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
National University of Singapore
m.vijay@comp.nus.edu.sg

Abstract

Program analysis has been so far dominated by Abstract Interpretation (AI), owing to its scalability. AI is typically path-insensitive, 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 *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 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*, and often, only one description of the analysis is maintained for each program point. While an AI algorithm is generally efficient and scalable, its precision could be arbitrarily low. Perhaps more importantly, the level of precision is *unknown*.

At the other extreme, one could consider a *path-sensitive* algorithm using symbolic execution which, for programs with only finite symbolic paths, is *exact*. While there has progress in the scalability of path-sensitive algorithms [5, 13, 15] 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 could result in no result at all.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, contact the Owner/Author. Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481. Copyright held by Owner/Author. Publication Rights Licensed to ACM.

Copyright © ACM [to be supplied]. . . $15.00
DOI: http://dx.doi.org/10.1145/(to come)

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. Termination occurs when (a) we have a resource *budget exhaustion*, where the limited resource is typically memory and/or 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.. that there is a final iteration which produces an exact analysis. This implies that each iteration, while not guaranteed to produce more precision than the previous iteration, nevertheless produces useful information that monotonically reduces the complexity of the remaining analysis problem.

There are two additional requirements, which are crucial for scalability, for the iterative refinement step:

- INCREMENTAL PERFORMANCE
  The results of the present iteration should be *persistent*, i.e.: they can be (re-)used in the next iteration, without re-computation. In particular, no path which has been analyzed is ever analyzed again.
- INCREASED PRECISION
  The choice of the refinement for the next iteration should be *goal-directed* to be more likely to achieve early termination.

We begin with a traditional formulation of program analysis defined over an "abstract domain" [9]. The domain provides an *analysis formula* for a set of abstract states that overapproximate the set of concrete states that are really 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 analyses, and therefore also the notion of an *exact* analysis. Overwhelmingly, example of analyses come from numerical bounds of variables, or from inclusion bounds for sets of variables, where the notion of lower and upper bounds 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. While this tree is often too big to compute explicitly, we instead compute a smaller *hybrid SET* (HSET) which is a SET but where some subtrees may be replaced *AI nodes*; such a node is, intuitively, an analysis of the subtree it replaces that is efficiently obtained. More specifically, an AI node contains two things: (a) a lower[1] and upper bound analysis σ_l and σ_u, and (b) a collection of *potential witness paths*, which is a subset of paths that the AI represents which is sufficient to give rise to σ_l and σ_u. This collection is usually small in size. Now these potential witness paths, because they are obtained using abstract interpretation, may not be "real" witnesses because (i) one of them may not be feasible in the SET, or (ii) they collectively do not realize an upper bound analysis in the SET. Apart from the AI nodes, every other node N in a HSET also has an analysis: if N is a terminal node, then its analysis is simply that arising from the one symbolic execution path this node represents. If however N has two successors, then it inherits an analysis as a "join" of the two successor analyses in a straightforward way.

Thus a HSET always provides an analysis of the program (via its root node representing all the symbolic execution paths). What we seek, however, is that this analysis is precise, and this can only happen when there are no more AI nodes in the way. In order to achieve this, we need to be able to identify certain subtrees in the HSET whose analysis is *redundant*, and then *prune* them from further consideration.

Toward this end, we now introduce the most important concept of this paper, *domination*.

> if a node N_1 has a lower-bound analysis and a node N_2 has an upper-bound analysis such that the lower bound is greater or equal to the upper bound, then N_1 *dominates* N_2.

This essentially means that the entire subtree N_2 can *pruned* because the analysis of N_2 is now known to be superfluous.

A starting point for our program analysis is a HSET comprising of a single AI node representing an abstract analysis of the entire program. The iterative step, applied to a general HSET, is to apply a *refinement strategy* to choose a particular AI node in this HSET to refine into a hybrid (sub-)tree. This in general increases its precision of the resulting HSET.

We now walk through the HSET refinement process on T, whose base case is when T constrains no AI nodes, in which case its root node will indicate an exact analysis. In the general case, we now need to choose one AI node in T to refine, that is, to replace it with a HSET which, hopefully, will contain a more precise analyses than the AI node. The following choice, in conjunction with the use of the potential witness paths, is what makes our algorithm *goal-directed*:

> choose an AI node N which is (a) *not dominated*, and (b) has a maximal upper bound.

In other words, do *not* choose an AI node N which, upon the refinement of a *different* AI node, could lead to the domination of N.

See Figure 1(a) depicting a HSET where the leftmost AI node @ with upper bound σ_0 is refined into the subtree labelled with the upper-bound analysis σ in the second tree in Figure 1(b). Note that this new subtree contains two AI nodes with upper bounds σ_1 and σ_3. Note also that the subtree T_2 does not contain any AI-nodes, and that σ_2 is a lower-bound analysis.

We now detail this refinement. For simplicity, assume that the analysis of the original AI node marked σ_0 contains one potential witness path p. We then construct the subtree by first constructing the edges and nodes as a symbolic path p. As each edge and destination node is constructed from a branching source node, we also construct a new edge and destination node corresponding to the alternative of the branch. For this second destination node, which is not in the path p, we now construct a new AI node. At the end of this process, we would have constructed the subtree which has a

[1] Typically the lower bound is not used in abstract analysis.

