# WPBOUND: Enforcing Spatial Memory Safety Efficiently at Runtime with Weakest Preconditions

Ding Ye Yu Su Yulei Sui Jingling Xue
School of Computer Science and Engineering
UNSW Australia
{dye, ysu, ysui, jingling}@cse.unsw.edu.au

***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 optimisation 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 optimisation is applicable to any spatial-error detection approach (in software or hardware or both).**

**To evaluate the effectiveness of our optimisation, we take SOFTBOUND, a compile-time tool with an open-source implementation in LLVM, as our baseline. SOFTBOUND adopts a pointer-based checking approach 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 optimisation. For a set of 12 SPEC C benchmarks evaluated, WPBOUND reduces the average runtime overhead of SOFTBOUND from 71% to 45% (by a reduction of 37%), with small code size increases.**

## I. INTRODUCTION

C, together with its OO incarnation C++, is the de facto standard for implementing systems software (e.g., operating systems and language runtimes), embedded software as well as server and client applications. Due to the low-level control provided over memory allocation and layout, software written in such languages makes up the majority of performance-critical code running on most 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), causing program crashes and security vulnerabilities in today's commercial software.

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 Figure 1, with the data taken from the NVD database [37]. In addition, a recent study shows that buffer overflows are the commonest vulnerability in the last quarter century [61]. Furthermore, spatial errors persist today, as demonstrated by a recently reported *Heartbleed* vulnerability in OpenSSL (CVE-2014-0160).

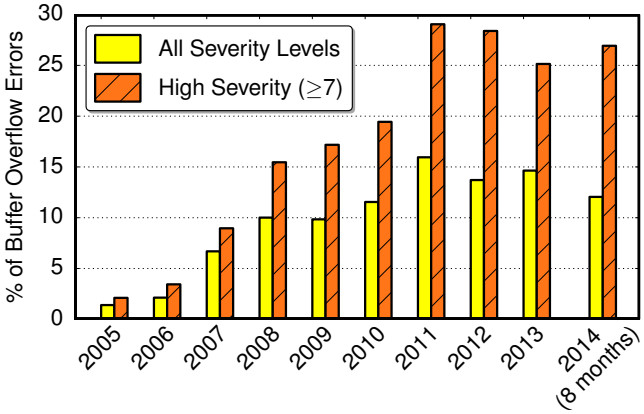


Fig. 1: Reported buffer overflow vulnerabilities in the past decade, listed as CWE-119 in the NVD database [37].

Several approaches exist for detecting and eliminating spatial errors for C/C++ programs at runtime: guard zone-based [22], [23], [39], [45], [60], object-based (by maintaining per-object bounds metadata) [1], [8], [10], [13], [25], [44], pointer-based (by maintaining per-pointer metadata) either inline [2], [24], [38], [41], [57] or in a disjoint shadow space [9], [18], [33], [35]. These approaches can be implemented in software via instrumentation, at source-level as in [13], [35], [45], or binary-level as in [23], [39], accelerated in hardware [9], [33] or by a combination of both [18], [34]. 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 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 if it is done 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 optimisations have been discussed before. For example, a simple dominator-based redundant check elimination [35] enables the compiler to avoid the redundant checks at any dominated memory accesses. As described in [34] and also in the recently announced MPX ISA extensions from Intel [7], new instructions are proposed to be added for accelerating bounds checking (and propagation).

In this paper, we present a new compile-time optimisation that not only complements prior bounds checking optimizations but also applies to any afore-mentioned 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 avoiding the corresponding bounds check 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 completely eliminated.

We demonstrate the effectiveness of our WP-based optimisation by taking SoftBound [35] as the baseline. SoftBound, with an open-source implementation available in LLVM, represents a state-of-the-art compile-time tool for detecting spatial errors. By adopting a pointer-based checking approach 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 [39], SoftBound can reduce MemCheck's overheads significantly as both the original and instrumentation code can be optimised 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 optimisation. WPBound supports separate compilation since its analysis and transformation phases are intraprocedural. Our evaluation shows that WPBound is effective in reducing SoftBound's instrumentation overheads while incurring some small code size increases.

In summary, the contributions of this paper are:

- a WP-based optimisation for reducing bounds checking overheads for C programs;
- a WP-based source-level instrumentation tool, WPBound, for enforcing spatial safety for C programs;
- an implementation of WPBound in LLVM; and
- an evaluation on a set of 12 C programs, showing that WPBound reduces SoftBound's average runtime overhead from 71% to 45% (by a reduction of 37%), with small code size increases.

The rest of this paper is organized as follows. Section II provides the 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

We review briefly how SoftBound [35] works as a pointer-based approach. Section V discusses additional related work on guard zone- and object-based approaches in detail.

Figure 2 illustrates the pointer-based metadata initialisation, propagation and checking abstractly in SoftBound with the instrumentation code highlighted in orange. Instead of maintaining the per-pointer metadata (i.e., base and bound) inline [2], [24], [38], [41], [57], SoftBound uses a disjoint metadata space to achieve source compatibility.

```
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) Memory allocation

```
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) Copying and pointer arithmetic

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

(c) Scalar loads and stores

```
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) Pointer loads and Stores

```
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) Spatial checks

Fig. 2: Pointer-based instrumentation with disjoint metadata.

