Incrementally Precise Program Analysis

Duc-Hiep Chu  
National University of Singapore  
hiepcd@comp.nus.edu.sg

Joxan Jaffar  
National University of Singapore  
joxan@comp.nus.edu.sg

Vijayaraghavan Murali  
Rice University  
vijay@rice.edu

Abstract

Program analysis has been dominated by Abstract Interpretation (AI), owing to its scalability. AI is typically not (or only partially) path-sensitive thus the obtained level of accuracy could be arbitrarily low. Recently, there have been works on path-sensitive program analysis, applied to domains where accuracy is critical. However, they suffer from the path explosion problem and are not scalable in general. In this paper, we present a general framework for program analysis that incrementally increases accuracy as it iterates.

We start with a formulation of analysis which is both a lower-bound as well as upper-bound analysis of the program’s behavior. This duality allows for a specification of precision in the overall analysis. We then define an abstract representation of the program which we can iteratively refine. The iterations allows for early termination when a user-definable level of precision in the analysis has been extracted. The critical performance factors are (a) the incrementality of the refinement step where results from previous iterations are persistent for future iterations, (b) reuse of analysis from subproblems that have precise analysis, and most importantly, (c) a concept of domination which allows a lower-bound analysis to prune away subproblems. Finally, we demonstrate the framework with real programs on a backward counting analysis and a forward data flow analysis. We show that in many cases, our iterative method is in fact superior to both AI as well as algorithms designed to run continuously till an exact analysis is found.

1. Introduction

Program analysis has been so far dominated by some form of Abstract Interpretation (AI) where abstract program properties are propagated through abstract transitions induced by the program. AI implementations are typically path-insensitive (or only partially path-sensitive, even by carefully designing the abstract domain), and often, only one description of the analysis problem and are not scalable in general. In this paper, we present a general framework for program analysis that incrementally increases accuracy as it iterates.

In the theoretical setting of unlimited resource budget, a naive algorithm that enumerates all possible symbolic paths would produce an “exact” analysis. This is the ideal precision we would aim for in this paper.

However, in practice, the scalability issue is what prevents us from getting ideal precision. While there has been progress in path-sensitive algorithms [5, 13, 15] by exploiting the concepts of interpolation and reuse; these may not always scale. In a realistic setting where a budget, typically in terms of memory or time, is limited, depending on such a completely accurate algorithm, one could end up with no analysis result at all.

The main contribution of this paper is a general method to combine these two extreme approaches by starting with an efficiently obtained but potentially imprecise analysis, then refining the problem, and then iterating. Our method constitutes an “anytime algorithm” [3]. Termination occurs when (a) we have a resource budget exhaustion, where the limited resource is typically memory and time, or (b) when the precision is known to be sufficiently precise, a condition that is user-definable, in which case we have “early termination”.

A basic requirement for the refinement process is convergence, i.e., there is a final iteration which produces an exact analysis. This implies that each iteration is guaranteed to produce equal or more precision than the previous iteration. Additionally, it produces useful information that monotonically reduces the complexity of the remaining analysis problem.

We begin with a traditional formulation of program analysis defined over an “abstract domain” [8]. The domain provides an analysis formula for a set of abstract states that over-approximate the set of concrete states that are indeed reachable. Thus an analysis is sound, applying to all states. We shall further specify our analysis as lower bound and upper bound, and there is an intuitive notion of precision when comparing analysis values, which assumed to form a lattice structure \( R \). Consequently, the notion of an exact analysis is straightforward. Overwhelmingly, examples of analyses come from numerical bounds of variables, or from inclusion bounds for sets of variables, where the notion of lower and upper bound is very clear. In the end, we seek an iterative analysis which monotonically decreases the gaps between the lower and upper bound formulas so that we can have a useful analysis after any number of iterations. More importantly, by defining a notion of “sufficiently close”, we have a flexible and rigorous way of obtaining early termination.

The conceptual core of our framework is centered on the symbolic execution tree (SET) of a program – a tree representing all possible symbolic paths – from which the exact analysis can be extracted. While this tree is often too big to compute explicitly due to the exponentiality of the number of symbolic paths, we instead compute a smaller hybrid SET (HSET), a SET where some subtrees of symbolic paths may be replaced by AI nodes. Such a node is, intuitively, an over-approximation of the analysis of the subtree it replaces but is efficiently obtained through AI. More specifically,
an AI node contains two things: (a) an upper bound analysis \( \mathcal{U} \) that soundly approximates the desired analysis, and (b) a collection of witness paths \( \omega \), which is a subset of paths that the AI node represents but sufficient to give rise to \( \mathcal{U} \). This collection is usually small in size, as not every individual path often contributes to the upper bound analysis. Now these witness paths, because they are obtained using abstract, as opposed to precise, interpretation, may not be “real” witnesses due to one of two reasons, one concerning feasibility and the other concerning optimality. In other words: (i) one of the witness paths in \( \omega \) may not actually be feasible in the SET, or (ii) the upper bound realized by \( \omega \) is not an exact analysis of the corresponding symbolic paths in the SET. Apart from the AI nodes, every other node \( v \) in a HSET also has an analysis: if \( v \) is a terminal node, then its analysis is simply that arising from the one symbolic execution path this node represents. If however \( v \) has two successors, then it inherits an analysis as a “join” of the two successor analyses in the obvious way.

Thus a HSET always provides an analysis of the program (via its root node representing all the symbolic execution paths). In the search for precision, we will embark on a series of refinement phases to the HSET. Each phase starts by choosing a particular AI node \( v' \) in the HSET to refine into a HSET by performing symbolic execution in order to generate one complete symbolic path \( \pi \) starting from \( v' \). Importantly, the search for this path shall start with a witnesses path of \( v' \), and proceed until we obtain one of two outcomes: (a) a confirmation that the witness path is “real”, or (b) a refutation that the witness path is real. In case (a) the benefit is that we discovered an optimal analysis for \( v' \). In case (b) the benefit is that we can discard the misleading witness paving the way to improve the (upper bound) analysis of \( v' \). Let us call the symbolic path that we obtained, a spine.

The new HSET formed by the refinement of \( v' \) is the tree rooted at \( v' \), such that any branch deviating from the spine \( \pi \) is now replaced by a (new) AI node. The key idea now is twofold:

- Because we have a symbolic path (the spine), we may perform exact analysis on it, in particular obtaining an exact analysis for its terminal node.
- We may use this exact analysis to obtain lower bound analyses for the intermediate nodes in the spine, which can then be used to prune away other AI nodes (equipped with upper bounds).

This is an instance of our general concept of domination: if a node \( v \) has an upper-bound analysis \( \mathcal{U} \) and a node \( v' \) has a lower-bound analysis \( \mathcal{L} \) such that \( \mathcal{U} \sqsubseteq \mathcal{L} \), where \( \sqsubseteq \) is the partial order relation in the lattice \( \mathcal{R} \), then \( v' \) dominates \( v \).

