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-

*The author performed this work during a summer internship at Microsoft Research.

Ex1(uint z_1,z_2)	Ex2(uint z_1,z_2)	Ex3(uint z_1,z_2)	Ex4(uint z_1,z_2)	Ex5(uint z_1, z_2)

$i := 0$;
$n := z_2$;
while (i++ $< z_1$)
    if (nondet())
        $n := n + 2$;
$t := n$;
	$i := 0$;
	$n := z_2$; flag $:= 1$;
	while (i++ $< z_1$)
	    if(flag $\wedge$ nondet())
	        $n := n + 2$; flag $:= 0$;
	$t := n$;
		$i := 0$;
		$n := 0$;
		while (i++ $< z_1$)
		    $j := 0$;
		    while (j++ $< z_2$)
		        $n := n + 1$;
		$t := n$;
			$i := 0$;
			$n := 0$;
			while (i++ $< z_1$)
			    $j := 0$; $n := 0$;
			    while (j++ $< z_2$)
			        $n := n + 1$;
			$t := n$;
				$i := 0$;
				$n := 0$;
				while (i++ $< z_1$)
				    $j := 0$; $m := n + 1$;
				    while (j++ $< z_2$)
				        $m := m + 1$;
				    $n := m$;
				$t := n$;

Iterations$(5, 2) \leq z_1 - i$	Iterations$(5, 2) \leq 1$	Iterations$(6, 2) \leq (z_1 - i) \times z_2$	Iterations$(6, 4) \leq z_2 - j$	Iterations$(\{4, 6\}, 2) \leq (z_1 - i) \times (1 + z_2)$
$t \leq z_2 + 2z_1$	$t \leq z_2 + 2$	$t \leq z_1 \times z_2$	$t \leq z_2$	$t \leq z_1 \times (1 + z_2)$

Figure 1: Examples of skeletons from .Net code-base that update the variable to be traced backwards (variable n in these examples) inside loops. nondet() refers to non-deterministic abstraction of a conditional. These examples illustrate the precision of our proof rule for loops for computing an upper bound on a variable.

ments, 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 abstract implementations of such method calls. We then address these challenges using our notion of *abstract* bound expressions that provide 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 lvalues 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 experimental results, wherein 90% percentage of bounds that would have otherwise been undefined, could use such abstract constructs that provided more meaningful information.

We have implemented our solution to the bound analysis problem as part of the SPEED tool for computing symbolic computational complexity bounds for procedures in user-written code. Such a tool can help programmers understand the performance characteristics of their code as well as that of unfamiliar APIs. The SPEED tool already implements algorithms to compute symbolic *local* bound on the number of iterations of a loop, but this bound is expressed in terms of variables that are live immediately before the loop. We translate this local bound to a bound in terms of the procedure inputs by using the bound analysis presented in this paper. Our bound analysis can successfully trace back 94% percent of a total of 9152 queries on an average of 0.23 seconds per query.

Contributions and Organization.

- We present a parametrized solution to the bound analysis problem (formally described in Section 3.1). Our solution is parametrized by a reaching definition analysis (Section 3.2) and an iteration count analysis (Sec. 3.3). This allows use of off-the-shelf technology and easy leverage of future improvements in these analyses.
- We introduce the notion of *abstract bound expression* for expressing results of bound analysis (Section 4). This helps alleviate the theoretical challenge of undecidability of bound analysis and addresses practical challenges of dealing with arbitrary heap locations and use of interfaces/inheritance in object-oriented languages.
- We describe the core backward symbolic execution for bound analysis in Section 5 and show how to incorporate information from conditional guards in Section 7 and perform virtual call resolution in Section 8.
- We present a proof rule based technique to reason about (bound of variables that get updated inside) loops. This involves leveraging power of SMT solvers and understanding of design patterns to allow the symbolic execution engine to perform a precise reasoning of loops in an efficient non-iterative manner (Section 6).
- We present experimental results illustrating the scalability and effectiveness of various components of our solution to the bound analysis problem (Section 9).

2. MOTIVATING EXAMPLES

In this section, we describe some examples from .Net code base 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 t 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 addressing only a specific challenge and/or do not scale to large programs. The simplest example in Figure 1 is Procedure $Ex4$, where the bound on n can obtained after establishing the inductive loop invariant $n \leq j$. Such invariants can be automatically established by using the abstract-interpretation based analysis over octagon abstract domain [17], which consists of inequality relationships between two variables with unit coefficients. This analysis scales well to large programs; however, it cannot be used to discover the required invariants in any of the other procedures. The procedure $Ex1$ requires establishing the inductive loop invariant $n - z_2 \leq 2i$, which requires performing abstract interpretation over the polyhedral domain [7], which can compute arbitrary *linear inequality relationships among multiple program variables.* The polyhedral domain is not a good choice because it does not scale to large programs and is restricted to discovering only linear relationships. Procedure $Ex2$ requires performing a *path-sensitive analysis* that combines boolean reasoning with numerical reasoning [13], (to establish the inductive loop invariant $\texttt{flag} \Rightarrow n = 0$ and $\neg\texttt{flag} \Rightarrow n = 2$); however such an analysis is inherently expensive. Procedures $Ex3$ and $Ex5$ both require *non-linear invariant* generation techniques [20] that do not scale beyond small programs.