The bounds metadata are associated with a pointer whenever a pointer is created (Figure 2(a)). The types of base and bound are typically as `char*` so that spatial errors can be detected

at the granularity of bytes. These metadata are propagated on pointer-manipulating operations such as copying and pointer arithmetic (Figure 2(b)).

When pointers are used to access memory, i.e., dereferenced at loads or stores, spatial checks are performed (Figures 2(c) and (d)) by invoking the `sChk` function shown in Figures 2(e). The base and bound of a pointer is available in a disjoint shadow space and can be looked up in a global map `GM`. `GM` can be implemented in various ways, including a hash table or a trie. For each spatial check, five x86 instructions, `cmp`, `br`, `lea`, `cmp` and `br`, are executed on x86, 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 scalar pointer (Figure 2(c)) or a non-scalar pointer (Figure 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 $*p = \cdots$.

## III. METHODOLOGY

WPBOUND, which is implemented in the LLVM compiler infrastructure, consists of one analysis and three transformation phases (as shown in Figure 3). Their functionalities are briefly described below, illustrated by an example in Section III-A, and further explained in Sections III-C and III-D. As its four phases are intraprocedural, WPBOUND provides transparent support for separate compilation.

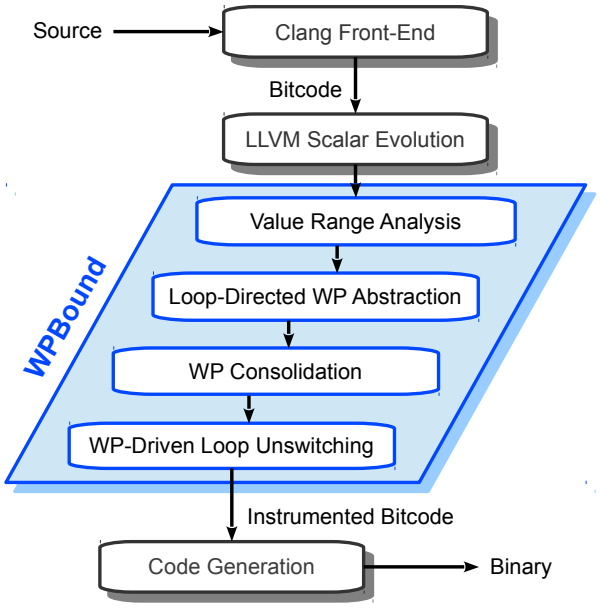


Fig. 3: Overview of the WPBOUND framework.

**Value Range Analysis** This analysis phase computes conservatively the value ranges of pointers dereferenced at loads and stores, leveraging LLVM's *scalar evolution pass*. The value range information is used for the WP computations in the following three transformation phases, where the instrumentation code is generated.

**Loop-Directed WP Abstraction** This phase inserts 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, causing its check to be avoided.

**WP Consolidation** As an optimisation, this phase consolidates the WPs for multiple accesses, which are always made to the same object, into a single one.

**WP-Driven Loop Unswitching** As another optimisation that trades code size for performance, loop unswitching is applied to a loop so that the instrumentation in its frequently executed versions is effectively eliminated.

### A. A Motivating Example

We explain how WPBOUND works with a program in C (rather than in its LLVM low-level code) given in Figure 4. In the program shown in Figure 4(a), there are a total of five memory accesses, four loads (lines 11, 14, 18, and 21) and one store (line 23), 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 orange). To avoid cluttering, we do not show the medadata initialisation and propagation, which are irrelevant to our WP-based optimisation.

**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 have:

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

where the two constants, `S` and `SP`, are defined at the beginning of the program in Figure 4(a), and `L` is the upper bound of the `for` loop in the program.

**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 Figure 4(b), as shown in Figure 4(c). The three spatial check calls to `sChk` at `a[i-1]`, `b[i]` and `a[i]` that are previously unconditional (in SOFTBOUND) are

```
1  #define S sizeof(int)
2  #define SP sizeof(int*)
3  ...
4  int i, k, L;
5  int t1, t2;
6  int *a, *b;
7  int **p;
8  ...
9  if(...) {
10   sChk(p+k, p_bs, p_bd, SP);
11   a = p[k];
12 }
13 sChk(p+k+1, p_bs, p_bd, SP);
14 b = p[k+1];
15 ...
16 for(i = 1; i <= L; i++) {
17   sChk(a+i-1, a_bs, a_bd, S);
18   t1 = a[i-1];
19   if(t < ...) {
20     sChk(b+i, b_bs, b_bd, S);
21     t2 = b[i];
22     sChk(a+i, a_bs, a_bd, S);
23     a[i] += t1 + t2;
24   }
25 }
```

(a) Unoptimized instrumentation

```
1  inline bool wpChk(
2       char *p_lb, char *p_ub,
3       char *p_bs, char *p_bd)
4  {
5    return p_lb < p_bs
6           || p_ub > p_bd;
7  }
```

