Bound Analysis using Backward Symbolic Execution

Sumit Gulwani  
Microsoft Research  
Redmond, WA, USA  
sumitg@microsoft.com

Sudeep Juvekar  
UC-Berkeley  
Berkeley, CA, USA  
sjuvekar@eecs.berkeley.edu

ABSTRACT
A fundamental problem that arises frequently in quantitative program analysis (e.g., resource usage analysis) is that of computing an upper bound for a given arithmetic expression at a given program location in terms of the procedure inputs. We refer to this problem as bound analysis. The problem is theoretically as well as practically challenging because of variable updates inside loops and presence of virtual methods.

Our solution to the bound analysis problem involves an inter-procedural (goal-directed) backward analysis built on top of an SMT solver. The analysis has the advantage of dealing with arbitrary operators that are understood by the underlying SMT solver. The analysis uses novel proof-rule based non-iterative technique to reason about updates inside loops, which makes it quite scalable. It uses user-defined abstract implementations to trace back across virtual methods arising from use of interfaces or extensible types.

We have implemented the analysis inside the SPEED tool, which computes symbolic computational complexity bounds for procedures. Our analysis is used to translate bounds on number of loop iterations and cost of method calls to respective bounds in terms of procedure inputs. We have evaluated the precision and scalability of the analysis over 4.NET assemblies that together contained thousands of methods and resulted in 9152 queries to the analysis. The analysis was able to answer 90% of the queries on an average of 0.23 seconds per query.

1. INTRODUCTION
The problem of bound analysis refers to the problem of computing an upper bound on a given arithmetic expression at a given program location in terms of the inputs of the enclosing procedure. Bound analysis has applications in the upcoming area of quantitative program analysis (as opposed to boolean program analysis), where the goal is to generate some quantitative information about the program or the data manipulated by the program.

An important class of quantitative program analyses are resource usage analyses [2], where the goal is to compute bounds on different kinds of resources consumed by a program such as time, memory, network bandwidth, and power. Computing bounds on such resources is important because of economic incentives or because the program might be running in a resource constrained environment (e.g., real-time systems or embedded systems). Most resource usage analyses (such as [1, 11, 10, 12]) compute some form of ranking functions for loops to obtain bounds. Since computation of ranking functions is usually a local analysis, the bounds obtained are arithmetic expressions expressed in terms of variables live at the beginning of the loop. Translating these arithmetic expressions in terms of the procedure inputs requires solving the bound analysis problem that is not formally addressed by these analyses. We consider this application in our experiments to demonstrate the effectiveness of our solution.

Another example of quantitative program analyses are analyses for computing bounds on numerical properties of data manipulated by the program, as in quantitative information flow analysis [15] for bounding the amount of secret data leaked by the program, or robustness analysis [5] for bounding the amount of uncertainty/error propagated during computation because of uncertainty/error in program inputs. Bound analysis can also serve as a key subroutine for these quantitative program analyses that can compute bounds in terms of local variables using domain-specific techniques (akin to ranking function computation for resource usage analysis), but require solving the bound analysis problem to translate the bounds in terms of procedure inputs.

In this paper, we address the problem of bound analysis by developing an inter-procedural backward symbolic execution engine. Our analyzer takes a program location and an arithmetic expression involving local program variables at that program location. It returns a set of expressions only involving program inputs and any reachable heap object from them, that upper bound the given expression at the given program location. The use of backward analysis is motivated by the problem domain, which requires a goal-directed analysis. Our backward analysis is more scalable than a forward symbolic execution like [21, 9] because it doesn’t explore unnecessary program paths (i.e., paths not leading to location of interest) and unnecessary regions of code (i.e., assignments that don’t determine the bound). Scalability of our approach is demonstrated by our experi-
mments, which explore across hundreds of procedures having $\sim 2^{50}$ execution paths.

A key technical challenge in bound analysis is to reason about variables that get updated inside loops. A common technique to reason about loops is to use iterative methods as in data-flow analyses [14], abstract interpretation [6] or model checking [8]. Data-flow analyses are relatively more scalable, but less precise than abstract interpretation and model checking. In contrast, we present a novel proof-rule based technique that allows for performing precise as well as scalable reasoning. Our proof-rule based technique captures the common design pattern wherein numerical variables that get updated inside loops either increase monotonically or decrease monotonically. Such a design pattern can be automatically identified by making an SMT (SAT modulo theory) query. Under such a design pattern, the value of the variable before and after the loop can be related using the number of visits to the program locations where the variable is updated inside loops. The number of such visits can be computed using a variety of existing techniques (e.g., those based on counter instrumentation [11], control-flow refinement [10], or ranking function based approach [12]). The effectiveness of our proof-rule based approach is demonstrated by our experiment results, wherein about 80% percentage of loops could be reasoned using the proof rules. Among some of the other technical challenges that we address in our backward symbolic execution are incorporating information from conditional guards and virtual call resolution.

A key practical challenge in bound analysis is to express or compute bounds when they depend on arbitrary heap locations or on implementations of (unavailable) method calls associated with input objects with interface/extensible types. In the latter case, we propose that the user provides some meaningful information in such circumstances. Our language of abstract bound expressions extends the program expression language by providing some constructs for referring to abstract heap locations or Ivalues such as access of a given array at an unknown index, dereference of a given field in an unknown object, or reference to an unknown object but one that is reachable from a given object via a certain set of fields. The abstract bound expressions also allow for expressing bounds conditional on validity of user-provided abstract implementations of method calls associated with interface/extensible types. The effectiveness of these abstract bounds is demonstrated by our experimen-
2. MOTIVATING EXAMPLES

In this section, we describe some examples from .Net codebase that illustrate the key technical and practical challenge in bound analysis.

2.1 Technical Challenge: Updates inside loops

The key technical challenge in bound analysis arises when the bound depends on variables that are updated inside loops. Consider the procedures shown in Figure 1. Computing an upper bound on \( n \) at the end of each of these procedures requires computing an upper bound on variable \( n \), which is updated inside loops.

