A Fast and Low-Overhead Technique to Secure Programs Against Integer Overflows

Raphael Ernani Rodrigues, Victor Hugo Sperle Campos, Fernando Magno Quintão Pereira

Department of Computer Science – The Federal University of Minas Gerais (UFMG) – Brazil

{raphael,victorsc,fernando}@dcc.ufmg.br

Abstract

The integer primitive type has upper and lower bounds in many programming languages, including C, and Java. These limits might lead programs that manipulate large integer numbers to produce unexpected results due to overflows. There exists a plethora of works that instrument programs to track the occurrence of these overflows. In this paper we present an algorithm that uses static range analysis to avoid this instrumentation whenever possible. Our range analysis contains novel techniques, such as a notion of "future" bounds to handle comparisons between variables. We have used this algorithm to avoid some checks created by a dynamic instrumentation library that we have implemented in LLVM. This framework has been used to detect overflows in hundreds of C/C++ programs. As a testimony of its effectiveness, our range analysis has been able to avoid 25% of all the overflow checks necessary to secure the C programs in the LLVM test suite. This optimization has reduced the runtime overhead of instrumentation by 50%.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers

General Terms Languages, Performance

Keywords Integer Overflow, Compiler, Range analysis

1. Introduction

The most popular programming languages, including C, C++ and Java, limit the size of primitive numeric types. For instance, the `int` type, in C++, ranges from -2^{31} to $2^{31} - 1$. Consequently, there exist numbers that cannot be represented by these types. In general, these programming languages resort to a *wrapping-arithmetics* semantics [27] to perform integer operations. If a number n is too large to fit into a primitive data type T, then n's value wraps around, and we obtain n modulo T_{max}. There are situations in which this semantics is acceptable [11]. For instance, programmers might rely on this behavior to implement hash functions and random number generators. On the other hand, there exist also situations in which this behavior might lead a program to produce unexpected results. As an example, in 1996, the Ariane 5 rocket was lost due to an arithmetic overflow – a bug that resulted in a loss of more than US$370 million [12].

Programming languages such as Ada or Lisp can be customized to throw exceptions whenever integer overflows are detected. Furthermore, there exist recent works proposing to instrument binaries derived from C, C++ and Java programs to detect the occurrence of overflows dynamically [4, 11]. Thus, the instrumented program can take some action when an overflow happens, such as to log the event, or to terminate the program. However, this safety has a price: arithmetic operations need to be surveilled, and the runtime checks cost time. Zhang *et al.* [28] have eliminated some of this overhead via a tainted flow analysis. We have a similar goal, yet, our approach is substantially different.

This paper describes the range analysis algorithm that we have developed to eliminate overflow checks in instrumented programs. As we show in Section 2, our algorithm has three core insights. Firstly, in Section 2.1 we show how we rely on strongly connected components to achieve speed and precision. It is well-known that this technique is effective in speeding up constraint resolution [22, Sec 6.3]. Yet, we go beyond: given our three-phase approach, we improve precision by solving strong components in topological order. Secondly, in Section 2.2 we describe this three-phase approach to extract information from comparisons between variables, e.g., $x < y$. Previous algorithms either deal with these comparisons via expensive relational analyses [9, 16, 19], or only consider comparisons between variables and constants [18, 23, 24]. Finally, in Section 2.3 we propose a new program representation that is more precise than other intermediate forms used to solve range analysis sparsely. This new live range splitting strategy is only valid if the instrumented program terminates whenever an integer overflow is detected. If we cannot rely on this guarantee, then our more

CGO '13 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE...$15.00

conservative live range splitting strategy produces the program representation that Bodik *et al.* [2] call Extended Static Single Assignment form. In Section 4.1 we show that an interprocedural implementation of our algorithm analyzes programs with half-a-million lines of code in less than ten seconds. Furthermore, the speed and memory consumption of our range analysis grows linearly with the program size.

We use our range analysis to reduce the runtime overhead imposed by a dynamic instrumentation library. This instrumentation framework, which we describe in Section 3, has been implemented in the LLVM compiler [17]. We have logged overflows in a vast number of programs, and in this paper we focus on SPEC CPU 2006. We have rediscovered the integer overflows recently observed by Dietz *et al.* [11]. The performance of our instrumentation library, even without the support of range analysis, is within the 5% runtime overhead of Brumley *et al.*'s [4] state-of-the-art algorithm. The range analysis halves down this overhead. Our static analysis algorithm avoids 24.93% of the overflow checks created by the dynamic instrumentation framework. With this support, the instrumented SPEC programs are only 1.73% slower. Therefore, we show in this paper that securing programs against integer overflows is very cheap.

2. Range Analysis

Following Gawlitza *et al.*'s notation [14], we shall be performing arithmetic operations over the complete lattice $\mathcal{Z} = \mathbb{Z} \cup \{-\infty, +\infty\}$, where the ordering is naturally given by $-\infty < \ldots < -2 < -1 < 0 < 1 < 2 < \ldots + \infty$. For any $x > -\infty$ we define:

$$\begin{array}{ll} x + \infty = \infty & x - \infty = -\infty, x \neq +\infty \\ x \times \infty = \infty \text{ if } x > 0 & x \times \infty = -\infty \text{ if } x < 0 \\ 0 \times \infty = 0 & (-\infty) \times \infty = \text{ not defined} \end{array}$$

Notice that $(\infty - \infty)$ is not well-defined. From the lattice $\mathcal{Z}$ we define the product lattice $\mathcal{Z}^2$ as follows:

$$\mathcal{Z}^2 = \{\emptyset\} \cup \{[z_1, z_2] |\ z_1, z_2 \in \mathcal{Z},\ z_1 \leq z_2,\ -\infty < z_2\}$$

This interval lattice is partially ordered by the subset relation, which we denote by "$\sqsubseteq$". Range intersection, "$\sqcap$", is defined by:

$$[a_1, a_2] \sqcap [b_1, b_2] = \begin{cases} [\max(a_1, b_1), \min(a_2, b_2)], \text{ if } a_1 \leq b_1 \leq a_2 \\ \quad \text{or } b_1 \leq a_1 \leq b_2 \\ [a_1, a_2] \sqcap [b_1, b_2] = \emptyset, \text{ otherwise} \end{cases}$$

And range union, "$\sqcup$", is given by:

$$[a_1, a_2] \sqcup [b_1, b_2] = [\min(a_1, b_1), \max(a_2, b_2)]$$

Given an interval $\iota = [l, u]$, we let $\iota_\downarrow = l$, and $\iota_\uparrow = u$, where $\iota_\downarrow$ is the lower bound and $\iota_\uparrow$ is the upper bound of a variable. We let $\mathcal{V}$ be a set of constraint variables, and $I : \mathcal{V} \mapsto \mathcal{Z}^2$ a mapping from these variables to intervals in $\mathcal{Z}^2$. Our objective is to solve a constraint system $\mathcal{C}$, formed by constraints such as those seen in Figure 1(left). We let the

$$\begin{array}{ll}
Y = [l, u] & e(Y) = [l, u] \\[2ex]
Y = \phi(X_1, X_2) & \dfrac{I[X_1] = [l_1, u_1] \qquad I[X_2] = [l_2, u_2]}{e(Y) = [l_1, u_1] \sqcup [l_2, u_2]} \\[3ex]
Y = X_1 + X_2 & \dfrac{I[X_1] = [l_1, u_1] \qquad I[X_2] = [l_2, u_2]}{e(Y) = [l_1 + l_2, u_1 + u_2]} \\[3ex]
Y = X_1 \times X_2 & \dfrac{L = \{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\} \\ I[X_1] = [l_1, u_1] \qquad I[X_2] = [l_2, u_2]}{e(Y) = [\min(L), \max(L)]} \\[3ex]
Y = aX + b & \dfrac{I[X] = [l, u] \qquad k_l = al + b \qquad k_u = au + b}{e(Y) = [\min(k_l, k_u), \max(k_l, k_u)]} \\[3ex]
Y = X \sqcap [l', u'] & \dfrac{I[X] = [l, u]}{e(Y) \leftarrow [l, u] \sqcap [l', u']}
\end{array}$$

Figure 1. A suite of constraints that produce an instance of the range analysis problem.

