Signedness-Agnostic Program Analysis: Precise Integer Bounds for Low-Level Code

Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey

Department of Computing and Information Systems,
The University of Melbourne, Victoria 3010, Australia

Abstract. Many compilers target common back-ends, thereby avoiding the need to implement the same analyses for many different source languages. This has led to interest in static analysis of LLVM code. In LLVM (and similar languages) most signedness information associated with variables has been compiled away. Current analyses of LLVM code tend to assume that either all values are signed or all are unsigned (except where the code specifies the signedness). We show how program analysis can simultaneously consider each bit-string to be both signed and unsigned, thus improving precision, and we implement the idea for the specific case of integer bounds analysis. Experimental evaluation shows that this provides higher precision at little extra cost. Our approach turns out to be beneficial even when all signedness information is available, such as when analysing C or Java code.

1 Introduction

The "Low Level Virtual Machine" LLVM is rapidly gaining popularity as a target for compilers for a range of programming languages. As a result, the literature on static analysis of LLVM code is growing (for example, see [2, 7, 9, 11, 12]). LLVM IR (Intermediate Representation) carefully specifies the bit-width of all integer values, but in most cases does not specify whether values are signed or unsigned. This is because, for most operations, two's complement arithmetic (treating the inputs as signed numbers) produces the same bit-vectors as unsigned arithmetic. Only for operations that must behave differently for signed and unsigned numbers, such as comparison operations, does LLVM code indicate whether operands are signed or unsigned. In general it is not possible to determine which LLVM variables are signed and which are unsigned.

Surprisingly, current analyses of LLVM code tend to either assume unlimited precision of integers, or else not to take the necessarily signedness-agnostic nature of LLVM analysis into account. An exception is Dietz *et al.* [1] who are very aware of the problems that come from the agnosticism but resolve, as a consequence, to move analysis to the front end of their compiler (Clang). They explain that this design decision has been made after LLVM IR analysis *"proved to be unreliable and unnecessarily complicated, due to requiring a substantial amount of C-level information in the IR . . . for any IR operation the transformation needs to know the types involved (including the size and signedness) . . . "* [1].

In this paper we show how precise static analysis *can* be performed on the LLVM IR, even when signedness information would seem essential to the analysis. As a consequence we can make the most of LLVM's unifying role and avoid any need to separately implement the same analyses for different source languages. Key to our solution is a way of allowing abstract operations (in the sense of abstract interpretation) to superpose the signed and unsigned cases.

At first it may seem surprising that there is any issue at all. On a fixed-width machine utilising two's complement, the algorithm for addition is the same whether the operands are signed or unsigned. The same holds for multiplication. Why should program analysis then have reason to distinguish the signed and unsigned cases? The answer is that most numeric-type abstract domains rely on an integer ordering $\leqslant$. Such an ordering is not available when we do not know whether a bit-vector such as $101 \cdots 100$ should be read as a negative or a positive integer. For example, in performing interval analysis, the rule for $[a, b] \times [c, d]$ varies with the signs of the integers that are being approximated. Any of $[bd, ac]$, $[ad, bc]$, $[bc, ad]$ and $[ac, bd]$ could be the right answer [5]. Note that we cannot conservatively appeal to the *minimum* and *maximum* of the set $\{ac, ad, bc, bd\}$, as these are ill-defined in the absence of signedness information.

We could treat all variables as signed, but this will dramatically lose precision in some cases. In the case of *unsigned* 4-bit variables[1] $x = 0110$ and y known to lie in the interval $[0001, 0011]$ clearly $x+y$ must lie in the interval $[0111, 1001]$, and if the variables are treated as unsigned, this is what is determined. However, if the variables are treated as signed, the addition will be deemed to "wrap around", giving $1000 = -8$ as the smallest possible sum, and $0111 = 7$ as the largest. Thus no useful information will be derived. A similar situation can arise if x and y are signed and we treat them as unsigned, such as when $x = 1110$ and y lies in the interval $[0001, 0011]$.

Surprisingly, the wrong signedness assumption can also lead to *stronger* results than the right assumption. Reflecting on these examples, whether it is better to treat a value as signed or unsigned is determined solely by the patterns of bits in each value, not by whether the value is intended to be interpreted as signed or unsigned. When a computation can wrap around if the operands are treated as signed, it is better to treat them as unsigned, and vice versa. As long as the set of bit-vectors specified by an interval is correct, it does not matter whether we consider the bounds to be signed or unsigned.

This suggests that it is better to treat the bounds of an interval as a superposition of signed and unsigned values, accommodating both signed and unsigned wrap-around. That is, we treat each bound as merely a bit pattern, considering its signedness only when necessary for the operation involved (such as comparison). Instead of representing bounds over fixed bit-width integer types as a range of values on the number line, we handle them as a range of values on a number circle (see Figure 1), or, in the n-dimensional case, as a (closed convex) region of an n-dimensional torus. The unsigned numbers begin with 0 near the "South Pole", proceeding clockwise to the maximum value back near the South Pole.

[1] We use 4-bit examples and binary notation to make examples more manageable.

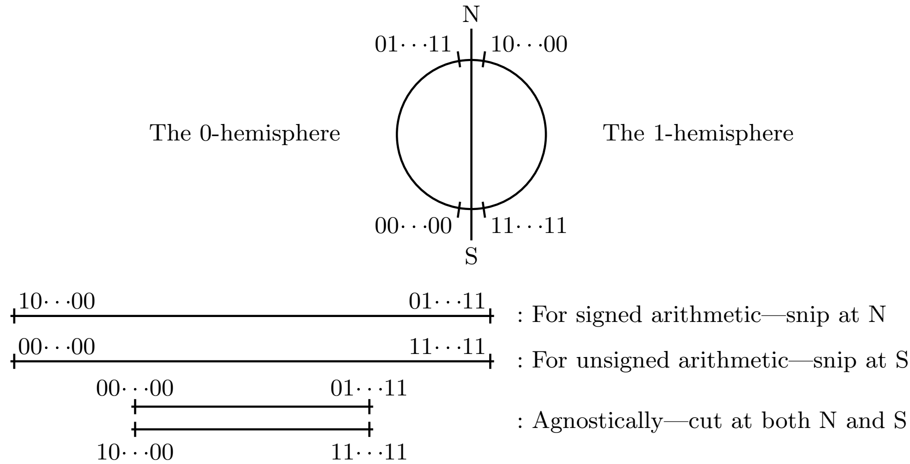


Fig. 1. Three different ways to cut the number circle open

The signed numbers begin with the smallest number near the "North Pole", moving clockwise through 0 to the largest signed number near the North Pole.