Computing an upper bound on \( n \) for each of these examples is challenging for various reasons. Existing techniques for computing numerical invariants, which are based on inductive loop invariant generation, are specialized to address problems is challenging for various reasons. Existing techniques for computing numerical invariants, which are based on inductive loop invariant generation, are specialized to address problems.

Consider the procedure \( Ex6 \) in Figure 2. Observe that bound on \( t_1 \) depends on the contents of array \( A \) and on \( L \) and \( t_2 \) depends on the data field of some element of list \( L \). We propose representing these bounds using abstract bound expressions \( A[\perp] \) (denoting any index of the array) and \( R(L, \{\text{next}\}) \) (denoting the data field of any object object reachable from \( L \) by following the next field) as opposed to representing it by \( \perp \) (which denotes undefined).

Consider the procedure \( Ex7 \) in Figure 2 that uses the ICollection interface for Collections, a commonly used datastructure in several object-oriented languages. The procedure copies contents of input array \( A \) to collection \( C \). The challenge in computing bounds in such procedures is the absence of source code for methods such as Add. We propose using user-defined abstract implementations as a substitute for such methods. Under the abstraction that the Add method increases the Count field of the receiver object by at least 1, we can compute a bound of \( C.Count + A.length \) for \( t_1 \). Since the validity of this bound is conditional on the concrete implementation of Add satisfying the abstract semantics provided by the user, we denote this bound as \( \text{Cond}(C.Count + A.length) \). However, note that the bound on \( t_2 \) does not rely on this assumption.

3. PRELIMINARIES

3.1 Problem Statement

Given an lvalue \( t \) and a program location \( \pi \), the goal is to compute a set of symbolic expressions in terms of procedure inputs that upper bound any value that \( t \) can take at location \( \pi \). A procedure consists of assignment statements and conditional guards of the following form:

\[
\text{Assignment Statement} \quad \pi: \quad t := e \\
\text{Guard} \quad e_1 \text{ relop } e_2
\]

where label \( \pi \) denotes a program location, relop denotes some relational operator, and the expressions \( e \) and lvalues \( t \) have the following syntax with the usual semantics.

\[
e := c \mid t \mid e_1 \pm e_2 \mid op(e_1, e_2) \mid x.m(t_1, t_2) \mid \text{new}(\tau)
\]

\[
t := x \mid x[e] \mid x.f
\]
We classify any definition $d$ for $(\ell, \pi)$ as either inductive or non-inductive depending on whether or not it flows to $\ell$ along a back-edge. Let $\text{DefsInd}(\ell, \pi)$ and $\text{DefsInit}(\ell, \pi)$ denote the disjoint partition of $\text{Definitions}(\ell, \pi)$ into inductive and non-inductive definitions respectively.

**Example 2.** For procedure $\text{Ex}5$ in Figure 1, we have:

$$\text{Definitions}(n, 4) = \{\text{Simple}(1 : n := 0);\},$$

$$\text{DefsInit}(n, 4) = \{\text{Simple}(1 : n := 0);\}$$

Computation of Definitions can be done by using any off-the-shelf alias analysis.

### 3.3 Iteration Count

Let $P_1$ be some set of program locations inside a loop, and $P_2$ be some set of program locations outside that loop. Let $\text{Iterations}(P_1, P_2)$ denote an upper bound on the number of times any program location $P_1$ is visited in between any two visits to some location in $P_2$, expressed as a function of variables that are live at the header of outermost loop that contains all locations in $P_1$, but none in $P_2$. We overload the notation $\text{Iterations}(\pi_1, \pi_2)$ to denote $\text{Iterations}(\{\pi_1\}, \{\pi_2\})$.

$\text{Iterations}(P_1, P_2)$ can be computed by using the procedure described in [12] for computing an upper bound on the number of times a given program location $\pi$ is visited in terms of variables that are live at the entry to the outermost loop containing $\pi$. Let’s refer to this procedure as $\text{Visits}(\pi)$. We can compute $\text{Iterations}(P_1, P_2)$ using the procedure $\text{Visits}$ as follows. First, we transform the procedure by deleting the outgoing edges $E$ from program locations in set $P_2$ and adding jumps from the procedure entry point to the targets of edges $E$. (This has the effect of deleting all paths that go between any two (possibly same) locations in set $P_1$ after visiting any location in set $P_2$.) Then, we simply sum up $\text{Visits}(\pi_1)$ for all $\pi_1 \in P_1$.

**Example 3.** The second row in Figure 1 gives examples of $\text{Iterations}(\pi_1, \pi_2)$ for respective program locations $\pi_1$ and $\pi_2$ for procedures $\text{Ex}1$ to $\text{Ex}5$. Since $\pi_2$ is located outside the outermost loop containing $\pi_1$, for the procedures $\text{Ex}1, \text{Ex}2$, and $\text{Ex}3$, $\text{Iterations}(\pi_1, \pi_2)$ is same as $\text{Visits}(\pi_1)$ for each of these procedures. For the procedure $\text{Ex}4$, computation of $\text{Iterations}(6, 4)$ essentially involves computing $\text{Visits}(6)$ in $\text{Ex}4$ after removing the outer loop.

### 4. ABSTRACT BOUND

In this section, we describe our language for bound expressions. Since the bound can depend on an unbounded number of memory locations that are reachable from the inputs of a procedure, we cannot simply use the expression language $e$ of the program to represent bounds.

We propose the following abstract language of expressions $\beta$ for representing bounds. The language $\beta$ is similar to that of the program expression language except that it uses abstract lvalues $\ell$ and two new constructs $\bot$ and $\text{Cond}(\beta)$. The abstract lvalue $\ell$ is similar to standard lvalues $\ell$ except that it uses one new construct $R(\ell, F)$.

$$\beta ::= c | \ell | \beta_1 \oplus \beta_2 | \text{op}(\beta_1, \beta_2) | \bot | \text{Cond}(\beta)$$

$$\ell ::= x | \ell[\beta] | \ell.f | R(\ell, F)$$

**Example 1.** Consider procedure $\text{Ex}8$ in Figure 3. We have $\text{Definitions}(t_1, 12) = \{\text{Simple}(1 : A[i] := z), \text{Index}(7 : i := z + 4)\}$, and $\text{Definitions}(t_2, 12) = \{\text{Simple}(1 : x.f := A[0]), \text{Simple}(5 : x := y), \text{Object}(9 : y.f := z), \text{SideEffect}(8 : t := x.m())\}$. The definition $\text{Object}(9 : y.f := z)$ is included because $y$ may alias with $x$.

