Lazy Symbolic Execution for Enhanced Learning

Duc-Hiep Chu, Joxan Jaffar, and Vijayaraghavan Murali

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

Abstract. The performance of symbolic execution based verifiers relies heavily on the quality of "interpolants", formulas which succinctly describe a generalization of states proven safe so far. By default, symbolic execution along a path stops the moment when infeasibility is detected in its path constraints, a property we call "eagerness". In this paper, we argue that eagerness may hinder the discovery of good quality interpolants, and propose a systematic method that ignores the infeasibility in pursuit of better interpolants. We demonstrate with a state-of-the-art system on realistic benchmarks that this "lazy" symbolic execution outperforms its eager counterpart by a factor of two or more.

1 Introduction

Symbolic execution has been largely successful in program verification, testing and analysis [16, 24, 28, 14, 13]. It is a method for program reasoning that uses symbolic values as inputs instead of actual data, and it represents the values of program variables as symbolic expressions on the input symbolic values. As symbolic execution reaches each program point along different paths, different "symbolic states" are created. For each symbolic state, a path condition is maintained, which is a formula over the symbolic inputs built by accumulating constraints that those inputs must satisfy in order for execution to reach the state. A symbolic execution tree depicts all executed paths during symbolic execution.

We say that a state is *infeasible* if its path condition is unsatisfiable, therefore one obviously cannot reach an **error** location from this state. Whenever an infeasible state is encountered, symbolic execution will backtrack along the edge(s) just executed. In that regard, symbolic execution by default is *eager*. This eagerness has been considered as a clear advantage of symbolic execution, in comparison with Abstract Interpretation (AI) [7] or Counterexample-Guided Abstraction Refinement (CEGAR) [6], since it avoids the exploration of *infeasible paths* which could block exponentially large symbolic trees in practice.

This paper considers symbolic execution in the context of software verification. One main challenge is to address the path explosion problem. The approaches of [16, 24, 15, 14] tackle this fundamental issue by eliminating from the model those facts which are irrelevant or too-specific for proving the unreachability of the error nodes. This "learning" phase consists of computing *interpolants* in the same spirit of conflict-driven learning in SAT solvers. Informally, the interpolant at a given program point can be seen as a formula that succinctly captures the reason of infeasibility of paths which go through that program

point. In other words it succinctly captures the reason why paths through the program point are safe. As a result, if the program point is encountered again through a different path such that its path condition implies the interpolant, the new path can be *subsumed*, because it can be guaranteed to be safe. Interpolation has been empirically shown to be crucial in scaling symbolic execution because it can potentially result in exponential savings by pruning large sub-trees. It is also generally known that the quality of interpolants greatly affects the amount of savings provided.

This is where a conflict between eagerness and learning arises. Eagerly stopping and backtracking at an infeasible state can make the learned interpolants unnecessarily too *restrictive* – while the interpolant would typically capture the reason for infeasibility of the state, the infeasibility could have nothing to do with the safety of the program. In practice, safety properties often involve a small number of variables whereas conditional expressions, which act as guards by causing infeasibility in paths, could be on any unrelated variable. Ultimately, this causes the (restrictive) interpolant to disallow subsumption in future, mitigating its benefit. In other words, eagerness hinders a *property-directed* approach.

In this paper, we propose a new method to enhance the learning of powerful interpolants but without losing the intrinsic benefits of symbolic execution. Whenever an infeasible path is encountered, instead of backtracking immediately, we *selectively abstract* the infeasible state so that it becomes feasible, and proceed with the search. By performing such an abstraction, we say that we have entered *speculation mode*. More generally, as we progressively abstract away infeasibility from a symbolic path, we are exhibiting a property-directed strategy, i.e., ignoring the infeasibility along the path until the real reason why the path is safe is found. Note that the sole purpose of speculation is to find better interpolants – we already know any path with an infeasible prefix is safe.

However, since exploration of infeasible states is in general a wasteful effort, we subject the speculation to a *bound*. This mitigates the potential blowup of the speculative search, while still retaining the possibility of discovering good interpolants. Intuitively, this bound should be at least linearly related to the program size: anything less than this could make the speculation phase arbitrarily short. It is a main contribution of this paper, that in the other direction, a linear bound is *good enough*.

Finally, we remark that though this paper studies and quantifies the "enhanced learning" for symbolic execution in the setting of static analysis, its impact is relevant to runtime verification as well. In our previous work [13], we have demonstrated the benefits of interpolation in speeding up *concolic* testing for better coverage. By being lazy, we expect the enhanced interpolants to result in further speedup. As another example, Navabpour et al. [25] propose a method, which leverages symbolic execution to predict the program's execution path, in order to effectively reduce the overhead of time-triggered runtime verification (TTRV). In fact, improvements in symbolic execution, as demonstrated in this paper, can lead to improvements in their runtime verification.

2 Examples

We first exemplify the case when (eager) symbolic execution is clearly not an efficient way to conduct a proof. For the programs in Fig. 1, assume (1) the boolean expressions e_i do not involve the variables x and y, and (2) the desired postcondition is $y \leq n$ for some constant $n > 0$. A *path expression* is of the form $E_1 \wedge E_2 \wedge \cdots \wedge E_n$ where each E_i is either e_i or its negation. Note that each of the (2^n) path expressions represents a unique path through each of the programs.

(a) Lazy is Good

```
x = y = 0
if (e1) y++ else x++
if (e2) y++ else x++
...
if (en) y++ else x++
```

(b) Eager is Good

```
x = y = 0
if (e1) y += 2
if (e2) y += 2
...
if (en) y += 2
```

(c) Lazy is Still Better

```
x = y = 0
if (e1) y++ else x++
...
if (ej) y++ else y = n+1
...
if (en) y++ else x++
```

Fig. 1: Proving $y \leq n$: Eager vs Lazy

Given the first program in Fig. 1(a), we can reason that the postcondition $y \leq n$ always holds, *without considering* the satisfiability of the path expressions. Using symbolic execution, in contrast, many of the unsatisfiable path expressions need to be detected and worse, their individual reasons for unsatisfiability (the "interpolants") need to be recorded and managed. Note that if we used a CEGAR approach [6] here, where *abstraction refinements* are performed only when a spurious counter-example is encountered, we would have a very efficient (linear) proof.

