools, the native (instrumentation-free) code is also generated under the "-O2" together with link-time optimization.

To maintain a good balance between code size and performance, our WP-based interprocedural optimization (given in Algorithm 4) is designed to be demand driven. In our experiments, we have applied it only to the top three most frequently executed functions in a program. Such hot functions are identified by profiling using gprof (with the "-pg" option turned on when each C file is compiled). In practice, the number of hot functions selected in a program can be determined as a user-tunable parameter to allow a tradeoff to be made between performance gains and code size increases.

D. Instrumentation Results

Let us first discuss the instrumentation results of the 20 benchmarks according to the statistics given in Table I.

In Column 6, we see that SoftBound inserts an average of 5572 spatial checks for each benchmark. Note that the number of spatial checks inserted is always smaller than the number of loads and stores added together. This is because SoftBound has eliminated some unnecessary spatial checks by applying some simple optimizations including its dominator-based redundant check elimination [36] (with its "BOUNDSCHECKOPT" option turned on). This set of optimizations is also performed by WPBound as well.

In Columns 7–12, we can observe some results collected for WPBound. According to Column 7, there is an average of 775 `wpChk` calls inserted in each benchmark by Algorithm 1 (for WP-based instrumentation), causing slightly over 1/7 of the spatial checks that are inserted by SoftBound to be guarded. According to Column 8, Algorithm 2 (for WP consolidation) has made an average of 2620 unconditional checks guarded (with respect to an average of 5572 spatial checks) for each benchmark. According to Column 9, Algorithm 3 (for loop unswitching) has succeeded in merging an average of 206 WPs at loop entries in each benchmark.

According to Column 10, our WP-based interprocedural optimization (Algorithm 4) has successfully hoisted a total of 120 WP checks in 13 out of 20 benchmarks out of their top three hot functions, which are listed in Fig. 9. For these 13 benchmarks, there are relatively more WP checks hoisted in `mpeg2enc`, `milc`, `mpeg2dec`, `sphinx3`, `djpeg`, and `h264ref` than in `gsmdecode`, `mesamipmap`, `gsmencode`, `ammp`, `hmmer`, `gzip`, and `bzip2`. For the remaining seven benchmarks, their hot functions do not benefit from our interprocedural optimization for various reasons. In `twolf` and `lbm`, the arrays in their hot functions are accessed via local variables. In `mcf`, its hot functions are involved in recursion. In these three benchmarks, no WP check can be successfully hoisted out of their hot functions. In `gobmk` and `sjeng`, there are fewer array operations in loops. In `art` and `libquantum`, some WP checks can be hoisted out of their hot functions but the hoisted WP checks reside in a loop such that they are loop-variant. So the overall effect is that no WP check can be further optimized.

Overall, the average number of the WPs combined to yield one disjunctive WP is 5.0 (Column 11), peaking at 235 constituent WPs in one disjunctive WP in the `Mode_Decision_for_4x4IntraBlocks` function in `h264ref` (Column 12).

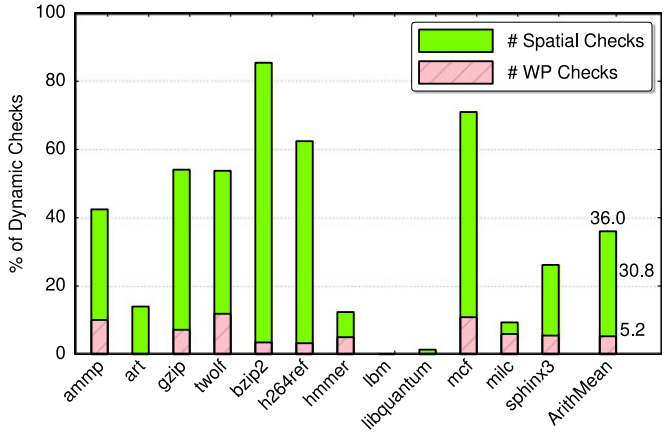


Fig. 10. Bitcode file sizes after instrumentation (normalized with respect to native code).

By performing WP-based compiler optimization, WPBound results in slightly larger code sizes than SoftBound, as compared in Fig. 10. This happens due to (1) the `wpChk` calls introduced, (2) the guards added to some spatial checks, (3) code duplication caused by loop unswitching, and (4) context-sensitive replication of WP checks by interprocedural WP hoisting. Compared to uninstrumented native code, the geometric means of code size increases for SoftBound and WPBound are $1.54\times$ and $1.87\times$, respectively. This implies that WPBound has made an instrumented program about 21.4% larger than SoftBound on average. In general, the code explosion problem is well contained due to the partitioning heuristics used in our WP-based loop unswitching as discussed in Section III-E3 and the demand-driven interprocedural optimization introduced in Section III-F. Note that there is no obvious correlation between bitcode file sizes and performance slowdowns (shown in Fig. 11 and discussed below). For example, some bounds checks are hoisted from a loop to prevent them from being repeatedly performed inside. However, such instrumented instructions may be infrequently or never executed at runtime (under certain inputs).