To make the presentation uniform, we assume henceforth that every node \( v \) in the HSET has both an upper bound and lower bound analysis. If \( v \) is a leaf representing an AI node, its upper bound is defined by \( \mathcal{U} \) and lower bound is usually trivial (\( \bot \)). If \( v \) is any other leaf representing the end of a spine, it has an exact analysis, i.e., its bounds are equal. Any intermediate node in the HSET has its upper (lower) bound defined by the join of the upper (lower) bounds of its successor nodes. If the joins resulted in the bounds being equal, the intermediate node also has an exact analysis.

We now summarize the HSET refinement process on a HSET \( T \). The base case is when \( T \) contains no AI nodes, in which case its root node can be subject to an exact analysis. For the case where there are AI nodes in \( T \), the big question is how to choose the next AI node to refine. The answer is simply: choose an AI node which is not dominated, and has a maximal upper bound, i.e., one whose upper bound is not included, as defined by the lattice \( \mathcal{R} \), in any other’s upper bound.

The following two results highlight key features of our algorithm, and distinguish our framework from existing methods.

**Proposition 1** (Linearity). No symbolic path which has been (exactly) analyzed is ever analyzed again. \( \square \)

More precisely, this means that the algorithm is, at worst, computationally proportional to the size of the original (theoretical) SET. Of course, we seek to do much better than this by discarding subtrees of this SET as we progress. This proposition follows easily from the structure of a HSET and the fact that refinement takes place in a new AI node. This property is the core of our “incrementality”, and is in contrast with CEGAR-style framework [4] where the work performed in one iteration is agnostic about previous iterations. We will elaborate more on this in the Related Work (Section 7).

**Proposition 2** (Progress). Let \( v \) be an AI node which is not dominated, and has a maximal upper bound. Then, performing a refinement phase on a node other than \( v \) will not give rise to a domination of \( v \). \( \square \)

This follows from the fact that domination of \( v \) can occur iff there is another node \( v' \) whose lower bound analysis includes \( v' \)’s upper bound analysis. But note that any future lower bound analysis of \( v' \) must be included in the current upper bound analysis of \( v' \). This leads to the fact that \( v' \)’s upper bound must be included in the current upper bound of \( v' \). Since \( v' \)’s upper bound is maximal among all current upper bounds, this is impossible. This mandates that the refinement step at this point pick \( v \) in order to achieve progress, because as long as \( v \) exists in the HSET, its upper bound will be included in the final analysis.

As a result of the above, we are able to produce an efficient implementation. We now outline some key reasons for the scalability of our method.

**Scenario 1 (Reuse of Abstract Analyses):**
In the refinement of a node \( v \) to produce a spine path of length \( n \), we generally produce \( n - 1 \) new AI nodes attached to the spine. However, it is typical in AI implementations (which was used on the node \( v \)) to have computed analysis for all program points that are reachable from \( v \)'s program point (via the CFG), and not just for that of the root \( v \). Therefore much of the analyses required for the new AI nodes are typically already at hand.

**Scenario 2 (Infeasibility):**
By exposing the spine, we are extending the path constraint that is subject to satisfiability testing, and therefore increasing the likelihood of discovering an infeasible path. In this scenario, we exploit the situation where one of the successors of the node being refined is unsatisfiable. This means that an entire subtree of symbolic paths, which previously had been included in the AI node \( v \), now has been removed.

**Scenario 3 (Reuse of Exact Analyses):**
Here we use a computed exact analysis of one subtree to derive an exact analysis of another subtree. Suppose we have an exact analysis \( E \) for a subtree rooted at \( v \). In this scenario, we exploit \( E \) to compute another exact analysis \( E' \) for another (yet unexplored) node associated with the same program point as \( v \). In general, the witness condition for such a reuse (here we are talking about real witnesses for the exact analysis), as well as the precise definition of the mapping from \( E \) to \( E' \), is quite involved because it depends on the kind of analysis in question. But in specific instances, this is

---

1 This path could either be feasible or infeasible.
2 We assume the join operator in \( \mathcal{R} \) is precise.
easily done. We thus omit a full description here but instead refer to [5, 13, 15], and use an example of reuse in Section 2.

**Scenario 4 (Domination):**

Here we exploit the situation when we have computed a nontrivial lower-bound analysis, say for node \( v \). Now we can in fact prune all subtrees in the entire HSET which are dominated by \( v \). Note here that domination does not require that the two entities involved represent the same program point, in contrast to reusing. In other words, any node/subtree can dominate any other. Another difference between reuse and domination is that both parties involved in a reuse contribute an analysis; it is just that we have a quick way to compute one of them from the other. Domination however means that we can simply ignore the dominated party.

In summary for these scenarios, the general idea is that during the refinement process, the effects of infeasible paths, reuse, and domination serve to produce more lower-bound analyses, and these, in turn, produce further opportunities for reuse and domination.

We assume in this paper that the program’s SET is finite and so the symbolic execution of loops always terminates. Dealing with unbounded loops, which make the SET infinite, is then relegated to standard approaches such as using loop invariants. We elaborate on this in Section 4.

Finally, in Section 6 we demonstrate the framework on two kinds of analyses. The first is (high-level) Worst-Case Execution Time (WCET) analysis. This is an example of a backward counting analysis. The second example analysis is a forward data flow analysis (such as that used to discover tainted variables). In running several realistic examples, we show that the incremental iterations indeed produce precision gains progressively, and the final analysis is almost always more precise than that obtained through AI. Importantly, in some examples, our algorithm terminates (i.e., producing an exact analysis) faster than the best custom algorithms that are designed to pursue an exact analysis in one iteration.

### 2. An Example

Consider here a trivial class of programs which contain only assignment statements of the form \( \text{tick} \leftarrow \kappa \) where \( \kappa \) is a positive number, and consider only non-nested if-then-else statements with unspecified guards \( b_i \) which do not depend on the variable \( \text{tick} \).

The example analysis is an abstraction of the well-known worst-case execution time (WCET) analysis, and in our simple setting, the analysis formulas are simply bounds on \( \text{tick} \), and the final analysis is to determine the (upper) bound of \( \text{tick} \) at the end.

Consider the program and its SET in Figure 1. Assuming that any combination of the unspecified guards is satisfiable, that is, that \( B_1 \land B_2 \land B_3 \) is satisfiable, where \( B_i \) is either \( b_i \) or \( \neg b_i \), \( 1 \leq i \leq 3 \), it is easy to see that the WCET is 6, obtained from the leftmost path. Before we proceed, note that in the general case where not all combinations of guards are satisfiable (as we will consider in this Section), and remaining within our trivial programming language, the problem to find the WCET is NP-hard [13].

Before we start our analysis we will first demonstrate a use of reuse. Assume that \( \neg b_1 \land b_2 \land b_3 \) is satisfiable, and that we already have an exact analysis, \( \text{tick} = 3 \) of the right subtree marked \((2')\) in Figure 1. We now can produce an exact analysis for the left subtree marked \((2)\) without having to traverse it. To do this, we take the longest path in the right subtree which gave rise to the analysis, i.e., the witness path, and this is the leftmost path under \((2')\). Call this path \( p_1 \). We now replay this path in the left subtree, getting the leftmost path starting from the root. Call this path \( p_2 \). Now the idea is that the analysis of \( p_2 \) is computed from the analysis of \( p_1 \), which is 3. However, since the prefix of \( p_2 \) from the root to node \((2')\), which increments \( \text{tick} \) by zero, differs from the prefix of \( p_2 \) from the root to node \((2)\), which increments \( \text{tick} \) by 3, we must adjust for this and now declare that the exact analysis of node \((2)\) is \( \text{tick} = 6 \). In other words, we assumed that the longest increment of \( \text{tick} \) from node \((2)\) downwards is the same as that from node \((2')\), which is 3. But since the prefix of node \((2)\) is 3 more than the prefix of node \((2')\), we add a further 3 to obtain the final value 6.