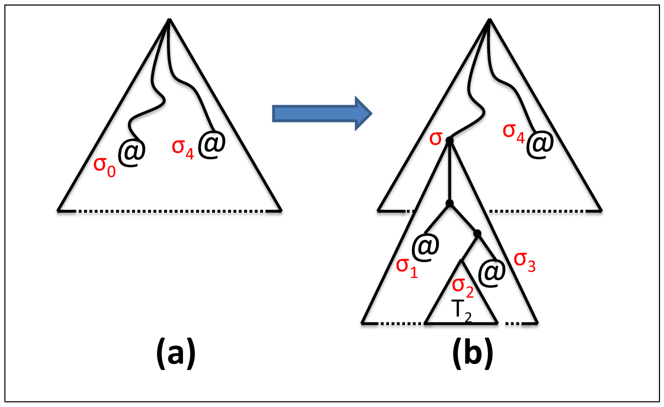

Figure 1: The Refinement Step

"spine" corresponding to the path p, and along the spine, we have constructed a number of AI nodes (two, in this example).

See Figure 1(b) and once again focus on the subtree labelled σ, and where the spine is the path to σ_2. There are two possible benefits of this refinement step. One is that this sub-HSET σ is more precise than the original analysis σ_0 because the join of σ_1, σ_2 and σ_3 is more precise than σ_0. Another benefit is when σ_2, which is a lower-bound analysis, can be used to *dominate* any other analysis. For example, if the lower bound of σ_2 not smaller than the upper bound of σ_4, then the entire subtree at σ_4 can be pruned from further consideration.

We remark that in the refinement step, each of the newly generated AI nodes require an (abstract) analysis, and although these analyses are efficient, there is the issue that the number of analyses could be as long as the path p. However, an important feature is that in the several invocations of abstract analysis performed here over the several AI nodes, the analysis of each of these is often produces the the same results, and hence can be cached and need not be redone. We will argue and demonstrate this important feature in detail later.

We now outline three key reasons for the scalability of our method.

SCENARIO 1 (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 has been included in the AI node N, now has been removed.

SCENARIO 2 (REUSE):
Here we use a computed exact analysis of one subtree to derive an exact analysis of another subtree. Suppose we have an exact analysis σ for a subtree T rooted at N. In this scenario, we exploit σ to compute another exact analysis σ' for another (yet unexplored) node associated with the same program point as N. In general, the *witness condition* for such a reuse (and here we are talking about real and not potential witnesses), as well as the precise definition of the mapping from σ to σ', is quite involved because it depends on the kind of analysis in question. But in specific instances, this is 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 3 (DOMINATION):
Here we exploit the situation when we have computed a nontrivial lower-bound analysis, say for node N. Now we can in fact prune

all subtrees in the *entire* HSET which are dominated by N. 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* 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 one of the two parties.

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

Finally, we mention that we do not explicitly consider unbounded loops in this paper. We assume that the symbolic execution of loops always terminates. Dealing with unbounded loops then is relegated to standard approaches such as using loop invariants.

The rest of this paper is organized as follows. 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 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 do indeed produce precision gains progressively. 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 *tick* $+= \kappa$ where κ 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 *tick*. The example analysis is an abstraction of the well-known worstscase execution time (WCET) analysis, and in our simple setting, the analysis formulas are simply upper bounds on *tick*, and the final analysis it is to determine the upper bound of $tick$ at the end.

Consider the program and its SET in Figure 2. Assuming that any combination of the unspecified guards is satisfiable, that is, that $B_1 \wedge B_2 \wedge B_3$ 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, and remaining within our trivial programming language, the problem to find the WCET is NP-complete [13].

Before we start our analysis we will first demonstrate a use of reuse. Assume that $\neg b_1 \wedge b_2 \wedge b_3$ is satisfiable, and that we already have an exact analysis, $tick = 3$ of the right subtree marked ⟨2'⟩ in Figure 2. 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 length p_2 is computed from the length of p_1, which is 3. However, since the prefix of p_1 from the root to node ⟨2'⟩, which increments *tick* by zero, differs from the prefix of p_2 from the root to node ⟨2⟩, which increments *tick* by 3, we must adjust for this and now declare that the exact analysis of node ⟨2⟩ is $tick = 6$. In other words, we assumed that the longest increment of *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 of 6.

```
⟨0⟩ tick = 0
⟨1⟩ if (b1) tick += 3
⟨2⟩ if (b2) tick += 2
⟨3⟩ if (b3) tick += 1
```

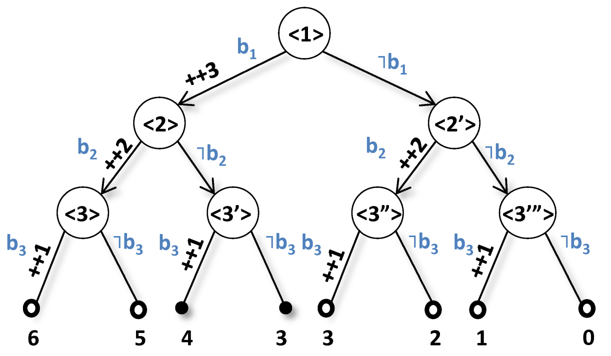

Figure 2: Example Program and its Symbolic Execution tree

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

- Suppose $b_1 \wedge b_2 \wedge b_3$ (corresponding to the leftmost path in the left subtree) is unsatisfiable. We can still perform reuse, i.e. 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 $\neg b_1 \wedge b_2 \wedge b_3$ (corresponding to the leftmost path in the right subtree) is unsatisfiable but $b_1 \wedge b_2 \wedge b_3$ (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 take place i.e.. that the path under the subtree to which reuse is being considered is feasible. 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.

We now proceed to analyze the nodes in the tree, i.e.. to provide a lower and upper bound for *tick*in the set of paths indicated by each node. For example, an upper-bound analysis for the left subtree in Figure 2, labelled ⟨2⟩, is *tick* $\leq$ 6. This subtree also can have a *lower-bound analysis* of any nonnegative number less than or equal to 6; for example, if we knew that the path proceeding to the left successor of ⟨3'⟩ were feasible, then we could record down that $4 \leq$ *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 right branches). Note that there may not actually be a real execution path resulting in *tick* $= 3$. Note also that spurious lower bounds, if their values are too low, are not very useful.