---

**Table 3:** Two examples (obtained from combining code snippets from .Net code base). $\text{Ex}8$ illustrates different types of Definitions for an lvalue (Section 3.2). $\text{Ex}9$ illustrates the significance of different tracing modes (even if the final goal is to compute an upper bound) and their corresponding proof rules for loops.

Above, $c$ denotes a constant, $x$ denotes a variable, $x[e]$ denotes an array dereference, $x.f$ denotes a field dereference, $o$ denotes an operator other than addition and subtraction, $x.m(t_1, t_2)$ denotes a method invocation to method $m$ of class that denotes the runtime type of object $x$, and $\text{new}(\tau)$ returns a fresh object of class $\tau$.

#### 3.2 Reaching Definitions

We use the notation $\text{Definitions}(\ell, \pi)$ to denote the set of immediate/reaching definitions that may determine the value of $\ell$ at location $\pi$. A definition $d$ is either a labeled assignment statement $\pi' : \ell' := e$ along with an associated type Simple, Index, Object and SideEffect, or simply has the type Input, with the following semantics.

- If $d$ is of the form Input, then $\ell$ at procedure entry is same as $\ell$ at location $\pi$.
- If $d$ is of the form Simple($\pi' : \ell' := e$), then $\ell'$ at location $\pi'$ is same as $\ell$ at location $\pi$.
- If $d$ is of the form Index($\pi' : \ell' := e$), then $\ell$ at location $\pi$ is equal to some index expression in $\ell$ at location $\pi$.
- If $d$ is of the form Object($\pi' : \ell' := e$), then $\ell$ at location $\pi$ is equal to some object dereference in $\ell$ at location $\pi$.
- If $d$ is of type SideEffect($\pi' : \ell' := o.m(t_1, \ldots, t_n)$), then the call to method $o.m$ may modify $\ell$.

Note that if $\ell$ is a scalar variable, then all definitions in $\text{Definitions}(\ell, \pi)$ are of type Input or Simple.

**Example 1.** Consider procedure $\text{Ex}8$ in Figure 3. We have $\text{Definitions}(t_1, 12) = \{\text{Simple}(1 : A[i] := z), \text{Index}(7 : i := z + 4)\}$, and $\text{Definitions}(t_2, 12) = \{\text{Simple}(1 : x.f := A[0]), \text{Simple}(5 : x := y), \text{Object}(9 : y.f := z), \text{SideEffect}(8 : t := x.m())\}$. The definition $\text{Object}(9 : y.f := z)$ is included because $y$ may alias with $x$.

---

**Ex8(C x, C y**

<table>
<thead>
<tr>
<th>Array&lt;INT&gt; A,</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
| $1 \ x.f := A[0]$
| $2 \ i := z$
| $3 \ (\text{if (nondet())})$
| $4 \ A[i] := z$
| $5 \ x := y$
| $6 \ else$
| $7 \ i := z +4$
| $8 \ t := x.m()$
| $9 \ y.f := z$
| $10 t_1 := A[i]$
| $11 t_2 := x.f$
| $12$ | $\text{Bounds computed by our analysis:}$
| $t_1 \leq z + 1, \ t_2 \leq 100, \ t_3 \leq A[1], \ t_4 \leq R(L, \text{next}).data$ |
\(\bot\) denotes arbitrary content. It is used as a bound expression whenever the analysis fails to compute a bound. However, more significantly, the recursive construction of abstract lvalues using \(\bot\) allows for providing more meaningful information. For example, \(A[\bot]\) denotes content at an arbitrary location in array \(A\), \(\bot[0]\) denotes the element at the first location of an arbitrary array, and \(\bot.f\) denotes the \(f\) field of an arbitrary object.

\(R(\ell, F)\) denotes an arbitrary object that is reachable from \(\ell\) by applying zero or more field dereferences from set \(F\) of fields. Note that \(R(\ell, \{f\})\) provides more meaningful information that \(\bot.f\).

To enable bound computation when a procedure has an input with a polymorphic type \(T\) (e.g., an interface or a base class that can be extended), we allow the user to optionally define abstract implementations of methods associated with the polymorphic type. These abstract implementations define how they update fields of various input objects as well as the receiver. \(\text{Cond}(\beta)\) either denotes \(\beta\) or \(\bot\) conditional on whether or not the user-provided abstract implementation of the methods associated with interfaces or extensible base classes is consistent with the concrete implementations of those methods.

**Example 4.** Consider the interface ICollection defined by the .Net Framework and widely used in .Net programs. The ICollection Interface declares methods Add, Remove, and Find, and a read-only field \(\{\text{declared as a property}\}\) Count. It is reasonable to define the following abstract implementations regarding updates to the read-only field Count by the various ICollection methods.

\[
\begin{align*}
\text{Clear}() & : \{\text{this.Count} := 0;\} \\
\text{Add}(\ell) & : \{\text{if (nondef())\{this.Count} := \text{this.Count} + 1;\}\} \\
\text{Remove}(\ell) & : \{\text{if (nondef())\{this.Count} := \text{this.Count} - 1;\}\} \\
\text{Contains}(\ell) & : \{\text{skip;}\}
\end{align*}
\]

Besides .Net Framework, other well-known examples of collections frameworks are collections classes are Java Framework, C++ Standard Template Library, and Smalltalk's collection hierarchy.

## 5. BACKWARD SYMBOLIC EXECUTION

In this section, we describe the core functionality of our backward symbolic execution engine for tracing expressions across definitions that are non-inductive and are different from virtual method calls. We address the issue of tracing across definitions that are inductive or those that are virtual method calls in Section 6 and Section 8 respectively.

### 5.1 Tracing Modes

The symbolic execution engine traces an expression at a program location backward in one of four possible modes.