There are two further points to note about reuse in general.

- **Suppose** \( b_1 \land b_2 \land b_3 \) (corresponding to the leftmost path in the left subtree) is unsatisfiable. We can still perform reuse, i.e., we declare that the analysis of node \((2)\) is 6, but this will be imprecise. To prevent this imprecision, one needs to check that the path under node \((2)\) that corresponds to the witness is feasible.

- **Now suppose** \( \neg b_1 \land b_2 \land b_3 \) (corresponding to the leftmost path in the right subtree) is unsatisfiable but \( b_1 \land b_2 \land b_3 \) (corresponding to the leftmost path in the left subtree) is satisfiable. Now it is unsound to reuse the exact analysis of node \((2')\) (which now is different from 3) in the analysis of node \((2)\). In previous implementations of reuse, e.g.: [5, 13, 15], the exact analysis would be accompanied by an interpolant which would ensure that the reuse can soundly take place. In a setting more general than the WCET example in this Section, we would further need to check that the subtree to which reuse is considered satisfies not just one or more feasible witness paths, but that the optimality of these witness paths carries over from the source analysis.

Our algorithm performs analysis by providing a lower and upper bound for \( \text{tick} \) in the set of paths indicated by each node. For example, an upper-bound analysis for the left subtree in Figure 1, labelled \((2)\), is \( \text{tick} \leq 6 \). This subtree also can have a lower-bound analysis of a nonnegative number less than or equal to 6; for example, if we knew that the path proceeding to the left successor of \((3')\) was feasible, then we could record down that \( 4 \leq \text{tick} \) is a lower bound. If on the other hand we did not care to check the feasibility of any path going through \((2)\), then we could quickly estimate that 3 is a lower bound (by choosing only rightmost branches that do not add to \( \text{tick} \)). Note that there may not actually be a real execution path resulting in \( \text{tick} = 3 \). Note also that lower bounds whose values are too low (e.g., \( 0 \leq \text{tick} \)) are not very useful.

We now proceed to analyze the program incrementally. See Figure 2 where \( "@" \) denotes an AI node, the \( l \) and \( u \) superscripts

![Figure 1: Example Program and its Symbolic Execution tree](image-url)
denote lower and upper bounds respectively, and the $T_i$s represent the HSET we construct in each iteration. We start with a single AI node at $T_1$ representing an (abstract) analysis of the program starting at the beginning. We could have used traditional abstract interpretation (AI) which over-approximates the set of paths in the SET in order to limit consideration to a small number of abstract states (typically, one state per program point). Thus AI analyzers are typically very efficient. We then quickly, because the analyzer is path-insensitive, determine a (trivial) lower bound of 0 and an upper bound of 6. Furthermore, the analyzer indicates that the leftmost path is a witness path, i.e., if it were feasible, then it would indicate the true WCET. In Figure 2, we show only upper bounds when the lower bound is trivial.

Next we refine the single AI node $T_1$ into the HSET $T_2$ which now contains new nodes, amongst them two AI nodes at $(3')$ and $(3')$. Using abstract interpretation, note that former has an upper bound of 3, while the latter has an upper bound of 4. We assume that the constraint $b_1 \land \neg b_2 \land b_3$ is unsatisfiable, and so the leftmost path in Figure 1 is in fact infeasible (at just before program point $(3)$). Now since node $(3')$ has a bound 4, this is inherited by the parent node $(2)$. Finally, the root node $(1)$ inherits the larger of the bounds of its successors, which are 3 and 4, and so we obtain a final bound of 4. Now since $T_2$ contains AI nodes which contribute to this answer, this analysis is not confirmed to be exact.

Finally we deal with the two remaining AI nodes in $T_2$, and choose one of them to refine. We choose the node $(3')$ over $(2')$ because its upper bound is higher. The intuition is this: if we instead chose to refine the AI node with the smaller bound, the other AI node will still need to refined in the future. If, as we will show next, we choose the AI node $(3')$ with the higher bound, there is a chance that the remaining AI node can be dominated. We now obtain $T_3$ by refining this AI node.

This refinement produced two successors, and by assuming that the constraint $b_1 \land \neg b_2 \land b_3$ is unsatisfiable, we have that the left subtree of node $(3')$ is an infeasible path. The right subtree is a terminal node, and so for the first time, we can declare that, since both subtrees of $(3')$ have no AI-nodes, $(3')$ has a lower bound\(^3\) of 3. The most interesting step now can be taken: the analysis here dominates the analysis at the one remaining AI node at $(2')$. Note that the set of paths represented by $(2')$ is nearly half of all the paths. By pruning away this subtree, we now have that the entire tree has no more AI nodes, and we can now declare that the root node has an exact analysis of 3.

### 3. Preliminaries

**Syntax.** We restrict our presentation to a simple imperative programming language where all basic operations are either assignments or assume operations, and the domain of all variables are integers. The set of all program variables is denoted by $\text{Vars}$. An assignment $x := e$ corresponds to assigning the evaluation of the expression $e$ to the variable $x$. In the assume operator, assume$(c)$, if the boolean expression $c$ evaluates to true, then the program continues, otherwise it halts. The set of operations is defined by $\text{Ops}$.

We then model a program by a transition system (Fig. 1). A transition system is a quadruple $(\text{SymStates}, \ell_{\text{start}}, \rightarrow, \text{Ops})$ where $\text{SymStates}$ is the set of symbolic states and $\ell_{\text{start}} \in \text{SymStates}$ is the unique initial program point. $\rightarrow \subseteq \text{SymStates} \times \text{Ops}$ is the transition relation that relates a state to its (possible) successors executing operations. This transition relation models the operations that are executed when control flows from one program point to another. We shall use $\ell \xrightarrow{\text{op}} \ell'$ to denote a transition relation from $\ell \in \text{SymStates}$ to $\ell' \in \text{SymStates}$ executing the operation $\text{op} \in \text{Ops}$. Finally, $\text{Ops} \subseteq \text{SymStates}$ is the set of terminal program points.

**Symbolic Execution.** A symbolic state $v$ is usually defined as a triple $(\ell, s, \Pi)$. The symbol $\ell \in \text{SymStates}$ corresponds to the current program point. The symbolic store $s$ is a function from program variables to terms over input symbolic variables. The evaluation $[c]_s$ of a constraint expression $c$ in a store $s$ is defined as: $[v]_s = s(v)$ (if $v$ is a variable), $[n]_s = n$ (if $n$ is an integer), $[e \text{ op } e']_s = [e]_s \text{ op } [e']_s$ (where $\text{op}$ is a relational or arithmetic operator). $\Pi$ is called path condition and it is a first-order formula over the symbolic inputs and it accumulates constraints which the inputs must satisfy in order for an execution to follow the particular corresponding path. The set of first-order formulas and symbolic states are denoted by $\rightarrow_{\text{FO}}$ and $\text{SymStates}$, respectively.