In the next program in Fig. 1(b), slightly modified from the previous, we present a dual and opposite situation. Note that the program is safe just if, amongst the path expressions that are satisfiable, less than $n/2$ of these involve a distinct and positive expression e_i (as opposed to the negation of e_i), for i ranging from 1 to n. This means that the number of times the "then" bodies of the if-statements are (symbolically) executed is less than $n/2$. Here, it is in fact necessary to record and manage the unsatisfiable path expressions as they are encountered during symbolic execution. CEGAR, in contrast, would require a large number of abstraction refinements in order to remove counter-examples arising from not recognizing the unsatisfiability of "unsafe" path expressions, i.e. those corresponding to $n/2$ or more increments of y.

In principle, a typical program would correspond to being in between the above two extreme cases in Fig. 1(a) and 1(b). Our key intuition, however, is that in fact a typical program lies closer to the first example rather than the second, because in practice safety properties are typically on a small subset of variables, whereas program guards, which are the cause of infeasibilities, can be on any (unrelated) variables. This intuition is later confirmed empirically.

For the final example program in Fig. 1(c), assume that all and only the path expressions which contain the subexpression e_j are unsatisfiable. (In other words, the only way to execute the jth if-statement is through its "then" body.) Here we clearly need to detect the presence of the expression e_j and not any of

the other expressions. More generally, we argue that while some path expressions must be recorded and managed, this number is small. The challenge is, of course, is how to *find* these important path expressions, which is precisely the objective of our speculation algorithm. We next exemplify this.

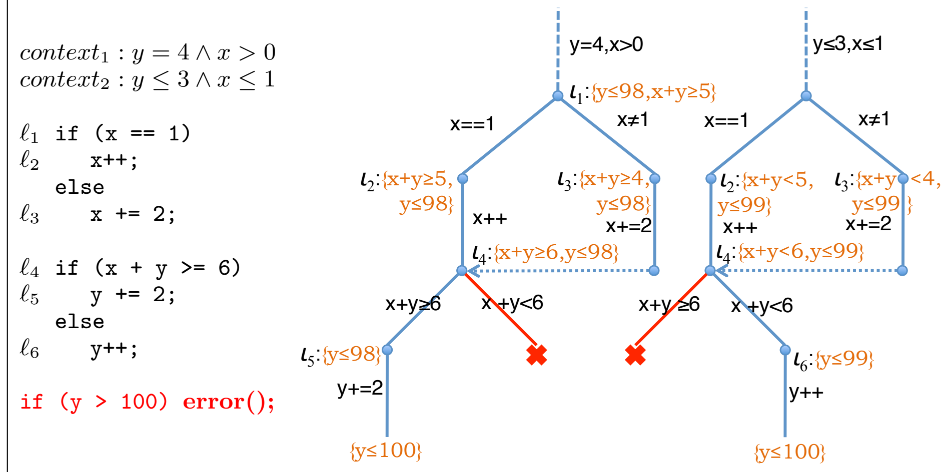

Fig. 2: A Symbolic Execution (Eager) Tree with Learning

Consider the program fragment in Fig. 2 executed under two different initial contexts: $y = 4 \wedge x > 0$ and $y \leq 3 \wedge x \leq 1$. In both contexts, the program is safe because $y \leq 100$ at the end. Throughout the example, assume weakest preconditions (WP) are used as interpolants.

Symbolic execution (eager) would start at program point ℓ_1 with the first context $y = 4 \wedge x > 0$. Assume it first takes the then branch with condition `x==1`, executing `x++` and reaching ℓ_4. Proceeding along the then branch from ℓ_4, it executes `y+=2` and reaches the end of the safe path, generating the (WP) interpolant $y \leq 98$ at ℓ_5. Now from ℓ_4, it finds that the else branch is infeasible as the path condition $y = 4 \wedge x > 0 \wedge x = 1 \wedge x' = x + 1 \wedge x' + y < 6$ is unsatisfiable. Being eager, symbolic execution would immediately backtrack, and to preserve this infeasibility, it would learn the interpolant $x' + y \geq 6$. Combining the then and else body's interpolants, it would generate $x' + y \geq 6 \wedge y \leq 98$ at ℓ_4 (note that in Fig. 2 we project the formula on the original variable names). Passing this back through WP propagation would result in $x + y \geq 5 \wedge y \leq 98$ at ℓ_2.

Now, executing the else body `x+=2` from ℓ_1, it would reach ℓ_4 with the path condition $y = 4 \wedge x > 0 \wedge x \neq 1 \wedge x' = x + 2$, which implies the interpolant $x' + y \geq 6 \wedge y \leq 98$. Therefore the path would be *subsumed* (dotted line). Propagating this interpolant through `x+=2` would result in $x + y \geq 4 \wedge y \leq 98$ at ℓ_3. Now, combining the then and else body's interpolant at ℓ_1 would result in the disjunction: $(x = 1 \Rightarrow (x+y \geq 5 \wedge y \leq 98)) \wedge (x \neq 1 \Rightarrow (x+y \geq 4 \wedge y \leq 98))$. For the sake of clarity, we strengthen this to $y \leq 98 \wedge x + y \geq 5$, but we assure the reader that our discussion is not affected by this. Thus, the final symbolic execution tree explored for this context will be the one on the left in Fig. 2.

Now, when the program fragment is reached along the second context $y \leq 3 \wedge x \leq 1$, subsumption cannot take place at ℓ_1 as the context does not imply the interpolant $y \leq 98 \wedge x + y \geq 5$. Symbolic execution would therefore proceed to generate the symbolic tree shown on the right. It is worth noting that even if the program was explored with the order of the contexts swapped, subsumption cannot take place at the top level.