- **U:** Upper Bound mode. The goal here is to compute an upper bound on the expression being traced backwards.
- **L:** Lower Bound mode. The goal here is to compute a lower bound on the expression being traced backwards.
- **E:** Equality mode. The goal here is to trace an expression backwards precisely.
- **O:** Object Equality Mode. A special case of equality mode where the value being traced back is an object (as opposed to a scalar value).

The difference in these modes show up primarily in our strategy for tracing back across loops. We use the notations \(B_{\text{pol}}(\ell, \pi)\) and \(B_{\text{pol}}(e, \pi)\) to denote the result obtained by tracing lvalue \(\ell\) and expression \(e\) respectively, at location \(\pi\) backwards in mode \(\text{pol}\).

### 5.2 Tracing Of Expressions

The backward symbolic execution engine traces expressions backward by tracing the constituent lvalues backward.

For tracing arithmetic expressions built using addition or subtraction operators, or for tracing return values of method calls, the symbolic execution engine passes down the appropriate contextual information concerning whether an upper bound, lower bound, or equivalent expression is to be computed. For tracing across a procedure, the symbolic execution engine first traces the returned value of the procedure from its exit location to the procedure entry. The resulting expression, however, is in terms of formal parameters of the called procedure. The symbolic execution engine then performs a syntactic transformation to replace formal arguments in the expression by actual procedure parameters.

### 5.3 Tracing Of Lvalues

Given a definition \(d \in \text{Definitions}(\ell, \pi)\), we overload the notation \(B_{\text{pol}}(\ell, \pi, d)\) to denote \(B_{\text{pol}}(\ell, \pi)\) under the assum-
tion that $d$ is the update that sets value of $\ell$. If all definitions in Definitions$(\ell, \pi)$ are non-inductive, then the symbolic engine traces each definition individually as follows.

$$B_{pol}(\ell, \pi) = \{ \beta \mid \beta \in B_{pol}(\ell, \pi, d), d \in \text{Definitions}(\ell, \pi) \} \quad (3)$$

The backward symbolic execution traces back across any non-inductive non-SideEffect definition as follows.

$$B_{pol}(\ell, \pi, \text{Input}) = \{ \ell \}$$
$$B_{pol}(\ell, \pi, \text{Simple}(\pi': \ell := e)) = B_{pol}(e, \pi')$$
$$B_{pol}(\ell, \pi, \text{Index}(\pi': \ell' := e)) = \{ \beta \mid \beta \in B_{pol}(\ell[\beta_1 / \ell'], \pi'), \beta_1 \in B_{pol}(e, \pi') \} \quad (4)$$
$$B_{pol}(\ell, \pi, \text{Object}(\pi': \ell' := e)) = \{ \beta \mid \beta \in B_{pol}(\ell[\beta_1 / \ell'], \pi'), \beta_1 \in B_{pol}(e, \pi') \} \quad (5)$$

$B_{E}$ and $B_{O}$ trace back array indices and objects respectively in exactly the same manner as $B_{U}$, except for inductive definitions. Hence, otherwise indicated, the definitions for $B_{E}$ and $B_{O}$ are supposed to be identical to that of $B_{U}$.

**Example 5.** Consider the example shown in Figure 3. Computing an upper bound on $t_3$ requires computing an upper bound on $n$ after the loop (Eq. 1). Computing an upper bound on $t_4$ requires computing a lower bound on $n$ after the loop (Eq. 2). Computing an upper bound on $t_5$ and $t_4$ requires identifying an expression($n$) equivalent to $n$ and $L$ after the loop (Eq. 4 and Eq. 5). As, we will see later, our proof rules for loops can provide a more precise estimate about $L$ if we identify that $L$ is an object variable (as opposed to being a scalar variable), and hence we use $B_{O}$ (as opposed to $B_{E}$) to trace back $L$.

The backward symbolic execution traces back across non-virtual method calls as follows.

$$B_{pol}(\ell, \pi, \text{SideEffect}(\pi': \ell' = o.m(\ell_1, \ell_2))) = \{ \beta \in B_{pol}(\ell_1, \pi') \mid \beta_1 \in B_{pol}(\ell, \pi_{\text{exit}})[\ell_1 / z_1, \ell_2 / z_2] \}$$

Above $\pi_{\text{exit}}$ denotes the exit location of method $m$, $r$ denotes the return variable of method $m$, and $z_1, z_2$ denote the formal parameters of method $m$.

### 6. TRACING ACROSS LOOPS

In this section, we describe a novel proof-rule based technique to reason about loops. The key idea is to use SMT solvers to verify/identify commonly occurring design patterns and then use the appropriate proof rule to conclude the effect of the loop. This allows us to short-circuit the backward symbolic execution across the loop to its start.

#### 6.1 Proof Rule for Upper Bound Mode

Suppose an lvalue increases by a bounded quantity $c$ in each iteration of the loop. Then its value at the end of the loop is bounded above by the sum of the value at the beginning of the loop and the number of increments multiplied by $c$. The following theorem captures a more general form of this principle involving multiple lvalues.

**Theorem 1.** Let $L$ be some set of lvalues and $P$ be some cut-set of a strongly connected region (i.e., $P$ is a set of program locations such that any cycle in the region goes through some location in $P$). Suppose that there exists a constant $c$ such that on any path between two locations in $P$ (with no intervening visit to any location in $P$), any update to an lvalue $\ell \in L$ is such that the updated value of $\ell$ is bounded above by $\ell + c$ for some $\ell' \in L$. Then, the value of any lvalue $\ell \in L$ outside the strongly connected component is bounded above by the sum of value of some $\ell' \in L$ before the strongly connected component and the number of times the locations in set $P$ are visited multiplied by $c$.

The proof of Theorem 1 follows easily by induction on the number of visits to the locations in $P$.