In contrast, our proof-rule based technique uniformly addresses the challenges of path-sensitivity, non-linearity, presence of multiple variables in a scalable manner. For example, in $Ex3$, we first establish that along all paths through the inner loop, the value of n increases by at most 1. Such a check can be done by transforming the loop update into an SMT query and using existing SMT solvers like Z3 [24]. The analysis then uses existing techniques [12, 10] to compute bound on the total number of visits to location 6 (the location where n is updated). This bound is denoted by $\texttt{Iterations}(6, 2)$ and is computed to be the value of $(z_1 - i) \times z_2$ at program location 2. Finally, our analysis computes bound on n at location 7 as the sum of the bound on n after the assignment $n := 0$ at location 3 and the value of $\texttt{Iterations}(6, 2) \times 1$, which is computed to be $z_1 \times z_2$. Same proof rule can be applied to all loops in Figure 1. We have also developed similar proof rules for monotonic-decrease and assignment to objects inside loops. Our observation is that our proof rules were sufficient to reason about 80% of the loops encountered on our benchmark programs.

2.2 Practical Challenge: Heap and Interfaces

The key practical challenge in bound analysis arises when the bound depends on heap data or on interface methods whose code is not available. The former is challenging because we require a language to refer to arbitrary heap location, especially when it cannot be determined statically. The latter is challenging because we do not have the source code available.

Ex6(uint z, Array<int> A, List L_1)	Ex7(ICollection<int> C, Array<int> A)
$n := 0;\ i := 0;$	$n := C.\texttt{Count};\ i := 0;$
$L := L1;$	while $(i\texttt{++} < A.\texttt{length})$
while $(i\texttt{++} < z)$	$n := n + 1;$
if (nondet())	$C.\texttt{Add}(A[i]);$
$n := n + 1;$	$t_1 := C.\texttt{Count};$
$L := L.\texttt{next};$	$t_2 := n;$
$t_1 := A[n];$	
$t_2 := L.\texttt{data}$	
$t_1 \leq A[\bot]$	$t_1 \leq \texttt{Cond}(C.\texttt{count}+A.\texttt{length})$
$t_2 \leq R(L_1, \{\texttt{next}\}).\texttt{data}$	$t_2 \leq C.\texttt{count} + A.\texttt{length}$

Figure 2: Examples of skeletons from .Net codebase that motivate the need for an *abstract* bound expression.

Consider the procedure $Ex6$ in Figure 2. Observe that bound on t_1 depends on the contents of array A and bound on t_2 depends on the data field of some element of list L. We propose representing these bounds using abstract bound expressions $A[\bot]$ (denoting any index of the array) and $R(L, \{\texttt{next}\}).\texttt{data}$ (denoting the data field of any object object reachable from L by following the next field) as opposed to representing it by $\bot$ (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 most 1, we can compute a bound of $C.\texttt{Count} + A.\texttt{length}$ 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 $\texttt{Cond}(C.\texttt{Count} + A.\texttt{length})$. 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 π, 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 π. A procedure consists of assignment statements and conditional guards of the following form:

$$\begin{array}{rl} \texttt{Assignment Statement} & \pi: \quad t := e \\ \texttt{Guard} & e_1 \ \texttt{relop} \ e_2 \end{array}$$

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

$$\begin{array}{rcl} e & := & c \mid t \mid e_1 \pm e_2 \mid op(e_1, e_2) \mid x.m(t_1, t_2) \mid new(\tau) \\ t & := & x \mid x[e] \mid x.f \end{array}$$

```
Ex8(C x, C y
        Array<int> A,
        uint z)
1   x.f := A[0];
2   i := z;
3   if (nondet())
4       A[i] := z;
5       x := y;
6   else
7       i := z + 4;
8       t := x.m();
9       y.f := z;
10  t1 := A[i];
11  t2 := x.f;
12
```

```
Ex9(uint z, uint[] A,
        List L, List L')
n := 0; i := 0;
while (i++ < z)
    if (nondet())
        n := n + 1;
    if (nondet())
        L := L.next;
t1 := n + 1;
t2 := 100 − n;
t3 := A[n];
t4 := L.data;
```