Finally, the analysis times spent by WPBound at compile time are all small, with 20.01 s for `gobmk` and less than 5 s for each of the remaining 19 benchmarks.

E. Performance Results

To understand the effects of our WP-based approach on performance, we compare WPBound and SoftBound in terms of their runtime overheads in terms of execution time, dynamic number of bounds checks performed, dynamic number of instructions performed, and memory consumption.

1) Execution Times: Fig. 11 compares WPBound and SoftBound in terms of their runtime slowdowns over the native code (as the uninstrumented baseline). The average overhead of a tool is measured as the geometric mean of overhead of all benchmarks analyzed by the tool.

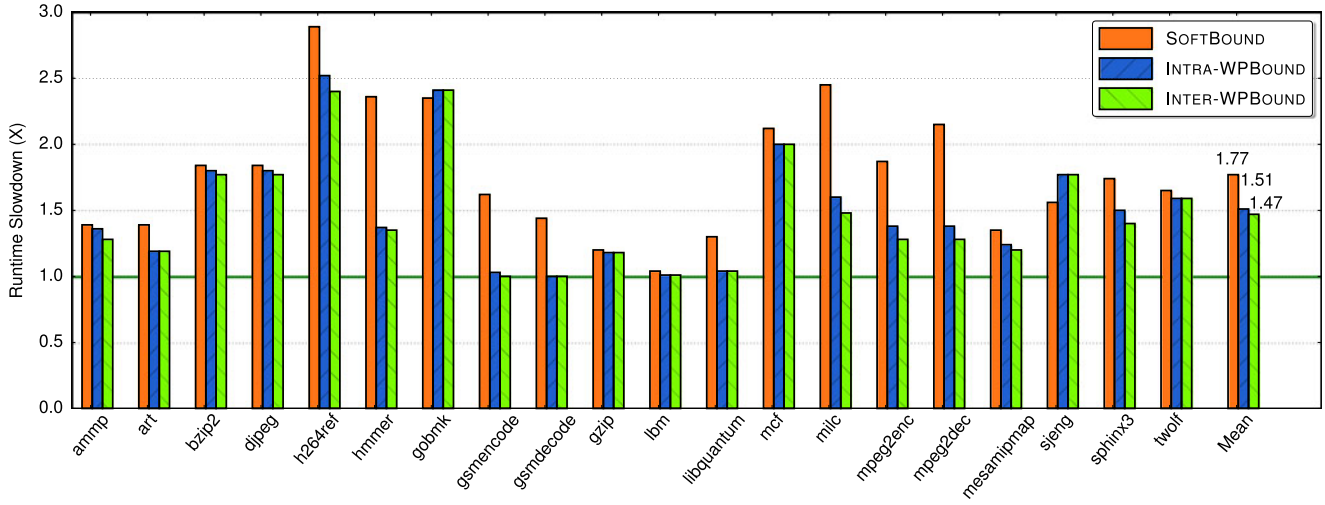


Fig. 11. Execution time (normalized with respect to native code).

SoftBound exhibits an average overhead of 77%, reaching 180% at `h264ref`. By using our intraprocedural WP-based instrumentations alone, WPBound$^{\text{Intra}}$ has reduced SoftBound's average overhead from 77% to 51%, with the top three largest reductions achieved at `hmmer` (41.7%), `gsmencode` (36.1%) and `mpeg2dec` (35.7%). For `lbm`, which is the best case for both tools, SoftBound and WPBound $^{\text{Intra}}$ suffer from only 3.7% and 0.9% overheads, respectively. In this benchmark, the pointer load and store operations that are costly for in-memory metadata propagations [as shown in Fig. 2(e)] are relatively scarce. In addition, SoftBound 's simple dominator-based redundant check elimination identifies 60% of the checks as unnecessary.

When our WP-based interprocedural optimization is also enabled, WPBound becomes more efficient than WPBound $^{\text{Intra}}$. The average runtime overhead has dropped further from 51% to 47%. We can observe relatively large overhead reductions achieved in 6 out of 20 benchmarks, 1) ammp (from 36% to 28%), 2) h264ref (from 52% to 40%), 3) milc (from 60% to 48%), 4) mpeg2enc (from 38% to 28%), 5) mpeg2dec (from 38% to 29%) and 6) sphinx3 (from 50% to 40%). Although some WP checks have also been hoisted interprocedurally in `bzip2`, `h264ref`, and `gzip` (see Table I), their improvements are negligible. There are two main reasons behind. First, the hoisted checks originate from a hot function that takes only a small percentage of the total execution time in its containing program (e.g., 6.46% for `sendMTFValues` in `bzip2`). Second, some WP checks in a hot function are hoisted only to its immediate callers but not further up along its call chains (as in `h264ref` and `gzip`).