**Example 6.** Consider procedure Ex5 in Figure 1. Consider the cut-set $P = \{4, 6\}$ for the strongly connected region defined by the outer loop. Let $L = \{n, m\}$. Observe that the choice of $c = 1$ satisfies the condition in Theorem 1 since: On the (shortest) path from 4 to 6, $m$ is assigned to $n + 1$. On the path from 6 to 4, $n$ and $m$ are assigned to $m + 1$. On the path from 4 to 4, $m$ and $n$ both are assigned to $n + 1$. The location 4 is visited at most $z_1$ times, while the location 6 is visited at most $z_1 \times z_2$ times. Note that the initial value of $n$ before the loop is 0. Hence, the values of both $n$ and $m$ outside the loop are bounded above by $0 + (z_1 + z_2 \times z_2) \times 1 = z_1 + z_2$.

We make use of Theorem 1 to define below the backward symbolic execution engine $B_{L}(\ell, \pi)$ when Definitions$(\ell, \pi)$ contains an inductive definition. For this purpose, we first define some helper functions. Let $T \equiv \text{Transitive}(\ell, \pi)$ be the set of all $(\ell', \pi')$ pairs visited during backward tracing of $(\ell, \pi)$ such that Definitions$(\ell', \pi')$ contains an inductive definition.

$$\text{Bin\text{it}_{pol}(\ell, \pi)} = \{ \beta \mid \beta \in B_{pol}^{T, \pi}(\ell', \pi', d), (\ell', \pi') \in T, \ \ d \in \text{DefsInd}(\ell', \pi') \}$$

$$\text{BInd\text{ind}_{pol}(\ell, \pi)} = \{ \beta \mid \beta \in B_{pol}^{T, \pi}(\ell', \pi', d), (\ell', \pi') \in T, \ \ d \in \text{DefsInd}(\ell', \pi') \}$$

$$\text{PInit\text{it}_{pol}(\ell, \pi)} = \{ \text{Label}(d) \mid (\ell', \pi') \in T, \ \ d \in \text{DefsInd}(\ell', \pi') \}$$

$$\text{PInd\text{ind}_{pol}(\ell, \pi)} = \{ \text{Label}(d) \mid (\ell', \pi') \in T, d \in \text{DefsInd}(\ell', \pi') \}$$

Above, $B_{pol}^{T, \pi}(\ell_1, \pi_1)$ is the function that traces back $\ell_1$ at location $\pi_1$ across one iteration of the loop containing $\text{PInd}_{pol}$. It can be implemented in exactly the same manner as $B_{pol}(\ell_1, \pi_1)$ with the exception that backward tracing is stopped when any $(\ell_2, \pi_2) \in T$ is encountered, and $\ell_2$ is returned. $\ell_2$ refers to some fresh variable that does not occur in the program and is meant to represent all lvalues in $T$. $B_{pol}(\ell_1, \pi_1)$ is the function that traces back $\ell_1$ at location $\pi_1$ along the paths that take go outside of the loop. It can be implemented in exactly the same manner as $B_{pol}(\ell_1, \pi_1)$ with the exception that backward tracing is not performed along paths that iterate inside the loop.

$$B_{pol}^{T, \pi}(\ell_1, \pi_1) = \emptyset \ if \ (\ell_1, \pi_1) \in T$$

$$B_{pol}^{T, \pi}(\ell_1, \pi_1) = \{ \ell_2 \} \ if \ (\ell_1, \pi_1) \in T$$

Using the above helper functions, the proof rule described in Theorem 1, which captures an extremely common design pattern, can now be translated into our backward symbolic execution engine as follows.

$$B_{L}(\ell, \pi) = \{ \beta + \text{iter} \times \text{Max} (0, c) \mid \beta \in \text{Bin\text{it}_{pol}(\ell, \pi)} \}$$

$$\text{if } \forall \ell' \in \text{BInd\text{ind}_{pol}(\ell, \pi)} : \ell' \leq \ell_2 + c \quad (6)$$

$$= \bot \ otherwise$$
where c is some constant, $\text{giter} = B_t((\text{iter}, \pi'))$ and
$\text{iter} = \text{Iterations}(\text{PInd}(t, \pi), \text{PInit}(t, \pi))$ denotes an upper bound on the number of combined visits to locations in $\text{PInd}_{\text{psL}}(t, \pi)$ in terms of the variables that are live at $\pi'$, where $\pi'$ is some program location that is outside the loop that contains program locations in $\text{PInd}_{\text{psL}}(t, \pi)$.

Example 7. Computing an upper bound on $t$ for each of the procedures in Figure 1 requires computing an upper bound on $n$ which is updated inside loops. We now explain how the proof rule in Eq. 6 facilitates computation of precise upper bound on $n$ for each of these examples.

Procedures $\text{Ex}1$, $\text{Ex}2$ lead to exactly identical tracing except for computation of $\text{Iterations}(\text{PInd}(n, 5), \text{PInit}(n, 5)) = \text{Iterations}(5, 2)$, which is computed to be $z_1 - i$ and 1 for $\text{Ex}1$ and $\text{Ex}2$ respectively. This leads to asymptotically different bounds of $z_1 + 2z_2$ and $z_1 + 2$ for $n$ and $t$ at the end of the loops in $\text{Ex}1$ and $\text{Ex}2$ respectively.

Procedures $\text{Ex}3$ and $\text{Ex}4$ also lead to exactly identical tracing except for computation of $\text{BInit}(n, 6)$, which is computed to be $\{2 : n := 0\}$ and $\{4 : n := 0\}$ respectively. This leads to $\text{Iterations}(\text{PInd}(n, 6), \text{PInit}(n, 6))$ being computed as $\text{Iterations}(6, 2) = (z_1 - i) \times z_2$ and $\text{Iterations}(6, 4) = z_2 - j$ for $\text{Ex}3$ and $\text{Ex}4$ respectively. This leads to asymptotically different bounds of $z_1 \times z_2$ and $z_2$ for $n$ and $t$ after the loops in $\text{Ex}3$ and $\text{Ex}4$ respectively.