For all purposes of this paper, we do not consider arbitrary symbolic states, but only those generated during our symbolic execution. Hence, we abuse notation to (re)define a symbolic state $v$ as the quadruple $(\ell, s, \Pi, \pi)$, where the additional parameter $\pi$ is a sequence of program transitions that were taken during SE in order to reach $v$.

Given a transition system $(\text{SymStates}, \ell_{\text{start}}, \rightarrow, \text{Ops})$ and a state $v \equiv (\ell, s, \Pi, \pi) \in \text{SymStates}$, the symbolic execution of $\ell \xrightarrow{\text{op}} \ell'$ returns another symbolic state $v'$ defined as:

$$\text{SYMSTEP}(v, \ell \xrightarrow{\text{op}} \ell') \equiv v' \triangleq \begin{cases} (\ell', s, \Pi \land [e]_s, \pi') & \text{if op } \equiv \text{assume}(c) \text{ and } \Pi \land [e]_s \text{ is satisfiable} \\ (\ell', s[x \mapsto [e]_s], \Pi, \pi) & \text{if op } \equiv x := e \end{cases}$$

where $\pi' \triangleq \pi \cdot \ell \xrightarrow{\text{op}} \ell'$. We call $v'$ a successor of $v$. Note that Eq. (1) queries a theorem prover for satisfiability checking on the path condition. In practice, we assume the theorem prover is sound but not necessarily complete. That is, the theorem prover must say a formula is unsatisfiable only if it is indeed so. Given a symbolic state $v \equiv (\ell, s, \Pi, \pi)$ we define $[v] : \text{SymStates} \rightarrow \text{FO}$ as the formula $\bigwedge_{v \in \text{Vars}} [v]_s \land \Pi$ where $\text{Vars}$ is the set of program variables.

\(^3\)In fact the upper bound of $(3')$ is also 3, i.e., it has an exact analysis.
A symbolic path \( v_0 \cdot v_1 \cdot \ldots \cdot v_n \) is a sequence of symbolic states such that \( 1 \leq i \leq n \) the state \( v_i \) is a successor of \( v_{i-1} \). A path \( v_0 \cdot v_1 \cdot \ldots \cdot v_n \) is feasible if \( v_n \equiv \langle \ell, s, \Pi, \pi \rangle \) such that \( \Pi \) is satisfiable. If \( \ell \in O \) and \( v_n \) is feasible then \( v_n \) is called terminal state, denoted \( \text{TERMINAL}(v_n) \). Otherwise, if \( \Pi \) is unsatisfiable the path is called infeasible and \( v_n \) is called an infeasible state, denoted \( \text{INFEASIBLE}(v_n) \). If there exists a feasible path \( v_0 \cdot v_1 \cdot \ldots \cdot v_n \) then we say \( v_n = (0 \leq k \leq n) \) is reachable from \( v_0 \) in \( k \) steps. We say \( v' \) is reachable from \( v \) if it is reachable from \( v \) in some number of steps. A symbolic execution tree contains all the execution paths explored during the symbolic execution of a transition system by triggering Eq. (1). The nodes represent symbolic states and the arcs represent transitions between states.

Finally, given a terminal state \( v \equiv \langle \ell, s, \Pi, \pi \rangle \), we assume the existence of a function \( \theta \) which extracts an “exact” analysis from \( \pi \). This exactness is a theoretical concept that helps us quantify the precision of our incremental analysis against a fully path-sensitive algorithm. We note that fully path-sensitive algorithms do exist for loop-free programs, but they often do not work in the setting of realistic memory/time budget.

**Abstract Interpretation.** An invocation of abstract interpretation (AI) at a symbolic state \( v \) constructs an AI node at \( v \), using an abstract domain \( A \). This step starts by making use of the abstraction function \( \alpha \) to map the current symbolic state \( v \) to an abstract value, i.e., computing \( \alpha(v) \), then performing a standard fixpoint computation over \( A \) and the input CFG.

We expect an invocation of AI would necessarily return an upper bound analysis \( U \), i.e., a safe over-approximation, of the set of paths through \( v \). In addition, we also assume that it produces witness paths denoted as \( \omega \) – a set of paths from which the upper bound analysis \( U \) is derived. In practice, \( \omega \) often is just a small subset of all the paths going through \( v \), since not all paths contribute to the returned analysis \( U \).

We assume the analysis values also form a lattice structure \( R \), or \( [S, \sqcup, \sqcap, 1, 0, \top, \bot] \), where \( S \) is the set of analysis values, \( \sqcup \) is the partial order relationship, \( \sqcap \) and \( 1 \) are the least upper bound and greatest lower bound operators, and \( \bot \) and \( \top \) are the bottom and top elements of the lattice respectively. We assume that \( \sqcup \) and \( \sqcap \) are precise so that exact analyses over different paths can be combined precisely, yielding an exact analysis over the collection of paths.

Note that unlike the abstract domain \( A \) to run AI, \( R \) does not concern (partially) the (in-)feasibility of the program paths. While \( A \) is typically designed to be coarse enough so that an invocation of AI is fast, \( R \) is designed with the precision criterion in mind. This in practice does not hamper the overall scalability of our algorithm, since operations on the analysis values defined by \( R \) are performed only a small number of times, bounded by the number of nodes in the HSET.

To simplify the presentation of the hybrid symbolic execution tree (HSET), we also assume that an invocation of AI also returns a lower bound analysis \( L \). In practice, one can always make use of a trivial lower bound, e.g., \( \bot \). If \( L \) is assumed to be the desired exact analysis of the set of paths, then \( L \subseteq E \subseteq U \) (we do not explicitly compute \( E \) but infer it when \( L \) and \( U \) coincide).

It now should be clear that under our notion of exactness and the above assumptions, the analysis for each terminal symbolic state \( v \equiv \langle \ell, s, \Pi, \pi \rangle \) is exact. Its lower bound and upper bound coincide at \( \theta(\pi) \).

We comment here that computing the witness paths is straightforward but tedious, so we do not detail it here. Typically an AI algorithm operates over a CFG. During its execution, it can “mark” certain edges in the CFG that are sufficient to produce the analysis of the target AI node where the analysis is invoked. The witness paths can then be obtained by traversing marked edges from the target AI node to a terminal node (of the CFG). In summary, it is reasonable to assume the existence of a procedure ABSTRACTINTERPRETATION which when invoked with a symbolic state \( v \) returns a triple \( \langle L, U, \omega \rangle \).

**Interpolation.** Given a pair of first order logic formulas \( A \) and \( B \) such that \( A \land B \) is false, an interpolant [9] is another formula \( \Psi \) such that \( (a) \; A \models \Psi, \; (b) \; B \land \Psi \models \bot, \; \text{and} \; (c) \; \Psi \) is formed using common variables of \( A \) and \( B \). An interpolant removes irrelevant information in \( A \) that is not needed to maintain the unsatisfiability of \( A \land B \).