(b) WP checks

```
1  ...
2  wp_a1 = wpChk(a, a+L, a_bs, a_bd);
3  wp_a2 = wpChk(a+1, a+L+1, a_bs, a_bd);
4  wp_b = wpChk(b+1, b+L+1, b_bs, b_bd);
5  for(i = 1; i <= L; i++) {
6    if(wp_a1) sChk(a+i-1, a_bs, a_bd, S);
7    t = a[i-1];
8    if(t < ...) {
9      if(wp_b) sChk(b+i, b_bs, b_bd, S);
10     t2 = b[i];
11     if(wp_a2) sChk(a+i, a_bs, a_bd, S);
12     a[i] += t1 + t2;
13   }
14 }
```

(c) Loop-directed WP abstraction

```
1  ...
2  cwp_p = wpChk(p+k, p+k+2, p_bs, p_bd);
3  if(...) {
4    if(cwp_p) sChk(p+k, p_bs, p_bd, SP);
5    a = p[k];
6  }
7  if(cwp_p) sChk(p+k+1, p_bs, p_bd, SP);
8  b = p[k+1];
9  ...
10 cwp_a = wpChk(a, a+L+1, a_bs, a_bd);
11 wp_b = wpChk(b+1, b+L+1, b_bs, b_bd);
12 for(i = 1; i <= L; i++) {
13   if(cwp_a) sChk(a+i-1, a_bs, a_bd, S);
14   t = a[i-1];
15   if(t < ...) {
16     if(wp_b) sChk(b+i, b_bs, b_bd, S);
17     t2 = b[i];
18     if(cwp_a) sChk(a+i, a_bs, a_bd, S);
19     a[i] += t1 + t2;
20   }
21 }
```

(d) WP consolidation

```
1  ...
2  cwp_p = wpChk(p+k, p+k+2, p_bs, p_bd);
3  if(...) {
4    if(cwp_p) sChk(p+k, p_bs, p_bd, SP);
5    a = p[k];
6  }
7  if(cwp_p) sChk(p+k+1, p_bs, p_bd, SP);
8  b = p[k+1];
9  ...
10 cwp_a = wpChk(a, a+L+1, a_bs, a_bd);
11 wp_b = wpChk(b+1, b+L+1, b_bs, b_bd);
12 // Merging the two WPs in the loop.
13 wp_loop = cwp_a || wp_b;
14 // Unswitched loop without checks.
15 if (!wp_loop) {
16   for(i = 1; i <= L; i++) {
17     t = a[i-1];
18     if(t < ...) {
19       t2 = b[i];
20       a[i] += t1 + t2;
21     }
22   }
23 }
24 // Unswitched loop with checks.
25 else {
26   for(i = 1; i <= L; i++) {
27     sChk(a+i-1, a_bs, a_bd, S);
28     t = a[i-1];
29     if(t < ...) {
30       sChk(b+i, b_bs, b_bd, S);
31       t2 = b[i];
32       sChk(a+i, a_bs, a_bd, S);
33       a[i] += t1 + t2;
34     }
35   }
36 }
```

(e) WP-Driven Loop unswitching

Fig. 4: A motivating example.

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 19 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 Figure 4(c).

**WP Consolidation** In this phase, the WPs for accesses to the same object are considered for consolidation. The code in Figure 4(c) is further optimised into the one in Figure 4(d), 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`. Thus, the number of `wpChk` calls has dropped from 5 to 3 (lines 2, 10, and 11).

**WP-Driven Loop Unswitching** This phase generates the final code in Figure 4(e). The two WPs in the loop, `cwp_a` and `wp_b`, are merged as `wp_loop`, enabling the loop to be unswitched. The `if` branch at lines 16 – 22 is instrumentation-free. The `else` branch at lines 26 – 35 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.