Bounds computed by our analysis: $t_1 \leq z + 1$, $t_2 \leq 100$, $t_3 \leq A[\perp]$, $t_4 \leq R(L, \{\texttt{next}\}).\texttt{data}$

Figure 3: Two examples (obtained from combining code snippets from .Net code base). $Ex8$ illustrates different types of `Definitions` for an lvalue (Section 3.2). $Ex9$ 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, op denotes an operator other than addition and subtraction, $x.m(t_1, t_2)$ denotes a method invocation[1] to method m of class that denotes the runtime type of object x, and $\texttt{new}(\tau)$ returns a fresh object of class τ.

3.2 Reaching Definitions

We use the notation $\texttt{Definitions}(\ell, \pi)$ to denote the set of immediate/reaching definitions that may determine the value of ℓ at location π. 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 ℓ at procedure entry is same as ℓ at location π.
- If d is of the form $\texttt{Simple}(\pi' : \ell' := e)$, then ℓ' at location π' is same as ℓ at location π.
- If d is of the form $\texttt{Index}(\pi' : \ell' := e)$, then ℓ at location π' is equal to some index expression in ℓ at location π.
- If d is of the form $\texttt{Object}(\pi' : \ell' := e)$, then ℓ at location π' is equal to some object dereference in ℓ at location π.
- If d is of type $\texttt{SideEffect}(\pi' : \ell' := o.m(\ell_1, \ldots, \ell_n))$, then the call to method $o.m$ may modify ℓ.

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

Example 1. *Consider procedure $Ex8$ in Figure 3. We have $\texttt{Definitions}(t_1, 12) = \{\texttt{Simple}(4 : A[i] := z), \texttt{Index}(7 : i := z + 4)\}$, and $\texttt{Definitions}(t_2, 12) = \{\texttt{Simple}(1 : x.f := A[0]), \texttt{Simple}(5 : x := y), \texttt{Object}(9 : y.f := z), \texttt{SideEffect}(8 : t := x.m())\}$. The definition $\texttt{Object}(9 : y.f := z)$ is included because y may-alias with x.*

[1]We assume, without loss of generality, that a method takes two arguments. Our theory can be trivially extended to any method taking arbitrary number of arguments.

We classify any definition d for (ℓ, π) as either *inductive* or *non-inductive* depending on whether or not it flows to ℓ along a back-edge. Let $\texttt{DefsInd}(\ell, \pi)$ and $\texttt{DefsInit}(\ell, \pi)$ denote the disjoint partition of $\texttt{Definitions}(\ell, \pi)$ into inductive and non-inductive definitions respectively.

Example 2. *For procedure $Ex5$ in Figure 1, we have:*

$$\begin{aligned} \texttt{Definitions}(n, 4) &= \{\texttt{Simple}(1 : n := 0;), \\ &\quad\ \texttt{Simple}(7 : n := m;)\} \\ \texttt{DefsInit}(n, 4) &= \{\texttt{Simple}(1 : n := 0;)\} \\ \texttt{DefsInd}(n, 4) &= \{\texttt{Simple}(7 : n := m;)\} \end{aligned}$$

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 $\texttt{Iterations}(P_1, P_2)$ 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 $\texttt{Iterations}(\pi_1, \pi_2)$ to denote $\texttt{Iterations}(\{\pi_1\}, \{\pi_2\})$.

$\texttt{Iterations}(P_1, P_2)$ can be computed by using the procedure described in [12] for computing an upper bound on the number of times a given program location π is visited in terms of variables that are live at the entry to the outermost loop containing π. Let's refer to this procedure as $\texttt{Visits}(\pi)$. We can compute $\texttt{Iterations}(P_1, P_2)$ using the procedure `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 $\texttt{Visits}(\pi_1)$ for all $\pi_1 \in P_1$.

Example 3. *The second row in Figure 1 gives examples of $\texttt{Iterations}(\pi_1, \pi_2)$ for respective program locations π_1 and π_2 for procedures $Ex1$ to $Ex5$. Since π_2 is located outside the outermost loop containing π_1 for the procedures $Ex1$,$Ex2$, and $Ex3$, $\texttt{Iterations}(\pi_1, \pi_2)$ is same as $\texttt{Visits}(\pi_1)$ for each of these procedures. For the procedure $Ex4$, computation of $\texttt{Iterations}(6, 4)$ essentially involves computing $\texttt{Visits}(6)$ in $Ex4$ 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 β for representing bounds. The language β is similar to that of the program expression language except that it uses abstract lvalues ℓ and two new constructs $\perp$ and $\texttt{Cond}(\beta)$. The abstract lvalue ℓ is similar to standard lvalues t except that it uses one new construct $R(\ell, F)$.

$$\begin{aligned} \beta &:= c \mid \ell \mid \beta_1 \pm \beta_2 \mid op(\beta_1, \beta_2) \mid \perp \mid \texttt{Cond}(\beta) \\ \ell &:= x \mid \ell[\beta] \mid \ell.f \mid R(\ell, F) \end{aligned}$$

$\perp$ 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 $\perp$ allows for providing more meaningful information. For example, $A[\perp]$ denotes content at an arbitrary location in array A, $\perp[0]$ denotes the element at the first location of an arbitrary array, and $\perp.f$ denotes the f field of an arbitrary object.