Procedure $\text{Ex}5$ demonstrates the need for tracing multiple inductive values. Since tracing of $(n, 8)$, the helper functions get invoked with $(n, 4)$ (since $(n, 4)$ contain an inductive definition) and return the following:

$$\text{Transitive}(n, 4) = \{(n, 4), (m, 6)\}, \text{BInd}_I(n, 4) = \{\ell_d + 1\}$$

$$\text{BInit}_I(n, 4) = \{0\}, \text{PInd}_I(n, 4) = \{4, 6\}, \text{PInit}_I(n, 4) = \{2\}$$

Since $\text{Iterations}(4, 2) = z_1 - i$ and $\text{Iterations}(6, 2) = (z_1 - i) \times z_2$, we obtain $\text{Iterations}(\text{PInd}(n, 4), \text{PInit}(n, 4)) = \text{Iterations}(6, 2)$ to be $(z_1 - i) \times (1 + z_2)$. $\text{BInd}_I(n, 4)$ leads to a choice of 1 for $c$. Together, these lead to the desired bound of $z_1 \times (1 + z_2)$ on $n$ at the end of the loop.

6.2 Proof Rule for Lower Bound Mode

The proof rule for tracing in lower bound mode is similar to that of in upper bound mode except that we need to assert that inductive values have bounded decrease instead of bounded increase. This leads to the following backward symbolic execution strategy.

$$B_{L}(\ell, \pi) = \left\{ \beta - \text{giter} \times \text{Max}(0, c) \mid \beta \in \text{BInit}_L(\ell, \pi), \begin{array}{l} 
\text{if } \forall \beta' \in \text{BInd}_L(\ell, \pi) : \beta' \geq \ell_d - c \end{array} \right\}$$

$$= \bot \text{ otherwise}$$

where $\text{giter}$ is as in Eq. 6.

6.3 Proof Rule for Equality Mode

It is not possible to trace back arbitrary expressions in equality mode across loops. So, we simply return $\bot$.

$$B_{E}(\ell, \pi) = \bot$$

6.4 Proof Rules for Object Equality Mode

For the special case, when the object to be traced back in equality mode is an object, we provide a proof rule for the common design pattern of iterating along a certain set

of recursive fields.

$$B_{O}(\ell, \pi) = \{ R(\beta, \text{Dereferences}(\ell, \pi)) \mid \beta \in \text{BInit}_O(\ell, \pi), 
\text{if } \text{Dereferences}(\ell, \pi) \neq \bot \Rightarrow \ell \text{ otherwise} \}$$

Example 8. As explained in Example 5, tracing $t_1$, $t_2$, and $t_3$ backwards require in procedure $\text{Ex}9$ in Figure 3 requires tracing $n$ backwards in different modes. This, in turn, requires making use of different proof rules.

- Tracing $n$ in upper bound mode makes use of proof rule in Eq. 6. Note that $c = 1$ and $\text{giter}$ is computed as $z_1$, which provides an upper bound of $z_1$ for $n$.
- Tracing $n$ in lower bound mode makes use of proof rule in Eq. 7. Note that $c = -1$ and hence $\text{Max}(0, c) \times \text{giter}$ yields 0, which provides a lower bound of 0 for $n$.
- Tracing $n$ in equality mode makes use of proof rule in Eq. 8, which abstracts the value of $n$ to $\bot$.

Tracing $t_4$ backwards requires tracing $L$ in object equality mode, which makes use of the proof rule in Eq. 9 to provide an upper bound of $R(L, \{\text{next}\}, \text{data})$. Note that if $L$ was traced back in equality mode, then it would have provided an upper bound of $\bot$, which is less precise than the information provided by $R(L, \{\text{next}\}, \text{data})$ since the former implies that $t_4$ is bounded above by the $\text{data}$ field of any object, while the latter implies that $t_4$ is bounded above by the $\text{data}$ field of only those objects that are reachable from input list $L_1$ via the $\text{next}$ pointers.

7. MAKING USE OF GUARDS

Until now, our discussion of backward symbolic execution engine did not explicitly make use of conditional guards. Use of guards was implicit in computation of $\text{Iterations}(P_1, P_2)$ used in our proof rules to perform backward symbolic execution across loops. In this section, we discuss how to incorporate information from conditional guards to improve our bound computation process.

A primary purpose of reasoning about conditional guards in programs is to establish infeasibility of certain paths. However, we did not observe any instance in practice where
isolating infeasible paths helps compute an asymptotically better bound. In contrast, our path insensitive analysis is quite efficient and scales to large real programs.

For the bound analysis problem, a more relevant application of conditional guards can be in establishing bounds when we fail otherwise. This can be useful during two situations: in our backward symbolic execution engine where information from guards may be useful. One such situation is when \( B_{\text{pol}}(l, \pi, d) \) returns \( \bot \) in (Eq. 3). Whenever, this happens, we can try the following alternative in its place.

\[
B_{\text{pol}}(l, \pi, d) = B_{\text{pol}}(e, \pi') \text{ if Guard}(l, \pi, d) \Rightarrow (l \text{ relop}_e) e \quad (11)
\]

where \text{relop}_e denotes the \( \leq \) operator, \text{relop}_p denotes the \( \geq \) operator, and \text{relop}_p, \text{relop}_G both denote the equality comparison operator. \text{Guard}(l, \pi, d) denotes the boolean condition (in terms of variables that are live at the program location corresponding to definition \( d \)) that must be true if the value of \( l \) at location \( \pi \) is determined by \( d \). \text{Guard}(l, \pi, d) can be computed by using the gating functions as defined in [22].

Example 9. Consider tracing back \( t \) in procedure Ex10 in Figure 4. If we do not use the rule in Eq. 11, we obtain \( B_{\text{Vt}}(n,3, \text{Simple}(1 : n := \text{nondet}(())) = \bot \). Note that \( \text{Guard}(n,3, \text{Simple}(1 : n := \text{nondet}(())) = (n \leq 100) \). Use of the rule in Eq. 11 with a choice of 100 for \( e \) helps compute a bound of 100 on \( n \) (at program location 3) and \( t \).