For `gobmk` and `sjeng`, WPBound runs slightly more slowly than SoftBound. There are three reasons behind. First, many array operations (e.g., in `gobmk`) reside in loops that have loop-variant bounds, which prevent subsequent WP-based optimizations such as WP consolidation, WP-driven loop unswitching, and interprocedural WP hoisting from being applied. Second, the SCEV analysis is imprecise in these two benchmarks, causing the WP checks for some loops to be quite conservative so that some redundant bounds checks are still performed inside these loops. Finally, SoftBound has already eliminated a substantial number of redundant checks in these two benchmarks by using its dominator-tree based optimization [38] (in the default setting). The opportunities for eliminating more bounds checks are relatively small.

2) Dynamic Check Count Reduction: Fig. 12 shows the ratios of the dynamic number of checks, i.e., calls to `wpChk` and `sChk` executed under WPBound over the dynamic number of checks, i.e., calls to `sChk` executed under SoftBound (in percentage terms). On average, WPBound performs only 35.8% of SoftBound's checks, comprising 30.2% `sChk` calls and 5.6% `wpChk` calls. For every benchmark considered, the number of checks performed by WPBound is always lower than that performed by SoftBound. This confirms that the WPs constructed by WPBound for real code typically evaluate to true, causing their guarded checks to be avoided.

By comparing Figs. 11 and 12, we observe that WPBound is usually effective in reducing bounds checking overheads in programs where it is also effective in reducing the dynamic number of checks performed by SoftBound. This has been the case for benchmarks such as `hmmer`, `gsmencode`, and `mpeg2dec`. As for `bzip2`, WPBound still preserves 85% of SoftBound's checks, thereby reducing its overhead from 78% to 73% only.

We also observe that a certain percentage reduction in the dynamic number of checks achieved by WPBound does not translate into execution time benefits at the same magnitude. On average, WPBound has reduced SoftBound's dynamic check count by 64.2% but its overhead by 39% only. There are two reasons. First, a `wpChk` call is more expensive than an `sChk` call since the first two arguments in the former case specifying a MAR can involve complex expressions. Second, WPBound is not designed to improve metadata propagation, which can be another major source of overheads.

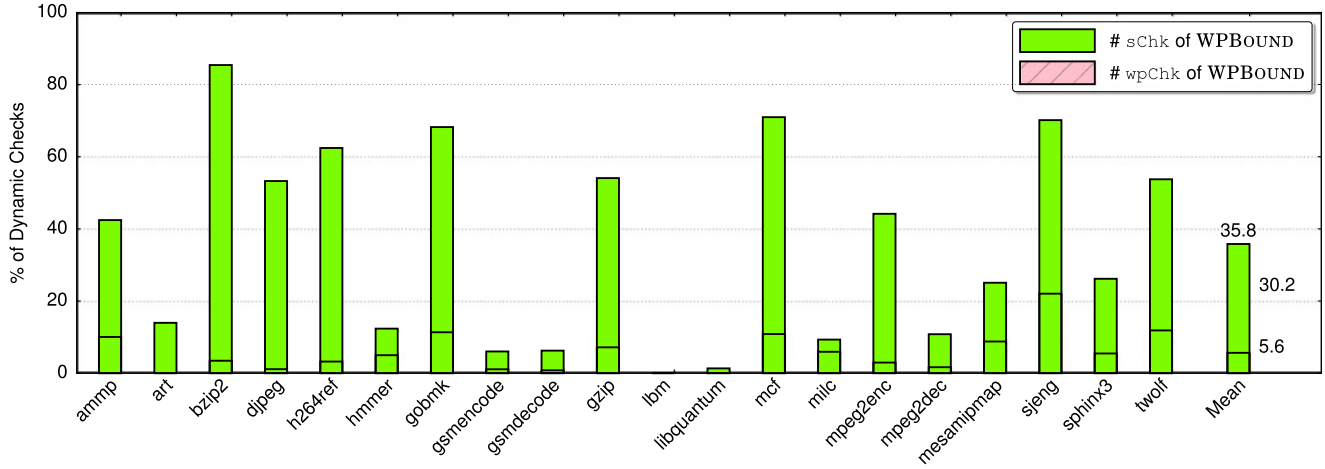


Fig. 12. Percentage of dynamic number of checks performed by WPBOUND over SOFTBOUND at runtime.