"Wrapped" intervals are permitted to cross either pole, or both. Letting an interval start and end anywhere allows for a limited type of disjunctive interval information. For example, an interval $x \in [0111, 1001]$ means $7 \leqslant x \leqslant 9$ if x is treated as unsigned, and $x = 7 \vee -8 \leqslant x \leqslant -7$ if it is treated as signed. This small broadening of the ordinary interval domain allows precise analysis of code where signedness information is unavailable. Equally importantly, it can provide more precise analysis results even where all signedness information *is* provided. We observe that this revised interval domain can pay off in the analysis of real-world programs, without incurring a significant additional cost.

It is important to note that the use of wrapped intervals, combined with signedness-agnosticism, can be worthwhile *even in the presence of signedness information.* Consider $[1000, 1010] \times [1111, 1111]$ in 4-bit arithmetic. Signed analysis gives $[-8, -6] \times [-1, -1] = [6, 8] = [0110, 1000]$, while unsigned analysis gives $[8, 10] \times [15, 15] = [120, 150]$. Here the range exceeds 2^4, so any 4-bit string is deemed possible. So even if the intervals represent *unsigned integers*, the *signed* analysis result is more accurate. Conversely, look at $[1000, 1011] \times [0000, 0001]$. Here signed analysis gives $[-8, -5] \times [0, 1] = [-8, 0]$, which we may write as $[1000, 0000]$ if we allow the interval to wrap. In this case unsigned analysis is the less accurate, as we have $[8, 11] \times [0, 1] = [0, 11]$ that is, we get $[0000, 1011]$. Each analysis provides a correct approximation to the set of five possible values, but the unsigned analysis finds 12 possible elements, whereas the signed analysis is tighter, yielding 9 possible values.

Wrapped interval arithmetic better reflects algebraic properties of the underlying arithmetic operations than intervals without wrapping, even if signedness information is available. Consider for example the computation $x + y - z$. If we know x, y, and z are all signed 4-bit integers in the interval $[0011, 0101]$, then

we determine $y - z \in [1110, 0010]$, whether using wrapped intervals or not. But wrapped intervals will also capture that $x+y \in [0110, 1010]$, while an unwrapped fixed-width interval analysis would see that this sum could be as large as the largest value 0111 and as small as the smallest 1000, so we derive no useful bounds. Therefore, wrapped intervals derive the correct bounds $[0001, 0111]$ for both $(x+y)-z$ and $x+(y-z)$. The use of ordinary intervals, on the other hand, can only derive these bounds for $x + (y - z)$, finding no useful information for $(x+y)-z$, although wrapping is not necessary to represent the final result. This ability to allow intermediate results to wrap around is a powerful advantage of wrapped intervals, even in cases where signedness information is available and final results do not require wrapping.

Consider the function below and assume it is entered with initial bounds information $0 \leqslant x \leqslant 100$ and $y = -10$.

```
char modulo(char x, char y) {
    while (x >= y) {            // line 1
        x = x - y;              // line 2
    }                           // line 3
    return x;                   // line 4
}
```

Traditional interval analysis [5] ignores the fixed-width nature of variables, in this case leading to the incorrect conclusion that line 4 is unreachable. A (traditional) *width aware* interval analysis can do better. During analysis it finds that, immediately before line 2, $0 \leqslant x \leqslant 120$. Performing line 2's abstract subtraction operation, it observes the wrap-around, and finds that x could be as small as -128. It concludes that $-128 \leqslant x \leqslant 127$ at line 3, and that $-128 \leqslant x \leqslant -11$ at line 4. Those are the bounds inferred for the result of the function call.

While this result is correct, it is also disappointingly weak. Inspection of the code tells us that the result of a call to `modulo` with $0 \leqslant x \leqslant 100$ and $y = -10$ must always be smaller than -118, since at each iteration, x increases by 10, and if $-118 \leqslant x \leqslant -11$ at line 4, then on the previous iteration $-128 \leqslant x \leqslant -21$, so the loop would have terminated. Thus we would have hoped for the analysis result $-128 \leqslant \texttt{modulo}(x, y) \leqslant -119$. The traditional bounds analysis misses this result because it considers “wrap around” to be a leap to the opposite end of the number line, rather than the incremental step it is. Using a domain of “wrapped” intervals yields the precision we hoped for. Finding again that $0 \leqslant x \leqslant 120$ at line 2, we derive the bounds $10 \leqslant x \leqslant 127 \vee -128 \leqslant x \leqslant -126$ at line 3 (since $\mathsf{trunc}_8(130) = -126$). At the next iteration of the analysis we have $0 \leqslant x \leqslant 127 \vee -128 \leqslant x \leqslant -126$ at line 1, $0 \leqslant x \leqslant 127$ at line 2, and $10 \leqslant x \leqslant 127 \vee -128 \leqslant x \leqslant -119$ at line 3. This yields $-128 \leqslant x \leqslant -119$ at line 4, and one more iteration proves this to be a fixed point.

In this paper we present a novel approach to signedness-agnostic analysis of programs that manipulate fixed-width integers. The key idea is that correctness and precision of analysis can be obtained by letting abstract operations deal with states that are superpositions of signed/unsigned states. This is done without incurring great running time cost. For a concrete example of the principle, we

define an abstract domain of "wrapped" intervals and associated operations and algorithms. This domain has features that appear to have been overlooked in other work. We report on the analysis cost and benefits, compared to ordinary non-wrapped intervals, as measured on a large set of benchmarks.

2 Related Work

Applications of bounds analysis include array bounds checking, overflow analysis, and bit-width frugal register allocation. However, as we have pointed out, in the context of C and similar languages using fixed-width integer types, correct bounds analysis needs to be aware of the limitations on integer precision. Simon and King [8] show how to make polyhedral analysis wrapping-aware without incurring a high additional cost. Regehr and Duongsaa [6] perform bounds analysis in a wrapping-aware manner, dealing also with bit-wise operations, but as their analysis uses conventional intervals, it is not able to maintain the precision offered by wrapped intervals. Sen and Srikant [7] utilise *strided* wrapped intervals, which they call *Circular Linear Progressions*, for the purpose of analysis of binaries. Their abstract domain is more expressive than what we use, allowing limited relational information. They give detailed abstract operations and refer to their abstract domain as a lattice. Setting the stride in their strided intervals to 1 results in precisely the concept of wrapped intervals that we use in this paper. Hence, as will become clear, their claim that the domain of (wrapped) strided intervals has lattice structure is not correct. More importantly, closer reading of their paper makes it clear that Sen and Srikant assume signed representation. The analysis they propose is not signedness-agnostic in our sense. Gotlieb, Leconte and Marre [3] also study wrapped, or "*clockwise*", intervals. Their aim is to provide constraint solvers for modular arithmetic for the purpose of software verification. They show how to implement abstract addition and subtraction and also how multiplication by a constant can be handled efficiently. Again, a claim that clockwise intervals form a lattice cannot be right. Gotlieb, Leconte and Marre assume unsigned representation, that is, the treatment is not signedness-agnostic. Signedness information is critical in the determination of the potential for under- or over-flow. In that context, the higher precision of bounds analysis that we offer is a useful additional contribution because, as shown by Dietz *et al.* [1], overflow is surprisingly common in real-world C/C++ code. They suggest, based on scrutiny of many programs, that much use of overflow is intentional and safe (though not portable), but also that the majority is probably accidental.