See Figure 3 where 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.

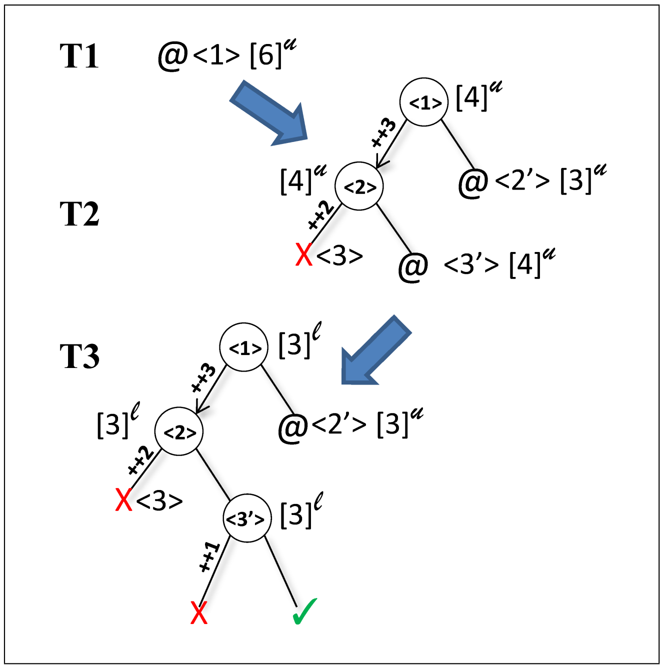

Figure 3: Detailed Refinement Step

Furthermore, the analyzer indicates that the leftmost path is a *potential witness* path, i.e.. if it were feasible, then it would indicate the true WCET. In Figure 3, we show only upper bounds, except in one instance in the final tree where we show one lower bound at the node ⟨3'⟩.

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 ⟨2'⟩ 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 $b_1 \wedge b_2$ is unsatisfiable, and so the leftmost path in Figure 2 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 can be dominated. We now obtain T_3 by refining this AI node.

This refinement produced two successors, and by assuming that the constraint $b_1 \wedge \neg b_2 \wedge b_3$ 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, that ⟨3'⟩ has a *lower* bound (and in fact also upper bound) of 3. The most interesting step now can be taken: that analysis here *dominates* the analysis at the one remaining AI node at ⟨2'⟩. Note that the set of paths represented by this node 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 *Vars*. An *assignment* $\mathsf{x} := \mathsf{e}$ corresponds to assign the evaluation of the expression e to the variable x. In the *assume* operator, $\mathsf{assume(c)}$, if the boolean expression c evaluates to *true*, then the program continues, otherwise it halts. The set of operations is denoted by *Ops*. We then model a program by a *transition system*. A transition system is a quadruple $\langle \mathcal{S}, I, \longrightarrow, O\rangle$ where $\mathcal{S}$ is the set of states and $I \subseteq \mathcal{S}$ is the set of initial states. $\longrightarrow \subseteq \mathcal{S} \times \mathcal{S} \times \mathit{Ops}$ 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 location to another. We shall use $\ell \xrightarrow{\mathsf{op}} \ell'$ to denote a transition relation from $\ell \in \mathcal{S}$ to $\ell' \in \mathcal{S}$ executing the operation $\mathsf{op} \in \mathit{Ops}$. Finally, $O \subseteq \mathcal{S}$ is the set of final states.

Abstract Interpretation and Symbolic Execution. An abstract domain $\mathcal{A}$ is defined as a lattice structure $[\sqsubseteq, \perp, \sqcup, \sqcap, \top]$, where $\sqsubseteq$ is the partial order relationship, $\sqcup$ and $\sqcap$ are the least upper bound and greatest lower bound operators, and $\perp$ and $\top$ are the bottom and top elements of the lattice respectively. $\mathcal{A}$ may also consider widening ∇ and narrowing $\triangle$ operators. We assume $\mathcal{A}$ is Galois-connected [8] with the powerset lattice of state sets, and the use of α and γ for the abstraction and concretization maps for this Galois connection.

A *symbolic state* v is defined usually as a triple $\langle \ell, s, \Pi\rangle$. The symbol $\ell \in \mathcal{S}$ corresponds to the current program location. We use special symbols for initial location, $\ell_{\mathsf{start}} \in I$, and final location, $\ell_{\mathsf{end}} \in O$. 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), $[\![e \;\mathsf{op}\; e']\!]_s = [\![e]\!]_s \;\mathsf{op}\; [\![e']\!]_s$ (where e, e' are expressions and op is a relational or arithmetic operator). Π 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 *FO* and *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 $\langle \ell, s, \Pi, \pi\rangle$, where the additional parameter π is a sequence of program transitions that were taken during SE in order to reach v.

Given a transition system $\langle \mathcal{S}, I, \longrightarrow, O\rangle$ and a state $v \equiv \langle \ell, s, \Pi, \pi\rangle \in \mathit{SymStates}$, the symbolic execution of $\ell \xrightarrow{\mathsf{op}} \ell'$ returns another symbolic state v' defined as:

$$\mathsf{SYMSTEP}(v, \ell \xrightarrow{\mathsf{op}} \ell') \equiv v' \triangleq \begin{cases} \langle \ell', s, \Pi \wedge [\![c]\!]_s, \pi'\rangle & \text{if } \mathsf{op} \equiv \mathsf{assume(c)} \text{ and } \Pi \wedge [\![c]\!]_s \\ & \text{is satisfiable} \\ \langle \ell', s[x \mapsto [\![e]\!]_s], \Pi, \pi'\rangle & \text{if } \mathsf{op} \equiv \mathsf{x} := \mathsf{e} \end{cases} \quad (1)$$

where $\pi' \triangleq \pi \cdot \ell \xrightarrow{\mathsf{op}} \ell'$. Note that Eq. (1) queries a *theorem prover* for satisfiability checking on the path condition. 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 $v \equiv \langle \ell, s, \Pi, \pi\rangle$ we define $[\![v]\!] : \mathit{SymStates} \rightarrow \mathit{FO}$ as the formula $(\bigwedge_{v \,\in\, \mathit{Vars}} [\![v]\!]_s) \wedge \Pi$ where *Vars* is the set of program variables.