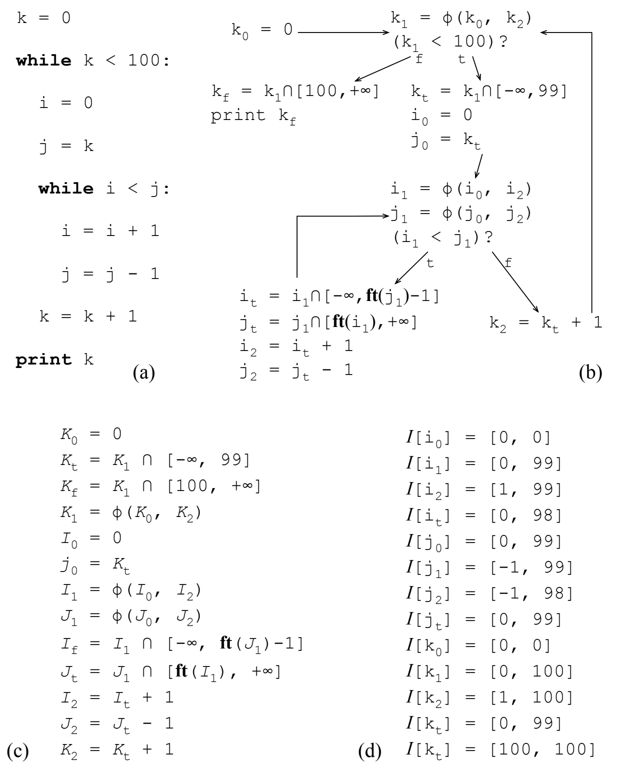
ϕ-functions be as defined by Cytron *et al.* [10]: they join different variable names into a single definition. Figure 1(right) defines a valuation function e on the interval domain. Armed with these concepts, we define the range analysis problem as follows:

DEFINITION: RANGE ANALYSIS PROBLEM
Input: a set $\mathcal{C}$ of constraints ranging over a set $\mathcal{V}$ of variables.
Output: a mapping I such that, for any $V \in \mathcal{V}$, e(V) = I[V].

We will use the program in Figure 2(a) to illustrate our range analysis. Figure 2(b) shows the same program in e-SSA form [2], and Figure 2(c) outlines the constraints that we extract from this program. There is a correspondence between instructions and constraints. Our analysis is sparse [7]; thus, it associates one, and only one, constraint with each integer variable. A possible solution to the range analysis problem, as obtained via the techniques that we will introduce in Section 2.1, is given in Figure 2(d).

2.1 Range Propagation

Our range analysis algorithm works in a number of steps. Firstly, we convert the program to a representation that gives us subsidies to perform a sparse analysis. We have tested our algorithm with two different representations, as we discuss in Section 2.3. Secondly, we extract constraints from the program representation. Thirdly, we build a constraint graph, following the strategy pointed by Su and Wagner [25]. However, contrary to them, in a next phase we find the strongly connected components in this graph, collapse them into super-nodes, and sort the resulting digraph topologi-



```
k = 0
while k < 100:
  i = 0
  j = k
  while i < j:
    i = i + 1
    j = j - 1
  k = k + 1
print k
```
(a)

```
k0 = 0  →  k1 = ϕ(k0, k2)
           (k1 < 100)?
     f                    t
kf = k1∩[100,+∞]     kt = k1∩[-∞,99]
print kf             i0 = 0
                     j0 = kt
                     i1 = ϕ(i0, i2)
                     j1 = ϕ(j0, j2)
                     (i1 < j1)?
            t                    f
it = i1∩[-∞,ft(j1)-1]
jt = j1∩[ft(i1),+∞]         k2 = kt + 1
i2 = it + 1
j2 = jt - 1
```
(b)

```
K0 = 0
Kt = K1 ∩ [-∞, 99]
Kf = K1 ∩ [100, +∞]
K1 = ϕ(K0, K2)
I0 = 0
j0 = Kt
I1 = ϕ(I0, I2)
J1 = ϕ(J0, J2)
It = I1 ∩ [-∞, ft(J1)-1]
Jt = J1 ∩ [ft(I1), +∞]
I2 = It + 1
J2 = Jt - 1
K2 = Kt + 1
```
(c)

```
I[i0] = [0, 0]
I[i1] = [0, 99]
I[i2] = [1, 99]
I[it] = [0, 98]
I[j0] = [0, 99]
I[j1] = [-1, 99]
I[j2] = [-1, 98]
I[jt] = [0, 99]
I[k0] = [0, 0]
I[k1] = [0, 100]
I[k2] = [1, 100]
I[kt] = [0, 99]
I[kt] = [100, 100]
```
(d)

Figure 2. (a) Example program. (b) Control Flow Graph in e-SSA form. (c) Constraints that we extract from the program. (d) Possible solution to the range analysis problem.

cally. Finally, for each strong component, we apply a three-phase approach to determine the ranges of the variables.

Building the Constraint Graph. Given a set $\mathcal{C}$ of constraints, which define and/or use constraint variables from a set $\mathcal{V}$, we build a constraint graph $G = (\mathcal{C} \cup \mathcal{V}, E)$. The vertices in $\mathcal{C}$ are the *constraint nodes*, and the nodes in $\mathcal{V}$ are the *variable nodes*. If $V \in \mathcal{V}$ is used in constraint $C \in \mathcal{C}$, then we create an edge $\overrightarrow{VC}$. If constraint $C \in \mathcal{C}$ defines variable $\mathcal{V} \in V$, then we create an edge $\overrightarrow{CV}$. Figure 3 shows the constraint graph that we build for the program in Figure 2(b). Our algorithm introduces the notion of *future ranges*, which we use to extract range information from comparisons between variables. In Section 2.3 we explain how futures are created. If V is used by constraint C as the input of a future range, then the edge from V to C represents what Ferrante *et al.* call a *control dependence* [13, p.323]. We use dashed lines to represent these edges. All the other edges denote *data dependences* [13, p.322]. As we will show later, control dependence edges increase the precision of our algorithm to solve future bounds.

Propagating Ranges in Topological Order. After building the constraint graph, we find its strongly connected components. We collapse these components in super nodes, and then propagate ranges along the resulting digraph. This approach is essential for scalability, because all the complexity of our algorithm lies in the resolution of strong components. Our tests show that the vast majority of the strongly connected components are singletons. For instance, 99.11% of the SCCs in SPEC CPU 2006 dealII (`447.dealII`) have only one node. Moreover, the composite components usually contain a small number of nodes. As an example, the largest component in dealII has 2,131 nodes, even though dealII's constraint graph has over one million nodes. This large SCC exists due to a long chain of mutually recursive function calls.