3 Wrapped Intervals

To accurately capture the behaviour of fixed bit-width arithmetic, we must limit the concrete domain to the values representable by the types used in the program, and correct the implementation of the abstract operations to reflect the actual behaviour of the concrete operations [8]. As we have seen, a commitment to

ordinary ordered intervals $[x, y]$ (either signed or unsigned), when wrap-around is possible, can lead to severe loss of precision. Wrapped intervals not only avoid much loss of precision in the case of such wrap-around, but also provide a natural setting for signedness-agnostic analysis.

We shall use the usual arithmetic operators with their usual meaning. Operators subscripted by a number suggest modular arithmetic; more precisely, $a +_n b = a + b \bmod 2^n$ (similarly for other operations). We use $\mathcal{B}$ for the set of all *bit-vectors* of size w. We will use sequence notation to construct bit-vectors: b^k, where $b \in \{0, 1\}$, represents k copies of bit b in a row, and $s_1 s_2$ represents the concatenation of bit-vectors s_1 and s_2. For example, $01^4 0^3$ represents 01111000.

We use $\leqslant$ for the usual lexicographic ordering of $\mathcal{B}$. For example, $0011 \leqslant 1001$. In the context of wrapped intervals, a relative ordering is more useful than an absolute one. We define

$$b \leqslant_a c \;\text{ iff }\; b -_w a \leqslant c -_w a$$

Intuitively, this says that starting from point a on the number circle and travelling clockwise, b is encountered no later than c. It also means that if the number circle were rotated to put a at the South Pole (the zero point), then b would be lexicographically no larger than c. Naturally, $\leqslant_0$ coincides with $\leqslant$, and reflects the normal behaviour of `<=` on unsigned w-bit integers. Similarly, $\leqslant_{2^{w-1}}$ reflects the normal behaviour of `<=` on *signed* w-bit integers. When restricted to a single hemisphere (Figure 1), these orderings coincide, but $\leqslant_0$ and $\leqslant_{2^{w-1}}$ do not agree across hemispheres.

We view LLVM as a bit-vector machine—this is the key to precise analysis in the absence of signedness information, and it is what LLVM truly is. However, for convenience, when operations on bit-vectors will be independent of the interpretation, we may now and then use integers (by default unsigned) to represent bit-vectors. This is just a matter of convenience: by slight extension it allows us to use congruence relations and other modular-arithmetic concepts to express bit-vector relations that are otherwise cumbersome to express. The following definition is a good example.

Definition 1. A *wrapped interval*, or *w-interval*, is either an empty interval, denoted $\bot$, a full interval, denoted $\top$, or a delimited interval $(\!|x, y|\!)$, where x, y are w-width bit-vectors and $x \neq y +_w 1$.[2]

Let $\mathcal{W}_w$ be the set of w-intervals over width w bit-vectors. The meaning of a w-interval is given by the function $\gamma : \mathcal{W}_w \to \mathcal{P}(\mathcal{B})$:

$$\begin{aligned}
\gamma(\bot) &= \varnothing \\
\gamma(\!|x, y|\!) &= \begin{cases} \{x, \ldots, y\} & \text{if } x \leqslant y \\ \{0^w, \ldots, y\} \cup \{x, \ldots, 1^w\} & \text{otherwise} \end{cases} \\
\gamma(\top) &= \mathcal{B}
\end{aligned}$$

[2] The condition, which is independent of signed/unsigned interpretation, avoids duplicate names (such as $(\!|0011, 0010|\!)$ and $(\!|1100, 1011|\!)$) for the full interval.

For example, $\gamma(\!|1111, 1001|\!) = \{1111, 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001\}$ represents the signed integers $[-1, 7] \cup \{-8, -7\}$ or the unsigned integers $[0, 9] \cup \{15\}$. The cardinality of a w-interval is therefore:

$$\begin{array}{ll} \#(\bot) & = 0 \\ \#(\!|x, y|\!) & = (y -_w x +_w 1) \\ \#(\top) & = 2^w \end{array}$$

In an abuse of notation, we define $e \in u$ iff $e \in \gamma(u)$. Note that $\mathcal{W}_w$ is complemented. We define the complement of a w-interval:

$$\begin{array}{ll} \overline{\bot} & = \top \\ \overline{\top} & = \bot \\ \overline{(\!|x, y|\!)} & = (\!|y +_w 1, x -_w 1|\!) \end{array}$$

3.1 Ordering wrapped intervals

We order $\mathcal{W}_w$ by inclusion: $t_1 \sqsubseteq t_2$ iff $\gamma(t_1) \subseteq \gamma(t_2)$. It is easy to see that $\sqsubseteq$ is a partial ordering on $\mathcal{W}_w$; the set is a finite partial order with least element $\bot$ and greatest element $\top$. While $(\mathcal{W}_w, \sqsubseteq)$ is partially ordered, it is *not* a lattice. For example, consider the w-intervals $(\!|0100, 1000|\!)$ and $(\!|1100, 0000|\!)$. Two minimal upper bounds are the incomparable $(\!|0100, 0000|\!)$ and $(\!|1100, 1000|\!)$, two sets of the same cardinality. So a join operation is not available. By duality, neither is a meet operation.

In fact there is no Galois connection (α, γ). For example, $\gamma(\!|1000, 0000|\!) \cap \gamma(\!|0000, 1000|\!) = \{0000, 1000\}$, a set which does not correspond to a w-interval. Furthermore, the two w-intervals $(\!|1000, 0000|\!)$ and $(\!|0000, 1000|\!)$ describe the set $\{0000, 1000\}$ equally well.

The obvious solution is a *biased* pseudo-join operation $\tilde{\sqcup}$ which selects, from the set of possible resulting w-intervals, the one with smallest cardinality, and in case of a tie, the one that contains the lexicographically smallest left bound. First we define membership testing and inclusion. Membership testing is defined:

$$e \in u \;=\; \begin{cases} \mathit{true} & \text{if } u = \top \\ \mathit{false} & \text{if } u = \bot \\ e \leqslant_x y & \text{if } u = (\!|x, y|\!) \end{cases}$$

In guarded definitions like this, the clause that applies is the first (from the top) whose guard is satisfied; that is, an 'if' clause should be read as 'else if'.

