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. Symbolic execution with interpolation has emerged as a powerful technique for software verification. Its performance heavily relies heavily on the quality of the computed "interpolants", formulas which succinctly describe a generalization of the symbolic states proved so far. Symbolic execution by default is *eager*, that is, execution along a symbolic path stops the moment when infeasibility is detected in the logical constraints describing the path so far. This may however hinder the discovery of better interpolants, i.e., more general abstractions of the symbolic state which are yet sufficient ensure the entire symbolic path remains error-free.
In this paper, we present a systematic method which speculates that an infeasibility may be temporarily ignored in the pursuit of better information about the path in question. This speculation does not lose the intrinsic benefits of symbolic execution because its operation shall be bounded. We argue that the trade-off between this 'enhanced learning' and incurring additional cost (which in principle may not be productive) is in fact in favor of speculation. Finally, we demonstrate with a state-of-the-art system on realistic benchmarks that this method enhances symbolic execution by a factor of 2 or more.

1 Introduction

Symbolic execution has been shown to be largely successful in program verification, testing and analysis [15, 20, 23, 13, 16]. 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 the 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) [6] or Counterexample-Guided Abstraction Refinement (CEGAR) [5], since it avoids the exploration of *infeasible paths* which could block exponentially large symbolic trees in practice.

The main challenge for symbolic execution is addressing the path explosion problem. The approaches of [15, 20, 14, 13] tackle this fundamental issue by eliminating from the concrete 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 no-good 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 error-free. As a result, if the program point is encountered again through a different path such that the interpolant is implied, the new path can be *subsumed*, because it can be guaranteed to be error-free. Interpolation has been proven 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 *goal-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 during symbolic execution, instead of backtracking immediately, we *selectively abstract* the infeasible state so that it becomes feasible, and proceed with the search. For instance, assuming forward symbolic execution, we can ignore the constraint from the most recent guard that caused the infeasibility, in order make it a feasible state.

By performing such an abstraction, we say that we have entered *speculation mode*. More generally, as we progressively abstract away infeasibility in the consideration of a symbolic path, we are exhibiting a goal (or property) directed strategy. Note again that this exercise does not have an immediate benefit because we already know that all paths containing an infeasible prefix subpath are in fact safe. The point here is to learn new interpolants. Now, recalling the mantra "*a little knowledge is dangerous, but so is a lot*", is it in fact true that the more interpolants we learn, the better? The answer is no: we could have an exponential number of interpolants and yet many or all of them may be useless for pruning the search space. What we really want are *good* interpolants. But of course the challenge is how to target our algorithm in this direction.

Our answer is speculation, but *subject to a bound*. This mitigates the potential blowup of what was already a workable method (symbolic execution with interpolation), but yet retaining the possibility of discovering good interpolants.

It is easy to see that this bound should be linearly related to the program size: anything less than this makes the speculation phase arbitrarily short. That is, we do need set aside at least a linear bound. It is a main contribution of this paper, that in other direction, a linear bound is *good enough.*

2 Examples

We begin with an exemplification of when (eager) symbolic execution is clearly not the most direct way to conduct a proof. For the programs in Figure 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.

In the first program in Figure 1(a), it is easy to see that we can reason about y *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 [5] here, where *abstraction refinements* are performed only when a spurious counter-example is encountered, that we would have a very efficient (linear) proof.

In the next program in Figure 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, is is in fact necessary to record and manage the unsatisfiable path expressions as they are encountered during symbolic execution. (Using CEGAR, in contrast, would require a large number of abstraction refinements in order to remove counter-examples arising from not recognizing the unsatisfibility of "unsafe" path expressions, i.e. those corresponding to $n/2$ or more increments of y.)

In practice, a typical program would correspond to being in between the above two extreme cases in Figures 1(a) and 1(b). Our key argument, however, is that in fact a typical program lies closer to the first example rather than the second. For the final example program in Figure 1(c), assume that all and only

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

(a) Lazy is Good

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

(b) Eager is Good

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

(c) Lazy is Still Better

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

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.