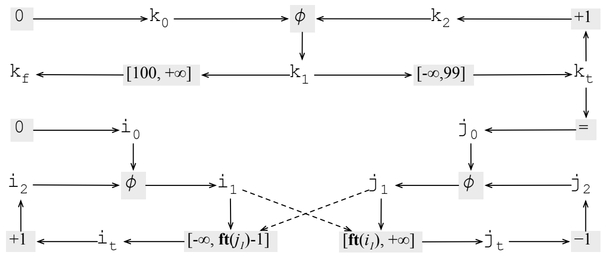


Figure 3. The constraint graph that we build for the program in Figure 2(b).

2.2 A Three-Phase Approach to Solve Strong Components

We find the ranges of the variables in each strongly connected component in three phases. First we determine the growth pattern of each variable in the component via widening. In the second step, we replace future bounds by actual limits. Finally, a narrowing phase starting from conditional tests improves the precision of our results.

Widening: we start solving constraints by determining how each program variable might grow. For instance, if a variable is only updated by sums with positive numbers, then it only grows up. If, instead, a variable is only updated by sums with negative numbers, then it grows down. Some variables can also grow in both directions. We discover these growth patterns by abstractly interpreting the constraints that constitute the strongly connected component. We ensure termination via a widening operator. Our implementation uses *jump-set widening*, which is typically used in range analysis [22, p.228]. This operator is a generalization of Cousot and Cousot's original widening operator [8], which we describe below:

$$I[Y] = \begin{cases} \text{if } I[Y] = [\bot, \bot] \text{ then } e(Y) \\ \text{elif } e(Y)_\downarrow < I[Y]_\downarrow \text{ and } e(Y)_\uparrow > I[Y]_\uparrow \text{ then } [-\infty, \infty] \\ \text{elif } e(Y)_\downarrow < I[Y]_\downarrow \text{ then } [-\infty, I[Y]_\uparrow] \\ \text{elif } e(Y)_\uparrow > I[Y]_\uparrow \text{ then } [I[Y]_\downarrow, \infty] \end{cases}$$

We let $[l, u]_\downarrow = l$ and $[l, u]_\uparrow = u$. We let $\bot$ denote non-initialized intervals, so that $[\bot, \bot] \sqcup [l, u] = [l, u]$. This operation only happens at ϕ-nodes, because we evaluate constraints in topological order. The map I and the abstract eval-

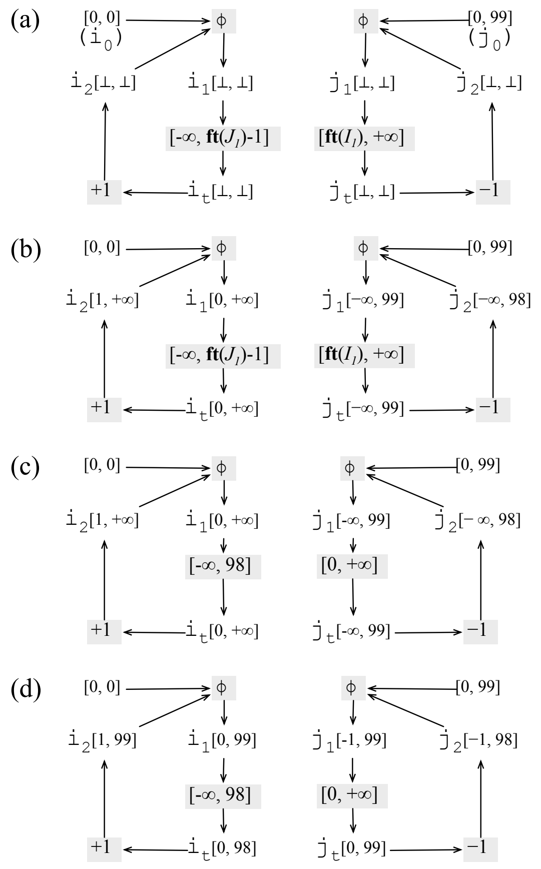


Figure 4. Four snapshots of the last SCC of Figure 3. (a) After removing control dependence edges. (b) After running the growth analysis. (c) After fixing the intersections bound to futures. (d) After running the narrowing analysis.

uation function e are determined as in Figure 1. We have an implementation of e for each operation that the target programming language provides. Our current LLVM implementation has 18 different instances of e, including signed and unsigned addition, subtraction, multiplication and division, plus truncation, the bitwise integer operators and ϕ-functions.

If we use the widening operator above, then the abstract state of any constraint variable can only change three times, e.g., $[\bot, \bot] \to [c_1, c_2] \to [c_1, \infty] \to [-\infty, \infty]$, or $[\bot, \bot] \to [c_1, c_2] \to [-\infty, c_2] \to [-\infty, \infty]$. Therefore, we determine the growth behavior of each constraint variable in a strong component in linear time on the number of constraints in that component. Figure 4(b) shows the abstract state of the variables in the largest SCC of the graph in Figure 3. As we see in the figure, this step of our algorithm has been able to determine that variables $\mathtt{i}_1$, $\mathtt{i}_2$ and $\mathtt{i}_t$ can only increase, and that variables $\mathtt{j}_1$, $\mathtt{j}_2$ and $\mathtt{j}_t$ can only decrease.

Future resolution: the next phase of the algorithm to determine intervals inside a strong component consists in replacing futures by actual bounds, a task that we accomplish by using the rules below:

$$\frac{Y = X \sqcap [l, \mathbf{ft}(V) + c] \qquad I[V]_\uparrow = u}{Y = X \sqcap [l, u + c]} \qquad u, c \in \mathbb{Z} \cup \{-\infty, \infty\}$$

$$\frac{Y = X \sqcap [\mathbf{ft}(V) + c, u] \qquad I[V]_\downarrow = l}{Y = X \sqcap [l + c, u]} \qquad l, c \in \mathbb{Z} \cup \{-\infty, \infty\}$$

To correctly replace a future $\mathbf{ft}(V)$ that limits a constraint variable V', we need to have already applied the growth analysis onto V. Had we considered only data dependence edges, then it would be possible that V''s strong component would be analyzed before V's. However, because of control dependence edges, this case cannot happen. The control dependence edges ensure that any topological ordering of the constraint graph either places V before V', or places these nodes in the same strongly connected component. For instance, in Figure 3, variables j_1 and i_t are in the same SCC only because of the control dependence edges. Figure 4(c) shows the result of resolving futures in our running example. The information that we acquire from the growth analysis is essential in this phase. For instance, the growth analysis has found out that the value stored in variable $\mathtt{i}_1$ can only increase. Given that this variable is assigned the initial value zero, we can replace $\mathbf{ft}(I_1)$ with this value.

Narrowing: the last step that we apply on the strongly connected component is the narrowing phase. In this step we use values extracted from conditional tests to restrict the bounds of the constraint variables. We use the narrowing operator firstly proposed by Cousot and Cousot [8], which we show below:

$$I[Y] = \begin{cases} \text{if } I[Y]_\downarrow = -\infty \text{ and } e(Y)_\downarrow > -\infty \text{ then } [e(Y)_\downarrow, I[Y]_\uparrow] \\ \text{elif } I[Y]_\uparrow = \infty \text{ and } e(Y)_\uparrow < \infty \text{ then } [I[Y]_\downarrow, e(Y)_\uparrow] \\ \text{elif } I[Y]_\downarrow > e(Y)_\downarrow \text{ then } [e(Y)_\downarrow, I[Y]_\uparrow] \\ \text{elif } I[Y]_\uparrow < e(Y)_\uparrow \text{ then } [I[Y]_\downarrow, e(Y)_\uparrow] \end{cases}$$

Figure 4(d) shows the result of our narrowing operator in our running example. Ranges improve due to the two conditional tests in the program. Firstly, we have that $I[I_t] = I[I_1] \sqcap [-\infty, 98]$, which gives us that $I[I_t] = [0, 98]$. We also have that $I[J_t] = I[J_1] \sqcap [0, \infty]$, giving $I[J_t] = [0, 99]$. From these new intervals, we can narrow the ranges bound to the other constraint variables.