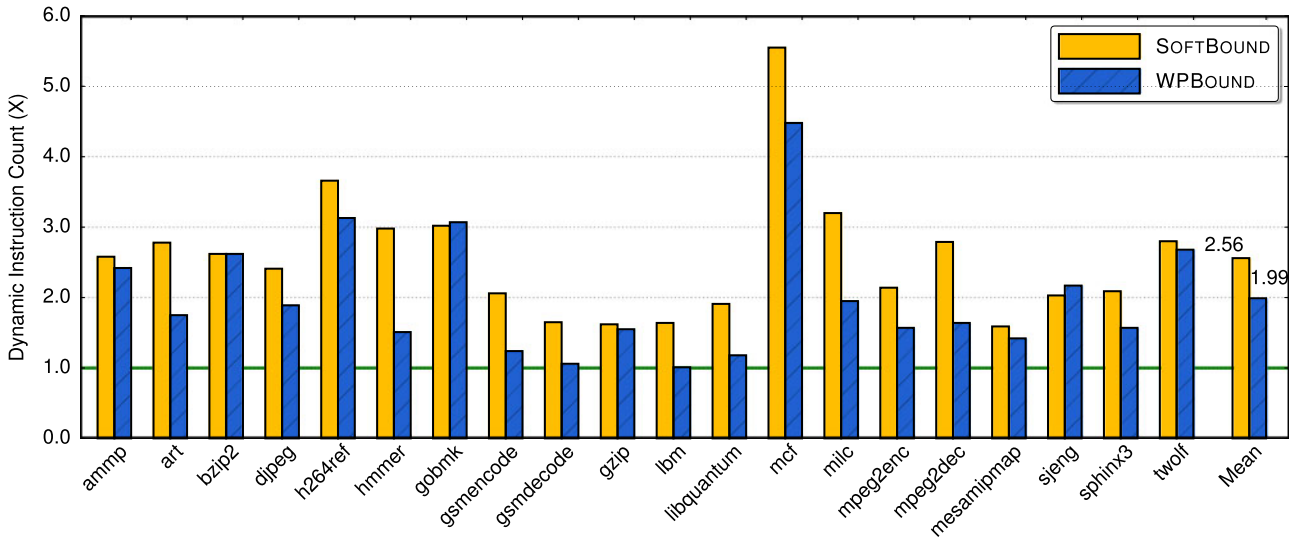


Fig. 13. Dynamic instruction counts (normalized with respect to native code).

3) Dynamic Instruction Count: Fig. 13 compares WPBOUND and SOFTBOUND in terms of the dynamic number of instructions executed (normalized with respect to native code). We used the perf tool, which rely on hardware performance counters, to measure dynamic instruction count. On average, SOFTBOUND results in 2.56× instructions executed per benchmark. By performing our WP-based compiler optimization, WPBOUND has reduced 2.56× to 1.99×, by avoiding many redundant bounds checks performed inside loops. Note that the results presented in Fig. 13 correlate reasonably well with those presented in Fig. 12. In the case of `lbm`, WPBOUND has achieved a reduction of 38.4% with respect to the number of instructions executed by SOFTBOUND but is only marginally faster (as explained earlier). According to [36], `lbm` has a high data cache miss rate of one miss every 20 instructions. As a result, the resulting low IPC provides plenty of spare execution capacity to hide SoftBound overheads.

4) Memory Consumption: Fig. 14 shows the peak memory usage of WPBOUND and SOFTBOUND at runtime (normalized over native code). As WPBOUND is designed to eliminate redundant runtime bounds checks performed by a spatial-error detection tool such as SOFTBOUND while maintaining the same metadata information, WPBOUND (1.57×) consumes almost the same amount of memory as SOFTBOUND (1.56×).

F. Discussion

We discuss two principal directions along which WPBOUND can be further improved. One possible avenue for future research is to develop sophisticated range analysis techniques to improve the precision of WP checks. In practice, the most of the source code in a program are free of buffer overflow bugs. The more precise the range analysis is, the more precise the WPs will be. With precise WPs in place, many redundant bounds checks inside loops can be avoided. In our current implementation, the

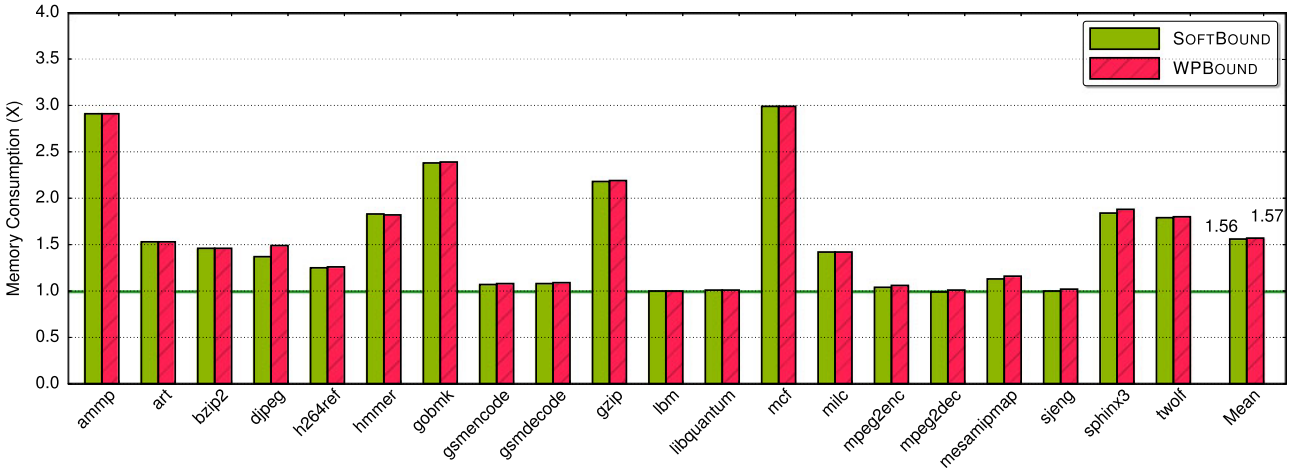


Fig. 14. Memory consumption (normalized with respect to native code).