A *symbolic path* $v_0 \cdot v_1 \cdot \ldots \cdot v_n$ is a sequence of symbolic states such that $\forall i \bullet 1 \leq i \leq n$ the state v_i is a *successor* of

v_{i-1}. A symbolic state $v' \equiv \langle \ell', \cdot, \cdot, \cdot \rangle$ is a successor of another $v \equiv \langle \ell, \cdot, \cdot, \cdot \rangle$ if there exists a transition relation $\ell \xrightarrow{\mathsf{op}} \ell'$. A path $v_0 \cdot v_1 \cdot \ldots \cdot v_n$ is *feasible* if $v_n \equiv \langle \ell, s, \Pi, \cdot \rangle$ such that $[\![\Pi]\!]_s$ is satisfiable. If $\ell \in O$ and v_n is feasible then v_n is called *terminal* state, denoted $\mathsf{TERMINAL}(v_n)$. Otherwise, if $[\![\Pi]\!]_s$ is unsatisfiable the path is called *infeasible* and v_n is called an *infeasible* state, denoted $\mathsf{INFEASIBLE}(v_n)$. If there exists a feasible path $v_0 \cdot v_1 \cdot \ldots \cdot v_n$ then we say v_k $(0 \leq k \leq n)$ 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, we assume the existence a function $\theta_{\mathcal{A}}$ which extracts an analysis, i.e., an element of the abstract domain $\mathcal{A}$, given a path π.

An invocation of *abstract interpretation* (AI) at a symbolic state v constructs an AI graph rooted at v. It performs the standard fixpoint computation on $\mathcal{A}$ and returns an *upper bound* analysis $\mathcal{U}$ and a possible *lower bound* analysis $\mathcal{L}$ of the set of paths through v. Formally, if E is assumed to be the desired *exact* analysis of the set of paths, then $\mathcal{L} \sqsubseteq E \sqsubseteq \mathcal{U}$ (we do not explicitly compute E but infer it when the bounds coincide).

Of course, being abstract, it need not consider the feasibility of individual paths and can analyze an over-approximation of them. In addition, we also assume that it produces *potential witness paths*[2] denoted as ω – a set of paths from which the upper bound analysis $\mathcal{U}$ is derived. In general this can be all paths in the AI graph, but often it is just a subset, since not all paths in an AI graph contribute to its analysis. Computing these witness paths is straightforward but tedious, so we do not detail it here. Briefly when AI executes, it can "mark" certain edges in the graph that are sufficient to produce the analysis of the entire graph. The witness paths can then be obtained by traversing marked edges from the root to a terminal node. In summary, we assume the existence of a procedure ABSTRACTINTERPRETATION which when invoked with a symbolic state v returns a triple $\langle \mathcal{L}, \mathcal{U}, \omega \rangle$.

Interpolation. Given a pair of first order logic formulas A and B such that $A \wedge B$ is $false$, an *interpolant* [10] is another formula $\overline{\Psi}$ such that (a) $A \models \overline{\Psi}$, (b) $\overline{\Psi} \wedge B$ is *false*, and (c) $\overline{\Psi}$ 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 $A \wedge B$.

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, as follows.

Given a current symbolic state $v \equiv \langle \ell, s, \cdot, \pi \rangle$ and an already explored symbolic state at the same program point $v' \equiv \langle \ell, \cdot, \cdot, \pi' \rangle$ annotated with the interpolant $\overline{\Psi}$, we say v' *subsumes* v, denoted as $\mathsf{SUBSUMES}(v', v)$ if (a) $[\![v]\!]_S \models \overline{\Psi}$ and (b) $c(\pi) \sqsubseteq_c c(\pi')$, where $c(\pi)$ is some contextual information extracted from π that monotonically determines the final analysis at 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 these paths have already been explored with a more general (in terms of the lattice of the context, possibly different from $\mathcal{A}$) context $c(\pi')$. 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.

[2] We say "potential" as they may not be real feasible paths.

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 presented and 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.

Efficient interpolation algorithms exist for quantifier-free fragments of theories 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. 4, can be expressed as one that starts with an abstract interpretation (AI) graph representing an abstract analysis of the program, and gradually refines the graph using symbolic execution (SE) until the desired level of analysis precision is obtained. During SE, a forward traversal collects path constraints and checks for path feasibility, and a backtracking phase *annotates* each state v with the following information: $\langle \mathcal{L}, \mathcal{U}, \omega, \overline{\Psi} \rangle$, representing the lower bound and upper bound analyses for the set of paths through v, the potential 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 symbolic state v annotated with $\langle \mathcal{L}, \mathcal{U}, \omega, \overline{\Psi} \rangle$ is* dominated *by a symbolic state v' annotated with $\langle \mathcal{L}', \mathcal{U}', \omega', \overline{\Psi}' \rangle$ if $\mathcal{U} \sqsubseteq \mathcal{L}'$. We also say that v'* dominates *v, denoted as $\mathsf{DOMINATES}(v', v)$.*

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 state can dominate itself if its lower and upper bounds are the same (i.e., it has an *exact* analysis).