The combination of widening, futures and narrowing, plus use of strong components gives us, in this example, a very precise solution. We emphasize that finding this tight solution was only possible because of the topological ordering of the constraint graph in Figure 3. Upon meeting the constraint graph's last SCC, shown in Figure 4, we had already determined that the interval $[0, 0]$ is bound to i_0 and that the interval $[0, 99]$ is bound to j_0, as we show in Figure 4(a). Had we applied the widening operator onto the whole graph, then we would have found out that variable j_1 is bound to $[-\infty, +\infty]$. This imprecision happens because, on one hand j_1's interval is influenced by k_t's, which is upper bounded by $+\infty$. On the other hand j_1 is part of a decreasing cycle of dependences formed by variables j_t and j_2 in addition to itself. Therefore, if we had applied the widening phase over the entire program followed by a global narrowing phase, then we would not be able to recover some of widening's precision loss. However, because in this example we only analyze j's SCC after we have analyzed k's, k only contribute the constant range $[0, 99]$ to j_0.

2.3 Live Range Splitting Strategies

A dense dataflow analysis associates information, i.e., a point in a lattice, with each pair formed by a variable plus a program point. If this information is invariant along every program point where the variable is alive, then we can associate the information with the variable itself. In this case, we say that the dataflow analysis is *sparse* [7]. A dense dataflow analysis can be transformed into a sparse one via a suitable intermediate representation. A compiler builds this intermediate representation by splitting the live ranges of variables at the program points where the information associated with these variables might change. To split the live range of a variable v, at a program point p, we insert a copy $v' = v$ at p, and rename every use of v that is dominated by p. In this paper we have experimented with two different live range splitting alternatives.

The first strategy is the *Extended Static Single Assignment* (e-SSA) form, proposed by Bodik *et al.* [2]. We build the e-SSA representation by splitting live ranges at definition sites – hence it subsumes the SSA form – and at conditional tests. The program in Figure 2(b) is in e-SSA form. Let $(v < c)$? be a conditional test, and let l_t and l_f be labels in the program, such that l_t is the target of the test if the condition is true, and l_f is the target when the condition is false. We split the live range of v at any of these points if at least one of two conditions is true: (i) l_f or l_t dominate any use of v; (ii) there exists a use of v at the dominance frontier of l_f or l_t. For the notions of dominance and dominance-frontier, see Aho *et al.* [1, p.656]. To split the live range of v at l_f we insert at this program point a copy $v_f = v \sqcap [c, +\infty]$, where v_f is a fresh name. We then rename every use of v that is dominated by l_f to v_f. Dually, if we must split at l_t, then we create at this point a copy $v_t = v \sqcap [-\infty, c - 1]$, and rename variables accordingly. If the conditional uses two variables, e.g., $(v_1 < v_2)$?, then we create intersections bound to *futures*. We insert, at l_f, $v_{1f} = v_1 \sqcap [\mathbf{ft}(v_2), +\infty]$, and $v_{2f} = v_2 \sqcap [-\infty, \mathbf{ft}(v_1)]$. Similarly, at l_t we insert $v_{1v} = v_1 \sqcap [-\infty, \mathbf{ft}(v_2) - 1]$ and $v_{2v} = v_2 \sqcap [\mathbf{ft}(v_1) + 1, +\infty]$. A variable v can never be associated with a future bound to itself, e.g., $\mathbf{ft}(v)$. This invariant holds because whenever the e-SSA conversion associates a variable u with $\mathbf{ft}(v)$, then u is a fresh name created to split the live range of v.

The second intermediate representation consists in splitting live ranges at (i) definition sites – it subsumes SSA, (ii) at conditional tests – it subsumes e-SSA, and at some use

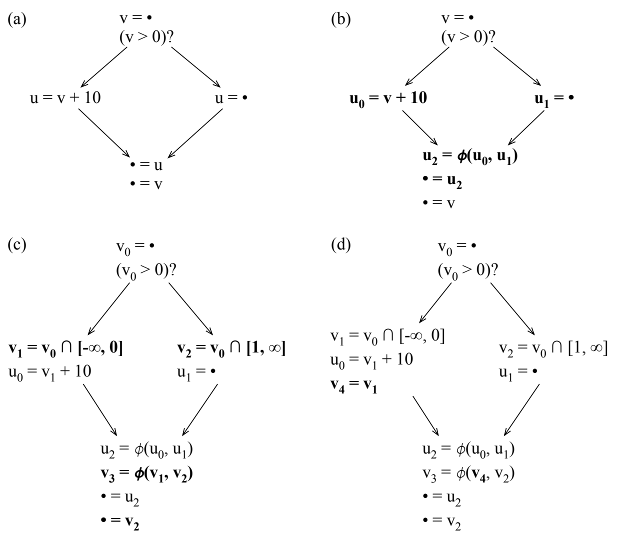


Figure 5. (a) Example program. (b) SSA form [10]. (c) e-SSA form [2]. (d) u-SSA form.

sites. This representation, which we henceforth call u-SSA, is only valid if we assume that integer overflows cannot happen. We can provide this guarantee by using our dynamic instrumentation to terminate a program in face of an overflow. The rationale behind u-SSA is as follows: we know that past an instruction such as $v = u + c, c \in \mathbb{Z}$ at a program point p, variable u must be less than $MaxInt - c$. If that were not the case, then an overflow would have happened and the program would have terminated. Therefore, we split the live range of u past its use point p, producing the sequence $v = u + c; u' = u$, and renaming every use of u that is dominated by p to u'. We then associate u' with the constraint $I[U'] \sqsubseteq I[U] \sqcap [-\infty, MaxInt - c]$.

Figure 5 compares the u-SSA form with the SSA and e-SSA intermediate program representations. We use the notation $v = \bullet$ to denote a definition of variable v, and $\bullet = v$ to denote a use of it. Figure 5(b) shows the example program converted to the SSA format. Different definitions of variable u have been renamed, and a ϕ-function joins these definitions into a single name. The SSA form sparsifies a dataflow analysis that only extracts information from the definition sites of variables, such as constant propagation. Figure 5(c) shows the same program in e-SSA form. This time we have renamed variable v right after the conditional test where this variable is used. The e-SSA form serves dataflow analyses that acquire information from definition sites and conditional tests. Examples of these analyses include array bounds checking elimination [2] and traditional implementations of range analyses [15, 23]. Finally, Figure 5(d) shows our example in u-SSA form. The live range of variable v_1 has been divided right after its use. This representation assists analyses that learn information from the

Instruction	Dynamic Check
$x = o_1 \;+_s\; o_2$	$(o_1 > 0 \wedge o_2 > 0 \wedge x < 0)\;\vee$ $(o_1 < 0 \wedge o_2 < 0 \wedge x > 0)$
$x = o_1 \;+_u\; o_2$	$x < o_1 \vee x < o_2$
$x = o_1 \;-_s\; o_2$	$(o_1 < 0 \vee o_2 > 0 \vee x > 0)\;\vee$ $(o_1 > 0 \vee o_2 < 0 \vee x < 0)$
$x = o_1 \;-_u\; o_2$	$o_1 < o_2$
$x = o_1 \;\times_{u/s}\; o_2$	$x \neq 0 \Rightarrow x \div o_1 \neq o_2$
$x = o_1 \ll n$	$(o_1 > 0 \wedge x < o_1) \vee (o_1 < 0 \wedge n \neq 0)$
$x = \downarrow_n\; o_1$	$\text{cast}(x, \text{type}(o_1)) \neq o_1$

Figure 6. Overflow checks. We use $\downarrow_n$ for the operation that truncates to n bits. The subscript s indicates a signed instruction; the subscript u indicate an unsigned operation.

way that variables are used, and propagate this information forwardly.