range analysis based on SCEV expressions is often conservative in the sense that the estimated ranges of values are sometimes crude overapproximations of their actual runtime ranges. In this case, some spatial checks inside a loop may still be performed unnecessarily. To sharpen the precision of WP checks, more advanced SCEV analysis techniques (e.g., by considering interprocedural path sensitivity) are needed.

Another possible research direction is to employ some sophisticated loop-oriented analysis and transformation techniques to increase the number of WP checks performed. Our WP-based approach is loop-oriented. WPBound may achieve some small performance improvements over SoftBound on some programs that are neither loop oriented nor array intensive. In these cases, relatively few bounds checks can be hoisted outside their containing loops. Furthermore, WPBound may not be effective even for some loop-oriented programs if the memory access regions (MARs) computed for some arrays by our range analysis are loop-variant (preventing the corresponding bounds checks from being hoisted outside). In this case, the effectiveness WPBound can be potentially improved if some advanced loop analysis and transformation frameworks such as Polly [20] are used.

Finally, WPBound can be applied directly to enforce spatial safety for systems programming languages such as C, C++, and Objective-C. For managed languages, such as Java and C#, array bounds are checked automatically by their underlying virtual machines at runtime. The WP-based idea can also be used to eliminate bounds checks when generating byte code in JIT compilers, such as Java HotSpot VM [57].

V. Related Work

In addition to the pointer-based approaches described in Section II, we now review guard zone-based and object-based approaches for enforcing spatial safety and discuss some other related work on bounds check elimination and static analysis.

A. Guard Zone-Based Spatial Safety

Guard zone-based approaches [23], [24], [41], [47], [64] enforce spatial safety by placing a guard zone of invalid memory between memory objects. Continuous overflows caused by walking across a memory object's boundary in small strides will hit a guard zone, resulting in an out-of-bounds error. In the case of overflows with a large stride that jumps over a guard zone and falls into another memory object, an out-of-bounds error will be missed. As a result, these approaches provide neither source compatibility nor complete spatial safety.

B. Object-Based Spatial Safety

In object-based approaches [1], [8], [10], [13], [26], [46], the bounds information is maintained per object (rather than per-pointer as in pointer-based approaches). In addition, the bounds information of an object is associated with the location of the object in memory. As a result, all pointers to an object share the same bounds information. On every pointer-related operation, a spatial check is performed to ensure that the memory access is within the bounds of the same object.

Compared to pointer-based approaches, object-based approaches usually have better compatibility with un-instrumented libraries. The metadata associated with heap objects are properly updated by interpreting `malloc` and `free` function calls, even if the objects are allocated or deallocated by uninstrumented code. Unlike pointer-based approaches, however, object-based approaches do not provide complete spatial safety, since sub-object overflows (e.g., overflows of accesses to arrays inside structs) are missed.

Note that our WP-based optimization can be applied to guard zone- and object-based approaches, although we have demonstrated its effectiveness in the context of a pointer-based approach, which has recently been embraced by Intel in a recently released commercial software product [17].

C. Bounds Check Elimination

Bounds check elimination, which reduces the runtime overhead incurred in checking out-of-bounds array accesses for Java, has been extensively studied in the literature [5], [14], [15], [33], [42], [44], [57], [58]. One common approach relies on solving a set of constraints formulated based on the program code [5], [15], [42], [44]. Another is to speculatively assume that

some checks are unnecessary and generate check-free specialized code, with execution reverted to unoptimized code when the assumption fails [14], [57], [58].

Loops in the program are also a target for bounds check elimination [33]. Some simple patterns can be identified, where unnecessary bound checks can be safely removed.

SoftBound [36] applies simple compile-time optimizations including a dominator-based redundant check elimination to eliminate unnecessary checks dominated by other checks.

In our earlier work [61], WPBound applies only intraprocedural optimizations to accelerate runtime detection of spatial memory errors. This paper extends that to include also interprocedural WP hoisting to reduce redundant bounds checks further. Our WP-based compiler approach complements prior work by making certain spatial checks guarded so that a large number of spatial checks are avoided conditionally.