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

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. $\texttt{Cond}(\beta)$ either denotes β or $\perp$ 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 (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.*

$\texttt{Clear}() : \{\texttt{this.Count} := 0;\}$

$\texttt{Add}(\ell) : \{\mathit{if}\ (\mathit{nondet}())\{\texttt{this.Count} := \texttt{this.Count} + 1;\}\}$

$\texttt{Remove}(\ell) : \{\mathit{if}\ (\mathit{nondet}())\{\texttt{this.Count} := \texttt{this.Count} - 1;\}\}$

$\texttt{Contains}(\ell) : \{\mathit{skip};\}$

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 $B_{\texttt{pol}}(\ell, \pi)$ and $B_{\texttt{pol}}(e, \pi)$ to denote the result obtained by tracing lvalue ℓ and expression e respectively, at location π backwards in mode `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.

$$B_{\texttt{pol}}(e_1+e_2, \pi) = \{\beta_1+\beta_2 \mid \beta_1 \in B_{\texttt{pol}}(e_1, \pi), \beta_2 \in B_{\texttt{pol}}(e_2, \pi)\} \quad (1)$$
$$B_U(e_1-e_2, \pi) = \{\beta_1-\beta_2 \mid \beta_1 \in B_U(e_1, \pi), \beta_2 \in B_L(e_2, \pi)\} \quad (2)$$
$$B_L(e_1 - e_2, \pi) = \{\beta_1 - \beta_2 \mid \beta_1 \in B_L(e_1, \pi), \beta_2 \in B_U(e_2, \pi)\}$$
$$B_E(e_1 - e_2, \pi) = \{\beta_1 - \beta_2 \mid \beta_1 \in B_E(e_1, \pi), \beta_2 \in B_E(e_2, \pi)\}$$
$$B_{\texttt{pol}}(o.m(\ell_1, \ell_2), \pi) = \{\beta \mid \beta \in B_{\texttt{pol}}(\beta_1, \pi), \beta_1 \in B_{\texttt{pol}}(r, \pi_{\texttt{exit}})[\ell_1/z_1, \ell_2/z_2]\}$$

Above $\pi_{\texttt{exit}}$ denotes the exit location of method m, r denotes the return variable of method m, z_1, z_2 denote the formal parameters of method m and $[\ell_1/z_1, \ell_2/z_2]$ denotes replacement of z_1 by ℓ_1 and z_2 by ℓ_2 respectively.

For expressions that use other operators, the symbolic execution engine drops the contextual information and traces using the criterion of exact equality.

$$B_{\texttt{pol}}(op(e_1, e_2), \pi) = \{op(\beta_1, \beta_2) \mid \beta_1 \in B_E(e_1, \pi), \beta_2 \in B_E(e_2, \pi)\}$$
$$B_{\texttt{pol}}(c, \pi) = \{c\}$$

Simplification: As described above, we trace an expression by tracing each of its lvalues individually. This individual tracing helps reducing the path space for exploration However, computation of $B_{\texttt{pol}}(e, \pi)$ in equation 1 requires one to combine the sets obtained by tracing individual lvalues and result in an explosion in set size. Our backward symbolic execution uses a custom simplifier to simplify the resulting set based on contextual information and mode under which tracing is done. When $\texttt{pol} = U$, we return the set of maximal expressions out of the total set of expressions obtained from the analysis. This set of maximal expressions can be found by issuing a simple SMT query. Similarly, when $\texttt{pol} = L$, our analysis returns the set of minimal expressions out of the total set of expressions. Our experiments have shown that this simplification helps in significantly reducing the sizes of resulting expressions.