Consider now our lazy symbolic execution process invoked on this program. We would perform symbolic execution exactly the same as before, except when the unsatisfiable path condition $y = 4 \wedge x > 0 \wedge x = 1 \wedge x' = x + 1 \wedge x' + y < 6$ is encountered, instead of backtracking, we selectively abstract the formula to make it satisfiable. Since we are doing forward symbolic execution, we selectively abstract by *deleting* the constraint(s) from the latest guard that we encountered (i.e., $x' + y < 6$) to make the formula satisfiable.[1]

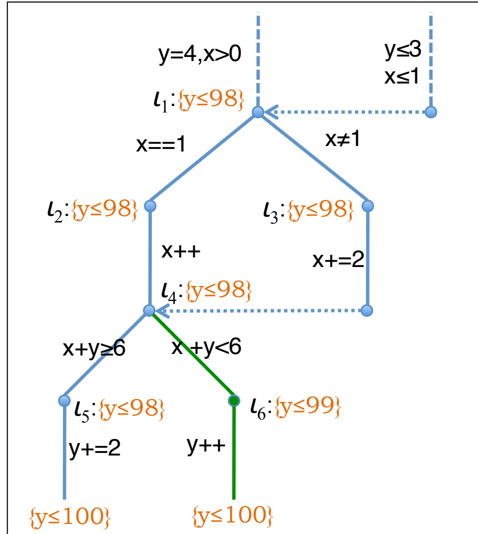

Fig. 3: Lazy Symbolic Execution Tree

After performing selective abstraction, we enter "speculation mode" with the abstracted path condition $y = 4 \wedge x > 0 \wedge x = 1 \wedge x' = x + 1$. A problem now is that in general, the sub-tree underneath the infeasible branch may be exponentially large, exploring which is wasteful as we already know that it is safe. Therefore it is necessary to impose a bound on the speculative search. We remark on our design choice of such a bound in later technical Sections.

Triggering speculation at ℓ_4, we execute the statement `y++` at ℓ_6 and reach the end of the (safe) path. Speculation has now succeeded, hence we annotate ℓ_6 with $y \leq 99$. Combining the interpolants at ℓ_4, we get $y \leq 98$. Propagating it back through the tree as shown in Fig. 3 we get the interpolant $y \leq 98$ at ℓ_1. Now, when the program fragment is reached along the second context $y \leq 3 \wedge x \leq 1$, the interpolant is implied at ℓ_1, and the entire tree can be subsumed at the top level. Note that we applied strengthening of WP as before, but we assure that even without strengthening the subsumption will still take place.

This example has shown that speculation can potentially result in exponential savings. The reason speculation works in practice is that safety properties are only on a small subset of variables whereas program guards that cause infeasibility can be on any of them. Temporarily ignoring the infeasibility helps

[1] In principle, selective abstraction can be done in many ways, for instance, by also deleting $y = 4$, $x' = x + 1$ or any combination. We defer to Section 5 the reasoning behind our design choice of deleting the latest guard.

in discovering interpolants closely related to the safety, such as those in Fig. 3, rather than interpolants that blindly preserve the infeasibility, such as those in Fig. 2. In Section 5, we empirically show that the exponential gains provided by speculation clearly outweigh its cost.

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 $[\Sigma, I, \longrightarrow, O]$ where Σ is the set of program locations and $I \subseteq \Sigma$ is the set of initial locations. $\longrightarrow \subseteq \Sigma \times \Sigma \times Ops$ 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 $\ell \in \Sigma$ to $\ell' \in \Sigma$ executing the operation $\mathsf{op} \in Ops$. Finally, $O \subseteq \Sigma$ is the set of final locations.

Symbolic Execution. A *symbolic state* s is a triple $\langle \ell, \sigma, \Pi \rangle$. The symbol $\ell \in \Sigma$ corresponds to the current program location. We will use special symbols for initial location, $\ell_{\mathsf{start}} \in I$, final location, $\ell_{\mathsf{end}} \in O$, and error location $\ell_{\mathsf{error}} \in O$ (if any). W.l.o.g we assume that there is only one initial, final, and error location in the transition system.

The symbolic store σ is a function from program variables to terms over input symbolic variables. Each program variable is initialised to a fresh input symbolic variable. The *evaluation* $[\![c]\!]_\sigma$ of a constraint expression c in a store σ is defined recursively as usual: $[\![v]\!]_\sigma = \sigma(v)$ (if $c \equiv v$ is a variable), $[\![n]\!]_\sigma = n$ (if $c \equiv n$ is an integer), $[\![e \ \mathsf{op}_r \ e']\!]_\sigma = [\![e]\!]_\sigma \ \mathsf{op}_r \ [\![e']\!]_\sigma$ (if $c \equiv e \ \mathsf{op}_r \ e'$ where e, e' are expressions and op_r is a relational operator $<, >, ==, !=, >=, <=$), and $[\![e \ \mathsf{op}_a \ e']\!]_\sigma = [\![e]\!]_\sigma \ \mathsf{op}_a \ [\![e']\!]_\sigma$ (if $c \equiv e \ \mathsf{op}_a \ e'$ where e, e' are expressions and op_a is an arithmetic operator $+, -, \times, \ldots$).

Finally, Π is called *path condition*, a first-order formula over the symbolic inputs that 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 *FOL* and *SymStates*, respectively. Given a transition system $[\Sigma, I, \longrightarrow, O]$ and a state $s \equiv \langle \ell, \sigma, \Pi \rangle \in SymStates$, a 'symbolic step' of transition $t : \ell \xrightarrow{\mathsf{op}} \ell'$ returns another symbolic state s' defined as:

$$s' \equiv \mathsf{SYMSTEP}(s, t) \triangleq \begin{cases} \langle \ell', \sigma, \Pi \wedge [\![c]\!]_\sigma \rangle & \text{if } \mathsf{op} \equiv \mathsf{assume(c)} \\ \langle \ell', \sigma[x \mapsto [\![e]\!]_\sigma], \Pi \rangle & \text{if } \mathsf{op} \equiv \mathsf{x} := \mathsf{e} \end{cases} \quad (1)$$

Given a symbolic state $s \equiv \langle \ell, \sigma, \Pi \rangle$ we define $[\![s]\!] : SymStates \to FOL$ as the formula $(\bigwedge_{v \,\in\, Vars} [\![v]\!]_\sigma) \wedge \Pi$ where *Vars* is the set of program variables.

