Synthesize Solving Strategy for Symbolic Execution

Zhenbang Chen*
College of Computer, National University of Defense Technology
Changsha, China
zbchen@nudt.edu.cn

Zehua Chen
College of Computer, National University of Defense Technology
Changsha, China
zehuachen2020@163.com

Ziqi Shuai
College of Computer, State Key Laboratory of High Performance Computing, National University of Defense Technology
Changsha, China
szq@nudt.edu.cn

Guofeng Zhang
College of Computer, National University of Defense Technology
Changsha, China
zhangguofeng16@nudt.edu.cn

Weiyu Pan
College of Computer, National University of Defense Technology
Changsha, China
panweiyu@nudt.edu.cn

Yufeng Zhang
College of Computer Science and Electronic Engineering, Hunan University
Changsha, China
yufengzhang@nudt.edu.cn

Ji Wang
College of Computer, State Key Laboratory of High Performance Computing, National University of Defense Technology
Changsha, China
wj@nudt.edu.cn

ABSTRACT

Symbolic execution is powered by constraint solving. The advancement of constraint solving boosts the development and the applications of symbolic execution. Modern SMT solvers provide the mechanism of solving strategy that allows the users to control the solving procedure, which significantly improves the solver's generalization ability. We observe that the symbolic executions of different programs are actually different constraint solving problems. Therefore, we propose to synthesize a solving strategy for a program to fit the program's symbolic execution best. To achieve this, we divide symbolic execution into two stages. The SMT formulas solved in the first stage are used to online synthesize a solving strategy, which is then employed during the constraint solving in the second stage. We propose novel synthesis algorithms that combine offline trained deep learning models and online tuning to synthesize the solving strategy. The algorithms balance the synthesis overhead and the improvement achieved by the synthesized solving strategy.

We have implemented our method on the *state-of-the-art* symbolic execution engine KLEE for C programs. The results of the extensive experiments indicate that our method effectively improves the efficiency of symbolic execution. On average, our method increases the numbers of queries and paths by 58.76% and 66.11%, respectively. Besides, we applied our method to a Java Pathfinder-based concolic execution engine to validate the generalization ability. The results indicate that our method has a good generalization ability and increases the numbers of queries and paths by 100.24% and 102.6% for the benchmark Java programs, respectively.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

Symbolic Execution, SMT Solving Strategy, Synthesis

ACM Reference Format:
Zhenbang Chen, Zehua Chen, Ziqi Shuai, Guofeng Zhang, Weiyu Pan, Yufeng Zhang, Ji Wang. 2021. Synthesize Solving Strategy for Symbolic Execution. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3460319.3464815

1 INTRODUCTION

Symbolic execution [2, 16] provides a general method for systematically exploring a program's path space. Recently, symbolic execution has been applied in many challenging problems in software

*Zhenbang Chen and Ji Wang are the corresponding authors.

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464815

engineering and security, *e.g.*, automatic software testing [5], vulnerability detection [1], and program repair [19]. Many successful stories [5, 12, 29] are resulted from these applications. However, the success of symbolic execution's applications is still challenged by the scalability problem caused by path explosion and constraint solving [2].

Symbolic execution analyzes a program $\mathcal{P}$ by executing $\mathcal{P}$ in a symbolic manner. $\mathcal{P}$'s inputs are symbolized (maybe partially) and assigned with symbolic values at the beginning. Symbolic execution maintains a constraint (denoted as PC) for each symbolic path p. If an input I satisfies PC, executing $\mathcal{P}$ under I results in path p. In the beginning, the PC is $true$. Then, when a non-branch statement $\mathcal{S}$ in $\mathcal{P}$ is executed, the symbolic computation corresponding to the symbolic values of the variables in $\mathcal{S}$ is carried out. When executing a branch statement $\mathcal{S}_b$ with condition b, symbolic execution calculates b's symbolic condition C_b and checks the feasibility of $\mathcal{S}_b$'s true and false branches with respect to the current PC. This feasibility checking invokes a constraint solver [17]. If $PC \wedge C_b$ is satisfiable [17], the *true branch* is feasible; if $PC \wedge \neg C_b$ is satisfiable, the *false branch* is feasible. When both branches are feasible, symbolic execution forks the state into two states and continues to execute the statements of the two branches, and the paths conditions are updated to $PC \wedge C_b$ and $PC \wedge \neg C_b$, respectively; otherwise, the infeasible branches are abandoned, *i.e.*, no input can steer $\mathcal{P}$ to this branch. In this way, $\mathcal{P}$'s path space can be systematically explored.

Symbolic execution invokes the constraint solver on-the-fly when exploring the program's path space. Constraint solving is one of the main technical challenges faced by symbolic execution [2, 6] and determines symbolic execution's scalability and feasibility. Most existing symbolic executors use the solver in a black-box manner. The existing approaches for optimizing the constraint solving in symbolic execution do the optimizations before invoking the solver, such as caching [5], reusing [30] and simplification [5]. On the other hand, existing high-performance constraint solvers (*e.g.*, SAT/SMT [17] solvers) are highly tuned for specific classes of problems. The solvers may perform poorly on new problems [8]. Hence, if we can customize the constraint solver in symbolic execution specifically for the program under analysis, symbolic execution's scalability can be improved further.

Modern SMT solvers (*e.g.*, Z3 [7] and CVC4 [4]) provide mechanisms for the users to control the solving procedure, *e.g.*, *solving strategy* [8] in Z3. We observe that solving an SMT formula with a different solving strategy may have a very different performance. For example, consider the following SMT formula in *floating-point* bit-vector theory [17], where the type of x is double.

$$x^3 = 8.0$$

If we use Z3 to solve this constraint by the *default strategy*, the solving time is around 56s[1]. However, if we use a customized solving strategy, *e.g.*, the following one, the solving time is only around 22s.

(`check-sat-using` (**then** `simplify smt`))

As far as we know, all the existing symbolic executors use the underlying SMT solver's default solving strategy. Besides, the path constraints collected during the symbolic executions of different programs may diverge in principle. Hence, customizing a *better* SMT solving strategy specifically for the program under symbolic execution can improve the solving performance, which directly improves symbolic execution's scalability.