Inclusion is defined in terms of membership: either the intervals are identical or else both endpoints of s are in t and at least one endpoint of t is outside s.

$$s \sqsubseteq t \;=\; \begin{cases} \mathit{true} & \text{if } s = \bot \vee t = \top \\ \mathit{false} & \text{if } s = \top \vee t = \bot \\ a \in t \wedge b \in t \wedge (c \notin s \vee d \notin s) & \text{if } s = (\!|a, b|\!), t = (\!|c, d|\!) \end{cases}$$

Consider the cases of possible overlap between two w-intervals shown in Figure 2. Only case (a) depicts containment, but case (b) shows a situation where each

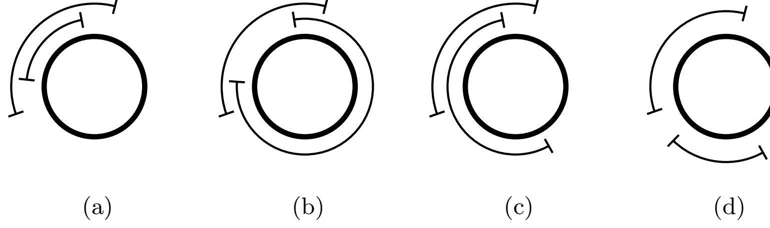


Fig. 2. Four cases of relative position of two w-intervals

w-interval has its bounds contained in the other. This explains why the third case in the definition of $\sqsubseteq$ requires that $c \notin s$ or $d \notin s$.

Then, a biased pseudo-join operation $\tilde{\sqcup}$ is finally defined:

$$s \mathbin{\tilde{\sqcup}} t = \begin{cases} t & \text{if } s \sqsubseteq t \\ s & \text{if } t \sqsubseteq s \\ \top & \text{if } s = (\!|a,b|\!) \wedge t = (\!|c,d|\!) \wedge a \in t \wedge b \in t \wedge c \in s \wedge d \in s \\ (\!|a,d|\!) & \text{if } s = (\!|a,b|\!) \wedge t = (\!|c,d|\!) \wedge b \in t \wedge c \in s \\ (\!|c,b|\!) & \text{if } s = (\!|a,b|\!) \wedge t = (\!|c,d|\!) \wedge d \in s \wedge a \in t \\ (\!|a,d|\!) & \text{if } s = (\!|a,b|\!) \wedge t = (\!|c,d|\!) \wedge \\ & \quad (\#(\!|b,c|\!) < \#(\!|d,a|\!) \vee (\#(\!|b,c|\!) = \#(\!|d,a|\!) \wedge \ a \leqslant c)) \\ (\!|c,b|\!) & \text{otherwise} \end{cases}$$

In this definition, the first two cases handle $\top$ and $\bot$, as well as Figure 2 (a); the third case handles Figure 2 (b); the fourth and fifth cases handle Figure 2 (c); and the final two cases handle Figure 2 (d). We can utilise the fact that $\mathcal{W}_w$ is complemented to define a pseudo-meet operation:

$$s \mathbin{\tilde{\sqcap}} t \ = \ \overline{\overline{s} \mathbin{\tilde{\sqcup}} \overline{t}}$$

Unfortunately the biased pseudo-join has certain shortcomings, inherited by the pseudo-meet. First, $\tilde{\sqcup}$ and $\tilde{\sqcap}$ are not associative. In fact, $(x \mathbin{\tilde{\sqcup}} y) \mathbin{\tilde{\sqcup}} z$ and $x \mathbin{\tilde{\sqcup}} (y \mathbin{\tilde{\sqcup}} z)$ may have different cardinality. This happens, for example, with $x = (\!|0010, 0110|\!)$, $y = (\!|1000, 1010|\!)$, and $z = (\!|1110, 0000|\!)$, since $(x \mathbin{\tilde{\sqcup}} y) \mathbin{\tilde{\sqcup}} z = (\!|1110, 1010|\!)$ has smaller cardinality than $x \mathbin{\tilde{\sqcup}} (y \mathbin{\tilde{\sqcup}} z) = (\!|0010, 0000|\!)$. Second, $\tilde{\sqcup}$ and $\tilde{\sqcap}$ are not monotone. For example, we have $(\!|1111, 0000|\!) \sqsubseteq (\!|1110, 0000|\!)$ and $(\!|0110, 1000|\!) \mathbin{\tilde{\sqcup}} (\!|1111, 0000|\!) = (\!|1111, 1000|\!)$. But owing to the lexicographic bias, $(\!|0110, 1000|\!) \mathbin{\tilde{\sqcup}} (\!|1110, 0000|\!) = (\!|0110, 0000|\!)$. As we do not have $(\!|1111, 1000|\!) \sqsubseteq (\!|0110, 0000|\!)$, $\tilde{\sqcup}$ is not monotone. We discuss the ramifications of this in Section 4, together with a work-around.

The lack of associativity means a generalised $\tilde{\bigsqcup}$ is not well-defined. Nevertheless, the requirement for $\tilde{\bigsqcup}$ is clear: Given a set S of w-intervals, we want $\tilde{\bigsqcup}$ to yield a w-interval t of minimal cardinality so that each interval in S is contained in t. Figure 3 gives an algorithm for computing t. Intuitively, the algorithm returns the complement of the largest un-covered gap among intervals from S. It identifies this gap by passing through S once, picking intervals lexicographically

function $\widetilde{\bigsqcup}(S)$
  $f \leftarrow g \leftarrow \bot$
  for $s \in S$ (in order of lex increasing left bound) **do**
    if $s = \top \vee (s = (\!|x, y|\!) \wedge y \leqslant_0 x)$ **then**
      $f \leftarrow \mathsf{extend}(f, s)$
  for $s \in S$ (in order of lex increasing left bound) **do**
    $g \leftarrow \mathsf{bigger}(g, \mathsf{gap}(f, s))$
    $f \leftarrow \mathsf{extend}(f, s)$
  return $\overline{\mathsf{bigger}(g, \overline{f})}$

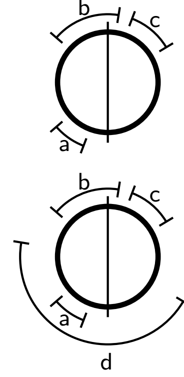


Fig. 3. Finding the (pseudo) least upper bound of a set of w-intervals

by their left bounds. However, care must be taken to ensure that any *apparent* gaps, which are in fact covered by w-intervals that cross the South Pole and may only be found later in the iteration, are not mistaken for *actual* gaps. We define the gap between two w-intervals as empty if they overlap, or otherwise the clockwise distance from the end of the first to the start of the second.