A *symbolic path* $\pi \equiv s_0 \cdot s_1 \cdot ... \cdot s_n$ is a sequence of symbolic states such that $\forall i \cdot 1 \leq i \leq n$ the state s_i is a *successor* of s_{i-1}, denoted as $\mathsf{SUCC}(s_{i-1},\, s_i)$. A path $\pi \equiv s_0 \cdot s_1 \cdot ... \cdot s_n$ is *feasible* if $s_n \equiv \langle \ell, \sigma, \Pi \rangle$ such that $[\![\Pi]\!]_\sigma$ is satisfiable. If $\ell \in O$ and s_n is feasible then s_n is called *terminal* state. If $[\![\Pi]\!]_\sigma$ is unsatisfiable the path is called *infeasible* and s_n is called an *infeasible* state. If there exists a feasible path $\pi \equiv s_0 \cdot s_1 \cdot ... \cdot s_n$ then we say s_k $(0 \leq k \leq n)$ is *reachable* from s_0. A *symbolic execution tree* contains all the execution paths explored during the symbolic execution of a transition system by triggering Equation (1). The nodes represent symbolic states and the arcs represent transitions between states. Verification is done by exploring the symbolic execution tree and ensuring that the error location ℓ_{error} is not reachable. Finally, we define a "selective abstraction" operator $\overline{\nabla} : FOL \times FOL$ that accepts an *unsatisfiable* formula Π and returns a satisfiable formula that is an abstraction of Π.

Interpolation. The main challenge for symbolic execution is the path explosion problem. This issue has been addressed using the concept of interpolation.

Definition 1 (Craig Interpolant). *Given two formulas A and B such that $A \wedge B$ is unsatisfiable, a Craig* interpolant *[8], $\mathit{INTP}(A, B)$, is another formula $\overline{\Psi}$ such that (a) $A \models \overline{\Psi}$, (b) $\Psi \wedge B$ is unsatisfiable, and (c) all variables in $\overline{\Psi}$ are common to A and B.*

An interpolant allows us to remove irrelevant information in A that is not needed to maintain the unsatisfiability of $A \wedge B$. That is, the interpolant captures the essence of the reason of unsatisfiability of the two formulas. 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 [5] for details) where interpolants can be extracted from the refutation proof in linear time on the size of the proof.

Definition 2 (Subsumption check). *Given a current symbolic state $s \equiv \langle \ell, \sigma, \cdot \rangle$ and an already explored symbolic state $s' \equiv \langle \ell, \cdot, \cdot \rangle$ annotated with the interpolant $\overline{\Psi}$, we say s is* subsumed *by s', $\mathit{SUBSUME}(s, \langle s', \overline{\Psi} \rangle)$, if $[\![s]\!]_\sigma \models \overline{\Psi}$.*

To understand the intuition behind the subsumption check, it helps to know what an interpolant at a node actually represents. An interpolant $\overline{\Psi}$ at a node s' succinctly captures the reason of infeasibility of all infeasible paths in the symbolic tree rooted at s'. Let us call this tree T_1. Then, if another state s at ℓ implies $\overline{\Psi}$, it means the tree rooted at s, say T_2, has exactly the same or more (in a superset sense), infeasible paths compared to T_1. In other words, T_2 has exactly the same, or *less feasible paths* (in a subset sense) compared to T_1. Since T_1 did not contain any feasible path that was buggy, we can guarantee the same for T_2 as well, thus subsuming it.

Eager vs. Lazy. We say that a symbolic execution approach is *eager* if the successor relation is defined only for feasible states. In other words, when we

encounter an infeasible state, we immediately backtrack and compute an interpolant, succinctly capturing the reason of the infeasibility. Though different systems might employ different search strategies for symbolic execution (our formulation above is called *forward* symbolic execution [20]), it is worth to note that all common symbolic execution engines are indeed eager. This eagerness has been considered as a clear advantage of symbolic execution, since it avoids the consideration of infeasible paths, which could be exponential in number.

However, with learning (interpolation), being eager might not give us the best performance. The intuition behind this is that we are using the learned interpolant from T_1 to subsume T_2, which has less feasible paths than T_1. Therefore, if T_1 itself has very few feasible paths due to eagerness, it is unlikely that the learned interpolant would be able to subsume many of such T_2s.

4 Algorithm

We present our algorithm as a symbolic execution engine with interpolation and speculative abstraction. In Fig. 4, the recursive procedure SymExec is of the type SymExec : $SymStates \times \mathbb{N} \rightarrow FOL \cup \{\epsilon\}$. It takes two parameters – a symbolic state s typically on which to do symbolic execution, and a number representing the current level of speculative abstraction, which we will define soon. Its return

Assume initial state $s_0 \equiv \langle \ell_{\mathsf{start}}, \cdot, true \rangle$
⟨1⟩ *Initially* : SymExec$(s_0, 0)$
function SymExec$(s \equiv \langle \ell, \sigma, \Pi \rangle,$ AbsLevel$)$
⟨2⟩ **if** AbsLevel > 0 **then**
⟨3⟩   **if** (bounds violated) or $(\ell \equiv \ell_{\mathsf{error}})$ **then return** ϵ **endif**
⟨4⟩ **else if** $\ell \equiv \ell_{\mathsf{error}}$ **then** report error and halt
⟨5⟩ **endif**