This paper proposes to synthesize a customized solving strategy for the program under symbolic execution. Our key idea is to use the SMT formulas solved in the early stage of symbolic execution to synthesize a strategy that can be used later. In principle, our synthesis is challenged by the trade-off between the synthesis overhead and the synthesized solving strategy's effectiveness. We propose to utilize deep learning and decision tree techniques to online synthesize a solving strategy during symbolic execution, which achieves a balance between the synthesis overhead and the efficiency improvement achieved by the synthesized solving strategy. We have implemented our synthesis method on KLEE [5], *i.e.*, a *state-of-the-art* symbolic executor for C programs, and a Symbolic Pathfinder (SPF) [20] based concolic testing [11, 25] tool for Java programs. The results of the extensive experiments on real-world open-source programs indicate the effectiveness and the generalization ability of our synthesis method.

The main contributions of this paper are as follows.

- We propose a synthesis method to online generate a solving strategy for the program under symbolic execution to improve efficiency.
- We formalize the tactic-based constraint solving procedure as a Markov Decision Process with cost and propose to use an offline trained deep reinforcement learning model to generate candidate tactic sequences for synthesis.
- We have implemented our method on two state-of-the-art symbolic execution engines and conducted extensive experiments on real-world open-source programs.
- Our synthesis method can, on average, increase the queries and paths in the symbolic execution for C programs by 58.76% and 66.11%, respectively. Besides, our method has a good generalization ability and achieves 100.24% and 102.6% relative increases of the queries and paths for Java programs, respectively.

The remainder of this paper is organized as follows. Section 2 gives a brief introduction to solving strategy and illustrates our synthesis method by an example. Section 3 gives the formalization of the tactic-based solving procedure. Section 4 presents the synthesis method in detail. Section 5 gives the experimental results. The related work is discussed and compared in Section 6, and the conclusion is drawn in Section 7.

2 ILLUSTRATION

2.1 Solving Strategy

Due to the background of symbolic execution, we only consider the SMT formulas in *quantifier-free* first-order logic [17]. In principle, most solving tactics transform a formula into another. Let Γ be the set of the SMT formulas in different theories and SAT formulas, and Θ be the set of solving results, *i.e.*, $\{\texttt{SAT}, \texttt{UNSAT}\}$.

Definition 2.1. (**Tactic**) A tactic $\mathbf{T}$ is a function $\Gamma \rightarrow \Gamma \cup \Theta$ that gives the resulted formula or solving result of an input formula.

[1] Z3's version is 4.6.2. The CPU is 2.5GHz.

$$\begin{array}{rcl} \mathcal{S} & ::= & \mathbf{T}(\bar{p}) \mid \mathcal{S} \fatsemi \mathcal{S} \mid \mathbf{ITE}(C, \mathcal{S}, \mathcal{S}) \\ C & ::= & \mathbf{P} \mid \mathbf{P} \oplus c \\ p & ::= & (n, \texttt{true}) \mid (n, \texttt{false}) \end{array}$$

Figure 1: Syntax of Solving Strategy

For example, `simplify` in Z3 is a tactic for simplifying and rewriting the formula into the normal form. If we apply `simplify` to $x > (2 - 1)$, we will get $\neg(x \leq 1)$, which is in the normal form for arithmetic relations. Suppose that x is a *bit-vector variable* with length 3. Then, for $\neg(x \leq 1)$, if we apply the tactic `bit-blast` that encodes a bit-vector formula into an SAT formula, we will get the following SAT formula, where x_2 and x_1 are the two boolean variables representing the third and second bits of x, respectively.

$$\neg x_2 \wedge x_1 \tag{1}$$

It means that the sign bit x_2 should be zero (*i.e.*, x is positive) and the second bit (*i.e.*, the highest bit) should be one, which implies that x is at least 2. Besides the tactics of transformations, there are some tactics that are in charge of solving, including `sat`, `smt`, `nlsat`, *etc.* These tactics are in charge of the last step in the solving procedure. Applying them will give a solving result in Θ or time out. Tactics also have some parameters that can be used to configure the transformation or solving. A tactic with different configurations may produce different results or have different performances.

To specify the best tactics for solving a formula, we need some necessary information about the formula, which can be obtained by the probe mechanism defined as follows. Let $\mathbb{I}$ be the integer set and $\mathbb{B}$ be the boolean value set, *i.e.*, $\{true, false\}$, a probe is defined as follows.

Definition 2.2. (**Probe**) A probe $\mathbf{P}$ is a function $\Gamma \rightarrow \mathbb{I} \cup \mathbb{B}$ that gives a feature value of an input formula.

For example, `is-qfbv` and `num-consts` are two probes in Z3 that give whether the formula is a quantifier-free bit-vector formula and the number of the constants in the formula, respectively. If the bit-vector formula is $x > 1$, `is-qfbv` returns *true*, and `num-consts` returns 1.

Modern SMT solvers provide a domain-specific language (DSL) to construct solving strategies in terms of tactics and probes. The language contains composition operators, including sequence, branch, loop, *etc.* Let $\mathbb{T}$ be the set of tactics, $\mathbb{P}$ be the probe set, $\mathbb{N}$ be the name set, and $\mathbb{C}$ be the set of constants. Figure 1 gives the language of the solving strategies considered in this paper, where $\mathbf{T} \in \mathbb{T}$, $c \in \mathbb{C}$, $\mathbf{P} \in \mathbb{P}$, $n \in \mathbb{N}$, $\oplus$ represents a commonly used numeric relation operator, and $\bar{p}$ represents a list of p.