Interpolation has been prominently used to reduce state space blowup in program verification [14, 19], analysis [5, 15] and testing [17]. Here we will use it for a similar purpose – to merge, or subsume, symbolic states and avoid redundant exploration. During symbolic execution, our algorithm will annotate certain states with an interpolant, which can be used to prune other symbolic trees.

Given a current symbolic state \( v \equiv \langle \ell, s, \cdot, \cdot \rangle \) and an already explored symbolic state at the same program point \( v' \equiv \langle \ell, s', \cdot, \cdot \rangle \) annotated with the interpolant \( \Psi \), we say \( v' \) subsumes \( v \), denoted as \( \text{SUBSUMES}(v', v) \) if (a) \( \Psi \models v \) and (b) \( \alpha(v') \subseteq \alpha(v) \).

The first condition ensures that the symbolic paths through \( v \) are a subset of the symbolic paths through \( v' \), and the second condition ensures that the HSET at \( v' \) has already been explored with a more general context \( \alpha(v') \). Therefore, by exploring \( v \) one cannot obtain more precise analysis than that has been already obtained by exploring \( v' \), and hence \( v \) can be subsumed.

We note that subsumption is a special form of reuse that has been briefly discussed in the early Sections. While reuse (with interpolation) has been exploited for different analysis problems [5, 15], formulating this concept for a general analysis framework is rather involved. For simplicity, we thus omit the detail.

To conclude this Section, we comment that efficient interpo- lation algorithms do exist for quantifier-free fragments of theorems such as linear real/integer arithmetic, uninterpreted functions, pointers and arrays, and bitvectors (e.g., see [6] for details) where interpolants can be extracted from the refutation proof in linear time on the size of the proof.

### 4. Algorithm

Our incremental analysis algorithm, whose pseudocode is shown in Fig. 3, can be expressed as one that starts with an abstract interpreta- tion (AI) node representing an abstract analysis of the whole program, and gradually refines the HSET using symbolic execution (SE) until the desired level of analysis precision is obtained. Since each node in the HSET corresponds to a symbolic state \( v \), we will call it node \( v \) for short. During SE, a forward traversal collects path constraints and checks for path feasibility, and a backtracking phase annotates each node \( v \) in the HSET with the following information: \( \langle L, U, \omega, \Psi \rangle \), representing the lower bound and upper bound analyses for the set of paths through \( v \), the witness paths for the upper bound analysis, and the interpolant at \( v \), respectively.

With this annotation, we now define our all important domination condition.

**Definition 1 (Domination).** A node \( v \) annotated with \( \langle L, U, \omega, \Psi \rangle \) is dominated by a node \( v' \) annotated with \( \langle L', U', \omega', \Psi' \rangle \) if \( U \sqsubseteq U' \). We also say that \( v' \) dominates \( v \), denoted as \( \text{DOMINATES}(v', v) \).

In other words, if a symbolic state produces an upper bound analysis that is already contained (lattice-wise) in the lower bound analysis of another state, it is considered dominated. Particularly, there is no use trying to refine it to reduce its upper bound analysis. Note that a node can dominate itself if its lower and upper bounds...
are the same (i.e., it has an exact analysis). Obviously a node with an exact analysis needs not to be refined further.

The main procedure, incrementalAnalysis, accepts the program P as a transition system, which we assume is a global variable to all procedures. In line 1, the initial state is created with $l_{\text{start}}$ as the program point, an empty store, the path condition true, and the empty sequence. In line 2 the initial HSET containing a single AI node is generated by calling abstractInterpretation with the initial state. This would return a possible lower bound, an upper bound and the witness paths $\omega$ for the upper bound.

Lines 3-10 define the main refinement loop. First, the set non-dominated AI nodes in the current HSET is collected in $R$. Any node in $R$ not a candidate for the refinement. Our important strategy is to choose a node with maximal upper bound analysis. In the case of WCET, it is easy since the analysis values range over positive integers. In other analyses, if possible, a “difference” metric can be defined to even measure the amount of (non) domination, and the AI node in $R$ with maximal difference can be chosen (see Section 5 for an example for taint analysis).

Once the AI node $v$ is chosen for refinement, the procedure refineUnfold is called along with the witness paths for its upper bound analysis $\omega$. When refineUnfold returns it would have annotated $v$ with new, possibly tighter, upper and lower bounds which are then propagated back to its ancestors by the procedure propagateBack. This process continues until the loop terminates by means of a BoundHeuristic, which is user-defined.

A straightforward BoundHeuristic check is to check if there are no non-dominated symbolic states. This forces the algorithm to terminate only when an exact analysis has been derived. However, a WCET analyzer could be content if, say, the difference between upper and lower bounds is less than 5%, in which case the heuristic can check if the root of the HSET (the initial state) is annotated with $(L, U, \omega, \emptyset)$ and $L \cap U = L \leq 0.05$. A taint analyzer may only care about whether the source of taint variables would flow to a particular subset of variables $S$, so the heuristic can check if $\forall (L, U, \omega, S) \subseteq L \Rightarrow S \in U$, or if $\exists (L, U, \omega, S) \subseteq L$, in order to ensure no-flow or flow of taint to the variables in $S$, respectively.

RefineUnfold is our main refinement procedure that accepts the current node $v$ and the set of witness paths $\omega$. It is a recursive procedure that refines an AI node by symbolically unfolding the paths in $\omega$, with the hope of either confirming or refute the current analysis of the node. There are four bases of this procedure:

- (Lines 27-28) If $v$ is an infeasible state, then it sets the lower and upper bounds to $\perp$, the set of witness paths for the upper bound to $\emptyset$, and the interpolant $\Psi$ to false to denote the infeasibility.
- (Lines 29-30) If $v$ is a terminal state, then an exact analysis for this symbolic path is achieved. Hence both the lower and upper bounds are set to $\theta(\pi)$ – the analysis extracted from this single path. The witness paths for this analysis can be set to $\emptyset$ because we will never refine an exact analysis in future. Finally, the interpolant is set to true. In addition, we set a (global) variable spine_done to true to signify that a spine (witness path) has been exercised fully, and can begin constructing AI nodes along the branches from this path later.
• (Lines 31-32) If \( v \) is subsumed by another state \( v' \), it simply sets \( \text{spine\_done} \) to \( \text{true} \). Implicitly, the lower and upper bounds, the witness paths and interpolant for \( v \) are copied over from \( v' \).

• (Lines 39-34) If \( \text{spine\_done} \) is \( \text{true} \), i.e., a spine has been explored already and we are exploring other branches from it, then it constructs an AI node at \( v \) by calling \text{ABSTRACTINTERPRETATION}. This would return a lower bound, upper bound and the witness paths for the upper bound. The interpolant is then set to \( \text{true} \), as there is no infeasibility to capture in the constructed AI node.