D. Static Analysis

A significant body of work exists on statically detecting and diagnosing buffer overflows [3], [6], [11], [12], [16], [19], [21], [28], [29], [32], [45], [56], [59]. Due to its approximation nature, static analysis alone finds it rather difficult to maintain both precision and efficiency, and generally has either false positives or false negatives. However, its precision can be improved by using modern pointer analysis [22], [27], [48], [49], [51], [63], [65] and value-flow analysis [30], [31], [52]–[54] techniques. Recently, static value-flow analysis has been combined with dynamic analysis to reduce instrumentation overheads in detecting uninitialised variables [62]. So existing static analysis techniques can be exploited to compute WPs more precisely for our WP-based instrumentation.

In addition, the efficiency of static analysis techniques can be improved if they are tailored to specific clients. Dillig *et al.* [11] have recently proposed a static analysis to compute the preconditions for dictating spatial memory safety conservatively. Rather than analyzing the entire program, their static analysis works in a demand-driven manner, where the programmer first specifies a code snippet as a query and then the proposed static analysis infers a guard to ensure spatial memory safety for the code snippet. Such analysis uses logical abduction and is thus capable of computing the weakest and simplest guards. In contrast, our work is based on the symbolic analysis of LLVM's scalar evolution and thus more lightweight as an optimization for whole-program spatial-error detection.

VI. Conclusion

In this paper, we introduce a new WP-based compiler approach comprising both intra and interprocedural optimizations to enforce spatial memory safety for C. Our approach complements existing bounds checking optimizations and can be applied to any spatial-error detection approaches. Implemented on top of SoftBound, a state-of-the-art tool for detecting spatial errors, our instrumentation tool, WPBound, provides compatible, comprehensive and efficient spatial safety. For a set of C benchmarks evaluated, WPBound, can substantially reduce the runtime overheads incurred by SoftBound with small code size increases. In future work, we will develop techniques to enforce effectively spatial safety for C++ and multithreaded code bases [50].

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments.

References

[1] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *Proc. USENIX Security Symp.*, 2009, pp. 51–66.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 1994, pp. 290–301.

[3] D. Babic and A. J. Hu, "Calysto: Scalable and precise extended static checking," in *Proc. Int. Conf. Softw. Eng.*, 2008, pp. 211–220.

[4] O. Bachmann, P. S. Wang, and E. V. Zima, "Chains of recurrences—A method to expedite the evaluation of closed-form functions," in *Proc. Int. Symp. Symbolic Algebraic Comput.*, 1994, pp. 242–249.

[5] R. Bodík, R. Gupta, and V. Sarkar, "ABCD: Eliminating array bounds checks on demand," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2000, pp. 321–333.

[6] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. Proc. USENIX Symp. Oper. Syst. Design Implement. ,2008*, vol. 8, pp. 209–224.

[7] I. Corporation, "Intel architecture instruction set extensions programming reference, 319433–015 edition," Jul. 2013. [Online]. Available: http://software.intel.com/sites/default/files/ 319433–015.pdf

[8] J. Criswell, A. Lenharth, D. Dhurjati, and V. S. Adve, "Secure virtual architecture: A safe execution environment for commodity operating systems," in *Proc. ACM Symp. Operating Syst. Principles*, 2007, pp. 351–366.

[9] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: Architectural support for spatial safety of the C programming language," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2008, pp. 103–114.

[10] D. Dhurjati and V. S. Adve, "Backwards-compatible array bounds checking for C with very low overhead," in *Proc. Int. Conf. Softw. Eng.*, 2006, pp. 162–171.

[11] T. Dillig, I. Dillig, and S. Chaudhuri, "Optimal guard synthesis for memory safety," in *Proc. Int. Conf. Comput. Aided Verification*, 2014, pp. 491–507.

[12] N. Dor, M. Rodeh, and M. Sagiv, "CSSV: Towards a realistic tool for statically detecting all buffer overflows in C," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2003, pp. 155–167.

[13] F. C. Eigler, "Mudflap: Pointer use checking for C/C++," in *Proc. GCC Developers Summit*, 2003, pp. 57–69.

[14] A. Gampe, J. von Ronne, D. Niedzielski, and K. Psarris, "Speculative improvements to verifiable bounds check elimination," in *Proc. 6th Int. Symp. Principles Practice Program. Java*, 2008, pp. 85–94.

[15] A. Gampe, J. von Ronne, D. Niedzielski, J. Vasek, and K. Psarris, "Safe, multiphase bounds check elimination in Java," *Softw., Pract. Exper.*, vol. 41, no. 7, pp. 753–788, 2011.

[16] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek, "Buffer overrun detection using linear programming and static analysis," in *Proc. Conf. Comput. Commun. Security*, 2003, pp. 345–354.

[17] K. Ganesh, "Pointer checker: Easily catch out-of-bounds memory accesses. Intel Corporation. 2012. [Online]. Available: http://software.intel.com/sites/products/parallelmag/singlearticles/issue11/7080_2_IN_ParallelMag_Issue11_Pointer_Checker.pdf

[18] S. Ghose, L. Gilgeous, P. Dudnik, A. Aggarwal, and C. Waxman, "Architectural support for low overhead detection of memory violations," in *Proc. Des., Autom. Test Eur. Conf. Exhib.*, 2009, pp. 652–657.