3. The Dynamic Instrumentation Library

We have implemented our instrumentation library as a LLVM transformation pass; thus, we work at the level of the compiler's intermediate representation [1]. This is in contrast to previous work, which either transforms the source code [11], or the machine dependent code [4]. We work at the intermediate representation level to be able to couple our library with static analyses, such as the algorithm that we described in Section 2. Our instrumentation works by identifying the instructions that may lead to an overflow, and inserting assertions after those instructions. The LLVM IR has five instructions that may lead to an overflow: ADD, SUB, MUL, TRUNC (also bit-casts) and SHL (left shift). Figure 6 shows the dynamic tests that we perform to detect overflows.

The instrumentation that we insert is mostly straightforward. We discuss in the rest of this section a few interesting cases. When dealing with an unsigned SUB instruction, e.g, $x = o_1 \;-_u\; o_2$, then a single check is enough to detect the bad behavior: $o_1 < o_2$. If o_2 is greater than o_1, then we assume that it is a bug to try to represent a negative number in unsigned arithmetics. Regarding multiplication, e.g., $x = o_1 \times o_2$, if $o_1 = 0$, then this operation can never cause an overflow. This test is necessary, because we check integer overflows in multiplication via the inverse operation, e.g., integer division. Thus, the test prevents a division by zero from happening. The TRUNC instruction, e.g., $x = \downarrow_n\; o_1$ assigns to x the n least significant bits of o_1. The dynamic check, in this case, consists in expanding x to the datatype of o_1 and comparing the expanded value with o_1. The LLVM IR provides instructions to perform these type expansions. Note that our instrumentation catches any truncation that might result in data loss, even if this loss is benign. To make the dynamic checks more liberal, we give users the possibility of disabling tests over truncations.

[1] `http://llvm.org/docs/LangRef.html`

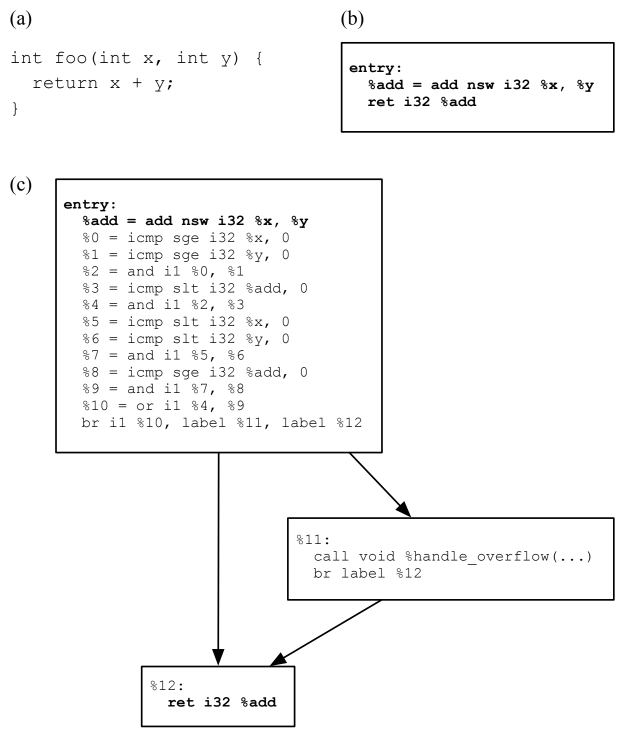
	ADD	SUB	MUL	SHL	TRUNC
signed	12	12	6	8	3
unsigned	4	2	6	2	3

Figure 7. Number of instructions used in each check.

Practical Considerations. Our instrumentation library inserts new instructions into the target program. Although the dynamic check depends on the instruction that is instrumented, the general modus operandi is the same. Dynamic tests check for overflows after they happen. The code that we insert to detect the overflow diverts the program flow in case such an event takes place. Figure 8 shows an actual control flow graph, before and after the instrumentation. Clearly the instrumented program will be larger than the original code. Figure 7 shows how many LLVM instructions are necessary to instrument each arithmetic operation. These numbers do not include the instructions necessary to handle the overflow itself, e.g., block `%11` in Figure 8, as this code is not in the program's main path. Nevertheless, as we show empirically, this growth is small when compared to the total size of our benchmarks, because most of the instructions in these programs do not demand instrumentation. Furthermore, none of the instructions used to instrument integer arithmetics access memory. Therefore, the overall slowdown that the instrumentation causes is usually small, and the experiments in Section 4.2 confirm this observation.

Which actions are performed once the overflow is detected depends on the user, who has the option to overwrite the `handle_overflow` function in Figure 8. Our library gives the user three options to handle overflows. First option: no-op. This option allows us to verify the slowdown produced by the new instructions. Second option: logging. This is the standard option, and it preserves the behavior of the instrumented program. Whenever an overflow is detected, we print `Overflow detected in FileName.cpp, line X.` in the standard error stream. Third option: abort. This option terminates the program once an overflow is detected. Thus, it disallows undefined behavior due to integer overflows, and gives us the opportunity to use the u-SSA form to get extra precision.

Using the static analysis to avoid some overflow checks. Our library can, optionally, use the range analysis to avoid having to insert some overflow checks into the instrumented program. We give the user the possibility of calling the range analysis with either the e-SSA or the u-SSA live range


(a)

```
int foo(int x, int y) {
  return x + y;
}
```

(b)

```
entry:
  %add = add nsw i32 %x, %y
  ret i32 %add
```

(c)

```
entry:
  %add = add nsw i32 %x, %y
  %0 = icmp sge i32 %x, 0
  %1 = icmp sge i32 %y, 0
  %2 = and i1 %0, %1
  %3 = icmp slt i32 %add, 0
  %4 = and i1 %2, %3
  %5 = icmp slt i32 %x, 0
  %6 = icmp slt i32 %y, 0
  %7 = and i1 %5, %6
  %8 = icmp sge i32 %add, 0
  %9 = and i1 %7, %8
  %10 = or i1 %4, %9
  br i1 %10, label %11, label %12
```

```
%11:
  call void %handle_overflow(...)
  br label %12
```

```
%12:
  ret i32 %add
```

Figure 8. (a) A simple C function. (b) The same function converted to the LLVM intermediate representation. (c) The instrumented code. The boldface lines were part of the original program.

splitting strategies. Our static analysis classifies variables into four categories, depending on their bounds:

- **Safe**: a variable is safe if its bounds are fully contained inside its declared type. For instance, if x is declared as an unsigned 8-bits integer, then x is safe if its bounds are within the interval $[0, 255]$.
- **Suspicious**: we say that a variable is suspicious if its bounds go beyond the interval of its declared type, but the intersection between these two ranges is non-empty. For instance, the same variable x would be suspicious if $I[x] = [0, 257]$, as $I[x]_\uparrow > \texttt{uint8}_\uparrow$.
- **Uncertain**: we classify a variable as uncertain if at least one of its limits is unbounded. Our variable x would be uncertain if $I[x] = [0, \infty]$. We distinguish suspicious from uncertain variables because we speculate that actual overflows are more common among elements in the former category.
- **Buggy**: a variable is buggy if the intersection between its inferred range and the range of its declared type is empty. This is a definitive case of an overflow. Continuing with our example, x would be buggy if, for instance, $I[x] = [257, \infty]$, given that $[257, \infty] \sqcap [0, 255] = \emptyset$.