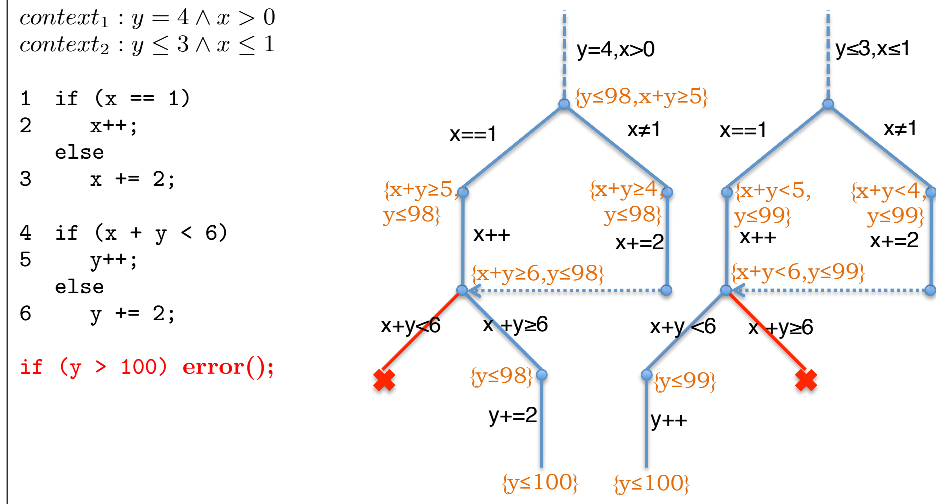

Fig. 2: A Symbolic Execution Tree with Learning

We now elaborate our method using a concrete example, in Figure 2. Consider two different initial contexts for the program fragment: $y = 4 \wedge x > 0$ and $y \leq 3 \wedge x \leq 1$. In both contexts, the program is safe because $y \leq 100$ at the end.

We start symbolic execution at program point 1 with the first context $y = 4 \wedge x > 0$. Both the then and else bodies being enabled, assume that it first takes the then body with condition `x==1`, executing `x++` at line 2 and reaching line 4. For this branch, the then body cannot be reached 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 an interpolant. Assuming weakest precondition (WP) is used as the interpolant, it would annotate line 4 with $x + y \geq 6$. Exploring the else body would prove that it is safe, and so line 6 would be annotated with the interpolant $y \leq 98$ to preserve its safety. Combining the then and else body's interpolants, it would generate $x' + y \geq 6 \wedge y \leq 98$ at line 4. (Note that we use x instead of x' in the Figure because we always project the formula onto the original program variable names.) Passing this back through WP propagation, it would generate $x + y \geq 5 \wedge y \leq 98$ at line 2.

Now, executing the else body `x+=2` of the first if-statement, it would reach line 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 line 3. Now, combining the then and else body's interpolant 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 Figure 2.

Now, when the program fragment is reached along the second context $y \leq 3 \wedge x \leq 1$, subsumption cannot take place at line 1 as the interpolant $y \leq 98 \wedge x + y \geq 5$ is not implied. 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.