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_{start} as the program point, an empty store, the path condition *true*, and the empty sequence. In line 2 the initial AI graph is generated by calling ABSTRACTINTERPRETATION with the initial state. This would return a possible lower bound, an upper bound and the potential witness paths ω for the upper bound.

Lines 3-9 define the main refinement loop. First, the set of symbolic states that root *non-dominated* AI graphs is collected in R. Any state in R now represents a potential AI graph to refine. A heuristic is then applied to select one state from R to refine in this iteration. Several meaningful heuristics can be applied here depending on the analysis being performed. In the case of WCET, the natural choice is to pick the AI graph that produces the maximum upper bound. In other analyses, if possible, a "difference" metric can be defined to measure the amount of (non) domination, and the AI graph in R with maximum difference can be chosen (see Section 5 for an example for taint analysis).

Once the state v representing the AI graph to refine is chosen, its current annotation is then removed (line 7) and the procedure REFINEUNFOLD is called along with the witness paths for its upper bound analysis ω. 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 *BoundsHeuristic*, which is user-defined.

```
INCREMENTALANALYSIS ($P$)
1:  $v := \langle \ell_{\mathsf{start}}, \emptyset, \mathit{true}, \cdot \rangle$
2:  $\langle \mathcal{L}, \mathcal{U}, \omega \rangle$ := ABSTRACTINTERPRETATION ($v$)
3:  do
4:    $R := \{v \mid \nexists v'$ s.t. DOMINATES$(v', v)\}$
5:    $v$ := RefinementHeuristic ($R$)
6:    let $v$ be annotated with $\langle \mathcal{L}, \mathcal{U}, \omega, \overline{\Psi} \rangle$
7:    remove $v$ annotation and REFINEUNFOLD($v$, $\omega$)
8:    PROPAGATEBACK($v$)
9:  until BoundsHeuristic

PROPAGATEBACK ($v' \equiv \langle \ell, s, \Pi, \pi \rangle$)
1:  if $\ell \equiv \ell_{\mathsf{start}}$ then return
2:  let $v'$ be the successor of some $v$
3:  $\langle \mathcal{L}, \mathcal{U}, \omega, \overline{\Psi} \rangle := \langle \bot, \bot, \emptyset, \mathit{true} \rangle$
4:  foreach successor $v''$ of $v$ do
5:    let $v''$ be annotated with $\langle \mathcal{L}'', \mathcal{U}'', \omega'', \overline{\Psi}'' \rangle$
6:    $\langle \mathcal{L}, \mathcal{U}, \omega \rangle := \langle \mathcal{L}'' \sqcup \mathcal{L},\ \mathcal{U}'' \sqcup \mathcal{U},\ \omega'' \cup \omega \rangle$
7:    $\overline{\Psi} := \overline{\Psi} \wedge \widehat{wlp}\ (\overline{\Psi}'', \mathsf{op})$
8:  endfor
9:  replace $v$ annotation with $\langle \mathcal{L}, \mathcal{U}, \omega, \overline{\Psi} \rangle$
10: PROPAGATEBACK($v$)
```

```
REFINEUNFOLD ($v \equiv \langle \ell, s, \Pi, \pi \rangle, \omega_V$)
1:  if INFEASIBLE ($v$) then
2:    $\langle \mathcal{L}, \mathcal{U}, \omega, \overline{\Psi} \rangle := \langle \bot, \bot, \emptyset, \mathit{false} \rangle$
3:  else if TERMINAL ($v$) then
4:    $\langle \mathcal{L}, \mathcal{U}, \omega, \overline{\Psi} \rangle := \langle \theta_{\mathcal{A}}(\pi), \theta_{\mathcal{A}}(\pi), \emptyset, \mathit{true} \rangle$
5:    spine_done := true
6:  else if $\exists\, v' \equiv \langle \ell, s, \cdot, \pi' \rangle$ annotated with $\langle \mathcal{L}, \mathcal{U}, \omega, \overline{\Psi} \rangle$ s.t.
7:    SUBSUMES ($v'$, $v$) then spine_done := true
8:  else if spine_done then
9:    $\langle \mathcal{L}, \mathcal{U}, \omega \rangle$ := ABSTRACTINTERPRETATION($v$),  $\overline{\Psi}$ := true
10: else
11:   $\langle \mathcal{L}, \mathcal{U}, \omega, \overline{\Psi} \rangle := \langle \bot, \bot, \emptyset, \mathit{true} \rangle$
12:   foreach transition $\ell \xrightarrow{\mathsf{op}} \ell' \in P$ do
13:     $v'$ := SYMSTEP($v, \ell \xrightarrow{\mathsf{op}} \ell'$)
14:     if $\ell \xrightarrow{\mathsf{op}} \ell' \in \omega_V$ then spine_done := false endif
15:     REFINEUNFOLD ($v', \omega_V$)
16:     let $v'$ be annotated with $\langle \mathcal{L}', \mathcal{U}', \omega', \overline{\Psi}' \rangle$
17:     $\langle \mathcal{L}, \mathcal{U}, \omega \rangle := \langle \mathcal{L}' \sqcup \mathcal{L},\ \mathcal{U}' \sqcup \mathcal{U},\ \omega' \cup \omega \rangle$
18:     $\overline{\Psi} := \overline{\Psi} \wedge \widehat{wlp}\ (\overline{\Psi}', \mathsf{op})$
19:   endfor
20: endif
21: if $\mathcal{L} \equiv \mathcal{U}$ then annotate $v$ with $\langle \mathcal{L}, \mathcal{U}, \emptyset, \overline{\Psi} \rangle$
22: else annotate $v$ with $\langle \mathcal{L}, \mathcal{U}, \omega, \mathit{false} \rangle$
23: endif
```