$$\mathsf{gap}(s, t) \;=\; \begin{cases} \overline{(\!|c, b|\!)} & \text{if } s = (\!|a, b|\!) \wedge t = (\!|c, d|\!) \wedge b \notin t \wedge c \notin s \\ \bot & \text{otherwise} \end{cases}$$

We also define an operation $\mathsf{extend}(s, t)$ to yield the w-interval from the start of s to the end of t, ensuring it includes all of s and t. This operation is identical to $\widetilde{\sqcup}$, except that the last cases are omitted, and the condition on the penultimate case is relaxed to apply regardless of cardinalities. Finally, we define $\mathsf{bigger}(s, t)$ to be t if $\# t > \# s$, and s otherwise. The two loops in Figure 3 traverse the set of w-intervals in order of lexicographically increasing left bound; it does not matter where $\top$ and $\bot$ appear in this sequence. The first loop assigns to f the least upper bound of all w-intervals that cross the South Pole. The invariant for the second loop is that g is the largest uncovered gap in f; thus the loop can be terminated as soon as $f = \top$. When the loop terminates, all w-intervals have been incorporated in f, so $\overline{f}$ is an uncovered gap, and g is the largest uncovered gap in f. Thus the result is the complement of the bigger of g and $\overline{f}$.

Consider Figure 3 (upper right) as an example. Here no intervals cross the South Pole, so at the start of the second loop, $f = g = \bot$, and at the end of the loop, g is the gap between a and b, and f is the interval clockwise from the start of a to the end of c. Since the complement of f is larger than g, the result in this case is f: the interval from the start of a to the end of c.

For the lower right example of Figure 3, interval d does cross the South Pole, so at the start of the second loop, $f =$ d and $g = \bot$. Now in the second loop, f extends clockwise to encompass b and c, and finally also d, at which point f becomes $\top$. But because the loop starts with $f =$ d, g never holds the gap between a and b; finally it holds the gap between the end of c and the start of d.

Now the complement of f is smaller than g so the final result is the complement of g, that is, the interval from the start (right end) of d to the end of c.

The $\widetilde{\bigsqcup}$ operation is useful because it may preserve information that would be lost by repeated use of the pseudo-join. Thus it should always be used when multiple w-intervals must be joined together, such as in the implementation of multiplication below. Furthermore, it will improve precision in other contexts to delay computation of pseudo-joins until all the w-intervals that will be joined are available, and substitute $\widetilde{\bigsqcup}$ for multiple uses of $\tilde{\sqcup}$.

The intersection of two w-intervals returns one or two w-intervals, and gives the accurate intersection, in the sense that $\bigcup\{\gamma(u) \mid u \in s \cap t\} = \gamma(s) \cap \gamma(t)$.

$$s \cap t = \begin{cases} \{\,\} & \text{if } s = \bot \text{ or } t = \bot \\ \{t\} & \text{if } s = t \vee s = \top \\ \{s\} & \text{if } t = \top \\ \{(\!|a, d|\!), (\!|b, c|\!)\} & \text{if } s = (\!|a, b|\!) \wedge t = (\!|c, d|\!) \wedge a \in t \wedge b \in t \wedge c \in s \wedge d \in s \\ \{s\} & \text{if } s = (\!|a, b|\!) \wedge t = (\!|c, d|\!) \wedge a \in t \wedge b \in t \\ \{t\} & \text{if } s = (\!|a, b|\!) \wedge t = (\!|c, d|\!) \wedge c \in s \wedge d \in s \\ \{(\!|a, d|\!)\} & \text{if } s = (\!|a, b|\!) \wedge t = (\!|c, d|\!) \wedge a \in t \wedge d \in s \wedge b \notin t \wedge c \notin s \\ \{(\!|b, c|\!)\} & \text{if } s = (\!|a, b|\!) \wedge t = (\!|c, d|\!) \wedge b \in t \wedge c \in s \wedge a \notin t \wedge d \notin s \\ \{\,\} & \text{otherwise} \end{cases}$$

3.2 Analysing expressions

On w-intervals, addition is defined as follows:

$$s + t = \begin{cases} \bot & \text{if } s = \bot \text{ or } t = \bot \\ (\!|a +_w c, b +_w d|\!) & \text{if } s = (\!|a, b|\!),\ t = (\!|c, d|\!), \text{ and } \#s + \#t \leqslant 2^w \\ \top & \text{otherwise} \end{cases}$$

Here, to detect a possible overflow when adding the two cardinalities, standard addition is used. Note that $+_w$ is signedness-agnostic: treating it as signed or unsigned makes no difference. The rule for subtraction $(s - t)$ is similar—just replace the delimited interval on the left by $(\!|a -_w d, b -_w c|\!)$. The definition of abstract unary minus then follows easily, by noting that $-s = 0 - s$.

Multiplication on w-intervals is more cumbersome, even when we settle for a less-than-optimal solution. The reason is that even though unsigned and signed multiplication are the same operation on bit-vectors, signed and unsigned interval multiplication retain *different* information. The solution requires separating each interval at the North and South poles, so the segments agree on ordering for both signed and unsigned interpretations, and then performing both signed and unsigned multiplication on the fragments.

It is convenient to have names for the smallest w-intervals that straddle the poles. Let $\mathsf{np} = (\!|01^{w-1}, 10^{w-1}|\!)$ and $\mathsf{sp} = (\!|1^w, 0^w|\!)$. Define the North Pole split of a delimited w-interval as follows:

$$\mathsf{nsplit}(s) = \begin{cases} \varnothing & \text{if } s = \bot \\ \{(\!|a, b|\!)\} & \text{if } s = (\!|a, b|\!) \text{ and } \mathsf{np} \not\sqsubseteq (\!|a, b|\!) \\ \{(\!|a, 01^{w-1}|\!), (\!|10^{w-1}, b|\!)\} & \text{if } s = (\!|a, b|\!) \text{ and } \mathsf{np} \sqsubseteq (\!|a, b|\!) \\ \{(\!|0^w, 01^{w-1}|\!), (\!|10^{w-1}, 1^w|\!)\} & \text{if } s = \top \end{cases}$$

and define the South Pole split ssplit similarly (in particular, the last case is identical). Then let the sphere cut be

$$\mathsf{cut}((\!|x, y|\!)) = \bigcup \{\mathsf{ssplit}(u) \mid u \in \mathsf{nsplit}((\!|x, y|\!))\}$$

For example, $\mathsf{cut}((\!|1111, 1001|\!)) = \{(\!|1111, 1111|\!), (\!|0000, 0111|\!), (\!|1000, 1001|\!)\}$.