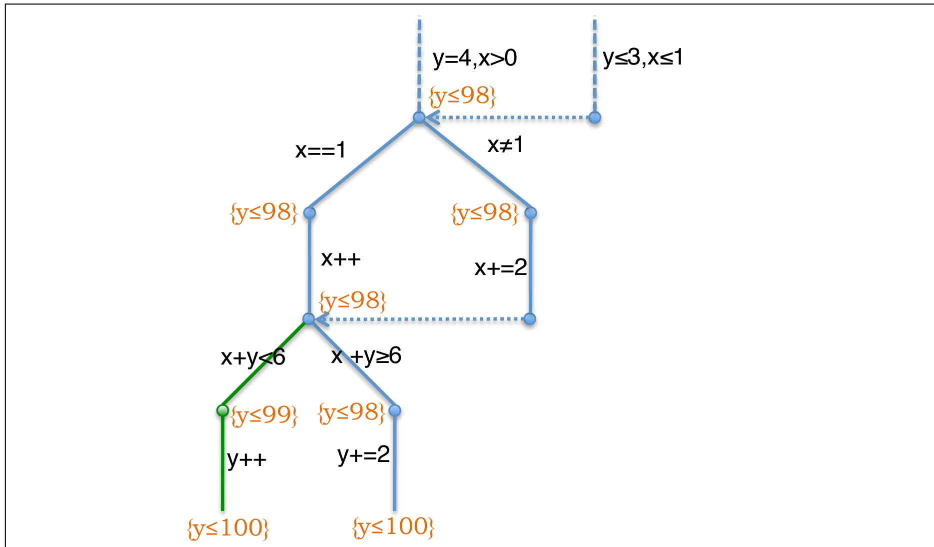

Fig. 3: Symbolic execution tree with selective abstraction and speculation

We now describe our lazy symbolic execution process with selective abstraction and speculation. On the same program, assume that we reach line 1 with the context $y = 4 \wedge x > 0$, and proceed to execute the then body `x++`. On reaching the second if-statement at line 4, as before the then body would be infeasible because of the unsatisfiable path condition $y = 4 \wedge x > 0 \wedge x = 1 \wedge x' = x+1 \wedge x'+y < 6$. This time, instead of immediately backtracking, we selectively abstract the path condition to make it satisfiable. Since we are doing forward symbolic execution, a simple way to do this is to ignore the constraint(s) from the last guard that we encountered. Here, this means deleting the constraint $x+y < 6$, which would make the formula satisfiable.

We now proceed to explore the then body with the abstracted path condition $y = 4 \wedge x > 0 \wedge x = 1 \wedge x' = x + 1$. We say that we have entered 'speculation mode' – we speculate that we can obtain a better interpolant by opening the infeasible branch than by simply backtracking. However there is a problem: in general the 'subtree' underneath the infeasible branch may be arbitrarily large, exploring which would be intractable. We definitely do not want to resort to an

exponential search, as we already know the entire tree is infeasible, and hence safe. Therefore we restrict the search by imposing a bound on it. A bound that we found works very well in practice from our experiments is a *linear bound*. That is, in speculation mode we explore each program point at most with one symbolic state. If another symbolic state is encountered corresponding to the same program point, we demand it to be subsumed. If not, we backjump to the last point where speculation was triggered and use the 'default' eager interpolant to block the infeasible branch. Note that speculation can be triggered recursively during another level of speculation.

Back to the example, we proceed to execute the statement `y++` and reach the end of the path to find that it is safe, as the safety $y \leq 100$ is implied. Speculation has now succeeded, hence we annotate line 5 with $y \leq 99$, assuming WP interpolants. Exploring the else body of the second branch, we execute `y+=2` and again reach the end of the safe path, generating the interpolant $y \leq 98$ at line 6. Combining the two we get the interpolant $y \leq 98$ at line 4. Propagating it back through the tree we get the interpolant $y \leq 98$ at line 1. Now, when the program fragment is reached along the second context $y \leq 3 \wedge x \leq 1$, the interpolant is implied at line 1, and the entire tree can be subsumed at the top level. The resulting final symbolic tree is shown in Figure 3. Note that we applied strengthening of WP as before in order to simplify the interpolants. (Without strengthening, the subsumption will also take place.)

This example has shown that speculation can potentially result in exponential savings for taking a 'risk' with merely linear cost. The reason why 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. Ignoring the infeasibility for the time being can help in discovering interpolants closely related to the safety, such as those in Figure 3, rather than interpolants that blindly preserve the infeasibility, such as those in Figure 2. In Section 5, we provide empirical evidence that the (potentially) exponential gains provided by speculation clearly outweigh its linear 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:

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

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* $\pi \equiv s_0 \cdot s_1 \cdot \ldots \cdot s_n$ is a sequence of symbolic states such that $\forall i \bullet 1 \leq i \leq n$ the state s_i is a *successor* of s_{i-1}, denoted as $\mathsf{SUCC}(s_{i-1}, s_i)$. A path $\pi \equiv s_0 \cdot s_1 \cdot \ldots \cdot s_n$ 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 $\pi \equiv s_0 \cdot s_1 \cdot \ldots \cdot s_n$ 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 special 'widening' operator $\overline{\nabla} : FOL \times FOL$ that accepts an *unsatisfiable* FOL formula Π and returns a satisfiable formula by abstracting (arbitrary) constraints from Π. In Section 4 we will describe a specific way to abstract constraints that is suitable to our algorithm.