5.3 Tracing Of Lvalues

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

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

The backward symbolic execution traces back across any non-inductive non-$\texttt{SideEffect}$ definition as follows.

$$\begin{array}{rcl}
B_{\texttt{pol}}(\ell,\pi,\texttt{Input}) &=& \{\ell\}\\
B_{\texttt{pol}}(\ell,\pi,\texttt{Simple}(\pi':\ell:=e)) &=& B_{\texttt{pol}}(e,\pi')\\
B_{\texttt{pol}}(\ell,\pi,\texttt{Index}(\pi':\ell':=e)) &=& \{\beta \mid \beta\in B_{\texttt{pol}}(\ell[\beta_1/\ell'],\pi'),\\
&& \beta_1\in B_E(e,\pi')\} \qquad (4)\\
B_{\texttt{pol}}(\ell,\pi,\texttt{Object}(\pi':\ell':=e)) &=& \{\beta \mid \beta\in B_{\texttt{pol}}(\ell[\beta_1/\ell'],\pi'),\\
&& \beta_1\in B_O(e,\pi')\} \qquad (5)
\end{array}$$

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_1 requires computing an upper bound on n after the loop (Eq. 1). Computing an upper bound on t_2 requires computing a lower bound on n after the loop (Eq. 2). Computing an upper bound on t_3 and t_4 requires identifying an expression(s) 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 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.

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

Above $\pi_{\texttt{exit}}$ 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 ℓ 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 6, m is assigned to $m+1$. On the path from 6 to 4, n and m both 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 $0+(z_1+z_1\times z_2)\times 1=z_1+z_1\times z_2$.*

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

$$\begin{array}{l}
\texttt{BInit}_{\texttt{pol}}(\ell,\pi)=\{\beta \mid \beta\in B^{T,a}_{\texttt{pol}}(\ell',\pi',d),(\ell',\pi')\in T,\\
\qquad\qquad d\in\texttt{DefsInit}(\ell',\pi')\}\\
\texttt{BInd}_{\texttt{pol}}(\ell,\pi)=\{\beta \mid \beta\in B^{T,b}_{\texttt{pol}}(\ell',\pi',d),(\ell',\pi')\in T,\\
\qquad\qquad d\in\texttt{DefsInd}(\ell',\pi')\}\\
\texttt{PInit}_{\texttt{pol}}(\ell,\pi)=\{\texttt{Label}(d) \mid (\ell',\pi')\in T,\\
\qquad\qquad d\in\texttt{DefsInit}(\ell',\pi'), B^{T,a}_{\texttt{pol}}(\ell',\pi',d)\neq\emptyset\}\\
\texttt{PInd}_{\texttt{pol}}(\ell,\pi)=\{\texttt{Label}(d) \mid (\ell',\pi')\in T, d\in\texttt{DefsInd}(\ell',\pi')\}
\end{array}$$

Above, $B^{T,b}_{\texttt{pol}}(\ell_1,\pi_1)$ is the function that traces back ℓ_1 at location π_1 across one iteration of the loop containing $\texttt{PInd}_{\texttt{pol}}$. It can be implemented in exactly the same manner as $B_{\texttt{pol}}(\ell_1,\pi_1)$ with the exception that backward tracing is stopped when any $(\ell_2,\pi_2)\in T$ is encountered, and ℓ_d is returned. ℓ_d refers to some fresh variable that does not occur in the program and is meant to represent all lvalues in T. $B^{T,a}_{\texttt{pol}}(\ell_1,\pi_1)$ is the function that traces back ℓ_1 at location π_1 along the paths take go outside of the loop. It can be implemented in exactly the same manner as $B_{\texttt{pol}}(\ell_1,\pi_1)$ with the exception that backward tracing is not performed along paths that iterate inside the loop.

$$\begin{array}{rcl}
B^{T,a}_{\texttt{pol}}(\ell_1,\pi_1) &=& \emptyset \text{ if } (\ell_1,\pi_1)\in T\\
B^{T,b}_{\texttt{pol}}(\ell_1,\pi_1) &=& \{\ell_d\} \text{ if } (\ell_1,\pi_1)\in T
\end{array}$$

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.

$$\begin{array}{rcl}
B_U(\ell,\pi) &=& \{\beta+\texttt{giter}\times Max(0,c) \mid \beta\in\texttt{BInit}_U(\ell,\pi)\}\\
&& \quad \text{if } \forall\beta'\in\texttt{BInd}_U(\ell,\pi):\ \beta'\leq\ell_d+c \qquad (6)\\
&=& \bot \text{ otherwise}
\end{array}$$

where c is some constant, $\texttt{giter} = B_U(\texttt{iter}, \pi')$ and $\texttt{iter} = \texttt{Iterations}(\texttt{PInd}_U(\ell,\pi), \texttt{PInit}_U(\ell,\pi))$ denotes an upper bound on the number of combined visits to locations in $\texttt{PInd}_{\texttt{pol}}(\ell,\pi)$ in terms of the variables that are live at π', where π' is some program location that is outside the loop that contains program locations in $\texttt{PInd}_{\texttt{pol}}(\ell,\pi)$.