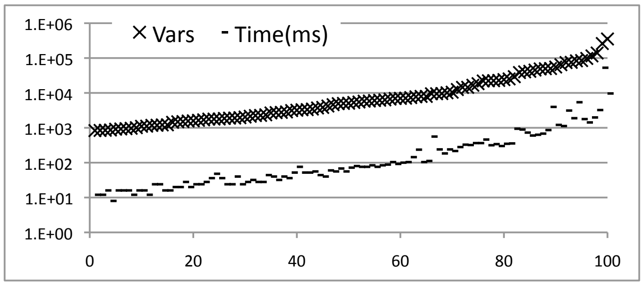


Figure 9. Correlation between program size (number of var nodes in constraint graphs) and analysis runtime (ms). Each point represents a benchmark. Coefficient of determination = 0.967.

Independent on the arithmetic instruction that is being analyzed, the instrumentation library performs the same test: if the result x of an arithmetic instruction such as $x = o_1 +_s o_2$ is safe, then the overflow check is not necessary, otherwise it must be created.

4. Experimental Results

We have implemented our range analysis algorithm in LLVM 3.0, and have run experiments on a Intel quad core CPU with a 2.40GHz clock, and 3.6GB of RAM. Each core has a 4,096KB L1 cache. We have used Linux Ubuntu 10.04.4. Our implementation of range analysis has 3,958 lines of commented C++ code, our e/u-SSA conversion module has 672 lines, and our instrumentation pass has 762 lines. We have analyzed 428 C programs that constitute the LLVM test suite plus the integer benchmarks in SPEC CPU 2006. Together, these programs contain 4.76 million assembly instructions. This section has two main goals. First, we want to show that our range analysis is fast and precise. Second, we want to demonstrate the effectiveness of our framework to detect integer overflows.

4.1 Static Range Analysis

Time and Memory Complexity: Figure 9 compares the time to run our range analysis with the size of the input programs. We show data for the 100 largest benchmarks in our test suite, considering the number of variable nodes in the constraint graph. We perform function inlining before running our analysis. Each point in the X line corresponds to a benchmark. We analyze the smallest benchmark in this set, `Prolangs-C/deriv2`, which has 1,131 variable nodes in the constraint graph, in 20ms. We take 9.91 sec to analyze our largest benchmark, `403.gcc`, which, after function inlining, has 1,419,456 assembly instructions, and a constraint graph with 679,652 variable nodes. For this data set, the coefficient of determination (R^2) is 0.967, which provides very strong evidence about the linear asymptotic complexity of our implementation.

The experiments also reveal that the memory consumption of our implementation grows linearly with the program

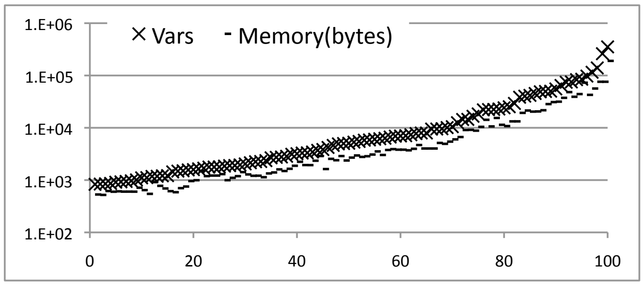


Figure 10. Comparison between program size (number of var nodes in constraint graphs) and memory consumption (KB). Coefficient of determination = 0.994.

size. Figure 10 plots these two quantities. The linear correlation, in this case, is even stronger than that found in Figure 9: the coefficient of determination is 0.994. The figure only shows our 100 largest benchmarks. Again, SPEC `403.gcc` is the largest benchmark, requiring 265,588KB to run. Memory includes stack, heap and the executable program code.

Precision: Our implementation of range analysis offers a good tradeoff between precision and runtime. Lakhdar *et al.*'s relational analysis [16], for instance, takes about 25 minutes to go over a program with almost 900 basic blocks. We analyze programs of similar size in less than one second. We do not claim our approach is as precise as such algorithms, even though we are able to find exact bounds to 4/5 of the examples presented in [16]. On the contrary, we present a compromise between precision and speed that scales to very large programs. Nevertheless, our results are not trivial. We have implemented a dynamic profiler that measures, for each variable, its upper and lower limits, given an execution of the target program. Figure 11 compares our results with those measured dynamically for the Stanford benchmark, which is publicly available in the LLVM test suite. We chose Stanford because these benchmarks do not read data from external files; hence, imprecisions are due exclusively to library functions that we cannot analyze.

We have classified the bounds estimated by the static analysis into four categories. The first category, called 1, contains tight bounds: during program execution, the variable has been assigned an upper, or lower limit, that equals the limit inferred statically. The second category, called n, contains the bounds that are within twice the value inferred statically. For instance, if the range analysis estimates that a variable v is in the range $[0, 100]$, and during the execution the dynamic profiler finds that its maximum value is 51, then v falls into this category. The third category, n^2, contains variables whose actual value is within a quadratic factor of the estimated value. In our example, v's upper bound would have to be at most 10 for it to be in this category. Finally, the fourth category contains variables whose estimated value lays outside a quadratic factor of the actual value. We call this category *imprecise*, and it contains mostly the limits

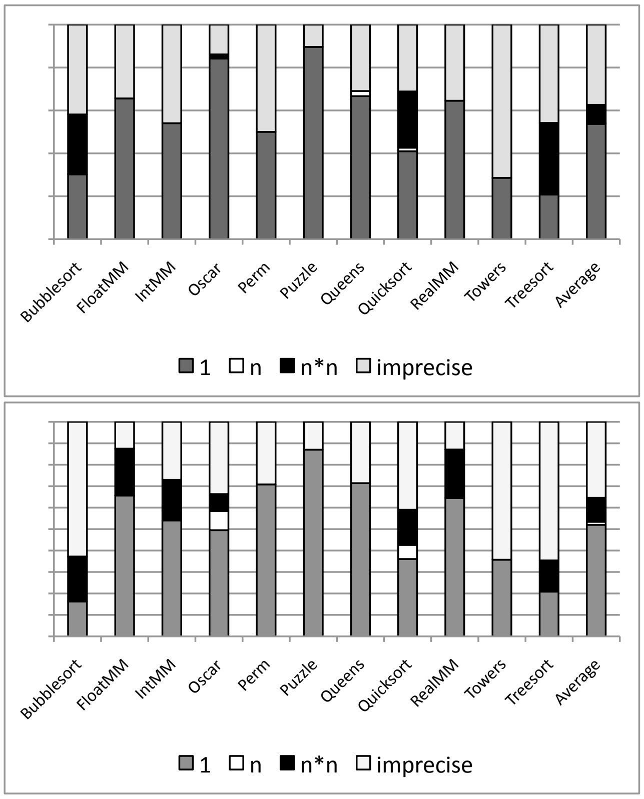


Figure 11. (Upper) Comparison between static range analysis and dynamic profiler for upper bounds. (Lower) Comparison between static range analysis and dynamic profiler for lower bounds.

that our static analysis has marked as either $+\infty$ or $-\infty$. As we see in Figure 11, 54.11% of the lower limits that we have estimated statically are exact. Similarly, 51.99% of our upper bounds are also tight. The figure also shows that, on average, 37.39% of our lower limits are imprecise, and 35.40% of our upper limits are imprecise. This result is on par with those obtained by more costly analysis, such as Stephenson *et al.*'s [24].