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 *[7], $\mathsf{INTP}(A, B)$, 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 [3] 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', $\mathsf{SUBSUME}(s, \langle s', \overline{\Psi} \rangle)$, 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, infeasible paths compared to T_1. In other words, T_2 has exactly the same, or *less feasible paths* 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 [18]), 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, of which the number could be exponential in practice.

However, with learning, i.e. interpolation, being eager might not give us the best performance. The intuition behind this is that, here, we are using the learned interpolant from T_1 to subsume other tree, say 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_2 instances.

In this paper, we propose a lazy symbolic execution approach that whenever an infeasible state is encountered, instead of backtracking immediately, we abstract the infeasible state into a feasible one, and allow our symbolic execution to proceed further. We delay the detailed description of our algorithm to the next section.

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 : $\mathit{SymStates} \times \mathbb{N} \rightarrow \mathit{FOL} \cup \{\epsilon\}$. 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 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, 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}), then 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, then the current interpolant $\overline{\Psi}$ is set to be the same as the subsuming node's interpolant $\overline{\Psi}'$. Note that this is an

```
Assume initial state s0 ≡ ⟨ℓstart, ·, true⟩
 ⟨1⟩ Initially : SymExec(s0, 0)
function SymExec(s ≡ ⟨ℓ, σ, Π⟩, AbsLevel)
 ⟨2⟩ if AbsLevel > 0 then
 ⟨3⟩     if (bounds violated) or (ℓ ≡ ℓerror) then return ϵ endif
 ⟨4⟩ else if ℓ ≡ ℓerror then report error and halt
 ⟨5⟩ endif

 ⟨6⟩ if TERMINAL(s) then Ψ̄ := true
 ⟨7⟩ else if ∃ s' ≡ ⟨ℓ, ·, ·⟩ annotated with Ψ̄ s.t. SUBSUME(s, ⟨s', Ψ̄'⟩) then Ψ̄ := Ψ̄'
 ⟨8⟩ else if INFEASIBLE(s) then
 ⟨9⟩     s' := ⟨ℓ, σ, ∇̄(Π)⟩
⟨10⟩     Ψ̄' := SymExec(s', AbsLevel + 1)
⟨11⟩     if Ψ̄' ≡ ϵ then Ψ̄ := false else Ψ̄ := Ψ̄' endif
⟨12⟩     if AbsLevel ≡ 0 then clear data on bounds endif
⟨13⟩ else
⟨14⟩     Ψ̄ := true
⟨15⟩     foreach transition t: ℓ ⟶ ℓ' do
⟨16⟩         s' := SYMSTEP(s, t)
⟨17⟩         Ψ̄' := SymExec(s', AbsLevel)
⟨18⟩         if Ψ̄' ≡ ϵ then return ϵ
⟨19⟩         else Ψ̄ := Ψ̄ ∧ INTP(Π, constraints(t) ∧ ¬ Ψ̄')
⟨20⟩     endfor
⟨21⟩ endif
⟨22⟩ annotate s with Ψ̄ and return (Ψ̄)
end function
```

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

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 selective abstraction 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 *widened* using $\overline{\nabla}$ to make the new path condition, which is satisfiable. In our implementation, we use a simple and effective widening operator as follows: since SymExec does forward symbolic execution, the path condition would have been feasible until the preceding state whose successor is the current infeasible state 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 is a selective abstraction of Π because we are in essence *ignoring* the recent constraint(s) that caused its infeasibility.