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 $Ex1$, $Ex2$ lead to exactly identical tracing except for computation of $\texttt{Iterations}(\textit{PInd}_U(n,5), \textit{PInit}_U(n,5))$ $= \texttt{Iterations}(5,2)$, which is computed to be $z_1 - i$ and 1 for $Ex1$ and $Ex2$ 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 $Ex1$ and $Ex2$ respectively.

Procedures $Ex3$ and $Ex4$ also lead to exactly identical tracing except for computation of $\textit{BInit}_U(n,6)$, which is computed to be $\{2 : \ n := 0\}$ and $\{4 : \ n := 0\}$ respectively. This leads to $\texttt{Iterations}(\textit{PInd}_U(n,6), \textit{PInit}_U(n,6))$ being computed as $\texttt{Iterations}(6,2) = (z_1 - i) \times z_2$ and $\texttt{Iterations}(6,4) = z_2 - j$ for $Ex3$ and $Ex4$ respectively. This leads to asymptotically different bounds of $z_1 \times z_2$ and z_2 for n and t after the loops in $Ex3$ and $Ex4$ respectively.

Procedure $Ex5$ demonstrates the need for tracing multiple inductive lvalues. During tracing of $(n,8)$, the helper functions get invoked with $(n,4)$ (since $(n,4)$ contain an inductive definition) and return the following:

$$\textit{Transitive}(n,4) = \{(n,4),(m,6)\}, \quad \textit{BInd}_U(n,4) = \{\ell_d + 1\}$$
$$\textit{BInit}_U(n,4) = \{0\}, \textit{PInd}_U(n,4) = \{4,6\}, \textit{PInit}_U(n,4) = \{2\}$$

Since $\texttt{Iterations}(4,2) = z_1 - i$ and $\texttt{Iterations}(6,2) = (z_1 - i) \times z_2$, we obtain $\texttt{Iterations}(\textit{PInd}_U(n,4), \textit{PInit}_U(n,6)) = \texttt{Iterations}(\{4,6\},2)$ to be $(z_1 - i) \times (1 + z_2)$. $\textit{BInd}_U(n,4)$ 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 lvalues have bounded decrease instead of bounded increase. This leads to the following backward symbolic execution strategy.

$$\begin{aligned} B_L(\ell,\pi) \quad &= \quad \{\beta - \texttt{giter} \times Max(0,c) \mid \beta \in \texttt{BInit}_L(\ell,\pi), \\ &\qquad \text{if } \forall \beta' \in \texttt{BInd}_L(\ell,\pi) : \ \beta' \geq \ell_d - c \qquad (7) \\ &= \quad \bot \text{ otherwise} \end{aligned}$$

where $\texttt{giter}$ 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 $\bot$.

$$B_E(\ell,\pi) \quad = \quad \bot \qquad (8)$$

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

Ex10()	Ex11(uint z_1, z_2)

`n := nondet();`
`if (n > 100) return;`
`t := n;`
	`n := z2;`
	`while (z1 < n)`
	`    if (nondet())`
	`        n := (z1 + n)/2;`
	`    else break;`
	`t := n;`

$t \leq 100$	$t \leq z_2$

Figure 4: Examples of skeletons from .Net codebase that require making use of conditional guards for computing bound on the variable to be traced backwards (variable t in these examples).

of recursive fields.

$$\begin{aligned} B_O(\ell,\pi) \quad &= \quad \{R(\beta, \texttt{Dereferences}(\ell,\pi)) \mid \beta \in \texttt{BInit}_O(\ell,\pi), \\ &\qquad \text{if } \texttt{Dereferences}(\ell,\pi) \neq \bot \qquad (9) \\ &= \quad \bot \text{ otherwise} \end{aligned}$$

where $\texttt{Dereferences}(\ell,\pi)$ is defined as follows.

$$\begin{aligned} \texttt{Dereferences}(\ell,\pi) \quad &= \quad \{f \mid \ell.f \in \texttt{BInd}_O(\ell,\pi)\} \\ &\qquad \text{if } \forall \beta \in \texttt{BInd}_O(\ell,\pi)\ \exists f : \ \beta \equiv \ell.f \qquad (10) \\ &= \quad \bot, \text{ otherwise} \end{aligned}$$