### B. The LLVM IR

WPBOUND, as shown in Figure 3, works directly on the LLVM-IR, LLVM's intermediate representation (IR). As illustrated in Figure 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

```
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) C (b) LLVM-IR (in pseudocode)

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

Single Assignment form) so that each variable use has a unique definition, but address-taken variables are not in SSA.

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 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 Figures 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 $p_{\text{bs}}$ and $p_{\text{bd}}$, respectively, as shown in Figure 2.

### C. 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 (Figure 3), which calculates closed-form expressions for all top-level scalar integer variables (including top-level pointers) in the way described in [52]. 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):

$$e := c \mid v \mid \mathcal{O} \mid e_1 + e_2 \mid e_1 \times e_2 \mid <e_1, +, e_2>_\ell$$

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 $<e_1, +, e_2>_\ell$ 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 $\ell$. For example, in Figure 4(a), the SCEV for the pointer `&a[i]` contained in the `for` loop in line 16 is $<\texttt{a}, +, \texttt{sizeof(int)}>_{16}$. Finally, the notation $\mathcal{O}$ is used to represent any value that is neither expressible nor computable in the SCEV framework.

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

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

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

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

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 behaviour. 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 optimised 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 Figure 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. 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 $\ell$, 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 non-executed 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 $\ll$`a`$, +, 2 \times$ `sizeof(int)`$\gg_{L1}, +,$ `sizeof(int)`$\gg_{L2}$ 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 Figure 6 as:

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

### D. WP-Based Instrumentation

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

*1) Loop-Directed WP Abstraction:* This phase computes 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 $[p_{\text{lb}}^{\text{mar}}, p_{\text{ub}}^{\text{mar}})$. Let the value range of $p$ be $[p_{\text{l}}, p_{\text{u}}]$. There are two cases:

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

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

  where $p_{\text{bs}}$ and $p_{\text{bd}}$ are the base and bound of $p$ (Section III-B). The result of evaluating this WP, called a *WP check*, can be obtained by a call to `wpChk`$(p_{\text{lb}}^{\text{mar}}, p_{\text{ub}}^{\text{mar}}, p_{\text{bs}}, p_{\text{bd}})$ in Figure 4(b).
- $p_{\text{l}} = \mathcal{O} \vee p_{\text{u}} = \mathcal{O}$: The MAR of $p$ is $[p_{\text{lb}}^{\text{mar}}, p_{\text{ub}}^{\text{mar}}) = [\mathcal{O}, \mathcal{O})$ 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 19 in Figure 4) affecting the execution of $*p$ are ignored during this analysis, as explained in Section III-A.

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.

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. 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.

**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 $s \leftarrow$ POSITIONINGWP$(p)$;
4 **if** $[p_{\text{lb}}^{\text{mar}}, p_{\text{ub}}^{\text{mar}}] \neq [\mathcal{O}, \mathcal{O}) \wedge s \neq p$ **then**
5 Insert a `wpChk` call for $*p$ at point $s$: $wp_p = \texttt{wpChk}(p_{\text{lb}}^{\text{mar}}, p_{\text{ub}}^{\text{mar}}, p_{\text{bs}}, p_{\text{bd}})$;
6 Insert a guarded spatial check before $*p$: `if` $(wp_p)$ `sChk`$(p, p_{\text{bs}}, p_{\text{bd}}, SIZE)$;
**else**
7 Insert an unguarded spatial check before $*p$: `sChk`$(p, p_{\text{bs}}, p_{\text{bd}}, SIZE)$;

**Procedure** POSITIONINGWP$(p)$
**begin**
8 $s \leftarrow p$; // denoting the point where $*p$ is
9 **while** $s$ is inside a loop **do**
10 Let $\ell$ be the innermost loop containing $s$;
11 **if** $p_{\text{lb}}^{\text{mar}}$ and $p_{\text{ub}}^{\text{mar}}$ are invariants in $\ell$ **then**
12 $s \leftarrow$ the point just before $\ell$;
13 **else break**;
14 **return** $s$;