$\mathbf{T}(\bar{p})$ is a tactic with parameters. We do not consider the parameters with integer values. $\mathbf{T}$ represents a tactic with the default parameter values. We only consider two kinds of compositions, *i.e.*, sequential and branch compositions. The condition in branch composition can be the probe whose value is boolean or a numeric relation $\mathbf{P} \oplus c$. For example, the following provides a strategy for bit-vector formulas.

$$\texttt{simplify} \fatsemi \mathbf{ITE}(\texttt{num-consts} < 3, \texttt{bit-blast} \fatsemi \texttt{sat}, \texttt{smt}) \tag{2}$$

Given a bit-vector formula φ, the strategy first applies `simplify` to φ and gets φ_1. Then, depending on the number of the constants in φ_1, the strategy uses a bit-blasting style of solving in case φ_1's number is less than three or directly applies `smt` for solving φ_1.

2.2 Framework

We want to synthesize an optimal solving strategy online for the target program to improve the efficiency of symbolic execution. Figure 2 shows the critical steps of our framework for symbolic execution. The framework divides the symbolic execution procedure into two stages. The first stage generates a solving strategy, which will then be used in the second stage to solve the SMT formulas.

Our key idea is to use the SMT formulas solved in the first stage to synthesize a solving strategy. The synthesis contains three key steps: tactic sequence generation, tactic parameter tuning, and strategy generation.

- **Tactic sequence generation**. This step's inputs are the SMT formulas collected in the first stage of symbolic execution. We randomly select a subset from these SMT formulas to represent the program's path constraints and based on which a solving strategy is synthesized. These selected formulas are divided into *training*, *validation* and *testing* sets. Then, we predicate a tactic sequence for each SMT formula in the training set. This sequence is supposed to solve the SMT formula in a shorter time. We formalize the tactic based solving procedure of an SMT formula as a Markov Decision Process (MDP) (Section 3) and use an *offline trained* deep reinforcement learning (DRL) [28] model to predicate a tactic sequence for an SMT formula (Section 4.2).
- **Tactic parameter tuning**. The tactics in the sequences generated by the first step use only default parameter values. Tactic parameters also influence the performance of solving. After getting a set of better tactic sequences in the first step, we tune the tactics' parameters in this step. The basic idea is to randomly generate the parameter values and keep the better tactic sequences. Here, a better tactic sequence is the one that can solve more formulas in the training set and solve faster. This step introduces overhead because of solving the formulas. To reduce the overhead, we employ the pre-trained deep neural networks (DNNs) [13] to predicate whether the solving will time out (Section 4.3).
- **Strategy generation**. The third step composes the tactic sequences produced in the last step into a solving strategy. The key idea is to select the best tactic at each step of solving an SMT formula. We employ the idea of decision tree [10] to generate the solving strategy. For each step, we greedily select the tactic that needs a smaller solving cost. More specifically, by employing different probes, we select different tactics for different formulas (Section 4.4). In this way, we want to achieve a global optimal solving performance for the SMT formulas in the validation set.

After the first stage, we get a synthesized strategy, which is supposed to make the solver more efficient in the second stage of the program's symbolic execution.

2.3 Motivation Example

Figure 3 shows a motivation program. The program has two double floating-point inputs of a and b. If b is greater than zero, there are

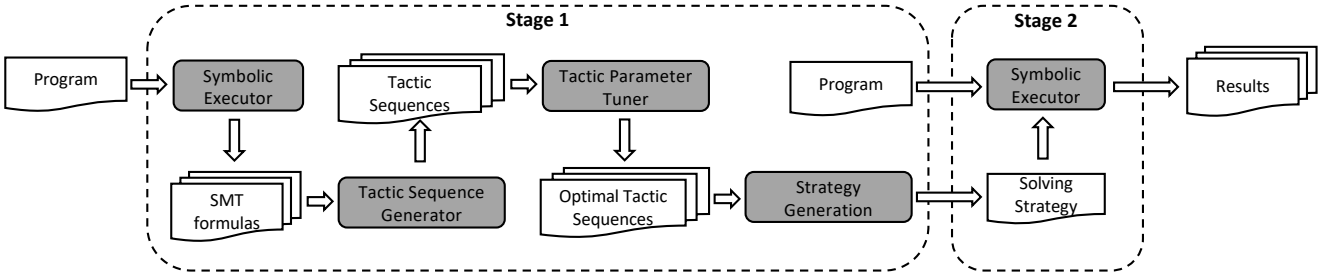

Figure 2: The two-stage procedure of symbolic execution.

```
1  void test(double a, double b) {
2    if (b > 0.0) { // true branch
3      double c = a; double num = 1.0;
4      for (int i = 0; i < 2; i++) {
5        if (c == num){
6          break;
7        }
8        c = c * a;
9        num = num * 2.0;
10     }
11   } else { // false branch
12     double c = a; int i = 0;
13     while (c != 27.0 && i < 2) {
14       c = c * a;
15       i++;
16     }
17   }
18 }
```

Figure 3: A motivation example program

two paths; otherwise, there are four paths in the false branch. If we use KLEE with Z3 as the SMT solver[2], we need **1362 seconds**[3] to explore all the paths in this program.

Suppose that the first stage of the program's symbolic execution explores the paths in the true branch. Then we use the following four formulas for synthesizing the solving strategy, where b and a are bit-vector floating-point variables with a length of 64.

$$\begin{array}{ll} (1) & b > 0 \wedge a = 1.0 \\ (2) & b > 0 \wedge a \neq 1.0 \\ (3) & b > 0 \wedge a \neq 1.0 \wedge a^2 = 2.0 \\ (4) & b > 0 \wedge a \neq 1.0 \wedge a^2 \neq 2.0 \end{array}$$

Suppose we select formulas (2) and (4) as the training set for synthesis. Then, the tactic sequences predicated are as follows, where the DRL model suggests the same sequence to the two formulas.

$$\texttt{simplify} \fatsemi \texttt{bit-blast} \fatsemi \texttt{smt}$$

After the second step, we will get the following two tactic sequences with parameters, where we use $para1 : \texttt{true}$ to denote the parameter whose name and value are $para1$ and `true`, respectively, and the other parameters of each tactic use the default values.