⟨6⟩ **if** TERMINAL(s) **then** $\overline{\Psi} := true$
⟨7⟩ **else if** $\exists\ s' \equiv \langle \ell, \cdot, \cdot \rangle$ annotated with $\overline{\Psi}$ s.t. SUBSUME$(s, \langle s', \overline{\Psi}' \rangle)$ **then** $\overline{\Psi} := \overline{\Psi}'$
⟨8⟩ **else if** INFEASIBLE(s) **then**
⟨9⟩ $s' := \langle \ell, \sigma, \overline{\nabla}(\Pi) \rangle$
⟨10⟩ $\overline{\Psi}' :=$ SymExec$(s',$ AbsLevel $+ 1)$
⟨11⟩   **if** $\overline{\Psi}' \equiv \epsilon$ **then** $\overline{\Psi} := false$ **else** $\overline{\Psi} := \overline{\Psi}'$ **endif**
⟨12⟩   **if** AbsLevel $\equiv 0$ **then** clear data on bounds **endif**
⟨13⟩ **else**
⟨14⟩ $\overline{\Psi} := true$
⟨15⟩   **foreach** transition t: $\ell \longrightarrow \ell'$ **do**
⟨16⟩ $s' :=$ SYMSTEP(s, t)
⟨17⟩ $\overline{\Psi}' :=$ SymExec$(s',$ AbsLevel$)$
⟨18⟩   **if** $\overline{\Psi}' \equiv \epsilon$ **then return** ϵ
⟨19⟩   **else** $\overline{\Psi} := \overline{\Psi} \wedge$ INTP$(\Pi, constraints(t) \wedge \neg\, \overline{\Psi}')$
⟨20⟩   **endfor**
⟨21⟩ **endif**
⟨22⟩ annotate s with $\overline{\Psi}$ and **return** $(\overline{\Psi})$
end function

Fig. 4: A Framework for Lazy Symbolic Execution with Speculative Abstraction

value is a FOL formula representing the interpolant it generated at s. A special value of ϵ is used to signify failure of speculation.

Initially, $\mathsf{SymExec}$ is called with the initial state s_0 with ℓ_{start} as the program point, an empty symbolic store, and the path condition *true*. For clarity, ignore lines 2-5 which we will come to later. Lines 6-12, represent the three base cases of eager symbolic execution in general – terminal, subsumed and infeasible node (of course, in our lazy method infeasible node is not a base case). In line 6, if the current symbolic state s is a terminal node (defined by ℓ being the same as ℓ_{end}), we simply set the current interpolant $\overline{\Psi}$ to *true*, as the path is safe and there is no infeasibility to preserve. In line 7, the subsumption check is performed to see if there exists another symbolic state s' at the same program point ℓ such that s' subsumes s (see Definition 2). If so, the current interpolant $\overline{\Psi}$ is set to be the same as the subsuming node's interpolant $\overline{\Psi}'$. Note that this is an important case for symbolic execution to scale as it can result in exponential savings.

In line 8, we check if the current state s is infeasible, defined by $[\![s]\!]_\sigma$ being unsatisfiable. Normally at this point, eager symbolic execution would simply generate the interpolant *false* to denote the infeasibility of s and return. For lazy symbolic execution, we begin our speculation procedure here. Line 9 creates a new symbolic state s' such that it has the same program point ℓ and symbolic store σ as s, but its (unsatisfiable) path condition Π is selectively abstracted using $\overline{\nabla}$ to make the new path condition, which is satisfiable. In our implementation of $\overline{\nabla}$, since $\mathsf{SymExec}$ does forward symbolic execution, the path condition would have been feasible until the preceding state whose successor is s. That is, the state s'' such that $\mathsf{SUCC}(s'', s)$ must have been a feasible state. Hence simply setting Π to the path condition of s'' would make it satisfiable. This mimics *deleting* the latest constraint(s) from Π that caused its infeasibility. In Section 5, we discuss the reasons for implementing $\overline{\nabla}$ in this way.

Once the abstraction is made, we now speculate by recursively calling $\mathsf{SymExec}$ with s' and incrementing the abstraction level by 1. An abstraction level greater than 0 means that we are under speculation mode. $\mathsf{SymExec}$ essentially performs symbolic execution on the selectively abstracted state but with a condition – focus now on lines 2-5. Running under speculation mode, if at any point the bound is violated or if the error location ℓ_{error} is encountered, it means the speculation failed. In this case, we return a special value ϵ to signify the failure (line 3). Of course, if we are not speculating and ℓ_{error} is encountered (line 4), then it is a real error to be reported and the entire verification process halts. Otherwise, $\mathsf{SymExec}$ proceeds to normally explore s and finally return an interpolant.

Now in line 10, the interpolant returned from speculation is stored in $\overline{\Psi}'$. If ϵ was returned, indicating that speculation failed, we simply resort to using *false* as the interpolant, just like a fully eager symbolic execution procedure. Otherwise, we use the interpolant computed by speculation (line 11). Finally, in line 12, if the current abstraction level is 0 (i.e., we are at the 'root' of the speculation tree), then regardless of whether we succeeded or not, we reset all the data that count towards the bounds. For instance, in our implementation, we restrict the speculation to not explore more than one state per program point

ℓ, which would result in a bound that is linear in the program's size. In this case, we have to maintain a count of the number of states explored for each program point. At line 12 this data is cleared since the speculation has finished.

Note that there are two reasons why speculation can fail. A first reason is simply that an abstracted guard is needed to *avoid a counter-example*. If this guard corresponds to abstraction level 0, speculation resulted in nothing learnt *at this program point* (but we could have learnt something from the start of speculation until encountering the counter-example, for descendant program points). If however the guard abstraction is at a deeper level, the top-level invocation of speculation still can learn new interpolants. The second reason why speculation can fail is that the *bound was exceeded*. In this case, we put forward that, by increasing the bound, it is not likely to result in significant learning. That is, increasing the bound is a strategy of diminishing returns. We will return to this point when we discuss certain statistics in Section 5.