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 Figure 4(c). The MAR of `b[i]` in line 10 is $[\texttt{b}+\texttt{SZ}, \texttt{b}+(\texttt{L}+1)\times\texttt{SZ})$, whose lower and upper bounds are invariants of the `for` loop in line 5. With the WP check, `wp_b`, evaluated in line 4, the spatial check for `b[i]` inserted in line 9 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 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:* This phase conducts 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 Figure 4(d)). 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

**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** $\epsilon$;
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** $\epsilon$;

such a check guarded as well (e.g., `cwp_p` in Figure 4(d)).

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:

**Prop. 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).

**Prop. 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, 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 Figure 4(d) again. The MARs for `a[i-1]` in line 14 and `a[i]` in line 19 are $[$`a, a + L × SZ`$)$ and $[$`a + SZ, a + (L + 1) × SZ`$)$, respectively. The consolidated MAR is $[$`a, a + (L + 1) × 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 10, 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:* In this last intraprocedural phase, we apply loop unswitching, a standard loop transformation, to a loop, as illustrated in Figure 4(e), 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 $\ell$ (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 $wp_\pi$ built from the WPs in each group $\pi$ just before $\ell$ (line 7). As $wp_\pi$ is weaker than each constituent $wp$, we can replace each $wp$ by $wp_\pi$ at the expense of more spatial checks (lines 8 – 9). Finally, we unswitch loop $\ell$ so that each spatial check guarded by $wp_\pi$ 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. Condi-

**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 $\ell$ 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) $(\nexists\ \ell' \in \mathcal{L}$ : $\ell'$ 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 $\ell$;
8 **foreach** $wp \in \pi$ **do**
9 Replace each $wp$ inside $\ell$ by $wp_\pi$;
10 Unswitch $\ell$ for every $wp_\pi$, where $\pi \in \Pi$;

tion (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 $wp_\pi$'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 un-instrumented 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 Figure 4(d) to unswitch the `for` loop, which contains two WP guards, `cwp_a` and `wp_b`. Replacing them with a weaker one, `wp_loop = cwp_a || wp_b` and then unswitching the loop yields the final code in Figure 4(e). The instrumentation-free version appears in lines 16 – 22 and the instrumented one in lines 26 – 35.

## 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 un-instrumented libraries, we resort to SoftBound's library function wrappers (Figure 7), 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 non-pointer 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 3.00GHz quad-core Intel Core2 Extreme X9650 CPU and 8GB DDR2 RAM, running on a 64-bit Ubuntu 10.10. The

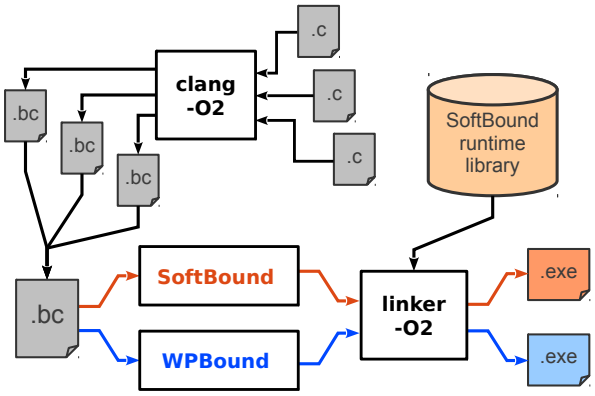


Fig. 7: Compilation workflow.

SoftBound tool is taken from the *SoftBoundCETS* open-source project (version 1.3) [35], [36], configurated to enforce spatial memory safety only.

Table I lists a set of 12 SPEC benchmarks used. These benchmarks are often used in the literature [1], [22], [34], [35], [45]. We have selected eight 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) and `gobmk` and `sjeng` since these two game applications are not loop-oriented. In addition to SPEC2006, we have included four loop-oriented SPEC2000 benchmarks, `ammp`, `art`, `gzip` and `twolf`, in our evaluation.

### C. Methodology

Figure 7 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 analyse the runtime overheads introduced by both tools, the native (instrumentation-free) code is also generated under the "-O2" together with link-time optimization.

### D. Instrumentation Results

Let us first discuss the instrumentation results of the 12 benchmarks according to the statistics given in Table I.

In Column 6, we see that SoftBound inserts an average of 5035 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 optimisations including its dominator-based redundant check elimination [35]. This set of optimisations is also performed by WPBound as well.

In Columns 7 – 11, we can observe some results collected for WPBound. According to Column 7, there are

| Benchmark | #Functions | #Loads | #Stores | #Loops | $\#SC_{SB}$ | WP-Based Instrumentation | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | $\#wp_a$ | $\Delta wp_c$ | $\#wp_l$ | $\overline{\lvert wp_l \rvert}$ | $\max_{\lvert wp_l \rvert}$ |
| ammp | 180 | 3,705 | 1,187 | 650 | 3,962 | 516 | 2,673 | 150 | 4.2 | 54 |
| art | 27 | 471 | 182 | 158 | 461 | 84 | 34 | 46 | 2.0 | 6 |