$$\begin{aligned} &\texttt{simplify}(elim\_and : \texttt{true}, hoist\_mul : \texttt{true}, \\ &\qquad local\_ctx : \texttt{true}, push\_ite\_bv : \texttt{true}) \fatsemi \\ &\qquad \texttt{bit-blast} \fatsemi \texttt{smt} \\ &\texttt{simplify}(elim\_and : \texttt{true}, hoist\_mul : \texttt{true}) \fatsemi \\ &\qquad \texttt{bit-blast} \fatsemi \texttt{smt} \end{aligned}$$

Finally, based on these two tactic sequences, we generate the following solving strategy, where the generation algorithm selects the first one as the optimal one.

$$\begin{aligned} &\texttt{simplify}(elim\_and : \texttt{true}, hoist\_mul : \texttt{true}, \\ &\qquad local\_ctx : \texttt{true}, push\_ite\_bv : \texttt{true}) \fatsemi \\ &\qquad \texttt{bit-blast} \fatsemi \texttt{smt} \end{aligned} \tag{3}$$

Then, we will use this solving strategy to explore the false branch's paths. In total, we need **764 seconds** to explore all the paths, in which strategy synthesis needs 3 seconds.

3 MDP FOR SMT SOLVING

We present the formalization for the tactic based SMT solving procedure of an SMT formula in terms of a Markov Decision Process (MDP) [9]. Different choices of solving strategies may produce different results or performances. Hence, we use the MDP with cost to formalize the different tactic choices in the solving procedure.

Definition 3.1. A Markov Decision Process (MDP) with cost is a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \beta_0, \mathcal{T}, \mathcal{S}_f, \mathcal{R}, C)$, where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, β_0 is the initial distribution of $\mathcal{S}$ such that $\sum_{s \in \mathcal{S}} \beta_0(s) = 1$, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{D}$ is the transition function that gives a distribution function $\beta : \mathcal{S} \to [0, 1]$ for a state and an action and β satisfies $\sum_{s \in \mathcal{S}} \beta(s) = 1$, $\mathcal{S}_f$ is the set of final states, $\mathcal{R} : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$ is the reward function, and $C : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the cost function.

MDP provides a general formal framework to specify a probabilistic system. Usually, given an MDP, we want to know the best choices to maximize the reward, *i.e.*, select the best action at a state. We use *policy* to formalize these choices.

Definition 3.2. A policy π for an MPD $\mathcal{M}$ with cost is a function $\pi : \mathcal{S} \to \mathcal{D}_{\mathcal{A}}$, which gives the action distribution for a state, and $\sum_{a \in \mathcal{A}} \mathcal{D}_{\mathcal{A}}(a) = 1$.

A policy gives the distribution of actions for each state. Hence, given an MDP policy, we can transfer the states of the MDP as follows.

[2]We use the version with FP support, and Z3's version is 4.6.2.
[3]The CPU is 2.5GHz.

Definition 3.3. A rollout $\zeta \sim \pi$ is a following tuple sequence in which each tuple is in $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathbb{R} \times \mathbb{R}$,

$$\langle (s_0, a_0, s_1, r_0, c_0), ..., (s_{n-1}, a_{n-1}, s_n, r_{n-1}, c_{n-1}) \rangle$$

and the sequence is randomly constructed as follows.

- Sample an initial state s_0 with respect to β_0.
- Sample an action a_i with respect to $\pi(s_i)$, and sample a transition with respect to $\mathcal{T}(s_i, a_i)$ which leads to state s_{i+1}. The reward r_i is $\mathcal{R}(s_i, s_{i+1})$ and the cost c_i is $C(s_i, a_i)$.
- Keep doing the second step until s_n is in $\mathcal{S}_f$.

A policy gives a distribution of rollouts. The optimal policy is the one adopting which the expected reward is maximized, and the expected cost is the minimized [4]. Hence, given an MDP $\mathcal{M}$ with cost, the optimal policy π^* is defined as follows.

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\zeta \sim \pi} \left[\sum_{i=0}^{n-1} (r_i - c_i) \right] \tag{4}$$

The solving procedure of an SMT formula φ is a special form of MDP with cost. The states are the SMT formulas and solving results. The actions are the tactics. There are the following specialties: first, only one initial state exists; second, the transition of a state with an action is deterministic; third, the rewards of internal states are zero, and the state's reward representing a successful solving is 1. Formally, the MDP with cost for solving an SMT formula is defined as follows.

Definition 3.4. An MDP with cost for solving an SMT formula φ by an SMT solver is a tuple $(Q, \mathbb{T}_p, \beta_\varphi, \mathcal{T}_\varphi, \{\text{SUCC}, \text{FAIL}\}, \mathcal{R}_\varphi, C_\varphi)$, where

- Q is the set containing the possible constraints during solving and two special states SUCC and FAIL.
- $\mathbb{T}_p$ is the set of parameterized tactics.
- β_φ is the initial distribution such that $\beta_\varphi(\varphi) = 1$.
- $\mathcal{T}_\varphi : Q \setminus \{\text{SUCC}, \text{FAIL}\} \times \mathbb{T}_p \to \mathcal{D}$ gives a deterministic transition for a constraint and a tactic, *i.e.*, $\mathcal{T}_\varphi(s, \mathbf{T}(\bar{p}))(s') = 1$ given $\mathbf{T}(\bar{p})(s) = s'$, where $\mathbf{T}(\bar{p}) \in \mathbb{T}_p$.
- SUCC is the final state representing the success of solving (*i.e.*, the solver returns SAT or UNSAT), and FAIL is the final state of a failed solving, *i.e.*, the solver times out.
- $\mathcal{R}_\varphi$ gives 1 to $(s, \text{SUCC}) \in Q \times Q$ and 0 to the others.
- $C_\varphi : Q \setminus \{\text{SUCC}, \text{FAIL}\} \times \mathbb{T}_p \to \mathbb{R}$ gives the cost of applying a tactic to the current constraint. The cost can be the solving time or the CPU cycles for solving.

Figure 4 shows a part of the MDP for solving the bit-vector floating-point formula $x^3 = (5.0 + 3.0)$, where the number under each transition line represents the cost of applying the tactic in terms of CPU cycles. For example, the cost of applying `smt` to $x^3 = 8.0$ is 16278808. The formulas after applying `fp2bv` and `bit-blast` are too large, and the detailed information is omitted for the sake of space.