EXAMPLE 8. *As explained in Example 5, tracing t_1, t_2, and t_3 backwards require in procedure $Ex9$ 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 is 1, and $\texttt{giter}$ 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 is -1 and hence $max(0,c) \times \texttt{giter}$ 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 $R(L, \{\texttt{next}\}).\texttt{data}$. Note that if L was traced back in equality mode, then it would have provided an upper bound of $\bot.\texttt{data}$, which is less precise than the information provided by $R(L, \{\texttt{next}\}).\texttt{data}$ since the former implies that t_4 is bounded above by the $\texttt{data}$ field of any object, while the latter implies that t_4 is bounded above by the $\texttt{data}$ field of only those objects that are reachable from input list L_1 via the $\texttt{next}$ 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 $\texttt{Iterations}(P_1, P_2)$ 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 $B_{\texttt{pol}}(\ell, \pi, d)$ returns $\perp$ in (Eq. 3). Whenever, this happens, we can try the following alternative in its place.

$$B_{\texttt{pol}}(\ell, \pi, d) = B_{\texttt{pol}}(e, \pi') \text{ if } \texttt{Guard}(\ell, \pi, d) \Rightarrow (\ell \; \texttt{relop}_{\texttt{pol}} \; e) \quad (11)$$

where $\texttt{relop}_U$ denotes the $\leq$ operator, $\texttt{relop}_L$ denotes the $\geq$ operator, and $\texttt{relop}_E$, $\texttt{relop}_O$ both denote the equality comparison operator. $\texttt{Guard}(\ell, \pi, d)$ 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 π is determined by d. $\texttt{Guard}(\ell, \pi, d)$ 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 $B_U(n, 3, \textit{Simple}(1 : n := nondet())) = \perp$. Note that $\textit{Guard}(n, 3, \textit{Simple}(1 : n := nondet())) = (n \leq 100)$. 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 $\tilde{B_U}(\ell, \pi)$ contains a symbolic expression β 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 $\texttt{Guard}(\ell, \pi, \beta)$, where $\texttt{Guard}(\ell, \pi, \beta)$ denote the boolean condition (in terms of variables that are live at $\texttt{PointsTInductive}(\ell, \pi)$) that must be true if ℓ if β flows into ℓ at location π. A similar refinement can also be performed for the check $\beta \geq \ell - c$ in Eq. 7 and the check $\beta \equiv \ell.f$ in Eq. 10. $\texttt{Guard}(\ell, \pi, \beta)$ can be computed by conjuncting together all gating functions $\texttt{Guard}(\ell, \pi, d)$ corresponding to the definitions d used in computation of β.

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 $\textit{DefsInd}(n, 6) = \{\textit{Simple}(4 : n := (z_1 + n)/2\}$ and $\tilde{B_U}(n, 6) = \{(z_1 + n)/2\}$. 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 $\textit{Guard}(\ell, \pi, \beta) = (z_1 < n)$ 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 $B_{\texttt{pol}}(o.m(\ell_1, \ell_2), \pi)$, 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 $E = B_E(o, \pi)$. If $\perp \in E$ or $\beta \in E$ such that $\texttt{TypeOf}(\beta)$ is extensible, then we define $B_{\texttt{pol}}(o.m(\ell_1, \ell_2), \pi)$ to be $\perp$. Otherwise, we obtain the set S of possible methods m from E as follows: $S = \{\texttt{TypeOf}(\beta) \mid \beta \in E\}$, where $\texttt{TypeOf}(\texttt{new}(\tau)) = \tau$, and $\texttt{TypeOf}(\beta)$, where β 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 overapproximation of $\texttt{Definitions}(\ell, \pi)$ for every lvalue and location in the program, by performing various sound dataflow 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 $\texttt{Iterations}$ 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

Procedure Depth	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16	17
Frequency	16088	2447	1122	424	308	63	26	45	18	1	5	8	12	10	1	2

Procedures visited	1	2	3	4	5	6	7	8	9	10	11-20	21-30	31-40	41-50	51-100	100-200
Frequency	16094	1873	857	576	428	178	130	133	69	40	152	15	26	7	7	1

Expression Set size	1	2	3	4	5	6	7	8	9	10	11-20	21-30	31-40	41-50	51-100	>100
Frequency	15308	308	118	24	19	4	8	9	2	7	16	2	8	1	1	1

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.

Benchmark	.NET Core	.NET Libraries	Facebook .NET API	Silverlight Media Player
# Queries	4259	2963	625	1305
# Successes	4050	2631	612	1243
Success (%)	95.09%	88.80%	97.92%	95.25%
# Functions	85834	78671	18116	34683
Size (LOC)	3560621	3187007	678128	2168345
Time (sec)	799.28	777.26	124.93	269.92

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

execution and one of the queries visited as many as 151 procedures. Similarly, more than 90% queries encountered no nested procedures. However, many queries encountered nested procedures, having procedure stack depth of more than 10, with two queries going through procedure stack of depth 17. The total number of execution paths in the procedures corresponding to these queries is huge, but our (goal-directed) analysis only visited a tiny fraction of these paths.