Unsigned and signed multiplication of two delimited w-intervals $(\!|a, b|\!)$ and $(\!|c, d|\!)$ that do not straddle poles is straightforward:

$$(\!|a, b|\!) \times_u (\!|c, d|\!) = \begin{cases} (\!|a \times_w c, b \times_w d|\!) & \text{if } b \times d - a \times c < 2^w \\ \top & \text{otherwise} \end{cases}$$

And, letting msb be the function that extracts the most significant bit:

$$(\!|a, b|\!) \times_s (\!|c, d|\!) = \begin{cases} (\!|a \times_w c, b \times_w d|\!) \text{ if } \mathsf{msb}(a) = \mathsf{msb}(b) = \mathsf{msb}(c) = \mathsf{msb}(d) \\ \qquad\qquad\qquad \wedge\, b \times d - a \times c < 2^w \\ (\!|a \times_w d, b \times_w c|\!) \text{ if } \mathsf{msb}(a) = \mathsf{msb}(b) = 1 \wedge \mathsf{msb}(c) = \mathsf{msb}(d) = 0 \\ \qquad\qquad\qquad \wedge\, b \times c - a \times d < 2^w \\ (\!|b \times_w c, a \times_w d|\!) \text{ if } \mathsf{msb}(a) = \mathsf{msb}(b) = 0 \wedge \mathsf{msb}(c) = \mathsf{msb}(d) = 1 \\ \qquad\qquad\qquad \wedge\, a \times d - b \times c < 2^w \\ \top \qquad\qquad\qquad \text{otherwise} \end{cases}$$

Now, signed and unsigned bit-vector multiplication agree for segments that do not straddle a pole. This is an important observation which gives us a handle on precise multiplication across arbitrary delimited w-intervals:

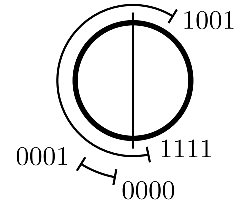
$$(\!|a, b|\!) \times_{us} (\!|c, d|\!) = ((\!|a, b|\!) \times_u (\!|c, d|\!)) \cap ((\!|a, b|\!) \times_s (\!|c, d|\!))$$

The use of intersection in this definition is the source of the added precision. Each of $\times_u$ and $\times_s$ gives a correct over-approximation of multiplication, and hence the intersection is also a correct over-approximation.

This now allows us to do general signedness-agnostic multiplication by joining the segments obtained from each piecewise hemisphere multiplication:

$$s \times t = \widetilde{\bigsqcup} \{m \mid u \in \mathsf{cut}(s), v \in \mathsf{cut}(t), m \in u \times_{us} v\}$$

Consider the multiplication $(\!|1111, 1001|\!) \times (\!|0000, 0001|\!)$. The two intervals are shown in the diagram here. The cut of the first w-interval is $\{(\!|1111, 1111|\!), (\!|0000, 0111|\!), (\!|1000, 1001|\!)\}$, the cut of the second is $\{(\!|0000, 0001|\!)\}$. The three separate segment multiplications give:



1. $(\!|1111, 1111|\!) \times_u (\!|0000, 0001|\!) = \top$,
   $(\!|1111, 1111|\!) \times_s (\!|0000, 0001|\!) = (\!|1111, 0000|\!)$,
   hence $(\!|1111, 1111|\!) \times_{us} (\!|0000, 0001|\!) = \{(\!|1111, 0000|\!)\}$.
2. $(\!|0000, 0111|\!) \times_u (\!|0000, 0001|\!) = (\!|0000, 0111|\!)$
   $(\!|0000, 0111|\!) \times_s (\!|0000, 0001|\!) = (\!|0000, 0111|\!)$
   hence $(\!|0000, 0111|\!) \times_{us} (\!|0000, 0001|\!) = \{(\!|0000, 0111|\!)\}$.
3. $(\!|1000, 1001|\!) \times_u (\!|0000, 0001|\!) = (\!|0000, 1001|\!)$,
   $(\!|1000, 1001|\!) \times_s (\!|0000, 0001|\!) = (\!|1000, 0000|\!)$,
   hence $(\!|1000, 1001|\!) \times_{us} (\!|0000, 0001|\!) = (\!|0000, 1001|\!) \cap (\!|1000, 0000|\!)$
   $= \{(\!|1000, 1001|\!), (\!|0000, 0000|\!)\}$.

Applying $\widetilde{\bigsqcup}$, we get the maximally accurate result $(\!|1111, 1001|\!)$. Note the crucial role played by $\times_{us}$ in obtaining this precision. For example, in the first case above, where we have no information about the result of unsigned multiplication ($(\!|1111, 1111|\!) \times_u (\!|0000, 0001|\!) = \top$), we effectively assume that multiplication is signed, obtaining a much tighter result. The role of $\times_{us}$ is to *do signed and unsigned multiplication simultaneously.* This is very different from the obvious case analysis that considers the unsigned and signed cases separately: For the example, either yields $\top$.

What is important about our approach is that the signed/unsigned case analysis happens as late as possible, at the "micro-level". This is what we have in mind when we say that the abstract operations deal with superposed signed/unsigned states. The superposition idea is general and works for other operations. However, it does not always add precision—for many abstract operations we can obtain equivalent but simpler definitions.

Unsigned and signed division are different operations, since the definition depends on the ordering of bit-vectors. Hence we need two abstract operations. Here, since the interpretation of the bit-vectors is given, the definition is straightforward but lengthy and slightly more complicated, owing to the need to carve out the sub-interval $(\!|0^w, 0^w|\!)$. The modulus operation is similar to division.

For the logical operations, it is tempting to simply consider the combinations of interval endpoints, at least when no interval straddles two hemispheres. But that does not work. For example, the endpoints of $(\!|1010, 1100|\!)$ are not sufficient to determine the endpoints of $(\!|1010, 1100|\!) \mid (\!|0110, 0110|\!)$. Namely, $1010 \mid 0110 = 1100 \mid 0110 = 1110$, but $1011 \mid 0110 = 1111$. Instead we use the unsigned versions of algorithms provided by Warren [10] (pages 58–62), but adapted to w-intervals using a South Pole split. We present the method for bitwise-or $\mid$; those for bitwise-and and bitwise-xor are similar.

$$s \mid t = \widetilde{\bigsqcup} \{u \mid_w v \mid u \in \mathsf{ssplit}(s), v \in \mathsf{ssplit}(t)\}$$

where $\mid_w$ is Warren's unsigned bitwise or operation for intervals [10], an operation with complexity $O(w)$.