If none of the base cases were activated, SymExec proceeds to unwind the path, in lines 13-20. It first initialises the interpolant $\overline{\Psi}$ to *true*. Then, for every transition from the current program point ℓ, it does the following. First it performs a symbolic step (SYMSTEP) to obtain the next symbolic state s' along the transition $t : \ell \longrightarrow \ell'$. Then, it recursively calls itself with s' to obtain an interpolant $\overline{\Psi}'$ for s' (note that we are not speculating here so the abstraction level is unchanged). Now, if the returned interpolant is ϵ, it means further down some speculation resulted in failure. Hence it simply propagates back this failure by returning ϵ (line 18). Otherwise, it computes the current interpolant by invoking INTP on the path condition Π and the conjunction of the constraints of the current transition, $constraints(t)$, with the negation of $\overline{\Psi}'$ (where $\Pi \wedge constraints(t) \wedge \neg\overline{\Psi}'$ is unsatisfiable). The result is conjoined with any existing interpolant (line 19). Finally, in line 22 the current state s is annotated with the interpolant $\overline{\Psi}$, which is then returned. This annotation is persistent such that the subsumption check at line 7 can utilise this information.

On loop handling: In the presence of unbounded loops, symbolic execution might not terminate in general. Handling unbounded loops, however, is often considered as an orthogonal problem. Indeed, we were able to incorporate the loop handling technique proposed by [15] into the implementation of our algorithm without difficulty.

We conclude this section with some insights about the new interpolants discovered by speculation. At the root of speculation, the eager algorithm would have returned *false* as an interpolant. Therefore any other valid interpolant is clearly better. However, is it the case that using the new (and better) interpolant here, results in better interpolants higher up in the tree? Intuitively the answer is yes, provided that the interpolation algorithm is, in some sense, well behaved. We formalize this as follows.

Definition 3 (Monotonic Interpolation). *The interpolation method used in our algorithm is said to be* monotonic *if for all transition t, path condition Π, and formulas $\overline{\Psi}_1, \overline{\Psi}_2 \bullet \overline{\Psi}_1 \models \overline{\Psi}_2$ implies $\mathsf{INTP}(\Pi, constraints(t) \wedge \neg\, \overline{\Psi}_1) \models \mathsf{INTP}(\Pi, constraints(t) \wedge \neg\, \overline{\Psi}_2)$*

Monotonicity ensures that better interpolants at a program point translate into better interpolants at a predecessor point. The supreme interpolation algorithm, based on the weakest precondition, is of course monotonic. A more practical algorithm, however, may not be guaranteed to be monotonic. For example, an algorithm which is based on computing an unsatisfiable core (i.e., simply disregarding constraints which do not affect unsatisfiability), is in general not monotonic because it can arbitrarily choose between choices of cores.

Nevertheless, we noticed in our experiments, detailed in Section 5, that new interpolants from speculation do translate into better interpolants and this, in turn, produces more subsumption. This indicates that the interpolation algorithm employed in [14], is indeed relatively well behaved. Some random inspections of the interpolants obtained in the experiments showed that we often have monotonic behavior in practice, although not theoretically. We show via concrete statistics that as a result of this, we obtain fewer and yet better interpolants.

5 Implementation and Evaluation

We implemented our lazy algorithm on top of TRACER [14], an eager symbolic execution system, and made use of the same interpolation method and theory solver presented in [14]. Let us now remark on our two design choices.

Selective abstraction: in principle, selective abstraction ($\overline{\nabla}$) can be done in many ways, formally, by deleting any "correction subset" [19] of the unsatisfiable formula. We implemented selective abstraction by deleting constraint(s) from the latest guard that we encountered during forward symbolic execution. The reason is two-fold. Firstly, deleting the latest guard *guarantees* the formula to become satisfiable without requiring to compute any of its correction subsets (the latest guard is trivially a *minimal* correction subset), which could be expensive. Secondly, given an *incremental* theory solver, deleting the latest constraints can be implemented more efficiently than deleting those encountered earlier. Although we believe that more sophisticated analysis can be employed to make a well-informed decision, the empirical results show that this approach works well in practice.

Speculation bound: we used a *linear* bound for the speculation. In particular, during speculation if a program point is visited more than once and it cannot be subsumed, we stop the speculative search, and use the interpolant *false* at the latest speculation point. Intuitively, anything less than a linear bound could make speculation arbitrarily short, hence we need to give each program point *at least* one chance to be explored. Our experiments confirm that often, a linear bound that gives each program point *at most* one chance, is good enough.

We used as benchmarks sequential C programs from a varied pool – five device drivers from the ntdrivers-simplified category of SV-COMP 2013 [2]: cdaudio, diskperf, floppy, floppy2 and kbfiltr, two linux drivers qpmouse and tlan, an air traffic collision avoidance system tcas, and two programs from the Mälardalen WCET benchmark [21] statemate and nsichneu for which the safety property was

the approximate WCET. We chose only safe programs for our benchmarks as they ensure a full search of the program's state space. With unsafe programs, if the error is encountered very early in the search process (e.g., due to good heuristics), hardly any useful comparison can be drawn. All experiments are carried out on an Intel 2.3 Ghz machine with 2GB memory.

Benchmark	CPA Time (sec)	IMP Time (sec)	TRACER								
			Time (sec)			States			#Interpolants		
			EAG	LZY	Speedup	EAG	LZY	Red.	EAG	LZY	Red.
cdaudio	19	30	41	23	1.78	4396	2864	35%	3854	2689	30%
diskperf	28	149	53	19	2.79	4309	1617	62%	4012	1514	62%
floppy	27	36	25	12	2.08	3535	1635	54%	3208	1534	52%
floppy2	98	40	42	29	1.45	5063	3153	38%	4536	2863	37%
kbfiltr	3	8	4	3	1.33	973	756	22%	860	691	20%
qpmouse	3	8	32	15	2.13	1313	779	41%	1199	723	40%
tlan	T/O	T/O	41	26	1.58	3895	2545	35%	3542	2324	34%
nsichneu	5	41	40	5	8.00	4481	1027	77%	4379	1018	77%
statemate	2	T/O	72	5	14.40	6680	616	91%	4370	471	89%
tcas	2	11	19	1	19.00	5500	369	93%	5248	348	93%
Total	**367**	**683**	**369**	**138**	**2.67**	**40145**	**15361**	**62%**	**35208**	**14175**	**60%**

Table 1. Verification Statistics for Eager and Lazy SE (A T/O is 180s (3 mins))

To give a perspective of where TRACER stands in the spectrum of verification tools, we compare its performance with two competitive verifiers CPA-CHECKER [30] (ABM version) and IMPACT [23]. Of these, IMPACT implements an interpolation-based model checking procedure, whereas CPA-CHECKER is a hybrid of SMT-based search and CEGAR. Since IMPACT is not publicly available, we use CPA-CHECKER's implementation of the IMPACT algorithm [23].