Another situation when conditional guards can be useful during bound computation is when \( B_t(l, \pi) \) contains a symbolic expression \( \beta \) for which the check \( \beta \leq \ell + c \) fails in the proof rule in Eq. 6. This problem can be alleviated by refining the check to be performed under the additional assumption of \( \text{Guard}(l, \pi, \beta) \), where \( \text{Guard}(l, \pi, \beta) \) denote the boolean condition (in terms of variables that are live at \( \text{PointsInductive}(\pi) \)) that must be true if \( \ell \) if \( \beta \) flows into \( l \) at location \( \pi \). A similar refinement can also be performed for the check \( \beta \geq \ell - c \) in Eq. 7 and the check \( \beta \equiv l \) in Eq. 10. \( \text{Guard}(l, \pi, \beta) \) can be computed by conjuncting together all gating functions \( \text{Guard}(l, \pi, d) \) corresponding to the definitions \( d \) used in computation of \( \beta \).

Example 10. Consider the process of tracing back \( t \) in procedure Ex11 in Figure 4. This requires computing an upper bound on \( n \), which is updated inside a loop. Computing an upper bound on \( n \) requires using the proof rule in Eq. 6, which requires computing \( \text{DefInd}(n,6) = \text{Simple}(4 : n := (z_1 + n)/2) \) and \( B_t(n,6) = \{z_1 + n)/2\} \). Since there does not exists a constant \( c \) such that \( \beta \leq n + c \), where \( \beta = (z_1 + n)/2 \), the proof rule fails to return a bound. Observe that \( \text{Guard}(l, \pi, \beta) = (z_1 < n) \) and can be used to prove that \( \beta \leq n + c \), where \( c = 0 \). This allows the proof rule to return an upper bound of \( z_2 \) for \( n \) (at location 6).

8. TRACING ACROSS VIRTUAL METHODS

In section 5.2, while defining \( B_{\text{pol}}(\alpha.m(l, t_{\ell})), \pi) \), we assumed that method \( m \) is non-virtual and can be statically resolved (i.e., can be resolved from the static type of receiver object \( o \)). In this section, we describe how to resolve method \( m \) if it cannot be resolved from the type of \( o \).

If the user has provided abstract implementation for method \( m \) (as discussed in Section 4), then we use those. Else, we trace back receiver object \( o \) in equality mode to obtain the set \( E = B_{\text{E}}(o, \pi) \). If \( L \subseteq E \) or \( \beta \in E \) such that \( \text{TypeOf}(\beta) \) is extensible, then we define \( B_{\text{pol}}(\alpha.m(l_{\ell_1}, \ell_2)), \pi) \) to be \( \bot \). Otherwise, we obtain the set \( S \) of possible methods \( m \) from \( E \) as follows: \( S = \{\text{TypeOf}(\beta) \mid \beta \in E\} \), where \( \text{TypeOf}(\text{neu}(\gamma)) = \tau \), and \( \text{TypeOf}(\beta) \), where \( \beta \) is some expression over inputs, can be obtained from types of inputs. We then trace back across each of those methods as defined in Section 5 and take the union of the results.

9. EVALUATION

In this section, we describe our experiments and results of our algorithms applied to four .Net benchmark projects.

9.1 Implementation and Benchmarks

We have implemented our algorithms in .NET framework and have integrated them with the SPEED [10, 11, 12] tool that computes the symbolic complexity of a loop in a program. SPEED computes this complexity in terms of live variables at the loop header. Our tool took these “local” loop bounds and converted these bounds into expressions in terms of procedure inputs. Our tool uses Phoenix Compiler Framework [18] to convert a source program into an intermediate representation. Phoenix provided us with an over-approximation of \( \text{Definitions}(\ell, \pi) \) for every value and location in the program, by performing various sound data-flow analyses on it. We also used SPEED tool [10, 11, 12] to give us the amortized number of visits to any location in the program, i.e., to compute the \( \text{Iterations} \) function defined in section 3.3. Our proof rules were implemented using Z3 [24] SMT solver.

Our experimental suite consists of four medium to large-sized .NET projects — (1) .NET core (includes main .NET libraries) (2) .NET libraries (include high level .NET libraries like Windows services) (3) Facebook .NET API client and (4) Silverlight Media Player.

9.2 Experimental Results

Table 2 shows the overall success of our experiments. The first row of the table gives the total number of queries issued to our backward analysis engine for every benchmark. Second row shows the total number of top-level queries that could be successfully traced backed to the procedure inputs. As we discussed earlier, failure in our analysis comes from two sources-(1) Not being able to reason about loops, and (2)not being able to resolve an abstract method. Our observation is that, for all benchmarks, on an average, 94% queries could be successfully traced back to the procedure inputs (as shown in the third row of the table). Next two rows show that every benchmark contains tens of thousands of methods and millions of lines of code. Finally, last row of the table shows the amount of time taken on every benchmark. All our benchmarks could be analyzed within few minutes, with an average time of 0.23 seconds per query. Our experiments were performed on a Desktop PC with Intel(R) Xeon(TM) 3.20 GHz processor, 2.00 GB of memory, running Windows XP.

9.3 Scalability of Our Analysis

We measured the total number of procedures visited and the depth of the procedure stack for any single query in our analysis. These numbers are shown in Table 1. The table shows that about more than 90% queries visited just one method during entire analysis. There were, however some queries that visited more than 10 methods during the entire
Table 1: Table showing the frequency of depth of procedure stack (First two rows), the frequency of number of visited procedures (next two lines) and size of the set of resulting expression (last two lines) for any query.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>.NET Core</th>
<th>.NET Libraries</th>
<th>Facebook</th>
<th>.NET API</th>
<th>Silverlight</th>
<th>Media Player</th>
</tr>
</thead>
<tbody>
<tr>
<td># Queries</td>
<td>4259</td>
<td>2963</td>
<td>625</td>
<td>1305</td>
<td></td>
<td></td>
</tr>
<tr>
<td># Successes</td>
<td>4050</td>
<td>2631</td>
<td>612</td>
<td>1244</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Success (%)</td>
<td>95.19%</td>
<td>88.80%</td>
<td>97.92%</td>
<td>95.25%</td>
<td></td>
<td></td>
</tr>
<tr>
<td># Functions</td>
<td>85834</td>
<td>78071</td>
<td>18116</td>
<td>34884</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size (LOC)</td>
<td>3569623</td>
<td>3187007</td>
<td>678128</td>
<td>2168345</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time (sec)</td>
<td>799.28</td>
<td>777.26</td>
<td>124.93</td>
<td>269.92</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Overall Success of Backward Symbolic Execution for Bound Analysis on 4 Benchmarks.