4.2 The Instrumentation Library

We have executed the instrumented programs of the integer benchmarks of SPEC 2006 CPU to probe the overhead imposed by our instrumentation. These programs have been executed with their "test" input sets. We have not been able to run the binary that LLVM produces for SPEC's `gcc` in our environment, even without any of our transformations, due to an incompatible `ctype.h` header. In addition, we have not been able to collect the statistics about the overflows that occurred in SPEC's `bzip2`, because the log file was too large. We verified more than 3,000,000,000 overflows in this program. Figure 12 shows the percentage of instructions that we instrument, without the intervention of the range analysis. The number of instrumented instructions is relatively low, compared to the total number of instructions, because we only instrument six different LLVM bitcodes, in a set of 57 opcodes, not counting intrinsics. Figure 12 also shows how many instructions have caused overflows. On the average, 4.90% of the instrumented sites have caused integer overflows.

Benchmark	#I	#II	#II/#I	#O
470.lbm	13,724	1,142	8.32%	0
433.milc	44,236	1,602	3.62%	11
444.namd	100,276	3,234	3.23%	12
447.dealII	1,381,408	36,157	2.62%	50
450.soplex	136,367	3,158	2.32%	13
464.h264ref	271,627	13,846	5.10%	167
473.astar	19,243	857	4.45%	0
458.sjeng	54,051	2,504	4.63%	68
429.mcf	4,725	165	3,49%	8
471.omnetpp	203,201	1,972	0.97%	2
403.gcc	1,419,456	18,669	1.32%	N/A
445.gobmk	308,475	14,129	4.58%	4
462.libquantum	16,297	928	5.69%	7
401.bzip2	38,831	2,158	5.56%	N/A
456.hmmer	114,136	4,001	3.51%	0
Total (Average)	275,070	6,968	3.96%	

Figure 12. Instrumentation without support of range analysis. #I: number of LLVM bitcode instructions in the original program. #II: number of instructions that have been instrumented. #O: number of instructions that actually overflowed in the dynamic tests.

Figure 13 shows how many checks our range analysis avoids. Some results are expressive: the range analysis avoids 1,138 out of 1,142 checks in `470.lbm`. In other benchmarks, such as in `429.mcf`, we have been able to avoid only 1 out of 165 tests. In general we fare better in programs that bound input sizes via conditional tests, as `lbm` does. Using u-SSA, instead of e-SSA, adds a negligible improvement onto our results. We speculate that this improvement is small because variables tend to be used a small number of times. Benoit *et al.* [3] have demonstrated that the vast majority of all the program variables are used less than five times in the program code. The u-SSA form only helps to avoid checks upon variables that are used more than once.

Figure 14 shows how our range analysis classifies instructions. Out of all the 102,790 instructions that we have instrumented in SPEC, 3.92% are suspicious, 17.19% are safe, and 78.89% are uncertain. This means that we found precise bounds to $3.92 + 17.19 = 21.11\%$ of all the program variables, and that 78.98% of them are bound to intervals with at least one unknown limit. We had, at first, speculated that overflows would be more common among suspicious instructions, as their bounds, inferred statically, go beyond the limits of their declared types. However, our experiments did not let us confirm this hypothesis. To check the correctness of our approach, we have instrumented the safe instructions, but have not observed any overflow caused by them.

Benchmark	#II	#E	%(II, E)	#U	%(II, U)
lbm	1,142	4	99.65%	4	99.65%
milc	1,602	1,070	33.21%	1,065	33.52%
namd	3,234	2,900	10.33%	2,900	10.33%
dealII	36,157	29,870	17.39%	28,779	20.41%
soplex	3,158	2,927	7.31%	2,897	8.26%
h264ref	13,846	11,342	18.38%	11,301	18.08%
astar	857	808	5.72%	806	5.95%
sjeng	2,504	2,354	5.99%	2,190	12.54%
mcf	165	164	0.61%	164	0.61%
omnetpp	1,972	1,313	33.42%	1,313	33.42%
gcc	18,669	15,282	18.14%	15,110	19.06%
gobmk	14,129	12,563	11.08%	12,478	11.69%
libquantum	928	820	11.64%	817	11.96%
bzip2	2,158	1,966	8.90%	1,966	8.90%
hmmer	4,001	3,346	16.37%	3,304	17.42%
Total	104,522	86,688		85,135	

Figure 13. Instrumentation library with support of static range analysis. #II: number of instructions that have been instrumented without range analysis. #E: number of instructions instrumented in the e-SSA form program. #U: number of instructions instrumented in the u-SSA form program.

Bench	#Sf	#S	#U	#SO	#SO/#S	#UO	#UO/#U
lbm	1138	0	4	0	0,00%	0	0,00%
milc	536	17	1048	0	0,00%	11	1,05%
namd	334	480	2420	0	0,00%	12	0,50%
dealII	6188	39	28740	0	0,00%	50	0,17%
soplex	229	16	2881	0	0,00%	13	0,45%
h264ref	2539	1195	10147	7	0,59%	160	1,58%
astar	48	11	795	0	0,00%	0	0,00%
sjeng	150	213	1977	0	0,00%	68	3,44%
mcf	1	0	164	0	0,00%	8	4,88%
omnetpp	659	25	1288	1	4,00%	1	0,07%
gcc	3365	1045	14065	N/A	N/A	N/A	N/A
gobmk	1509	742	11736	0	0,00%	4	0,03%
libqtum	104	12	805	0	0,00%	7	0,87%
bzip2	192	40	1926	N/A	N/A	N/A	N/A
hmmer	663	222	3082	0	0,00%	0	0,00%

Figure 14. How the range analysis classified arithmetic instructions in the u-SSA form programs. #Sf: safe. #S: suspicious. #U: uncertain. #SO: number of suspicious instructions that overflowed. #UO: number of uncertain instructions that overflowed.

Figure 15 shows, for the entire LLVM test suite, the percentage of overflow checks that our range analysis, with the e-SSA intermediate representation, could avoid. Each bar refers to a specific benchmark in the test suite. We only consider applications that had at least one instrumented instruction; the total number of benchmarks that meet this requirement is 333. On the average, our range analysis avoids 24.93% of the overflow checks. Considering the benchmarks in SPEC 2006 only, this number is 20.57%.

Figure 16 shows the impact of our instrumentation in the runtime of the SPEC benchmarks. We ran each benchmark 20 times. The largest slowdown that we have observed, 11.83%, happened in `h264ref`, the benchmark that presented the largest number of distinct sites where overflows happened dynamically. On the average, the instrumented programs are 3.24% slower than the original benchmarks. If we use the range analysis to eliminate overflow checks, this

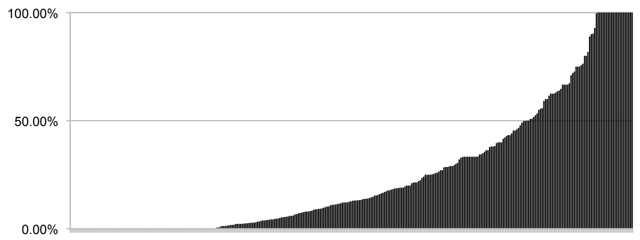


Figure 15. Percentage of overflow checks that our range analysis removes. Each bar is a benchmark in the LLVM test suite. Benchmarks have been ordered by the effectiveness of the range analysis. On average, we have eliminated 24.93% of the checks (geomean).