In principle, adopting different tactics may lead to different costs (or failure). Then, a *successful policy* π_φ for a formula φ is the one whose distribution satisfies the following condition.

$$\Pr_{\zeta \sim \pi_\varphi} \{s_n = \text{SUCC}\} > 0 \tag{5}$$

[4]We do not consider a discount because there are only finite steps in our SMT solving scenario.

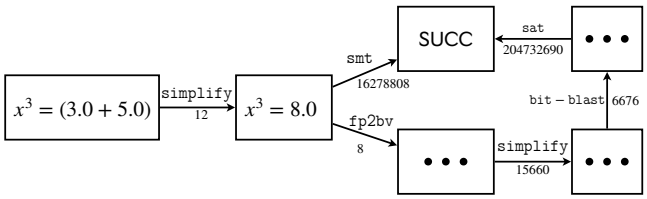

Figure 4: MDP for solving $x^3 = (5.0 + 3.0)$.

It means that it is possible to solve φ by employing the policy π_φ. For example, if we only consider the MDP in Figure 4, $\langle$`simplify`, `smt`$\rangle$ is a successful policy (*i.e.*, $\pi(s_0)$(`simplify`) = 1 and $\pi(s_1)$(`smt`) = 1) with the successful solving's probability 1. We use Π_φ to denote all the successful policies that are deterministic on each state, *i.e.*, only one action can be taken on each state. So, each policy π_φ in Π_φ satisfies $\Pr_{\zeta \sim \pi_\varphi} \{s_n = \text{SUCC}\} = 1$. Then, the optimal policy π^*_φ for φ is defined as follows.

$$\pi^*_\varphi = \arg\max_{\pi \in \Pi_\varphi} \mathbb{E}_{\zeta \sim \pi} \left[1 - \sum_{i=0}^{n-1} c_i \right] \tag{6}$$

For example, Π_φ of the MDP in Figure 4 has two policies, and the optimal policy is $\langle$`simplify`, `smt`$\rangle$, which needs the least cost to solve the formula.

It is natural to employ deep reinforcement learning (DRL) [28] to train a model from existing SMT benchmarks for SMT solving. Then, we can use the model to guide an SMT formula's solving procedure step by step. However, there are two technical challenges: (1) integrating the DRL model into the existing solvers also introduces much overhead, which may doom the model's advantage; (2) the generality problem of the model, which may perform poorly on new formulas. Hence, in this paper, we propose to use a DRL model trained offline to predicate a set of the tactic sequences using default parameter values for the representative SMT formulas of a program (Section 4.2). Then, we synthesize a composed solving strategy online (Section 4.4) to avoid the re-engineering of the solver and the overhead of employing the model during solving. Besides, we tune the parameters online for each program to tackle the generalization problem (Section 4.3).

4 SYMBOLIC EXECUTION WITH STRATEGY SYNTHESIS

This section presents the details of our framework. The first subsection depicts the two-stage symbolic execution framework. Then, the three key steps are explained in the following three sub-sections.

4.1 Symbolic Execution Framework

Algorithm 1 gives the symbolic execution framework in which a solving strategy is synthesized online. The inputs are the program under symbolic execution, the two search strategies S_1 and S_2 that will be used during the two stages of the symbolic execution, and the set of the probes used in strategy synthesis. Our framework adopts a state-based symbolic execution [16] and employs a worklist-based implementation. In the beginning, there is only initial state s_i in the worklist (Line 2). We use $\mathcal{G}$ to record the SMT formulas generated by

Algorithm 1: Symbolic Execution With Strategy Synthesis

SE($\mathcal{P}, S_1, S_2, Probs$)
Data: $\mathcal{P}$ is a program, S_1 and S_2 are the search strategies in the first and second stages, respectively, and $Probs$ is a set of probes for strategy synthesis.

```
1  begin
2      worklist, stage ← {s_i}, 0
3      G, S ← ∅, DS
4      while worklist ≠ ∅ do
5          if stage = 0 then
6              s ← Select(worklist, S_1)
7          end
8          else
9              s ← Select(worklist, S_2)
10         end
11         Q ← Execute(s, S)
12         G ← G ∪ Q
13         if First stage ends then
14             S ← Synthesize(G, Probs)
15             stage ← 1
16         end
17     end
18 end
```

Algorithm 2: Strategy Synthesis

Synthesize($\mathcal{G}, Probs$)
Data: $\mathcal{G}$ is a set of SMT formulas, and $Probs$ is a set of probes.

```
1  begin
2      (S_t, S_v, S_test) ← RandomSelect(G)
3      TS_o ← ∅
4      for each φ ∈ S_t do
5          TS_o ← TS_o ∪ {ChooseTS(φ)}
6      end
7      TS_s ← ParTuning(S_t, TS_o)
8      V_p ← ∅
9      for each φ ∈ S_t do
10         V_p ← V_p ⊎ ⨄_{ts∈TS_s} CPV(φ, ts, Probs)
11     end
12     C ← GenPredicates(V_p, Probs)
13     S ← GenStrategy(S_v, TS_s, C)
14     if Solve(S_test, S) > Solve(S_test, DS) then
15         S ← DS
16     end
17     return S
18 end
```

symbolic execution. $\mathcal{S}$ is the solving strategy used by the underlying solver. In the beginning, $\mathcal{S}$ is the default solving strategy (denoted as $\mathcal{DS}$).

When exploring the state space in the first stage, the symbolic executor uses the search strategy S_1 to select a state from the worklist to explore the state space and collect the SMT formulas (Line 6); otherwise, S_2 is employed (Line 9). The details of each statement's symbolic execution are traditional [16] and omitted for the sake of space. The symbolic execution of a statement employs the solving strategy $\mathcal{S}$ for SMT solving and returns the set of generated SMT formulas during symbolic execution (Line 11). The end of the first stage is also parametric, *e.g.*, the number of the explored paths reaches a threshold, or the time of the first stage's symbolic execution is up. If the first stage ends, we synthesize a solving strategy (Algorithm 2) and replace the default solving strategy for the later symbolic execution (Lines 14&15).