For each benchmark, we record in the shaded columns in Table 1 the verification time (in seconds) of CPA-CHECKER (CPA), IMPACT (IMP) and TRACER with *eager* symbolic execution (TRACER EAG.), respectively. As it can be seen TRACER is generally faster than IMPACT but sometimes slower than CPA-CHECKER so it can be roughly positioned between the two (closer to CPA-CHECKER) in terms of performance. This comparison is to show that we chose a competitive verifier to implement our algorithm and we expect the same benefits to be provided to other similar verifiers.

We now present the main results in the rest of Table 1. In the set of columns labelled Time (sec) we show the verification time of TRACER in seconds for each benchmark. In this, the (shaded) column EAG which we just saw, performed eager symbolic execution, while the LZY column performed lazy symbolic execution, and Speedup is the ratio of the two. It can be seen that in all programs, laziness makes the verification much faster, providing an average speedup of 2.67. This also makes lazy TRACER perform much better than eager TRACER. We notice enormous improvement for nsichneu, statemate and tcas, as these are programs with a large number of infeasible paths and the safety property on a small number of variables, the perfect scenario for our speculation to shine.

We move on to a more fine-grained measurement than time in the next set of columns States, which shows the number of symbolic states TRACER encountered during verification. In total, we found that 40145 states were encountered without speculation (EAG) and just 15361 states with speculation (LZY), a reduction of about 62%. This shows that speculation results in more subsumption, which thereby causes a reduction in the search space.

Next, we measure the improvement in memory provided by speculation. In the set of columns #Interpolants, we show the total number of interpolants stored by TRACER at the end of the verification process. Interpolants typically contribute to a major part of memory used by modern symbolic execution verifiers. In this regard, laziness reduced the number of interpolants in TRACER from 35208 (EAG) to 14175 (LZY), a reduction of 60% across all benchmarks.

We focus on the two metrics seen above: number of interpolants (#Interpolants), and amount of subsumption, in terms of states (States) encountered. The critical point is the inverse relationship: *laziness provided a much smaller number of interpolants while simultaneously increasing subsumption.* In other words, the *quality* of interpolants discovered through speculation is enhanced.

We conclude this section with a few more statistics which, while not directly linked to absolute performance, nevertheless shed additional insight. First, consider the number of distinct program variables that are involved in the interpolants. In the case without speculation, we noticed across all benchmarks that there were **363** such variables. In contrast, with speculation, the number is only **229**. This means that many (134) variables were not required to determine the safety of the program. They were being needlessly tracked by interpolants simply to preserve infeasible paths.

Next consider the "success rate" of speculation: how often does speculation find an alternative interpolant? For simplicity, consider only those speculations triggered at the top-level of the algorithm (from abstraction level 0 to 1). We found, across the benchmark programs, a rate of **40-90%**, more often at the higher end. This means that speculation returns something useful most of the time. However, note importantly that even when speculation was not successful at the top-level, there is likely to have been interpolants discovered at the lower levels. These are interpolants one would have not found without speculation.

To elaborate on the success rate of 40-90%, programs having large number of infeasible paths tend to produce a high success rate, because as per our key intuition, many such paths will be unrelated to the safety. Similarly, programs with few infeasible paths produce a low success rate. In our experiments, the highest success rates (90%) were from nsichneu, tcas and statemate, which have a large number of infeasible paths as mentioned before.

Finally, reconsider the bound. The above success rate also indicates that there are a significant, though minor, number of failures. We wish to mention that when we do fail, the overwhelming reason is *not the bound*, but instead, the (spurious) counter-examples. In summary, the rather high rate of success, and the rather low rate of failure caused by the bound, together suggest that increasing the bound would be a strategy of diminishing returns.

6 Related Work and Concluding Discussion

Symbolic execution [17] has been widely used for program understanding and program testing. We name a few notable systems: KLEE [3], Otter [26], and SAGE [11]. Traditionally, execution begins at the first program point and then proceeds according to the program flow. Thus symbolic execution is actually *forward* execution. Recently, [20] proposed a variation, *directed* symbolic execution, making use of heuristics to guide symbolic execution toward a particular target. This has shown some initial benefits in program testing.

For the purpose of having scalability in program verification, however, symbolic execution needs to be equipped with *learning*, particularly in the form of interpolation [16, 24, 15, 14, 1]. Due to the requirement of *exhaustive* search, as in the case of this paper, these systems naturally implement forward symbolic execution. Recently, interpolation has also been applied to the context of concolic testing [13]. All the above-mentioned approaches can be classified as *eager* symbolic execution. In other words, we do not continue a path when the accumulated constraints are enough to decide its infeasibility.

In the domain of SAT solving and hardware verification, *property directed reachability* (PDR) [10] has recently emerged as an alternative to interpolation [22]. Some notable extensions of PDR are [12, 4, 29]. However, the impact of PDR to the area of software verification is still unclear. While such "backward" execution has merits in terms of being goal directed, it has lost the advantage of using the (forward) computation to limit the scope of consideration.