| gzip | 72 | 936 | 711 | 257 | 1,096 | 83 | 118 | 56 | 1.5 | 7 |
| twolf | 188 | 9,781 | 3,304 | 1,253 | 9,328 | 532 | 2,683 | 195 | 2.8 | 32 |
| bzip2 | 68 | 2,570 | 1,680 | 545 | 2,414 | 324 | 1,114 | 116 | 3.2 | 59 |
| h264ref | 517 | 20,984 | 8,277 | 2,698 | 25,626 | 3,820 | 10,668 | 743 | 5.9 | 235 |
| hmmer | 472 | 8,345 | 3,608 | 1,667 | 8,644 | 1,586 | 3,434 | 502 | 3.7 | 48 |
| lbm | 18 | 244 | 114 | 32 | 319 | 278 | 282 | 10 | 27.8 | 76 |
| libquantum | 96 | 604 | 317 | 144 | 572 | 140 | 358 | 34 | 4.1 | 35 |
| mcf | 26 | 347 | 224 | 76 | 472 | 37 | 216 | 13 | 2.6 | 9 |
| milc | 236 | 3,443 | 1,094 | 544 | 3,266 | 571 | 1,556 | 97 | 7.7 | 49 |
| sphinx3 | 320 | 4,628 | 1,359 | 1,240 | 4,260 | 654 | 1,735 | 343 | 2.2 | 41 |
| ArithMean | 185 | 4,672 | 1,838 | 772 | 5,035 | 719 | 2,073 | 192 | 5.6 | 54.3 |

TABLE I: Benchmark statistics. $\#SC_{SB}$ 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). $\Delta wp_c$ 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). $\overline{\lvert wp_l \rvert}$ 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 non-empty $S$ at line 3 of Algorithm 3).

an average of 719 `wpChk` calls inserted in each benchmark by Algorithm 1 (for WP-based instrumentation), causing ∼1/7 of the spatial checks inserted by SOFTBOUND to be guarded. According to Column 8, Algorithm 2 (for WP consolidation) has made an average of 2073 unconditional checks guarded (a reduction of 41%) for each benchmark. According to Column 9, Algorithm 3 (for loop unswitching) has succeeded in merging an average of 192 WPs at loop entries for each benchmark. Overall, the average number of the WPs combined to yield one disjunctive WP is 5.6 (Column 10), peaking at 235 constituent WPs in one disjunctive WP in the `Mode_Decision_for_4x4IntraBlocks` function in `h264ref` (Column 11).

Finally, as compared in Figure 8, WPBOUND results in slightly larger code sizes than SOFTBOUND due to (1) the `wpChk` calls introduced, (2) the guards added to some spatial checks, and (3) code duplication caused by loop unswitching. Compared to un-instrumented native code, the geometric means of code size increases for SOFTBOUND and WPBOUND are 1.72X and 2.12X, respectively. This implies that WPBOUND has made an instrumented program about 23% 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-D3.

### E. Performance Results

To understand the effects of our WP-based approach on performance, we compare WPBOUND and SOFTBOUND in terms of their overheads and the number of checks performed.

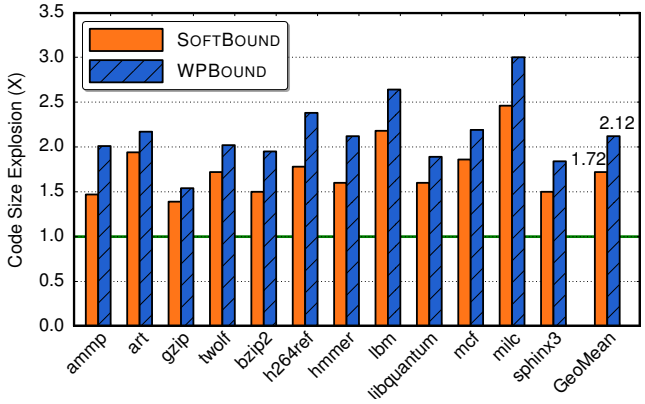


Fig. 8: Bitcode file sizes after instrumentation (normalized with respect to native code).

*1) Runtime Overheads:* Figure 9 compares WPBOUND and SOFTBOUND in terms of their runtime slowdowns over the native code (as the un-instrumented baseline). The average overhead of a tool is measured as the geometric mean of overhead of all benchmarks analyzed by the tool.

SOFTBOUND exhibits an average overhead of 71%, reaching 180% at `h264ref`. In the case of our WP-based instrumentation, WPBOUND has reduced SOFTBOUND's average overhead from 71% to 45%, with significant reductions achieved at `hmmer` (73%), `libquantum` (91%) and `milc` (57%). For `lbm`, which is the best case for both tools, SOFTBOUND and WPBOUND suffer from only 3.7% and 0.9% overheads, respectively. In this benchmark, the pointer load

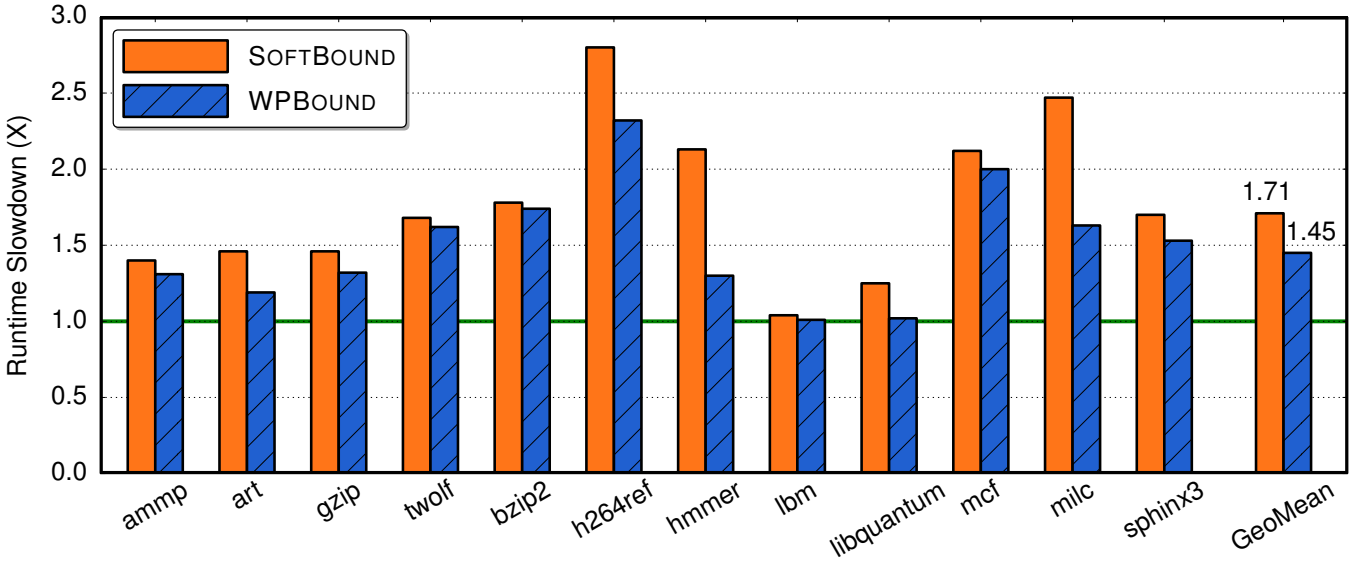


Fig. 9: Execution time (normalized with respect to native code).

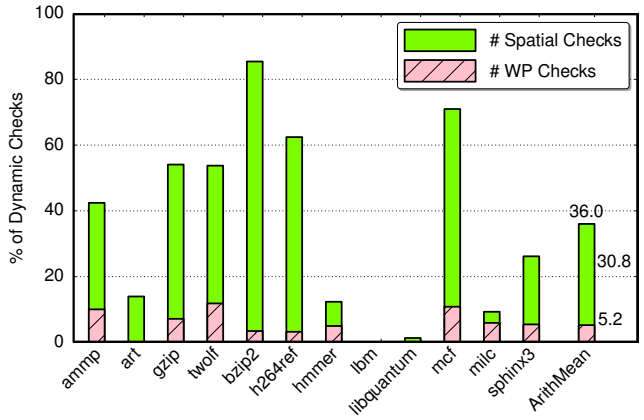


Fig. 10: Percentage of dynamic number of checks performed by WPBound over SoftBound at runtime.

and store operations that are costly for in-memory metadata propagations (as shown in in Figure 2(e)) are relatively scarce. In addition, SoftBound's simple dominator-based redundant check elimination identifies 60% of the checks as unnecessary.

*2) Dynamic Check Count Reduction:* Figure 10 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 36.0% of SoftBound's checks, comprising 5.2% `wpChk` calls and 30.8% `sChk` 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.

*3) Correlation:* By comparing Figures 9 and 10, we can 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`, `lbm`, `libquantum` and `milc`. As for `bzip2`, WPBound still preserves 85% of SoftBound's checks, thereby reducing its overhead from 78% to 74% 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 64.0% but its overhead by 37% only. There are two reasons behind. 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.