Algorithm 2 gives the strategy synthesis procedure. The inputs are the set of SMT formulas collected during the first stage of Algorithm 1 and the probes for strategy generation. The output is the synthesized solving strategy. The algorithm mainly contains the three steps introduced in Section 2.2. First, we *randomly* select three subsets S_t, S_v and S_{test} from the input set of formulas. S_t is used to generate tactic sequences and probe-based conditions. S_v is used to generate a composed solving strategy, and S_{test} is used to compare the synthesized strategy and the default strategy. Then, we predicate a tactic sequence for each formula in S_t by an offline trained DRL model (ChooseTS at Line 5), whose details of the design and training will be given in Section 4.2. After getting the tactic sequences in TS_o, we tune the parameters of the tactics in each sequence and get a better sequence subset TS_s (ParTuning at Line 7, Algorithm 3). Finally, we will use the tactic sequences in TS_s to generate a composite solving strategy $\mathcal{S}$ that performs better on the validation set S_v.

The key idea of generating the composite solving strategy is to separate the validation formulas into different groups and select a best candidate solving strategy for each group. The predicates for separating validation formulas are constructed by probes and the probe values collected by solving the formulas of the training set S_t. CPV($\varphi, ts, Probs$) at Line 10 contains the probe values during the procedure of using ts to solve φ. CPV($\varphi, ts, Probs$) is $\emptyset$ when ts is empty sequence; otherwise, CPV($\varphi, \langle t_0 \rangle \hat{} ts, Probs$) is defined as follows.

$$\{\mathbf{P} \mapsto \{\mathbf{P}(\varphi)\} \mid \mathbf{P} \in Probs\} \cup \text{CPV}(t_0(\varphi), ts, Probs) \tag{7}$$

Besides, we use $M_1 \uplus M_2$ to denote the merging of two maps whose values are sets, which is defined as follows, where $k \hat{\in} M$ represents that k is defined in map M.

$$M_1 \uplus M_2[k] := \begin{cases} M_1[k] \cup M_2[k] & k \hat{\in} M_1 \wedge k \hat{\in} M_2 \\ M_1[k] & k \hat{\in} M_1 \\ M_2[k] & k \hat{\in} M_2 \end{cases} \tag{8}$$

Based on the collected values of the probes, we select the representative values to construct the predicate set C (Line 12). Then, we construct the composite solving strategy based on the predicates in C and the candidate tactic sequences in TS_s with respect to the validation set S_v (Line 13, Algorithm 4). Finally, we compare the synthesized strategy with the default strategy $\mathcal{DS}$ with respect to the testing set S_{test} (Line 14), where Solve($V, \mathcal{S}$) represents the cost of solving the SMT formulas in V by employing the solving strategy $\mathcal{S}$. We use the default strategy if the synthesized strategy is not better.

Algorithm 3: Parameter Tuning

ParTuning(Q, TS)
Data: Q is the set of training SMT formulas, TS is a set of tactic sequences.
1 **begin**
2 $TS_p \leftarrow \emptyset$
3 **for** *each* $ts \in TS$ **do**
4 $TS_p \leftarrow TS_p \cup \{\text{ParaGenerate}(ts)\}$
5 **end**
6 $TS_p \leftarrow TS_p \cup TS$
7 **for** *each* $ts \in TS_p$ **do**
8 $M[ts] \leftarrow |\{\varphi \mid \varphi \in Q \wedge \text{Predicate}(\varphi, ts)\}|$
9 **end**
10 $TS_{N_1} \leftarrow \text{Top}(M, N_1)$
11 **for** *each* $ts \in TS_{N_1}$ **do**
12 $M_1[ts] \leftarrow \text{Solve}(Q, ts)$
13 **end**
14 **return** $\text{Top}(M_1, N_2)$
15 **end**

4.2 Tactic Sequence Generation

We employ an offline trained DRL model to predicate a tactic sequence for an SMT formula. The ReLU DNN for Q-learning contains five layers. To train the model, we generate the training data from the existing SMT formulas. Each element in the training data is a tuple $(\mathcal{E}(\varphi), \mathcal{E}(T_s), t, p)$ consisting of four parts: $\mathcal{E}(\varphi)$ is the embedding of the current formula φ; $\mathcal{E}(T_s)$ is the embedding of the tactic sequence applied until now to generate φ; t is the following tactic to apply; and p is the probability of applying the tactic. We can generate many training elements from an SMT formula. The generation greedily searches the tactic sequences with small solving costs. During the search, we record each step's choice as a training element and calculate the probability *w.r.t.* the consumed resources for applying the tactic. More resources imply a smaller probability. This greedy search and the probability calculation make the model predicate a tactic sequence that tends to have a smaller solving cost.

4.3 Tactic Parameter Tuning

Algorithm 3 shows the details of tuning the tactic parameters in the candidate tactic sequences. The inputs are the set of the training SMT formulas and the set of candidate tactic sequences predicated by the DRL model. The basic idea is to randomly generate the values for the parameters and keep the better tactic sequences. We use ParaGenerate (Line 4) to generate a tactic sequence with random parameter values for each candidate tactic sequence in TS. Then, together with the ones in TS (Line 6), we evaluate the effectiveness and efficiency of the tactic sequences in TS_p. In principle, the evaluation needs to solve the formulas in Q by employing each tactic sequence in T_p, which introduces a large overhead. To reduce this overhead, we use a pre-trained deep neural network (DNN) model to predicate whether the solving of a formula by a tactic sequence will time out (Line 8), where Predicate predicates whether the formula φ can be solved by employing the tactic sequence ts. Then, we select the top N_1 tactic sequences with respect to its value in M.