In contrast, our lazy symbolic execution preserves the intrinsic benefits of symbolic execution while at the same time, by opening the infeasible paths selectively, it enables the learning of *property directed* interpolants. We believe this is indeed the reason for the efficiency achieved and demonstrated in Section 5.

The traditional CEGAR-based approach to verification may also be thought of a "lazy". This is because it starts from a coarsely abstracted model and subsequently refines it. Such concept of laziness is, therefore, different from what discussed in this paper. In the context of this paper, given a refined abstract domain, a CEGAR-based approach is in fact considered as eager, since it avoids traversal of infeasible paths, which are blocked by the abstract domain. Some of such paths are indeed counter-examples learned from the previous phases. The work [24] discussed this as a disadvantage of CEGAR-based approaches: they might not recover from over-specific refinements. Our contribution, therefore, is plausibly applicable in a CEGAR-based setting.

There is now an emerging trend of employing generic SMT solvers for (bounded) symbolic execution, and since modern SMT solvers, e.g. [9], do possess the similar power of interpolation – in the form of conflict clause learning or lemma generation – we now make a few final comments in this regard.

First, note that lazy symbolic execution has no relation with the concept of *lazy* SMT. In particular, the dominating architecture DPLL(T), which underlies most state-of-the-art SMT tools, is based on the integration of a SAT solver and one (or more) T-solver(s), respectively handling the Boolean and the theory-specific components of reasoning. On the one hand, the SAT solver enumerates

truth assignments which satisfy the Boolean abstraction of the input formula. On the other hand, the T-solver checks the consistency in T of the set of literals corresponding to the assignments enumerated. This approach is called lazy (encoding), and in contrast to the eager approach, it encodes an SMT formula into an equivalently-satisfiable Boolean formula and feeds the result to a SAT solver. See [27] for a survey.

Second, we note that though the search strategies used in modern DPLL-based SMT solvers would be more dynamic and different from the forward symbolic execution presented in this paper, it is safe to classify these SMT solvers as *eager* symbolic execution. This is because, in general, whenever a conflict is encountered, a DPLL-based algorithm would analyze the conflict, learn and/or propagate new conflict clauses or lemmas, and then immediately backtrack (backjump) to some previous decision, dictated by its heuristics [18].

We believe that for the purpose of program verification, the benefit of being lazy by employing speculative abstraction, would also be applicable to SMT approaches. This is because, in general, we can always miss out useful (good) interpolants if we have not yet seen the complete path. In this paper, we have demonstrated that in verification, property directed learning usually outperforms learning from "random" infeasible paths. Eagerly stopping when the set of constraints is unsatisfiable might prevent a solver from learning the conflict clauses which are more relevant to the safety of the program. In SMT solvers, the search, however, is structured around the decision graph. Therefore, some technical adaptations to our linear bound need to be reconsidered. For example, a bound based on the number of decisions seems to be a good possibility. Moreover, in SMT setting, there are no error locations. One possible idea is, when compiling a verification problem into SMT input format, we specially "mark" the constraints guarding the error locations, so that unsatisfiable cores involving marked constraints can be favored over those not.

7 Conclusion

We presented a systematic approach to perform speculative abstraction in symbolic execution in pursuit of program verification. The basic idea is simple: when a symbolic path is first found to be infeasible, we abstract the cause of infeasibility and enter speculation mode. In continuing along the path, more abstractions may be performed, while remaining in speculation mode. Crucially, speculation is only permitted up to a given bound, which is a linear function of the program size. A number of reasonably sized and varied benchmark programs then showed that our speculative abstraction produced speedups of a factor of two and more.

References

1. A. Albarghouthi, A. Gurfinkel, and M. Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *VMCAI*, 2012.
2. D. Beyer. Second competition on software verification. In *TACAS*, 2013.

3. C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.
4. A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. Ic3 modulo theories via implicit predicate abstraction. *CoRR*, 2013.
5. A. Cimatti, A. Griggio, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo theories. In *TACAS'08*, pages 397–412, 2008.
6. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. CounterrExample-Guided Abstraction Refinement. In *CAV*, 2000.
7. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis. In *POPL*, 1977.
8. W. Craig. Three uses of Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Computation*, 22, 1955.
9. L. De Moura and N. Bjørner. Z3: an efficient smt solver. In *TACAS*, 2008.
10. N. Een, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *FMCAD*, 2011.
11. P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 2012.
12. K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, 2012.
13. J. Jaffar, V. Murali, and J. Navas. Boosting Concolic Testing via Interpolation. In *FSE*, 2013.
14. J. Jaffar, V. Murali, J. Navas, and A. Santosa. TRACER: A symbolic execution engine for verification. In *CAV*, 2012.
15. J. Jaffar, J. Navas, and A. Santosa. Unbounded Symbolic Execution for Program Verification. In *RV*, 2011.
16. J. Jaffar, A. E. Santosa, and R. Voicu. An interpolation method for clp traversal. In *CP*, 2009.
17. J. C. King. Symbolic Execution and Program Testing. *Com. ACM*, 1976.
18. D. Kroening and O. Strichman. Decision procedures: An algorithmic point of view, 2008.
19. M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Automated Reasoning*, 2008.
20. K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *SAS*, 2011.
21. Mälardalen WCET research group benchmarks. URL `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, 2006.
22. K. L. McMillan. Interpolation and SAT-based model checking. In *15th CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.
23. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.
24. K. L. McMillan. Lazy annotation for program testing and verification. In *CAV*, 2010.
25. S. Navabpour, B. Bonakdarpour, and S. Fischmeister. Path-aware time-triggered runtime verification. In *RV*, 2012.
26. E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE*, 2010.
27. R. Sebastiani. Lazy satisability modulo theories. *JSAT*, 2007.
28. S. W. Visser, C. Păsăreanu. Test input generation with java pathfinder. In *ISSTA*, 2004.
29. T. Welp and A. Kuehlmann. Qf bv model checking with property directed reachability. In *DATE*, 2013.
30. D. Wonisch. Block Abstraction Memoization for CPAchecker. In *TACAS*, 2012.