Figure 4: Algorithm for incrementally precise analysis

A straightforward *BoundsHeuristic* check is $\forall v.\exists v'$ s.t. DOMINATES(v', v), i.e., there are no non-dominated symbolic states. This forces the algorithm to terminate only when the final analysis is *exact*. 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 $\forall v$ annotated with $\langle \cdot, \mathcal{U}, \cdot, \cdot \rangle.\exists v'$ annotated with $\langle \mathcal{L}', \cdot, \cdot, \cdot \rangle$ s.t. $(\mathcal{U} - \mathcal{L}')/\mathcal{U} \leq 0.05$. A taint analyzer may only care about flow of taint to a particular subset of variables S, so the heuristic can check if $\forall v$ annotated with $\langle \cdot, \mathcal{U}, \cdot, \cdot \rangle$, $S \nsubseteq \mathcal{U}$, or $\exists v$ annotated with $\langle \mathcal{L}, \cdot, \cdot, \cdot \rangle$ s.t. $S \subseteq \mathcal{L}$, 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 symbolic state v and the set of potential witness paths to refine ω_V. It is a recursive procedure that refines an AI graph by symbolically unfolding the paths in ω_V. There are four bases of this procedure:

- (Lines 1-2) If v is an infeasible state, then it sets the lower and upper bounds to $\bot$, the set of witness paths for the upper bound to $\emptyset$, and the interpolant $\overline{\Psi}$ to *false* to denote the infeasibility.
- (Lines 3-5) If v is a terminal state, then this signifies an *exact analysis*. Hence both the lower and upper bounds are set to $\theta_{\mathcal{A}}(\pi)$ – the analysis extracted from this single path. A minor optimization is that 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 6-7) If v is subsumed by another state v', it simply sets spine_done to *true*. Implicitly, the lower and upper bounds, the witness paths and interpolant for v are copied over from v'.
- (Lines 8-9) If spine_done is *true*, i.e., a spine has been explored already and we are exploring other branches from it, then it constructs an AI graph at v by calling ABSTRACTINTERPRETATION. This would return a lower bound, upper bound and the witness paths for the upper bound. The interpolant is then set to *true*, as there is no infeasibility to capture in the constructed AI graph.

If the four bases fail, REFINEUNFOLD symbolically executes the current state v (lines 10-20). It first initializes the lower and upper bounds, the witness paths, and interpolant to $\bot$, $\emptyset$ and *true* respectively, which will be modified after the recursive call. Then, for each transition from the current program point ℓ to ℓ', it does the following (lines 12-19). It constructs the next symbolic state v' applying SYMSTEP on v, and recursively calls itself. As a minor technicality, if the transition is part of a witness path that is being refined (line 14), it sets spine_done to *false* before making the call. If this is not done, the fourth base case would simply construct an AI graph along a witness path, defeating the purpose of refinement.

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 *joining* it with the existing information at v (line 17). 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 $\overline{\Psi}'$ at v', the interpolant at v is computed by conjoining the current interpolant $\overline{\Psi}$ with $\widehat{wlp}(\overline{\Psi}', \mathsf{op})$ — the *weakest liberal precondition* [11] of $\overline{\Psi}'$ w.r.t. the transition op. $\widehat{wlp} : FO \times Ops \rightarrow FO$ ideally returns the weakest formula on the current state such that the execution of op results in $\overline{\Psi}'$. In practice we approximate the *wlp* by making a linear number of calls to a theorem prover, using techniques outlined in [14], which usually results in a formula stronger than the *wlp*.

Finally, once either a base case or the recursive case is executed, REFINEUNFOLD annotates (lines 21-23) 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 *false* before annotating v so that for all states v'', $\mathsf{SUBSUMES}(v, v'')$ 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 most likely assigns them differently. The recursive case is then dependent on the the bounds of the successors of v.

The final procedure PROPAGATEBACK simply propagates the annotation at a given state v' to its ancestors upto the root of the entire tree at ℓ_{start}. In line 2, it obtains the parent state v, and in lines 3-8 it performs the backward propagation from all successors of v, in exactly the same way as lines 11,16-18 of REFINEUNFOLD. For brevity, we provide its pseudocode but omit a detailed description.

The whole algorithm is guaranteed to terminate provided 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 "any-time terminate" algorithm.

Discussion on Loops

Unbounded Loops pose a technical problem as they make the symbolic execution tree infinite, thereby making REFINEUNFOLD non-terminating. The only hope 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 external means such as AI with octagon or polyhedral domains [22], and a function getassrt which, given a program point ℓ and symbolic store s, returns an assertion in the form of a FOL formula, renamed using s, that holds at ℓ. Note that when ℓ is a loop header, getassrt will return a loop invariant. Then, we modify SYMSTEP by adding a new case as follows:

$\mathsf{SYMSTEP}(v, \ell \xrightarrow{\mathsf{op}} \ell') \triangleq \mathsf{invariant}(\langle \ell', s, \Pi \rangle, \ell \rightarrow \ell_n)$ if ℓ is the header for a loop from ℓ to ℓ_n

where $\mathsf{invariant}(\langle \ell, s, \Pi, \pi \rangle, \ell \rightarrow \ell_n) \triangleq$

$$\begin{cases} \textbf{let } s' := \textsc{Havoc}(s, \textsc{Modifies}(\ell \rightarrow \ell_n)) \\ \quad \overline{\Pi} := \mathsf{getassrt}(\ell, s') \wedge \Pi \\ \textbf{in } \langle \ell, s', \overline{\Pi}, \pi \rangle \end{cases}$$

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

$\textsc{Modifies}(\ell \rightarrow \ldots \rightarrow \ell_n)$ 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 HAVOC function) and then strengthens the path condition Π of the symbolic state with the invariants from the abstract interpreter. Finally, REFINEUNFOLD is modified so that if it detects a cycle at ℓ, it simply stops unfolding and returns, as ℓ has already been explored with the loop invariant. Note that abstraction using loop invariants entails a possible loss of analysis precision, but this is a general problem outside the scope of this paper.