Once the abstraction is made, we now speculate by recursively calling SymExec with s' and incrementing the abstraction level by 1. An abstraction level greater than 0 means that we are under speculation mode. SymExec essentially performs symbolic execution on the widened 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, 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 the encounter with 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 signifcant 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 in line 17 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 was done and it 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 $\overline{\Psi}'$ (note that $\Pi \wedge constraints(t) \wedge \neg\overline{\Psi}'$ is unsatisfiable). The result is conjoined with any existing interpolant (line 19).

Finally, in line 22 we annotate the current state s with the interpolant $\overline{\Psi}$ (computed from one of the above cases) and returns $\overline{\Psi}$. This annotation is persistent such that the subsumption check at line 7 can utilise this information.

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, which is based on the weakest precondition, is of course monotonic. When using a more practical algorithm, however, it is not always easy to guarantee that it is monotonic. For example, an algorithm which is based on computing an unsatisfiable core (i.e., it simply disregards some 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 [13], is indeed relatively well behaved. Some random inspections of the interpolants obtained in the experiments showed that we often have monotonic behavior. We show below via concrete statistics that as a result of this, we obtain fewer and yet better interpolants.

5 Experiments

We implemented our lazy algorithm on top of the TRACER [13]. We note that originally TRACER is an eager symbolic execution system, and we make use of the same interpolation method presented in [13]. We re-emphasize that we used a linear bound for the speculation, i.e., during speculation if a program point is visited more than once, we demand it to be subsumed. If it cannot be subsumed, we stop the speculative search, and annotate the symbolic state with *false*. We implemented the selective abstraction (the widening operator $\overline{\nabla}$) by simply ignoring the constraints in the last guard that was encountered during forward symbolic execution, which would make the unsatisfiable path condition satisfiable. Given an *incremental* theory solver, this abstraction step can be performed efficiently.

We used as benchmarks several device drivers from the ntdrivers-simplified category of the Software Verification Competition (SV-COMP) 2013: cdaudio, diskperf, floppy, floppy2 and kbfiltr. We also used two linux device driver programs qpmouse and tlan, an air traffic collision avoidance system tcas, and statemate a program generated by the STAtechart Real-time-Code generator STARC. All experiments were carried out on an Intel 2.3 Ghz machine with 2GB memory.

Bench	CPA Time (sec)	IMP Time (sec)	TRACER								
			Time (sec)			States			#Interpolants		
			EAG	LZY	Speedup	EAG	LZY	Red.	EAG	LZY	Red.
cdaudio	21	28	16	9	1.78	5158	1264	75%	2006	1129	44%
diskperf	28	152	56	14	4.00	6746	1240	82%	1766	509	71%
floppy2	98	40	20	12	1.67	6182	2283	63%	1424	900	37%
floppy	27	34	13	6	2.17	4384	1052	76%	1020	437	57%
kbfiltr	3	8	2	2	1.00	980	510	48%	247	185	25%
qpmouse	3	8	27	13	2.08	1313	718	45%	1452	761	48%
statemate	2	115	18	5	3.60	5955	852	86%	3922	1135	71%
tcas	2	13	10	3	3.33	6718	689	90%	3425	531	84%
tlan	OOM	OOM	26	16	1.63	3895	2311	41%	1859	1023	45%
Total	**184**	**398**	**188**	**80**	**2.35**	**41331**	**10919**	**74%**	**17121**	**6610**	**61%**

Table 1. Verification Statistics for Eager and Lazy Symbolic Execution

To give a perspective of where TRACER stands in the spectrum of verification tools, we compare its performance with two competitive verifiers CPA-CHECKER [25] (ABM version) and IMPACT [20]. Of these, IMPACT implements an eager symbolic-execution based search 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 its algorithm.

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 performace. We note that CPA-CHECKER and IMPACT ran out of memory for the tlan benchmark, so we exclude it from the total time giving those

verifiers the benefit of the doubt. In the end, this comparison is to show that we chose a competitive verifier to implement our algorithm and we fully 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 (with the exception of kbfiltr), selective abstraction makes the verification much faster, providing an average speedup of 2.35. This also makes lazy TRACER perform much better than eager TRACER.