Algorithm 4: Strategy Generation

GenStrategy(S_v, TS, C)
Data: S_v is the set of validation formulas, TS is a set of tactic sequences, and C is a set of predicates.
1 **begin**
2 **if** $|S_v| < K$ **then**
3 **return** $\arg\min_{ts \in TS} \text{Solve}(S_v, ts)$
4 **end**
5 $ts_c \leftarrow \arg\max_{ts \in \{t \mid \forall ts \in TS \bullet t \preceq ts\}} \text{length}(ts)$
6 $S'_v \leftarrow \{ts_c(\varphi) \mid \varphi \in S_v\}$
7 $TS' \leftarrow \{ts_t \mid ts_c \hat{} ts_t \in TS\}$
8 $C_{min} \leftarrow \arg\min_{c \in C} \text{Cost}(c, S'_v, TS')$
9 $F_s \leftarrow \{t_1 \mid \langle t_1, ..., t_n \rangle \in TS'\}$
10 $t^1_m \leftarrow \arg\min_{t \in F_s} \sum_{q \in S'_v \downarrow C_{min}} \min\{\text{Solve}(\{q\}, ts) \mid ts \in TS' \downarrow t\}$
11 $t^2_m \leftarrow \arg\min_{t \in F_s} \sum_{q \in S'_v \downarrow \neg C_{min}} \min\{\text{Solve}(\{q\}, ts) \mid ts \in TS' \downarrow t\}$
12 $\mathcal{S}_1 \leftarrow \text{GenStrategy}(S'_v \downarrow C_{min}, TS' \downarrow t^1_m, C)$
13 $\mathcal{S}_2 \leftarrow \text{GenStrategy}(S'_v \downarrow \neg C_{min}, TS' \downarrow t^2_m, C)$
14 **return** $ts_c \fatsemi \text{ITE}(C_{min}, \mathcal{S}_1, \mathcal{S}_2)$
15 **end**

For the top N_1 tactic sequences in TS_{N_1}, we select the top N_2 ones by using the solver to solve the formulas in Q (Lines 10-14).

Given an SMT formula φ and a tactic sequence ts, the timeout predication is carried out as follows, where ts_1 is a non-empty sequence.

$$\text{Predicate}(\varphi, ts) := \begin{cases} \text{Predicate}(t(\varphi), ts_1) & ts = \langle t \rangle \hat{} ts_1 \wedge t(s) \in \Gamma \\ true & ts = \langle t \rangle \hat{} ts_1 \wedge t(s) \in \Theta \\ \text{NNPredciate}_t(\varphi) & ts = \langle t \rangle \end{cases} \quad (9)$$

Here, if a formula can be solved before the final tactic, the predication result is *true*; otherwise, we apply the tactics before the final one to the formula and predicate the result by the DNN for the formula before the last step ($\text{NNPredciate}_t(\varphi)$). To improve the effectiveness, we trained a DNN specifically for each tactic in charge of final solving, *e.g.*, `sat` and `smt`. The ReLU DNN for `smt` contains nine layers, and the other ReLU DNNs contain seven layers.

4.4 Strategy Generation

Algorithm 4 shows the details of generating the composite solving strategy. The inputs are the set S_v of validation formulas, the set TS of candidate tactic sequences, and the predicates for grouping the validation formulas. The algorithm uses the idea of decision tree [10] to generate a composite solving strategy, which selects the best candidate in TS for a subset of S_v. At the beginning, we get the longest common prefix ts_c of all the tactic sequences (if it exists), where $t \preceq ts$ represents that t is a prefix of ts. Then, there is a difference between the tactic sequences after ts_c, and the algorithm selects the tactic greedily with respect to the cost of solving the formulas in S_v after applying ts_c (S'_v at Line 6). The best predicate is the one using which to separate the formulas will have a least solving cost, and $\text{Cost}(c, S'_v, TS')$ (Line 8) is defined as

follows [3, 10], where $S \downarrow c$ represents the set of S's formulas that satisfy the predicate c.

$$(|S'_v \downarrow c|/|S'_v|)\times\mathcal{H}_{TS}(S'_v \downarrow c)+(|S'_v \downarrow \neg c|/|S'_v|)\times\mathcal{H}_{TS}(S'_v \downarrow \neg c) \quad (10)$$

$\mathcal{H}_{TS}(S_v)$ represents the entropy of S_v with respect to the tactic sequences in TS and is defined as follows, where $R(S_v, ts)$ is the ratio of the solved formulas in S_v using the tactic sequence ts.

$$-\sum_{ts\in TS} R(S_v, ts)\times\log(R(S_v, ts))+(1-R(S_v, ts))\times\log(1-R(S_v, ts)) \quad (11)$$

If $\mathcal{H}_{TS}(S_v)$ is less, it means that it is more possible to divide S_v into different groups, and each group's formulas can be solved by a tactic sequence in TS. Hence, in principle, if $\text{Cost}(c, S'_v, TS')$ is less, it is more possible to solve all the formulas in S'_v when dividing the formulas by c.

Then, the algorithm greedily select the best next tactic for the two groups (*i.e.*, $S'_v \downarrow C_{min}$ and $S'_v \downarrow \neg C_{min}$) from dividing S'_v by C_{min}. Here, the best tactic t_m is the one using the tactic sequences starting from which has the least solving cost when solving the formulas in $S'_v \downarrow C_{min}$ or $S'_v \downarrow \neg C_{min}$ (Lines 10&11), where $TS \downarrow t$ represents the sequence subset of TS whose element starts with t, *i.e.*, $\{\langle t\rangle\hat{}ts \mid \langle t\rangle\hat{}ts \in TS\}$. Then, we recursively generate the decision tree by generating the best strategy for the two groups under the tactic sequences starting with the best tactics (Lines 12&13). Finally, we compose the two groups' strategies by the **ITE** composition and return the synthesized strategy.

The strategy generation needs to balance the effectiveness and generation overhead. In principle, we can have a strategy that can recommend the best tactic sequence for each SMT formula in the validation set. However, the generation introduces more overhead, and the strategy may also carry out more decisions. This balance is controlled by a threshold of K of the validation set (Line 2). If S_v's size is less than K, the algorithm directly selects the best tactic sequence.