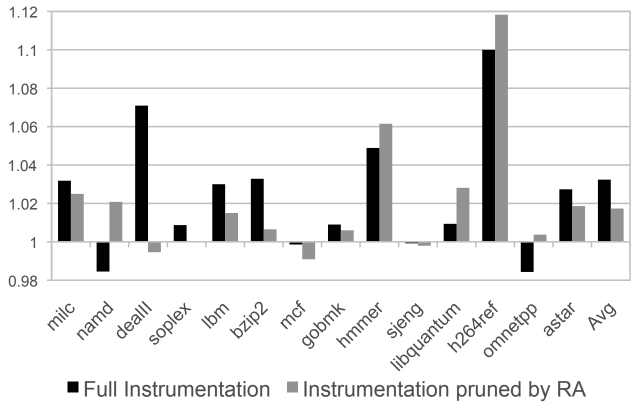


Figure 16. Comparison between execution times with and without pruning, normalized by the original program's execution time.

slowdown falls to 1.73%. The range analysis, in this case, reduces the instrumentation overhead by 46.60%. This improvement is larger than the percentage of overflow checks that we avoid, e.g,. 20.57%. We believe that this difference is due to the fact that we are able to eliminate checks on induction variables, as our range analysis can rely on the loop boundaries to achieve this end. We have not noticed any runtime difference between programs converted to e-SSA form or u-SSA form. Surprisingly, some of the instrumented programs run faster than the original code. This behavior has also been observed by Dietz *et al.* [11].

5. Related Work

Dynamic Detection of Integer Overflows: Brumley *et al.* [4] have developed a tool, RICH, to secure C programs against integer overflows. The author's approach consists in instrumenting every integer operation that might cause an overflow, underflow, or data loss. The main result of Brumley *et al.* is the verification that guarding programs against integer overflows does not compromise their performance significantly: the average slowdown across four large applications is 5%. RICH, Brumley *et al*'s tool, uses specific features of the x86 architecture to reduce the instrumentation overhead. Chinchani *et al.* [6] follow a similar approach. In this work, the authors describe each arithmetic operation formally, and then use characteristics of the computer architecture to detect overflows at runtime. Contrary to these previous works, we instrument programs at LLVM's intermediate representation level, which is machine independent. Nevertheless, the performance of the programs that we instrument is on par with Brumley's, even without the support of the static range analysis. Furthermore, our range analysis could eliminate approximately 45% of the tests that a naive implementation of Brumley's technique would insert; hence, halving down the runtime overhead of instrumentation.

Dietz *et al.* [11] have implemented a tool, IOC, that instruments the source code of C/C++ programs to detect integer overflows. They approach the problem of detecting integer overflows from a software engineering point-of-view; hence, performance is not a concern. The authors have used IOC to carry out a study about the occurrences of overflows in real-world programs, and have found that these events are very common. It is possible to implement a dynamic analysis without instrumenting the target program. In this case, developers must use some form of code emulation. Chen *et al.* [5], for instance, uses a modified Valgrind [21] virtual machine to detect integer overflows. The main drawback of emulation is performance: Chen *et al.* report a 50x slowdown. We differ from all this previous work because we focus on generating less instrumentation, an endeavor that we accomplish via static analysis.

Static Detection of Integer Overflows: Zhang *et al.* [28] have used static analysis to sanitize programs against integer overflow based vulnerabilities. They instrument integer operations in paths from a source to a sink. In Zhang *et al.*'s context, sources are functions that read values from users, and sinks are memory allocation operations. Thus, contrary to our work, Zhang *et al.*'s only need to instrument about 10% of the integer operations in the program. However, they do not use any form of range analysis to limit the number of checks inserted in the transformed code. Wang *et al.* [26] have implemented a tool, IntScope, that combines symbolic execution and taint analysis to detect integer overflow vulnerabilities. The authors have been able to use this tool to successfully identify many vulnerabilities in industrial quality software. Our work, and Wang *et al.*'s work are essentially different: they use symbolic execution, whereas we rely on range analysis. Contrary to us, they do not transform the program to prevent or detect such event dynamically. Still in the field of symbolic execution, Molnar *et al.* [20] have implemented a tool, SmartFuzz, that analyzes Linux x86 binaries to find integer overflow bugs. They prove the existence of bugs by generating test cases for them.

6. Final Remarks

This paper has presented a static range analysis algorithm that reduces the overhead necessary to secure programs against integer overflows. This algorithm analyzes interprocedurally programs with half-a-million lines of code, i.e., almost one million constraints, in ten seconds. We proposed the notion of "future bounds" to handle comparisons between variables, and tested different program representations to improve our precision. Although the overhead of guarding programs against integer overflows is small, as previous work has demonstrated, we believe that our technique is still important, as some of these programs will be executed millions of times.

Software: Our implementation is publicly available at `http://code.google.com/p/range-analysis/`.

Acknowledgments

We thank Douglas do Couto and Igor Costa for helping us with the LLVM code base. We also thank the anonymous reviewers for comments and suggestions. This project is supported by FAPEMIG and Intel.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[2] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.

[3] Benoit Boissinot, Sebastian Hack, Daniel Grund, Benoit Dupont de Dinechin, and Fabrice Rastello. Fast liveness checking for SSA-form programs. In *CGO*, pages 35–44. IEEE, 2008.

[4] David Brumley, Dawn Xiaodong Song, Tzi cker Chiueh, Rob Johnson, and Huijia Lin. RICH: Automatically protecting against integer-based vulnerabilities. In *NDSS*. USENIX, 2007.

[5] Ping Chen, Yi Wang, Zhi Xin, Bing Mao, and Li Xie. BRICK: A binary tool for run-time detecting and locating integer-based vulnerability. In *ARES*, pages 208–215, 2009.

[6] Ramkumar Chinchani, Anusha Iyer, Bharat Jayaraman, and Shambhu Upadhyaya. ARCHERR: Runtime environment driven program safety. In *European Symposium on Research in Computer Security*. Springer, 2004.

[7] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL*, pages 55–66, 1991.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[9] P. Cousot and N.. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM, 1978.

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[11] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in c/c++. In *ICSE*, pages 760–770. IEEE, 2012.

[12] Mark Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84–, 1997.

[13] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349, 1987.

[14] T. Gawlitza, J. Leroux, J. Reineke, H. Seidl, G. Sutre, and R. Wilhelm. Polynomial precise interval analysis revisited. *Efficient Algorithms*, 1:422 – 437, 2009.

[15] John Gough and Herbert Klaeren. Eliminating range checks using static single assignment form. Technical report, Queensland University of Technology, 1994.

[16] Lies Lakhdar-Chaouch, Bertrand Jeannet, and Alain Girault. Widening with thresholds for programs with complex control graphs. In *ATVA*, pages 492–502. Springer-Verlag, 2011.

[17] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.

[18] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1355–1371, 2001.

[19] Antoine Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19:31–100, 2006.

[20] David Molnar, Xue Cong Li, and David A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *SSYM*, pages 67–82. USENIX, 2009.

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

[22] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[23] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *PLDI*, pages 67–78. ACM, 1995.

[24] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *PLDI*, pages 108–120. ACM, 2000.

[25] Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computeter Science*, 345(1):122–138, 2005.

[26] T. Wang, T. Wei, Z. Lin, and W. Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*. Internet Society, 2009.

[27] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[28] Chao Zhang, Tielei Wang, Tao Wei, Yu Chen, and Wei Zou. Intpatch: automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *ESORICS*, pages 71–86. Springer-Verlag, 2010.