[19] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2008, vol. 8, pp. 151–166.

[20] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly-polyhedral optimization in LLVM," in *Proc. 1st Int. Workshop Polyhedral Compilation Techn.*, vol. 2011, 2011, pp. 1–6.

[21] B. Hackett, M. Das, D. Wang, and Z. Yang, "Modular checking for buffer overflows in the large," in *Proc. Int. Conf. Softw. Eng.*, 2006, pp. 232–241.

[22] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2011, pp. 289–298.

[23] N. Hasabnis, A. Misra, and R. Sekar, "Light-weight bounds checking," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2012, pp. 135–144.

[24] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *Proc. Winter USENIX Conf.*, 1991, pp. 125–138.

[25] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *Proc. USENIX Annu. Tech. Conf., General Track*, 2002, pp. 275–288.

[26] R. W. Jones and P. H. Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs," in *Proc. Automat. Algorithmic Debugging*, 1997, pp. 13–26.

[27] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2013, pp. 423–434.

[28] W. Le and M. L. Soffa, "Marple: A demand-driven path-sensitive buffer overflow detector," in *Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2008, pp. 272–282.

[29] L. Li, C. Cifuentes, and N. Keynes, "Practical and effective symbolic analysis for buffer overflow detection," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 317–326.

[30] L. Li, C. Cifuentes, and N. Keynes, "Boosting the performance of flow-sensitive points-to analysis using value flow," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2011, pp. 343–353.

[31] L. Li, C. Cifuentes, and N. Keynes, "Precise and scalable context-sensitive pointer analysis via value flow graph," in *Proc. Int. Symp. Memory Manage.*, 2013, pp. 85–96.

[32] A. Mine, D. Monniauxli, and X. Rival, "The ASTREE analyzer," in *Proc. Eur. Symp. Program.*, 2005, pp. 21–30.

[33] J. E. Moreira, S. P. Midkiff, and M. Gupta, "From Flop to Megaflops: Java for technical computing," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 2, pp. 265–295, Mar. 2000.

[34] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *Proc. Annu. Int. Symp. Comput. Archit.*, 2012, pp. 189–200.

[35] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdog-Lite: Hardware-accelerated compiler-based pointer checking," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2014, pp. 175–184.

[36] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2009, pp. 245–258.

[37] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "CETS: Compiler enforced temporal safety for C," in *Proc. Int. Symp. Memory Manage.*, 2010, pp. 31–40.

[38] S. G. Nagarakatte, "Practical low-overhead enforcement of memory safety for C programs," Ph.D. dissertation, Univ. Massachusetts Amherst, Amherst, MA, USA, 2012.

[39] "National vulnerability database," [Online]. Availanle: http://nvd.nist.gov/

[40] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, pp. 477–526, 2005.

[41] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2007, pp. 89–100.

[42] D. Niedzielski, J. Ronne, A. Gampe, and K. Psarris, "A verifiable, control flow aware constraint analyzer for bounds check elimination," in *Proc. 16th Int. Symp. Static Anal.*, 2009, pp. 137–153.

[43] H. Patil and C. N. Fischer, "Low-cost, concurrent checking of pointer and array accesses in C programs," *Softw.—Pract. Exp.*, vol. 27, no. 1, pp. 87–110, 1997.

[44] F. Qian, L. J. Hendren, and C. Verbrugge, "A comprehensive approach to array bounds check elimination for Java," in *Proc. Compiler Construction*, 2002, pp. 325–342.

[45] R. Rugina and M. C. Rinard, "Symbolic bounds analysis of pointers, array indices, and accessed memory regions," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2000, pp. 182–195.

[46] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2004, pp. 159–169.

[47] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. USENIX Conf. Annu. Tech. Conf., 2012*, vol. 2012, pp. 309–318.

[48] L. Shang, Y. Lu, and J. Xue, "Fast and precise points-to analysis with incremental CFL-reachability summarisation: Preliminary experience," in *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2012, pp. 270–273.

[49] L. Shang, X. Xie, and J. Xue, "On-demand dynamic summary-based points-to analysis," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2012, pp. 264–274.

[50] Y. Sui, P. Di, and J. Xue, "Sparse flow-sensitive pointer analysis for multithreaded C programs," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2016, pp. 160–170.

[51] Y. Sui, Y. Li, and J. Xue, "Query-directed adaptive heap cloning for optimizing compilers," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2013, pp. 1–11.

[52] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proc. Int. Symp. Softw. Testing Anal.*, 2012, pp. 254–264.

[53] Y. Sui, D. Ye, and J. Xue, "Detecting memory leaks statically with full-sparse value-flow analysis," *IEEE Trans. Softw. Eng.*, vol. 40, no. 2, pp. 107–122, Feb. 2014.