5. Implementation

We implemented the incremental analysis algorithm in Fig. 4 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 abstract domain $\mathcal{A}$ is defined as $[\leq, 0, \sqcup, \sqcap, \infty]$, with the lattice being the non-negative integers, $\theta_1 \sqcup \theta_2 \triangleq max(\theta_1, \theta_2)$. The pre-operation, defined as $\widehat{pre}(\theta_{v'}, \mathsf{op}) \triangleq \theta_{v'} + \mathtt{len}(\mathsf{op})$, adds the length of the transition op to the post-state analysis to obtain the current-state analysis.

We implemented the heuristics in Fig. 4 as follows. *RefinementHeuristic* is quite straightforward as the abstract domain $\mathcal{A}$ imposes a total order on its elements. Hence we simply pick for refinement the AI graph that produced the maximum upper bound WCET, with ties being resolved non-deterministically. *BoundsHeuristic* implements the following check: $\forall v.\exists v' s.t.\mathsf{DOMINATES}(v', v)$, i.e., all symbolic states are dominated by another, possibly the same, state. This makes the algorithm terminate only when the final WCET is exact.

5.2 Taint Analysis

For forward taint analysis, $\mathcal{A}$ is defined as $[\subseteq, \emptyset, \cup, \cap, V]$, with the lattice being the powerset of V, the set of all program variables. In the literature, there are various definitions of how taint information is to be propagated, and we follow the one in [21], 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 `if(t==x)`, which intuitively makes the other argument x also tainted, as one can observe its value to find the value of t. Therefore the post-operation is defined as follows:

$$\widehat{post}(\theta_v, \mathsf{op}) \triangleq \begin{cases} \theta_v \cup \{x\} & \text{if } \mathsf{op} \equiv \mathsf{x} := \mathsf{e} \text{ and } \exists \mathsf{t} \in \theta_v \\ & \text{that occurs in e} \\ \theta_v \cup \{x\} & \text{if } \mathsf{op} \equiv \mathsf{assume}(\mathsf{x}{=}{=}\mathsf{e}) \text{ and (2)} \\ & \exists \mathsf{t} \in \theta_v \text{ that occurs in e} \\ \theta_v & \text{otherwise} \end{cases}$$

Typically, the source of taint greatly affects the propagated taint information. We chose a realistic and objective model for our experiments, where all variables whose values are obtained from outside the program were tainted. This includes all `extern` variables, user input, environment variables etc. This models the execution of the program in an "untrusted" environment, a common scenario in realistic taint analyses.

We now discuss how the heuristics were implemented here. For *RefinementHeuristic*, unlike WCET, the abstract domain $\mathcal{A}$ 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 ($\sqcup$) of all lower bound taint sets in, say, a set L_B. Then, for each v that roots an AI graph annotated with $\langle \mathcal{L}, \mathcal{U}, \omega, \overline{\Psi} \rangle$, we compute the difference set $\delta_v \equiv \mathcal{U} \setminus L_B$. We then pick the AI graph at v that produces the maximal value of $|\delta_v|$. That is, we pick the AI graph 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 graphs, 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 maximal difference $\{c, d\}$ as opposed to just $\{c\}$ from v. Finally, *BoundsHeuristic* implements the same check as for WCET analysis, which ensures that the taint information is exact when the algorithm terminates.

Benchmark	LOC	AI-based	SE [5]			Incremental			
		WCET	WCET	Time	Mem	WCET$_{\mathcal{U}}$	WCET$_{\mathcal{L}}$	Time	Mem
cdaudio	1288	167	154	27 s	212 MB	154	154	12 s	47 MB
diskperf	1255	446	*	*	*	411	411	230 s	400 MB
floppy	1524	243	216	19 s	136 MB	216	216	14 s	38 MB
ssh	2213	107	59	17 s	39 MB	59	59	17 s	51 MB
nsichneu	3370	1551	*	*	*	1431	483	*	*
tcas	235	249	*	*	*	244	207	*	*
statemate	1187	433	*	*	*	431	319	*	*

Table 1. WCET Analysis results for AI based, SE based, and our incremental algorithm. A * represents a timeout of 5 minutes.

Benchmark	# V	AI-based	SE [15]			Incremental			
		# TV	# TV	Time	Mem	#TV$_{\mathcal{U}}$	#TV$_{\mathcal{L}}$	Time	Mem
cdaudio	330	318	318	45 s	528 MB	318	318	9 s	125 MB
diskperf	185	166	166	45 s	359 MB	166	166	1 s	17 MB
floppy	197	179	179	9 s	140 MB	179	179	1 s	22 MB
ssh	63	57	53	1 s	8 MB	53	53	1 s	8 MB
nsichneu	22	16	16	4 s	24 MB	16	16	2 s	11 MB
tcas	41	29	29	1 s	8 MB	29	29	1 s	2 MB
statemate	119	96	#	#	#	96	96	2 s	39 MB

Table 2. Taint Analysis results. # TV measures the number of tainted variables. A # is an out of memory within the timeout.

6. Experimental Evaluation

We used as benchmarks sequential C programs from a varied pool – three device drivers cdaudio, diskperf, floppy from the ntdrivers-simplified category and SSH Client protocol from the ssh-simplified category of SV-COMP 2014 [2], an air traffic collision avoidance system tcas, and two programs from the Mälardalen WCET benchmark [18] statemate 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, and a timeout of 5 minutes considering our nominal benchmark size.