Finally, to show the effectiveness of our expression simplification, we measured the size of set of expressions obtained for each query. The statistics are shown in last two lines of Table 1. This shows that in about 95% of cases, the analysis returned a singleton set of expressions. The largest set size observed during our analysis was 109. We then turned off all simplifications and again ran our experiments on one of the benchmarks. The analysis threw an OutOfMemoryException since the internal data structures could not handle sets of extremely large sizes.

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 plot shows that about 51% 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 $A[\perp]$), defined in section 4. We also computed the number of bound expressions that were arbitrary, i.e. $\perp$. 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 ($\perp$).

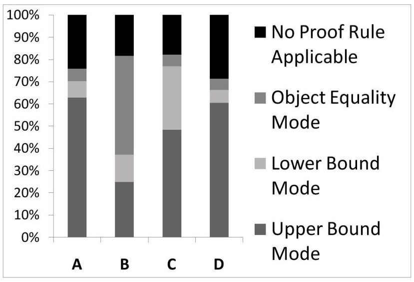
10. RELATED WORK

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.

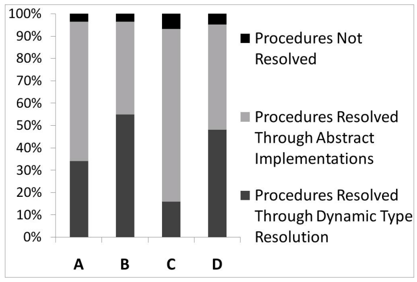
PSE [16] presents a static analysis to diagnose software failures. It tracks the flow of a single value from a program location to the program entry. It performs a novel dataflow analysis and pointer analysis to reason about heap. But like SNUGGLEBUG, PSE does not use any sophisticated reasoning about 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).

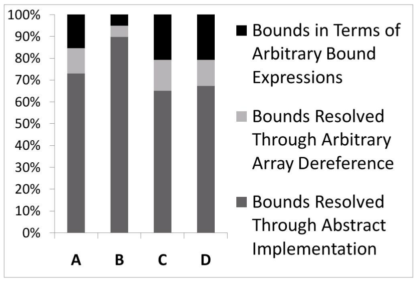
Program testing based forward analysis techniques generally aim at high code coverage and are not goal directed. KLEE [3] performs an interprocedural forward symbolic execution for finding bugs. It performs dynamic path pruning, expression simplification and uses a variety of heuristics to increase the scalability their approach. Systems like DART [9] and CUTE [21] combine symbolic analysis with



(I) % of loops that can be analyzed using different proof rules



(II) % of abstract methods resolved using different techniques



(III) % of expressions expressed in terms of abstract bounds

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

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 experience shows that a large number of expressions provided more meaningful information than undefined (i.e. $\perp$) because of these techniques.

12. REFERENCES

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *SAS*, pages 221–237, 2008.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Resource usage analysis and its application to resource certification. In *FOSAD*, 2009.

[3] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[4] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *PLDI*, pages 363–374, 2009.

[5] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *POPL*, 2010.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

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

[8] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.

[9] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, 2005.

[10] S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, pages 375–385, 2009.

[11] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, 2009.

[12] S. Gulwani and F. Zuleger. The reachability-bound problem. Technical Report MSR-TR-2009-146, Microsoft Research, Oct 2009.

[13] A. Gurfinkel and S. Chaki. Combining predicate and numeric abstraction for software model checking. In *FMCAD*, pages 1–9, 2008.

[14] G. A. Kildall. A unified approach to global program optimization. In *POPL*, pages 194–206, 1973.

[15] P. Malacaria. Assessing security threats of looping constructs. In *POPL*, pages 225–235, 2007.

[16] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. Pse: explaining program failures via postmortem static analysis. *SIGSOFT SEN*, 29, 2004.

[17] A. Miné. The octagon abstract domain. In *WCRE*, pages 310–319, 2001.

[18] Phoenix Compiler. research.microsoft.com/phoenix/.

[19] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM TOPLAS*, 27(2):185–235, 2005.

[20] S. Sankaranarayanan, H. Sipma, and Z. Manna. Non-linear loop invariant generation using gröbner bases. In *POPL*, pages 318–329, 2004.

[21] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE*, 2005.

[22] P. Tu and D. Padua. Gated ssa-based demand-driven symbolic analysis for parallelizing compilers. In *International Conf. on Supercomputing (ICS)*, pages 414–423, 1995.

[23] A. Venet and G. P. Brat. Precise and efficient static array bound checking for large embedded c programs. In *PLDI*, pages 231–242, 2004.

[24] Z3 SMT solver. research.microsoft.com/projects/Z3/.