If the four bases fail, \text{REFINENO}\textit{FOLD} proceeds to the successors of \( v \) (lines 35-50). It first initializes the lower and upper bounds, the witness paths, and interpolant to \( \emptyset \), \( \emptyset \) and \( \text{true} \) respectively, which will be modified. Then we target the refinement to either confirm or refute the given witness path \( \sigma_v \) (lines 37-42). This is done by following the witness, applying \text{SYMSTEP} on \( v \) to construct the next symbolic state \( v' \). Then \text{REFINENO}\textit{FOLD} is called recursively. For each remaining transition, which is not part of the witness path, the algorithm proceeds similarly (lines 43-49). But note that, now a spine has been constructed, indicated by \( \text{spine\_done} \) being set to \( \text{true} \), a number of AI nodes will be constructed along the spine. We further comment that a typical AI algorithm, when invoked, will follow the input CFG and compute an analysis for each program point, not just for the point of invocation. Thus the number of AI invocations while seemingly overwhelming, can indeed be optimized by a simple caching mechanism. In our implementation of our practical applications in Section 5, this is never an issue.

We now detail on how analysis answer and the interpolant are aggregated. Upon returning from the recursive call, \( v' \) would have been annotated with some lower and upper bounds, witness paths, and interpolant. From this, the same information for \( v \) is computed by \text{joining} it with the existing information at \( v \) (line 47) using the straightforward \text{COMBINE} procedure. That is, the analysis of the set of paths through \( v \) is computed as the (lattice) join of the analysis of each individual path. The interpolant deserves some special treatment due to its back propagation. From the interpolant \( \Psi \) at \( v' \), the interpolant at \( v \) is computed by \text{joining} it with the existing interpolant \( \Psi \) with \( \text{wp}(\Psi, \text{op}) \) — the weakest liberal precondition [10] of \( \Psi \) w.r.t. the transition op. \( \text{wp} : FO \times Ops \to FO \) ideally returns the weakest formula on the current state such that the execution of op results in \( \Psi \). In practice we approximate \( \text{wp} \) by making a linear number of calls to a theorem prover, using techniques outlined in [14], which usually results in a formula stronger than \( \text{wp} \).

Finally, once either a base case or the recursive case is executed, \text{REFINENO}\textit{FOLD} annotates (lines 52-54) the current state with the information defined by one of the cases. An important check is made here: if the lower and upper bounds are the same, then we have an exact analysis at \( v \). Therefore, the witness paths can be set to \( \emptyset \) since we will never refine an exact analysis. But most importantly, if the check failed, then the bounds do not coincide, and the analysis is imprecise. A state with an imprecise analysis should not subsume any other state. Hence we change the interpolant to \( \text{false} \) before annotating \( v \) so that for all states \( v'' \), \text{SUBSUMES}(v, v'') \) would fail. A subtle corollary of this is that the first three base cases assign the same lower and upper bounds at \( v \), and the fourth base case (AI) usually assigns them different values. The recursive case is then dependent on the the bounds of the successors of \( v \).

The final procedure \text{PROPAGATEBACK} simply propagates the annotation at a given state \( v' \) to its ancestors up to its root of the entire tree at \( v_{\text{root}} \). In line 12, it obtains the parent state \( v \), and in lines 13-18 it performs the backward propagation from all successors of \( v \), in exactly the same way as lines 36,46-48 of \text{REFINENO}\textit{FOLD}. For brevity, we provide its pseudocode but omit a detailed description.

The whole algorithm is guaranteed to terminate provided \text{ABSTRACTINTERPRETATION} terminates (see discussion on unbounded loops below). In case the algorithm is interrupted and forced to terminate, the current lower bound and upper bound can be extracted straightforwardly from the symbolic states and presented to the user, making this an “anytime algorithm”.

**Discussion on Loops**

**Bounded Loops** cause no problem to our described algorithm. For example, in the domain of WCET analysis, most loops are required to be statically bounded. A common treatment in this analysis domain is that loops are statically unrolled. In this case, all the technical description so far is very much applicable, as the program’s SET is finite.

**Unbounded Loops** pose a technical problem as they make the SET infinite, thereby making \text{REFINENO}\textit{FOLD} non-terminating. As we only work with finite SE trees, the only possibility to get termination is by using an abstraction such as a loop invariant. We employ invariant generation techniques outlined in [14, 15]. Particularly, we assume that program points are labeled with invariants inferred from an external invariant generator, typically using abstract domains such as octagons or polyhedra [23], and a function getasrt which, given a program point \( \ell \) and symbolic store \( s \), returns an assertion in the form of a FOL formula, renamed using \( s \), that holds at \( \ell \). Note that when \( \ell \) is a loop header, getassert will return a loop invariant. Then, we add a new case to \text{SYMSTEP}:

\[
\text{SYMSTEP}(v, \ell \xrightarrow{\text{op}} \ell') \triangleq \text{invariant}(\ell', s, \Pi, \ell \rightarrow \ell_n) \quad \text{if } \ell
\]

where \( \text{invariant}(\ell', s, \Pi, \ell \rightarrow \ell_n) \triangleq \{ \text{let } s' \leftarrow \text{HAVOC}(s, \text{MODIFIES}(\ell \rightarrow \ell_n)) \}
\]

where \( \text{HAVOC}(s, Vars) \triangleq \forall v \in Vars \bullet s[v \mapsto z] \) where \( z \) is a fresh variable (implicitly \( \exists \)-quantified).

\( \text{MODIFIES}(\ell \rightarrow ...) \) takes a sequence of transitions and returns the set of variables that may be modified during its symbolic execution.

Intuitively, invariant clears the symbolic store of all variables modified in the loop (using the \text{HAVOC} function) and then strengthens the path condition \( \Pi \) of the symbolic state with the externally obtained invariants. These invariants, once and for all, define the structure of the SET that will be explored iteratively by our algorithm. Note that we do not attempt to refine these invariants — they are provided to us by an “oracle”.

The notion of a spine is also affected by unbounded loops: a spine is no longer just a straight symbolic path from the root to a terminal point, but one that also contains looping points (cycles) at the loop headers. We can still extract a lower bound analysis from this spine by performing the standard fixpoint computation on the abstract domain \( A \), but undecidability restricts us from guaranteeing that this analysis is exact. Therefore, dominations performed using this lower bound may result in an imprecision of the overall analysis, but soundness is still guaranteed. The imprecision is affected by the quality (i.e., logical strength) of the given loop invariants, but this topic is outside the scope of this paper.

Finally, \text{REFINENO}\textit{FOLD} is modified so that if it detects a cycle at \( \ell \), it simply stops unfolding and returns, as \( \ell \) has already been explored with the loop invariant, thus guaranteeing its termination.
5. Implementation

We implemented the incremental analysis algorithm in Fig. 3 on the TRACER framework for symbolic execution, using the same interpolation method and theory solver presented in [16]. We instantiated our algorithm for (1) a backward WCET analysis and (2) a forward taint (data-flow) analysis.

5.1 WCET Analysis

For backward WCET analysis, the analysis values form the lattice $\mathcal{R}_1 \equiv \{ \mathbb{N}, \leq, 0, \cup, \cap, \infty \}$, with $\mathbb{N}$ is the set of non-negative integers, and $A_1 \cup A_2 \triangleq \max(A_1, A_2)$. The abstract domain $\mathcal{A}$ used for our AI component is the domain of intervals, denoted by $\mathcal{I}$, which is well-known for its efficiency.