## 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 [22], [23], [39], [45], [60] 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], [25], [44], 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 de-allocated by un-instrumented 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 optimisation 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], [32], [40], [42], [54], [55]. One common approach relies on solving a set of constraints formulated based on the program code [5], [15], [40], [42]. 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], [54], [55].

Loops in the program are also a target for bounds check elimination [32]. Some simple patterns can be identified, where unnecessary bound checks can be safely removed.

SOFTBOUND [35] applies simple compile-time optimisations including a dominator-based redundant check elimination to eliminate unnecessary checks dominated by other checks.

Our WP-based optimisation 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], [20], [27], [28], [31], [43], [53], [56]. 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 [21], [26], [46], [47], [48], [59], [62] and value-flow analysis [29], [30], [49], [50], [51] techniques. Recently, static value-flow analysis has been combined with dynamic analysis to reduce instrumentation overheads in detecting uninitialised variables [58]. 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 analysing 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 optimisation for whole-program spatial-error detection.

## VI. CONCLUSION

In this paper, we introduce a new WP-based compile-time optimisation to enforce spatial memory safety for C. Our optimisation complements existing bounds checking optimisations and can be applied to any spatial-error detection approaches (in software or hardware). Implemented on top of SOFTBOUND, a state-of-the-art tool for detecting spatial errors, our WP-based instrumentation tool, WPBOUND, provides compatible and comprehensive spatial safety (by maintaining disjoint per-pointer metadata as in SOFTBOUND) and supports separate compilation (since all its four phases are intraprocedural). For a set of 12 SPEC C benchmarks evaluated, WPBOUND, can substantially reduce the runtime overheads incurred by SOFTBOUND with small code size increases.

## VII. ACKNOWLEDGMENT

The authors wish to thank the anonymous reviewers for their valuable comments. This work is supported by Australian Research Grants, DP110104628 and DP130101970, and a generous gift by Oracle Labs.

## REFERENCES

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, pages 51–66, 2009.

[2] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI*, pages 290–301, 1994.

[3] Domagoj Babic and Alan J Hu. Calysto: Scalable and precise extended static checking. In *ICSE*, pages 211–220, 2008.

[4] Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *ISSAC*, pages 242–249, 1994.

[5] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bounds checks on demand. In *PLDI*, pages 321–333, 2000.

[6] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[7] Intel Corporation. Intel architecture instruction set extensions programming reference, 319433-015 edition, July 2013. http://software.intel.com/sites/default/files/319433-015.pdf.

[8] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram S. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *SOSP*, pages 351–366, 2007.

[9] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the C programming language. In *ASPLOS*, pages 103–114, 2008.

[10] Dinakar Dhurjati and Vikram S. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *ICSE*, pages 162–171, 2006.

[11] Thomas Dillig, Isil Dillig, and Swarat Chaudhuri. Optimal guard synthesis for memory safety. In *CAV*, pages 491–507, 2014.

[12] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI*, pages 155–167, 2003.

[13] Frank Ch. Eigler. Mudflap: Pointer use checking for C/C++. In *GCC Developers Summit*, pages 57–69, 2003.

[14] Andreas Gampe, Jeffery von Ronne, David Niedzielski, and Kleanthis Psarris. Speculative improvements to verifiable bounds check elimination. In *PPPJ*, pages 85–94, 2008.

[15] Andreas Gampe, Jeffery von Ronne, David Niedzielski, Jonathan Vasek, and Kleanthis Psarris. Safe, multiphase bounds check elimination in Java. *Softw., Pract. Exper.*, 41(7):753–788, 2011.

[16] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer overrun detection using linear programming and static analysis. In *CCS*, pages 345–354, 2003.

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

[18] Saugata Ghose, Latoya Gilgeous, Polina Dudnik, Aneesh Aggarwal, and Corey Waxman. Architectural support for low overhead detection of memory violations. In *DATE*, pages 652–657, 2009.

[19] Patrice Godefroid, Michael Y Levin, and David Molnar. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[20] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *ICSE*, pages 232–241, 2006.

[21] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO*, pages 289–298, 2011.