[54] Y. Sui, S. Ye, J. Xue, and P.-C. Yew, "SPAS: Scalable path-sensitive pointer analysis on full-sparse SSA," in *Proc. 9th Asian Conf. Program. Lang. Syst.*, 2011, pp. 155–171.

[55] R. van Engelen, "Efficient symbolic analysis for optimizing compilers," in *Proc. Compiler Construction*, 2001, pp. 118–132.

[56] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2000, pp. 1–15.

[57] T. Würthinger, C. Wimmer, and H. Mössenböck, "Array bounds check elimination for the Java HotSpot client compiler," in *Proc. 5th Int. Symp. Principles Practice Program. Java*, 2007, pp. 125–133.

[58] T. Würthinger, C. Wimmer, and H. Mössenböck, "Array bounds check elimination in the context of deoptimization," *Sci. Comput. Program.*, vol. 74, no. 5/6, pp. 279–295, Mar. 2009.

[59] Y. Xie, A. Chou, and D. R. Engler, "ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2003, pp. 327–336.

[60] W. Xu, D. C. DuVarney, and R. Sekar, "An efficient and backwards-compatible transformation to ensure memory safety of C programs," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2004, pp. 117–126.

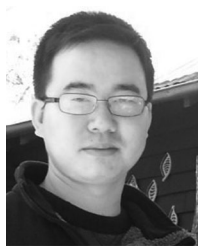
[61] D. Ye, Y. Su, Y. Sui, and J. Xue, "Wpbound: Enforcing spatial memory safety efficiently at runtime with weakest preconditions," in *Proc. 25th IEEE Int. Symp. Softw. Rel. Eng.*, 2014, pp. 88–99.

[62] D. Ye, Y. Sui, and J. Xue, "Accelerating dynamic detection of uses of undefined values with static value-flow analysis," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2014, pp. 154–164.

[63] S. Ye, Y. Sui, and J. Xue, "Region-based selective flow-sensitive pointer analysis," in *Proc. Int. Symp. Static Anal.*, 2014, pp. 319–336.

[64] S. H. Yong and S. Horwitz, "Protecting C programs from attacks via invalid pointer dereferences," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2003, pp. 307–316.

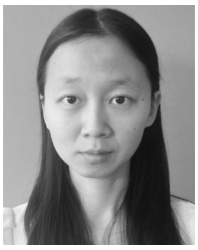
[65] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang, "Level by level: Making flow-and context-sensitive pointer analysis scalable for millions of lines of code," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2010, pp. 218–229.



Yulei Sui received the Bachelor's and Master's degrees in computer science from Northwestern Polytechnical University, Xi'an, China, in 2008 and 2011, and the Ph.D. degree in computer science from the University of New South Wales, Sydney, Australia.

He has been a Postdoctoral Fellow in Programming Languages and Compilers Group, University of New South Wales, since 2014. He is broadly interested in the research field of software engineering and programming languages, particularly interested in static and dynamic program analysis for software bug detection and compiler optimizations. He worked as a Research Intern in Program Analysis Group for Memory Safe C project in Oracle Lab Australia in 2013.

Dr. Sui was an Australian IPRS scholarship holder, a keynote speaker at EuroLLVM, and a Best Paper Award winner at CGO.



Yu Su received the Bachelor's degree in information security and the Master's degree in computer science from the Huazhong University of Science and Technology, Wuhan, China, in 2008 and 2011, respectively, and the Ph.D. degree under the supervision of Prof. J. Xue from University of New South Wales, Australia, in 2015.

Ding Ye received the Bachelor's degree in information security from the Huazhong University of Science and Technology, Wuhan, China, in 2008, and the Master's degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2011.

After receiving the Ph.D. degree from UNSW Australia in 2015, he became a Research Associate and stayed with his supervisor Prof. J. Xue to continue his work. His research interests include program analysis and compiler techniques.

Jingling Xue (SM'01) received the B.Sc. and M.Sc. degrees in computer science and engineering from Tsinghua University, Beijing, China, in 1984 and 1987, respectively, and the Ph.D. degree in computer science and engineering from Edinburgh University, Edinburgh, U.K., in 1992.

He is currently a Professor in the School of Computer Science and Engineering, University of New South Wales, Australia, where he heads the Programming Languages and Compilers Group. His main research interest has been programming languages and compilers for about 20 years. He is currently supervising a group of postdocs and the Ph.D. students on a number of topics including programming and compiler techniques for multi-core processors and embedded systems, concurrent programming models, and program analysis for detecting bugs and security vulnerabilities.

Dr. Xue is presently an Associate Editor of the IEEE TRANSACTIONS ON COMPUTERS, *Software: Practice and Engineering*, the *International Journal of Parallel, Emergent* and *Distributed Systems*, and the *Journal of Computer Science and Technology*. He has served in various capacities on the Program Committees of many conferences in his field.