We implemented the heuristics in Fig. 3 as follows. RefinementHeuristic is quite straightforward as the lattice $\{ \mathbb{N}, \leq, 0, \cup, \cap, \infty \}$ imposes a total order on its elements. Hence we simply pick for refinement the AI node that produced the maximum$^4$ upper bound WCET, with ties being resolved non-deterministically. BoundHeuristic implements the following check:

$$\forall v \exists v' \text{s.t. DOMINATES}(v', v),$$

that is, every symbolic state is dominated by another state, possibly by itself if it produces an exact analysis. This makes the algorithm terminate only when the final WCET is exact.

We emulated the low-level cache using a coarse model of an “instruction cache”. The cache size is 32 instructions, and the replacement policy on a cache miss is to simply erase the cache and fetch the next 32 instructions in the current basic block. It takes 1 unit of time to execute a program statement, and the cache miss penalty is 128 units of time.

5.2 Taint Analysis

For forward taint analysis, the analysis values form the lattice $\mathcal{R}_2 \equiv \{ \mathbb{P}, \subseteq, \emptyset, \cup, \cap, \setminus, \mathbb{V} \}$, with $\mathbb{P}$ is the powerset of $\mathbb{V}$; the set of all program variables, and standard operators on sets. The abstract domain $\mathcal{A}$ is defined as $\mathcal{I} \times \mathcal{R}_2$, where $\mathcal{I}$, as before, is the domain of intervals. In the literature, there are various definitions of how taint information is to be propagated, and we follow the one in [22], which considers both both explicit and implicit tainting. Explicit taint occurs when there is a direct data-flow from a tainted variable, say $t$, to another variable $x$ (e.g., through an assignment $x=t+r$). Implicit taint occurs when there is an equality check on a tainted variable, such as $\text{if}(t=x)$, which intuitively makes the other argument $x$ also tainted, as one can observe its value to find the value of $t$.

Typically, the source of taint affects the propagated taint information. But for lack of any information regarding the source, we picked all variables whose values were obtained from outside the program, including external variables, user input, environment variables, etc., and tainted some random subset of them. This models the usual scenario where some parts of the environment are “sensitive” (such as a user’s password input) and others are not (such as the system time).

For RefinementHeuristic unlike WCET, the lattice $\mathcal{R}_2$ does not impose a total order since two variable sets may be incomparable. Therefore we defined a notion of “difference” in domination as follows. First, we compute the lattice join $(\cup)$ of all lower bound taint sets in, say, a set $L_B$. Then, for each $v$ that roots an AI node annotated with $(\mathcal{L}, \cup, \omega, \overline{\mathcal{V}})$, we compute the difference set $\delta_v \equiv \mathcal{L} \setminus L_B$. We then pick the AI node at $v$ that produces the maximal value of $|\delta_v|$. That is, we pick the AI node that causes a maximum difference in the cardinality of tainted variable sets compared to the collective lower bound. Ties, when two sets produce the same difference in cardinality, are resolved non-deterministically. For instance, if two AI nodes, at $v$ and $v'$, produced the taint sets $(a, b, c)$ and $(a, c, d)$, and the collective lower bound $L_B$ is $(a, b)$, then we pick $v'$ to refine, as it produces the difference $(c, d)$ as opposed to just $(c)$ from $v$. Finally, BoundHeuristic implements the same check as for WCET analysis, which ensures that the taint information is exact when the algorithm terminates.

6. Experimental Evaluation

We used as benchmarks sequential C programs from a varied pool – three device drivers edaudio, diskperf, floppy from the rtdrivers-simplified category and SSH Client protocol from the ssh-simplified category of SV-COMP 2014 [2], an air traffic collision avoidance system teas, and two programs from the Mälardalen WCET benchmark [18] statemachine and nsichneu. We removed the safety properties from the SV-COMP benchmarks as we are not concerned with their verification. All experiments are carried out on an Intel 2.3 Ghz machine with 2GB memory, with a timeout of 5 minutes for WCET analysis and 1 minute for taint analysis, considering our nominal benchmark size. The lower timeout for taint analysis is because it is relatively cheap to perform than WCET, as will be evident from the timings in Table 2. This is because the abstract domain of taint analysis uses a finite lattice bounded by the number of program variables, and hence converge is achieved faster when compared to WCET which uses an infinite lattice of integers.

In both WCET and taint analysis, we compare our incremental algorithm with two adversaries: abstract interpretation (AI) on one hand, and state-of-the-art SE based algorithms on the other. For WCET analysis, we chose the algorithm presented in [5]. For taint analysis, we modified the algorithm in [15] to propagate forward taint information instead of slice information. These algorithms are highly path-sensitive, designed to produce exact analysis, and employ aggressive pruning techniques such as interpolation and reuse to achieve scalability.

In both experiments, we present the following statistics for each benchmark: (a) the final analysis produced by the AI-based, SE-based, and our incremental algorithm with upper $(\mathcal{U})$ and lower $(\mathcal{L})$ bounds (b) the time taken and (c) the total memory usage as given by the underlying TRACER system. We do not show the time and memory for the AI based algorithm as they are quite negligible compared to those of the other two algorithms. For instance, it always terminates in less than 1 second.

6.1 WCET Analysis

Table 1 shows the results of running WCET analysis on our benchmarks. The AI based algorithm produces an analysis quickly for all programs as mentioned above, but it is in fact not precise. As we will see, there is at least a 10% imprecision in most benchmarks, and an alarming 75% in nsichneu, a well-known program in the WCET community. So the only hope to produce an exact analysis is if the SE based algorithm terminates. However this fails to terminate by either timing out or running out of memory for four out of our seven benchmarks, leaving no useful analysis information.

On the other hand, our incremental algorithm is able to accomplish two things. It either terminates well before the SE algorithm, as in all but the last two programs, thus producing an exact analysis using much less budget (time and memory), or it produces a more precise range for the analysis using tighter upper and lower bounds, as in the last two programs. For instance, in nsichneu, AI produced the imprecise WCET 206788 and the SE-based algorithm ran out of budget. However we were able to produce the exact WCET of 52430 in less than half the budget and using only about 1/4th of the memory used by SE. This seems to be a common trend across all
Table 1. WCET Analysis results for AI based, SE based, and our incremental algorithm. An ∞ represents a timeout or out-of-memory.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1288</td>
<td>10668</td>
<td>9370</td>
<td>13784</td>
<td>28</td>
<td>212 MB</td>
</tr>
<tr>
<td></td>
<td>1255</td>
<td>33598</td>
<td>∞</td>
<td>29723</td>
<td>∞</td>
<td>2 GB</td>
</tr>
<tr>
<td></td>
<td>1524</td>
<td>16627</td>
<td>13784</td>
<td>39 MB</td>
<td>19 s</td>
<td>136 MB</td>
</tr>
<tr>
<td></td>
<td>2213</td>
<td>12394</td>
<td>6075</td>
<td>29723</td>
<td>17 s</td>
<td>39 MB</td>
</tr>
<tr>
<td></td>
<td>2540</td>
<td>206788</td>
<td>∞</td>
<td>52430</td>
<td>∞</td>
<td>522 MB</td>
</tr>
<tr>
<td></td>
<td>235</td>
<td>29305</td>
<td>∞</td>
<td>28788</td>
<td>∞</td>
<td>1.4 GB</td>
</tr>
<tr>
<td></td>
<td>1187</td>
<td>31281</td>
<td>∞</td>
<td>31515</td>
<td>285 s</td>
<td>∞</td>
</tr>
</tbody>
</table>