[22] Niranjan Hasabnis, Ashish Misra, and R Sekar. Light-weight bounds checking. In *CGO*, pages 135–144, 2012.

[23] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Winter 1992 USENIX Conference*, pages 125–138, 1991.

[24] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[25] Richard WM Jones and Paul HJ Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG*, pages 13–26, 1997.

[26] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *PLDI*, pages 423–434, 2013.

[27] Wei Le and Mary Lou Soffa. Marple: A demand-driven path-sensitive buffer overflow detector. In *FSE*, pages 272–282, 2008.

[28] Lian Li, Cristina Cifuentes, and Nathan Keynes. Practical and effective symbolic analysis for buffer overflow detection. In *FSE*, pages 317–326, 2010.

[29] Lian Li, Cristina Cifuentes, and Nathan Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *FSE*, pages 343–353, 2011.

[30] Lian Li, Cristina Cifuentes, and Nathan Keynes. Precise and scalable context-sensitive pointer analysis via value flow graph. In *ISMM*, pages 85–96, 2013.

[31] Antoine Mine, David Monniauxli, and Xavier Rival. The ASTREE analyzer. In *ESOP*, pages 21–30, 2005.

[32] José E. Moreira, Samuel P. Midkiff, and Manish Gupta. From Flop to Megaflops: Java for technical computing. *ACM Trans. Program. Lang. Syst.*, 22(2):265–295, March 2000.

[33] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *ISCA*, pages 189–200, 2012.

[34] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. WatchdogLite: Hardware-accelerated compiler-based pointer checking. In *CGO*, pages 175–184, 2014.

[35] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *PLDI*, pages 245–258, 2009.

[36] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. CETS: Compiler enforced temporal safety for C. In *ISMM*, pages 31–40, 2010.

[37] National vulnerability database. http://nvd.nist.gov/.

[38] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[39] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007.

[40] David Niedzielski, Jeffery Ronne, Andreas Gampe, and Kleanthis Psarris. A verifiable, control flow aware constraint analyzer for bounds check elimination. In *SAS*, pages 137–153, 2009.

[41] Harish Patil and Charles N. Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Softw., Pract. Exper.*, 27(1):87–110, 1997.

[42] Feng Qian, Laurie J. Hendren, and Clark Verbrugge. A comprehensive approach to array bounds check elimination for Java. In *CC*, pages 325–342, 2002.

[43] Radu Rugina and Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *PLDI*, pages 182–195, 2000.

[44] Olatunji Ruwase and Monica S Lam. A practical dynamic buffer overflow detector. In *NDSS*, pages 159–169, 2004.

[45] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX ATC*, volume 2012, pages 309–318, 2012.

[46] Lei Shang, Yi Lu, and Jingling Xue. Fast and precise points-to analysis with incremental CFL-reachability summarisation: preliminary experience. In *ASE*, pages 270–273, 2012.

[47] Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. In *CGO*, pages 264–274, 2012.

[48] Yulei Sui, Yue Li, and Jingling Xue. Query-directed adaptive heap cloning for optimizing compilers. In *CGO*, pages 1–11, 2013.

[49] Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA*, pages 254–264, 2012.

[50] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Trans. Software Eng.*, 40(2):107–122, 2014.

[51] Yulei Sui, Sen Ye, Jingling Xue, and Pen-Chung Yew. SPAS: Scalable path-sensitive pointer analysis on full-sparse SSA. In *APLAS*, pages 155–171, 2011.

[52] Robert van Engelen. Efficient symbolic analysis for optimizing compilers. In *CC*, pages 118–132, 2001.

[53] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, pages 1–15, 2000.

[54] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination for the Java HotSpot client compiler. In *PPPJ*, pages 125–133, 2007.

[55] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination in the context of deoptimization. *Science of Computier Programing*, 74(5-6):279–295, March 2009.

[56] Yichen Xie, Andy Chou, and Dawson R. Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *FSE*, pages 327–336, 2003.

[57] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *FSE*, pages 117–126, 2004.

[58] Ding Ye, Yulei Sui, and Jingling Xue. Accelerating dynamic detection of uses of undefined values with static value-flow analysis. In *CGO*, pages 154–164, 2014.

[59] Sen Ye, Yulei Sui, and Jingling Xue. Region-based selective flow-sensitive pointer analysis. In *SAS*, 2014 (To appear).

[60] Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *FSE*, pages 307–316, 2003.

[61] Yves Younan. 25 years of vulnerabilities: 1988-2012.

[62] 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 *CGO*, pages 218 – 229, 2010.