Signed and zero extension are defined as follows. We assume words of width w are being extended to width $w + k$, with $k > 0$.

$$\mathsf{sext}(s,k) = \widetilde{\bigsqcup} \{(\!|(\mathsf{msb}(a))^k a, (\mathsf{msb}(b))^k b|\!) \mid (\!|a,b|\!) \in \mathsf{nsplit}(s)\}$$
$$\mathsf{zext}(s,k) = \widetilde{\bigsqcup} \{(\!|0^k a, 0^k b|\!) \mid (\!|a,b|\!) \in \mathsf{ssplit}(s)\}$$

Truncation to $k < w$ bits (integer downcasting) keeps the lower k bits of a bit-vector of length w. Accordingly, $\mathsf{trunc}(s,k)$ is a w width w-interval s truncated to a k width w-interval. Truncation is defined as:

$$\mathsf{trunc}(s,k) = \begin{cases} \bot & \text{if } s = \bot \\ (\!|\mathsf{trunc}(a,k), \mathsf{trunc}(b,k)|\!) & \text{if } s = (\!|a,b|\!) \wedge a \texttt{>>}_a k = b \texttt{>>}_a k \\ & \quad \wedge\ \mathsf{trunc}(a,k) \leqslant \mathsf{trunc}(b,k) \\ (\!|\mathsf{trunc}(a,k), \mathsf{trunc}(b,k)|\!) & \text{if } s = (\!|a,b|\!) \wedge (a \texttt{>>}_a k) + 1 \equiv_{2^w} b \texttt{>>}_a k \\ & \quad \wedge\ \mathsf{trunc}(a,k) \nleqslant \mathsf{trunc}(b,k) \\ \top & \text{otherwise} \end{cases}$$

where $\texttt{>>}_a$ is arithmetic right shift. Once truncation is defined, we can easily define left shift:

$$s \texttt{<<} k = \begin{cases} \bot & \text{if } s = \bot \\ (\!|a \texttt{<<} k, b \texttt{<<} k|\!) & \text{if } \mathsf{trunc}(s, w-k) = (\!|a,b|\!) \\ (\!|0^w, 1^{w-k}0^k|\!) & \text{otherwise} \end{cases}$$

Logical right shifting ($\texttt{>>}_l$) requires testing if the South Pole is covered:

$$s \texttt{>>}_l k = \begin{cases} \bot & \text{if } s = \bot \\ (\!|0^w, 0^k 1^{w-k}|\!) & \text{if } \mathsf{sp} \sqsubseteq s \\ (\!|a \texttt{>>}_l k, b \texttt{>>}_l k|\!) & \text{if } s = (\!|a,b|\!) \end{cases}$$

and arithmetic right shifting ($\texttt{>>}_a$) requires testing if the North Pole is covered:

$$s \texttt{>>}_a k = \begin{cases} \bot & \text{if } s = \bot \\ (\!|1^k 0^{w-k}, 0^k 1^{w-k}|\!) & \text{if } \mathsf{np} \sqsubseteq s \\ (\!|a \texttt{>>}_a k, b \texttt{>>}_a k|\!) & \text{if } s = (\!|a,b|\!) \end{cases}$$

Shifting with variable shift, for example, $s \texttt{<<} t$, can be defined by calculating the (fixed) shift for each $k \in (\!|0, w-1|\!)$ which is an element of t, and pseudo-joining the resulting w-intervals.

3.3 Dealing with control flow

In LLVM, comparison operations are explicitly signed or unsigned. Taking the 'then' branch of a conditional with condition $s \leqslant_0 t$ can be thought of as prefixing the branch with '`assume` $s \leqslant_0 t$', and this assume statement can narrow the bounds for s and t. We update the information for s as follows:

$$s = \begin{cases} \bot & \text{if } t = \bot \\ s & \text{if } 1^w \in t \\ s \,\widetilde{\sqcap}\, (\!|0^w, b|\!) & \text{if } t = (\!|a,b|\!) \end{cases}$$

Signed comparison ($\leqslant_{2^{w-1}}$) is similar, but replaces 1^w by 01^{w-1} and $(\!|0^w, b|\!)$ by $(\!|10^{w-1}, b|\!)$. Updating the second argument t in a context $s \leqslant_{2^{w-1}} t$ is defined analogously. Finally, φ-nodes in the LLVM control-flow graph are handled as usual, in our case using $\widetilde{\bigsqcup}$.

4 Non-Termination and Widening

As shown in Section 3, $\tilde{\sqcup}$ lacks desirable algebraic properties and fails to be monotone. Although $\mathcal{W}_w$ is finite, the fact that $\tilde{\sqcup}$ is not monotone raises a major problem: a *least fixed point* may not exist because multiple fixed points could be equally precise, and even worse, the analysis may not terminate. Fortunately, in practice, there is an easy solution to this problem. While $\mathcal{W}_w$ is finite, it does contain chains of length $O(2^w)$. Hence, for efficient analysis, fixed point acceleration is needed in any case. Judicious use of widening ensures termination of our analysis, side-stepping the non-monotonicity problem. We define an upper bound operator ∇, based on the idea of widening by (roughly) doubling the size of a w-interval. First, $s \nabla \bot = \bot \nabla s = s$, and $s \nabla \top = \top \nabla s = \top$. Additionally,

$$\llparenthesis u, v \rrparenthesis \nabla \llparenthesis x, y \rrparenthesis = \begin{cases} \llparenthesis u, v \rrparenthesis & \text{if } \llparenthesis x, y \rrparenthesis \sqsubseteq \llparenthesis u, v \rrparenthesis \\ \top & \text{if } \# \llparenthesis u, v \rrparenthesis \geqslant 2^{w-1} \\ \llparenthesis u, y \rrparenthesis \mathbin{\tilde{\sqcup}} \llparenthesis u, 2v -_w u +_w 1 \rrparenthesis & \text{if } \llparenthesis u, v \rrparenthesis \mathbin{\tilde{\sqcup}} \llparenthesis x, y \rrparenthesis = \llparenthesis u, y \rrparenthesis \\ \llparenthesis x, v \rrparenthesis \mathbin{\tilde{\sqcup}} \llparenthesis 2u -_w v -_w 1, v \rrparenthesis & \text{if } \llparenthesis u, v \rrparenthesis \mathbin{\tilde{\sqcup}} \llparenthesis x, y \rrparenthesis = \llparenthesis x, v \rrparenthesis \\ \llparenthesis x, y \rrparenthesis \mathbin{\tilde{\sqcup}} \llparenthesis x, x +_w 2v -_w 2u +_w 1 \rrparenthesis & \text{if } u, v \in \llparenthesis x, y \rrparenthesis \\ \top & \text{otherwise} \end{cases}$$

Then ∇ is an upper bound operator [5] and we have the property

$$s \nabla t = s \;\vee\; s \nabla t = \top \;\vee\; \# s \nabla t \geqslant 2 \# s$$

Given $f : \mathcal{W}_w \to \mathcal{W}_w$, we define the accelerated sequence $\{f_\nabla^n\}_n$ as follows:

$$f_\nabla^n = \begin{cases} \bot & \text{if } n = 0 \\ f_\nabla^{n-1} & \text{if } n > 0 \wedge f(f_\nabla^{n-1}) \sqsubseteq f_\nabla^{n-1} \\ f_\nabla^{n-1} \nabla f(f_\nabla^{n-1}) & \text{otherwise} \end{cases}$$

Since $\{f_\nabla^n\}_n$ is increasing (whether f is monotone or not) and $\mathcal{W}_w$ has finite height, the accelerated sequence eventually stabilises. In our implementation we perform a widening step after every fifth iterative step.