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 that of the other two algorithms (for instance, it typically terminates in less than 1 second).

6.1 WCET Analysis

Table 1 shows the results of running WCET analysis on our benchmarks. As it can be seen, the AI based algorithm produces an analysis quickly for all programs but it is in fact not precise. So the only hope to produce an exact analysis is if the SE based algorithm terminates. However the latter fails to terminate on several programs by timing out, 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 the first four programs, thus producing the same exact analysis as SE but 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 three programs. For instance, in nsichneu, a well-known program in the WCET community that is notoriously hard to analyze, AI produced the imprecise WCET 1551 whereas SE and our incremental algorithm ran out of budget. However we were able to reduce the WCET (i.e., our upper bound) to 1431 before dying. Moreover, we provided a lower bound of 483 to *quantify* the precision of our analysis. That is, we are able to provide a *range* for the WCET to exist, whereas AI is only able to provide an imprecise upper bound for it.

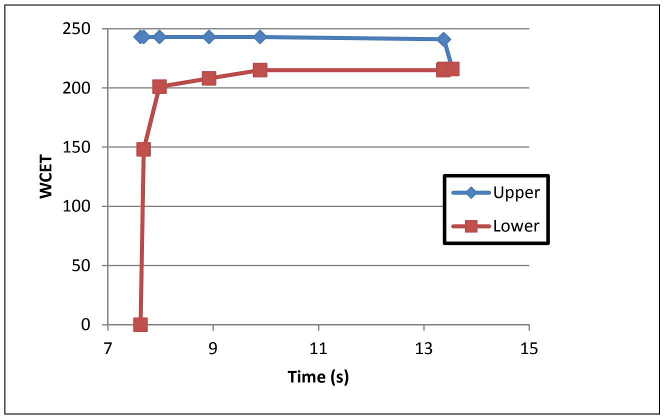

Figure 5: Progressive Upper and Lower bounds over time for floppy

To observe how the upper and lower bounds incrementally converge in our algorithm, we take a closer look the floppy benchmark that best exposes this phenomenon. Fig. 5 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 8 seconds, and when they coincide we get the exact analysis at around 14 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.

Unlike WCET analysis, we see that the analysis produced by AI in fact turns out to be exact in most cases, except ssh. This might be because our formulation of taint analysis propagates "similar" taint information across all paths, and so AI does not lose much precision due to its lack of detection of infeasible paths or merging. However, the important point is that AI cannot *systematically* guarantee or quantify the precision of its analysis.

On the other hand, the SE based algorithm is indeed able to terminate on most benchmarks and provide an exact analysis. But as it can be seen it consumes a much larger budget (time and memory) than our incremental algorithm to do so. Moreover, in statemate, it is unable to terminate within the memory bounds and produce any exact analysis.

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, 24] for survey), power consumption [23], 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 *potential 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 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) [12] 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" [12].
- 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.

8. Concluding Remarks

We presented an analysis framework whose algorithm starts with an efficienly obtained but not necessarily precise analysis, and then iterately 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 both lower and upper bounds and so the final reported analysis comes with some certification of precision. A third feature is that the algorithm is equipped with, due to having both lower and upper bounds, a concept of domination whcih 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 utilises less memory and time. For examples that 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 certification. In short, unless an uncertified abstract analysis is deemed acceptable, our algorithm has been evaluated to be always superior than the state-of-the-art.

References

[1] A. Banerjee, S. Chattopadhyay, and A. Roychoudhury. Static analysis driven cache performance testing. In *RTSS*, pages 319–329, 2013.

[2] D. Beyer. Third competition on software verification. In *TACAS*, 2014.

[3] M. Boddy. Anytime problem solving using dynamic programming. In *AAAI*, pages 738–743, 1991.

[4] P. Cerny, T. A. Henzinger, and A. Radhakrishna. Quantitative abstraction refinement. In *POPL*, pages 115–128, 2013.

[5] D. H. Chu and J. Jaffar. Symbolic simulation on complicated loops for wcet path analysis. In *EMSOFT*, 2011.

[6] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo theories. In *TACAS'08*, pages 397–412, 2008.

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. CounterrExample-Guided Abstraction Refinement. In *CAV*, 2000.

[8] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis. In *POPL*, 1977.

[9] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th POPL*, pages 269–282. ACM Press, 1979.

[10] W. Craig. Three uses of Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Computation*, 22, 1955.

[11] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 1975.

[12] V. D'Silva, L. Haller, and D. Kroening. Abstract conflict driven learning. In *POPL*, pages 143–154, 2013.

[13] J. Jaffar, A. E. Santosa, and R. Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In *AAAI*, 2008.

[14] J. Jaffar, A. E. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *15th CP, LNCS 5732*, 2009.

[15] J. Jaffar, V. Murali, J. Navas, and A. Santosa. Path sensitive backward analysis. In *SAS*, 2012.

[16] J. Jaffar, V. Murali, J. Navas, and A. Santosa. TRACER: A symbolic execution engine for verification. In *24th CAV*, 2012.

[17] J. Jaffar, V. Murali, and J. Navas. Boosting Concolic Testing via Interpolation. In *FSE*, 2013.

[18] Mälardalen. Mälardalen WCET research group benchmarks. URL `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, 2006.

[19] K. L. McMillan. Lazy annotation for program testing and verification. In *CAV*, 2010.

[20] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 2000.

[21] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *S&P*, pages 317–331, 2010.

[22] S. Seo, H. Yang, and K. Yi. Automatic construction of hoare proofs from abstract interpretation results. In *APLAS*, pages 230–245, 2003.

[23] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. In *ICCAD*, pages 384–390, 1994.

[24] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 2008.