<table>
<thead>
<tr>
<th>Procedure Depth</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>1008</td>
<td>247</td>
<td>112</td>
<td>124</td>
<td>308</td>
<td>63</td>
<td>26</td>
<td>45</td>
<td>48</td>
<td>83</td>
<td>78</td>
<td>100</td>
<td>16</td>
<td>9</td>
<td>6</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Procedures visited</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11-20</td>
<td>21-30</td>
<td>31-40</td>
<td>41-50</td>
<td>51-100</td>
<td>100-200</td>
<td></td>
</tr>
<tr>
<td>Frequency</td>
<td>16994</td>
<td>197</td>
<td>587</td>
<td>576</td>
<td>428</td>
<td>178</td>
<td>140</td>
<td>143</td>
<td>109</td>
<td>49</td>
<td>124</td>
<td>15</td>
<td>13</td>
<td>26</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Expression Set size</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11-20</td>
<td>21-30</td>
<td>31-40</td>
<td>41-50</td>
<td>51-100</td>
<td>&gt;100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frequency</td>
<td>13308</td>
<td>308</td>
<td>118</td>
<td>24</td>
<td>19</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

9.4 Effectiveness of Proof Rules

During our analysis, 1243 queries encountered reasoning about loops. For every loop encountered during our analysis, we determined the fraction of loops that could be analyzed using any of our proof rules defined in Equations 6, 7, and 9. The plots for percentage of loops that can be analyzed using different proof rules for different benchmarks are shown in figure 3(I). The plots show that about 80% loops could be analyzed using the proof rule for upper bound mode (Eq 6), 11% loops could be analyzed using the proof rule for lower bound mode (Eq 7) and 18% loops could be analyzed using the proof rule for object equality mode (Eq 9). Our experiments show that about 80% loops satisfied one of the proof rules across all benchmarks.

9.5 Effectiveness of Virtual Call Resolution

In our experiments, about 90% of the method calls could be uniquely resolved and did not require any virtual call resolution. To demonstrate the utility of our abstract method resolution techniques, we collected all unresolved/virtual methods encountered during our backward analysis on the benchmark projects. These methods were unresolved for two reasons—(1) The methods were native methods, implemented in a different language and our analysis could not get a code for them. This practical challenge was handled by defining our own abstract implementations for these methods as described in section 4. We used abstract implementations for about 30 methods in two classes—System.String and System.Array. (2) The methods were virtual methods and the type of the client objects for these methods could not be statically resolved. In this case, we applied our virtual call resolution technique (defined in Section 8) to uniquely resolve the methods.

The results are shown in figure 3(II) The figure shows that our virtual call resolution was effective in resolving method calls in about 20% of abstract methods, while the abstract implementations for two classes were successful in resolving method calls in about 76% of cases.

9.6 Effectiveness of Abstract Bounds

In order to measure the effectiveness of our abstract bounds, we first computed the number of bounds computed in terms of abstract implementations and unbounded array expressions (of the form $A[\bot]$), defined in section 4. We also computed the number of bound expressions that were arbitrary, i.e. $\bot$. The resulting plots are shown in figure 3(III). Our plots show that about 82% of the bound expressions were computed using abstract bounds, about 8% of bounds were computed using arbitrary array dereferences while remaining bounds were arbitrary expressions (⊥).

10. RELATED WORK

Backward Analyses: Snugglebug [4] performs a path-sensitive backward symbolic analysis on object-oriented programs to find bugs. Snugglebug reasons about loops by unfolding them a fixed number of times. Such an approach is useful for bug-finding, but would be unsound in our case.

PSE [16] presents a static analysis to diagnose software failures. It tracks the flow of a single value from a program location to the program entry. It performs a novel dataflow analysis and pointer analysis to reason about heap. But like SNUGGLEBUG, PSE does not use any sophisticated reasoning about loops. Nor do any of these techniques address the issue of presence of virtual methods that arise from use of interfaces and extensible classes.

Forward Analyses: Existing static analysis techniques based on forward analysis [7, 17, 20, 13] for computing arithmetic inequality relationships do not simultaneously address the technical challenges of path-sensitivity, non-linear operators, and multiple variables, and additionally do not scale to very large programs (See detailed discussion in Section 2.1). Program testing based forward analysis techniques generally aim at high code coverage and are not goal directed. KLEE [3] performs an interprocedural forward symbolic execution for finding bugs. It performs dynamic path pruning, expression simplification and uses a variety of heuristics to increase the scalability their approach. Systems like DART [9] and CUTE [21] combine symbolic analysis with
concrete execution to improve the coverage of random testing. These techniques, in worst case, need to explore huge number of program paths (path explosion problem). In this work, path explosion is partly overcome by goal directed backward analysis and our custom simplification of expressions.

**Array Bound Analyses:** There has been a large body of work in the area of bounding array index expressions for static detection of buffer overflow errors [19, 23]. These techniques are typically based on linear relational analysis and are usually precise enough to keep track of constant terms. In contrast, the focus of our work is to identify (possibly non-linear) bounds that are asymptotically precise.

11. CONCLUSION

We have presented a precise and scalable analysis for computing upper bound on an expression at any program location. There are two novel features of our approach—

(1) Proof rules for reasoning about loops; our experiments have shown that large number of loops could be handled by using relatively few proof rules. We, therefore, argue that with a sufficient understanding of the problem domain, one can better reason about the loops using few observable patterns.

(2) Abstract bounds and virtual call resolution; our experiments have shown that large number of loops could be handled by using different techniques.

12. REFERENCES


**Table 3:** Statistics for Proof Rules, Virtual Call Resolution and Abstract Bounds on 4 .NET Benchmarks—(A) .NET Core, (B) .NET Libraries, (C) Facebook .NET API, (D) Silverlight Media Player