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. Again, it can be seen that there is a large reduction in the states in the lazy (LZY) column across all benchmarks compared to the eager (EAG) column. In kbfiltr, we notice a reduction of 48% of states. This benefit was not shown in time because of the small size of the program which was verified in just 2 seconds. In total, we found that about 41331 states were encountered without speculation and just 10919 states with speculation, a reduction of about 74%. This shows that speculation is resulting in more subsumption, which thereby causes a reduction in the search space.

Next, we measure the improvement in space 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, selective abstraction reduced the number of interpolants in TRACER from 17121 (EAG) to 6610 (LZY), a reduction of 61% across all benchmarks.

We now focus on the two metrics: 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 **339** such variables. In contrast, with speculation, the number is only **234**. This means that many (105) 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 many times does a speculation find an alternative interpolant? For simplicity, let us consider only those speculations triggered at the top-level of the algorithm (from abstraction level 0 to abstraction level 1). We found, across the benchmark programs, a rate of **40-80%**, and more often at the higher end. This means that speculation

returns something useful, most of the time. However, it is important to note 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 if there had been no speculation. Finally, reconsider the bound. The above success rate also indicates that there are a significant, though minority, number of failures. We want mention that when we do fail, the overwhelming reason is *not the bound*, but instead, the counterexamples. 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 Discussion

Symbolic execution [17] has been widely used for program understanding and program testing. We name a few notable systems: KLEE [2], Otter [21], 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, [18] 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 [15, 20, 14, 13, 1]. Due to the requirement of *exhaustive* search, as in the case of this paper, these systems naturally implement forward symbolic execution. All the above-mentioned systems 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 [19]. Some notable extensions of PDR are [12, 4, 24]. 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 [20] 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 [22] 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 [8].

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.

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. Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *VMCAI*, 2012.
2. Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.
3. A. Cimatti, A. Griggio, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo theories. In *TACAS'08*, pages 397–412, 2008.
4. Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Ic3 modulo theories via implicit predicate abstraction. *CoRR*, 2013.
5. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. CounterrExample-Guided Abstraction Refinement. In *CAV*, 2000.
6. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis. In *POPL*, 1977.
7. W. Craig. Three uses of Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Computation*, 22, 1955.
8. Ofer Strichman Daniel Kroening. Decision procedures: An algorithmic point of view, 2008.
9. L. De Moura and N. Bjørner. Z3: an efficient smt solver. In *TACAS*, 2008.
10. Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *FMCAD*, 2011.
11. Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 2012.
12. Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *SAT*, 2012.
13. J. Jaffar, V. Murali, J.A. Navas, and A. Santosa. TRACER: A symbolic execution engine for verification. In *CAV*, 2012.
14. J. Jaffar, J.A. Navas, and A. Santosa. Unbounded Symbolic Execution for Program Verification. In *RV*, 2011.
15. J. Jaffar, A. E. Santosa, and R. Voicu. An interpolation method for clp traversal. In *CP*, 2009.
16. Joxan Jaffar, Vijayaraghavan Murali, and Jorge Navas. Boosting Concolic Testing via Interpolation. In *FSE*, 2013.
17. J. C. King. Symbolic Execution and Program Testing. *Com. ACM*, 1976.
18. Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *SAS*, 2011.
19. K. L. McMillan. Interpolation and SAT-based model checking. In *15th CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.
20. K. L. McMillan. Lazy annotation for program testing and verification. In *CAV*, 2010.
21. Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE*, 2010.
22. Roberto Sebastiani. Lazy satisability modulo theories. *JSAT*, 2007.
23. S.Khurshid W. Visser, C. Psreanu. Test input generation with java pathfinder. In *ISSTA*, 2004.
24. Tobias Welp and Andreas Kuehlmann. Qf bv model checking with property directed reachability. In *DATE*, 2013.
25. D. Wonisch. Block Abstraction Memoization for CPAchecker. In *TACAS*, 2012.