5 Experimental Evaluation

We implemented our wrapped interval analysis for LLVM 3.0 and ran the experiments on an Intel Core with a 2.70Gz clock and 7.8Gb of memory. For comparison purposes we also implemented an unwrapped fixed-width interval analysis using the same fixpoint algorithm and assuming signed numbers. We used the Spec CPU 2000 benchmark suite widely used by LLVM testers.

Fig. 4 shows the results of an evaluation. Columns 2 and 3 show analysis times (average of 5 runs) in seconds for the unwrapped interval analysis (T_U) and our wrapped analysis (T_W). Column 4 (#TI) is the total number of integer intervals considered by the analyses. Column 5 (#RTI) shows the total number of integer intervals after removing cases in which both analyses inferred $\top$ or $\bot$.

Program	T_U	T_W	#TI	#RTI	$\#G_W$	$\%G_W$
164.gzip	0.87	0.97	9869	940	113	10
175.vpr	3.88	7.2	10324	1631	473	29
176.gcc	98.12	104.39	795500	5194	1121	21
186.crafty	4.24	4.81	47664	1461	210	14
197.parser	3.53	6.51	19613	1502	347	23
255.vortex	43.46	49.46	276751	1511	214	14
256.bzip2	0.72	1.1	5548	1008	113	11
300.twolf	6.97	9.44	49349	1185	149	12

Fig. 4. Comparison between unwrapped and wrapped interval analyses

Finally, Column 6 ($\#G_W$) shows the gains of our wrapped interval analysis by counting the number of more precise intervals inferred by our analysis, and the last column ($\%G_W$) shows the ratio of improvement ($\frac{\#G_W}{\#RTI} \times 100$).

Although our prototype is not optimized and can be improved in several ways, the two analyses run fast for most programs with the exceptions of 176.gcc and 255.vortex (the biggest programs). It is worth mentioning that we perform function inlining before running the two analyses, which increases benchmark sizes and hence, analysis times. Nevertheless, the cost of wrapped interval analysis is comparable to that of an unwrapped version.

Since we analyse LLVM IR, signedness information is in general not available. Therefore, to compare the precision of "unwrapped" and "wrapped" analysis, we ran the unwrapped analysis assuming all integers are signed, similarly to [9]. The use of wrapped intervals pays off, improving precision by 19% on average, once trivial results are excluded from comparison.

6 Conclusion

Analysis of programs written in LLVM IR and similar low-level languages is hampered by the fact that, for many variables, signedness information has been stripped away. While it is possible to analyse programs correctly under the assumption that such variables are unsigned (or signed, depending on taste), such an assumption leads to a serious loss of precision.

It is far better for analysis to be signedness-agnostic. We have shown that, if implemented carefully, signedness-agnosticism amounts to more than simply "having a bet each way". Our key observation is that one can achieve higher accuracy of analysis by making each individual abstract operation signedness-agnostic, whenever its concrete counterpart is signedness-agnostic. This applies to important operations like addition, subtraction and multiplication.

Signedness-agnostic *bounds analysis* naturally leads to wrapped intervals, since signed and unsigned representation correspond to two different ways of ordering bit-vectors. In this paper we have detailed the first signedness-agnostic bounds analysis, based on wrapped intervals. The resulting analysis is efficient

and accurate, and is beneficial even for programs where all signedness information is present. Future work involves better assessing the practical benefits of our approach, for example, in the context of software verification. Another line of research is to generalise wrapped interval analysis to relational analyses, such as those based on difference logic (constraints $x - y \leqslant k$) or octagons [4].

Acknowledgment

This work was supported by the Australian Research Council through grant DP110102579. We thank Fernando Pereira, Victor Campos, Douglas do Couto, and Igor Rafael for fruitful discussions and for making their LLVM SSI (Static Single Information) construction pass available.

References

1. W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *Proc. 34th Int. Conf. Software Eng.*, pages 760–770. IEEE, 2012.
2. S. Falke, D. Kapur, and C. Sinz. Termination analysis of imperative programs using bitvector arithmetic. In R. Joshi, P. Müller, and A. Podelski, editors, *Verified Software: Theories, Tools, and Experiments*, volume 7152 of *LNCS*, pages 261–277. Springer, 2012.
3. A. Gotlieb, M. Leconte, and B. Marre. Constraint solving on modular integers. In *Proc. Ninth Int. Workshop Constraint Modelling and Reformulation*, 2010.
4. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
5. F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
6. J. Regehr and U. Duongsaa. Deriving abstract transfer functions for analyzing embedded software. In *LCTES'06: Proc. Conf. Language, Compilers, and Tool Support for Embedded Systems*, pages 34–43. ACM Press, 2006.
7. R. Sen and Y. N. Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *Proc. Fifth IEEE/ACM Int. Conf. Formal Methods and Models for Codesign*, pages 39–48. IEEE, 2007.
8. A. Simon and A. King. Taming the wrapping of integer arithmetic. In H. R. Nielson and G. Filé, editors, *Static Analysis*, volume 4634 of *LNCS*, pages 121–136. Springer, 2007.
9. D. d. C. Teixera and F. M. Q. Pereira. The design and implementation of a non-iterative range analysis algorithms on a production compiler. In *Proc. 2011 Brasilian Symp. Programming Languages*, 2011.
10. H. S. Warren Jr. *Hacker's Delight*. Addison Wesley, 2003.
11. C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou. IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In D. Gritzalis, B. Preneel, and M. Theoharidou, editors, *Proc. ESORICS 2010*, volume 6345 of *LNCS*, pages 71–86. Springer, 2010.
12. C. Zhang, W. Zou, T. Wang, Y. Chen, and T. Wei. Using type analysis in compiler to mitigate integer-overflow-to-buffer-overflow threat. *Journal of Computer Security*, 19(6):1083–1107, 2011.