Table 2. Taint Analysis results. # TV measures the number of tainted variables. An ∞ represents a timeout or out-of-memory.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># TV</th>
<th>AI-based # TV</th>
<th>SE [15] # TV</th>
<th>Incremental # TV</th>
<th>Time (sec)</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>330</td>
<td>50</td>
<td>574 MB</td>
<td>45</td>
<td>45</td>
<td>227 MB</td>
</tr>
<tr>
<td></td>
<td>185</td>
<td>36</td>
<td>425 MB</td>
<td>31</td>
<td>31</td>
<td>101 MB</td>
</tr>
<tr>
<td></td>
<td>330</td>
<td>27</td>
<td>581 MB</td>
<td>20</td>
<td>20</td>
<td>229 MB</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td>57</td>
<td>8 MB</td>
<td>53</td>
<td>53</td>
<td>8 MB</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>16</td>
<td>24 MB</td>
<td>16</td>
<td>16</td>
<td>11 MB</td>
</tr>
<tr>
<td></td>
<td>41</td>
<td>15</td>
<td>55 MB</td>
<td>15</td>
<td>15</td>
<td>12 MB</td>
</tr>
<tr>
<td></td>
<td>119</td>
<td>67</td>
<td>770 MB</td>
<td>63</td>
<td>63</td>
<td>79 MB</td>
</tr>
</tbody>
</table>

Figure 4: Progressive Upper and Lower bounds over time for diskperf

our benchmarks, except ssh where we use relatively more memory than SE.

To observe how the upper and lower bounds incrementally converge in our algorithm, we take a closer look the diskperf benchmark that best exposes this phenomenon. Fig. 4 shows the progressive upper and lower bound WCET of this program over time. The monotonicity can be clearly seen – the lower bound always increases and the upper bound always decreases. Any point the algorithm is terminated, the bounds can be reported to the user. Observe that the difference between the bounds reduces to less than 20% in just over 15 seconds, and when they coincide we get the exact analysis at around 230 seconds. We noted that similar trends were exhibited among other benchmarks as well.

6.2 Taint Analysis

Table 2 shows the results of taint analysis on our benchmarks. The column # V shows the total number of variables in the program, and the columns labelled # TV shows the number of tainted variables. Of course, the analysis considers variable sets, but we show only the cardinality for presentation.

Except for the programs nsichneu and ssh, the analysis produced by AI is about 10-25% imprecise. Again, SE fails to terminate for four out of seven benchmarks, leaving only the imprecise result from AI to work with. In contrary, our incremental algorithm is able to terminate with an exact analysis for all benchmarks, and confirms the exactness in the three benchmarks that SE terminated on (ssh, nsichneu, tcas) using less resources than SE.

We conclude by stating that in both Tables 1 and 2, our incremental algorithm excels in some form in every row. We surpass the exact SE-based analyzer in performance by almost always using less resources, and we surpass the AI-based analyzer in accuracy by producing more precise analysis, together with a quantification of precision using the lower bound. In the end, we have empirically shown that our method makes “best use” of the resources than both our adversaries put together and fulfills our motive to be an “anytime algorithm”.

7. Related Work

Quantitative analysis, where executions are given a quality measure, covers a wide range of important applications such as for WCET; see [20, 25] for survey), power consumption [24], performance testing [1], to name a few. Such analyses are well-suited for “anytime algorithms” [3], which generate imprecise answers quickly, and then iteratively produce better solutions. The recent work [4] proposed both state-based and segment-based abstraction schemes, coupled with algorithms for counterexample-guided abstraction refinement (CEGAR) extended for quantitative properties. The refinement strategy here is based on an extremal counterexample trace, called the ext-trace. (In our paper, we called it the witness path.) The reason for choosing an extremal trace is, obviously, because a refinement which does not eliminate this trace would not improve the analysis.

Our choice of refinement step shares this motivation [4], by choosing, in some sense, in terms of likelihood of improvement. But there are significant differences:

- Our approach is not limited to a quantitative property, as we have demonstrated with a taint analysis.
- Our approach uses both lower and upper bound analyses, thus providing a precision measure for a more flexible terminating condition.
Our work possesses incremental performance. The results of the present iteration are be persistent, and can be (re-)used in the next iteration. In particular, paths are never analyzed twice.

The fundamental reason for this is that we operate using symbolic execution, which is in some sense the best abstract domain, and refine the Control Flow Graph (CFGs) with appropriate splitting. In contrast, [4] refine the abstract domain, thus affecting both CFGs and the quantitative properties (e.g. timing over abstract caches) at the end of each iteration. In other words, the relationship of our work to [4] is like Abstract Conflict Driven Clause Learning (ACDCL) [11] to traditional CEGAR [7]. We quote: “ACDCL never changes the domain, and this immutability is crucial for efficiency (over CEGAR), because the implementations of the abstract domain and transformers can be highly optimized” [11].

We have a concept of domination which leads to effective pruning. This is used in conjunction with a refinement strategy which chooses to refine an abstract node that is not just un-dominated, but one that remains un-dominated no matter which other choice of abstract node is made.

Finally, we mention some other related works, which share similar motivations as our proposed framework. Many customized abstract interpreters have been injected with some form of path-sensitivity to enhance the precision of the analysis results. A notable example is [21]. There have also been work on path-sensitive algorithms (under SMT setting) equipped with abstract interpretation in order to prune (potentially infinite number of) paths [12]. However, our framework differs significantly in the way the spines are interactively constructed. At one hand, we quickly refute spurious analysis from previous iteration while computing realistic lower bounds to exploit the new concept of domination for pruning. On the other hand, we can reach early termination when the spines confirm previously computed upper bound analyses are indeed precise.

8. Concluding Remarks

We presented an analysis framework whose algorithm starts with an efficiently obtained but not necessarily precise analysis, and then iteratively refines the problem in order to get more precision. A first feature is that a sound analysis is obtained after any number of iterations, with increasing precision, and exact precision is obtained eventually. A second feature is that our analysis comprises of both lower and upper bounds and so the final reported analysis comes with some quantification of precision. A third feature is that the algorithm is equipped with, due to having both lower and upper bounds, a concept of domination which can prune the search space. A fourth feature is that the algorithm is both incremental and goal-directed in its refinement process, and therefore pruning is, arguably, often effective.

Finally, our realistic benchmarks, on two complementary kinds (backward and forward) of analyses, show that our algorithm outperforms in almost all respects. On examples for which a non-iterative exact analyzer can terminate within a budget, our algorithm almost always utilizes less memory and time. For examples on which no known exact analyzer can terminate within that budget, our algorithm not only produces a more accurate analysis than an abstract analysis, but it does so with a precision quantification. In short, unless an abstract analysis that provides no quantification of its precision is deemed acceptable, our algorithm has been evaluated to be always superior than the state-of-the-art.

References