5 EVALUATION

We have implemented our method on KLEE[5] [5] (*i.e*, a *state-of-the-art* symbolic execution engine for C programs) and an SPF-based concolic execution engine [31] for Java programs. Both engines use Z3[6] as the backend solver and bit-vector SMT theory for encoding the path constraints. We train the DRL model and the DNN models by Pytorch. The synthesis procedure is implemented in Python 3.6.

We have conducted extensive experiments to answer the following two research questions:

- **RQ1**: *effectiveness*, How effective is our solving strategy synthesis method? Here, effectiveness means solving more queries (*i.e.*, SMT formulas) and exploring more paths during symbolic execution.
- **RQ2**: *generalization ability*, How general is our synthesis method when applied to the symbolic execution of other kinds of programs?

5.1 Experimental Setup

To evaluate the effectiveness of our method, we use Coreutils as the benchmark. Coreutils is the mainstream benchmark for the symbolic execution researches whose implementations are based on KLEE. The used Coreutils's version is 6.11. There are 89 programs in total. We use 80 programs (87159 SLOCs in total). We filter the remaining 9 programs because the errors happened in the symbolic execution or the time of symbolic execution is less than 1 minute.

We train the first step's DRL model and the second step's DNNs used in the first stage as follows.

- For the DRL model, we randomly selected 14 programs from the 80 Coreutils programs. Then, we carried out symbolic execution for these 14 program and collected the SMT formulas generated during symbolic execution. We randomly selected 300 from the formulas of each program and created a dataset consisting of 4200 formulas for training the DRL model. We generated the dataset for training the DRL model by greedily search the strategy space and record each tactic applying step's formula and cost.
- We trained four DNNs of predicating timeout for `sat`, `smt`, `qfnra-nlsat` and `qfnra`, respectively. These four tactics are the final solving steps. Besides the formulas from the randomly selected 8 Coreutils programs, we also use the `qf_bv`, `qf_abv`, `qf_abvfp`, `qf_bvfp` SMT-LIB2 benchmarks [26] for generating the dataset. We randomly generated a set of tactic sequences that end with any of these four tactics. We applied the tactic sequences to the SMT formulas. We collected the formulas before applying the last tactic and the results after applying the tactic to generate the datasets for the timeout predication DNNs. The timeout threshold is set to 30 seconds.

We use the bag of words (BOW) model [35] and the one-hot encoding [14] as the embedding of the SMT formulas and the solving strategies, respectively.

We analyze each Coreutils program in 1 hour. In both stages, we use BFS as the search strategy. We set the end condition of the first stage as reaching 100 seconds. RandomSelect in Algorithm 2 selects 20, 30, and 30 formulas from the formula set generated in the first stage for training, validation, and testing datasets, respectively. In Algorithm 3, N_1 and N_2 are set to 10 and 5, respectively. The K in the strategy generation algorithm is set to 10. The baseline method is vanilla KLEE using the BFS search strategy.

To evaluate the generalization ability, we use our method to analyze Java programs. Note that we directly use the model trained for Coreutils C programs during the first and second steps of the strategy synthesis when analyzing Java programs. Table 1 shows the Java programs in evaluation. All the programs are open-source Java programs. Most programs are the parsing programs of different grammars, including Java, Json, XML, *etc.*

We create a driver for each program and provide an initial input to the program's main interface. The input can be a string or a file. We symbolize each byte in the input to do symbolic execution. Each program is analyzed in 15 minutes. The parameters of strategy synthesis are the same as those for analyzing Coreutils programs. The first stage ends when exploring 100 paths. Both of the search strategies of the first and the second stages are BFS. The baseline method is the original concolic execution using BFS.

All the experiments were carried out on a Server with 64G memory and 16 2.5GHz cores. The operating system is Ubuntu 14.04. To

[5]KLEE's version is 2.1-pre.
[6]Z3's version is 4.6.2.

Table 1: The Java benchmark programs.

Subject	SLOC	Brief Description
txtmark	4255	A Markdown parser
barcode	3793	A library for QR code recognition
javaparser	22286	A Java code parser
html5parser	14266	A complete HTML5 parser
jsqlparser	30250	A SQL statement parser
actson	623	A Json parser
nanoxml	1429	An XML parser
rhino	20042	A Speech-to-Intent engine
htmlparser	22231	A Java HTML parser
toba	6029	A Java bytecode to C compiler
jericho	9542	A Java HTML parser
minimal-json	1859	A Json parser
xml	3367	An XML parser
pobs	3024	A Java object parser
Antlr	27118	A parser generator
fastjson-dev	19329	Alibaba Json parser
jmp123	3273	A MP3 decoder
nanojson	2185	Nano Json parser
foxykeep	3865	A Java code generator
jsoniter	13005	A Json parser
univocity	18263	A Java code parser
fastcsv	807	A CSV parser
argo	2687	A Jdom parser
htmlcleaner	8328	A Java HTML parser
jsoup	12512	A Java HTML parser
url-detector	1705	A URL parser
jcsv	1039	A CSV parser
commons-csv	1452	A CSV parser
super-csv	3734	A CSV parser
markdown4j	3740	A markdown parser
simple-csv	1583	A simple CSV parser
jaad	35984	An AAC decoder and MP4 demultiplexer library
jtidy	18937	A HTML cleaner
commonmark	4964	A markdown parser
Total	327506	**34 open source Java programs**

alleviate experimental errors and randomness, we carried out each task three times. We use the average value of the two closed values in the three values generated by the three runs.

5.2 Experimental Results

5.2.1 Effectiveness. We use the numbers of the paths explored by KLEE and the queries (*i.e.*, SMT formulas) solved during symbolic execution as the main factors to evaluate the effectiveness. Figure 5 shows the results, and the first x-value whose y-value is larger than zero is 18. The X-axis displays the programs ordered by the values. The value of relative increase is calculated as follows, where N_{opt} represents the number of paths or queries after employing our method, and $N_{baseline}$ represents the number of original symbolic execution.

$$\frac{N_{opt} - N_{baseline}}{N_{baseline}} \tag{12}$$

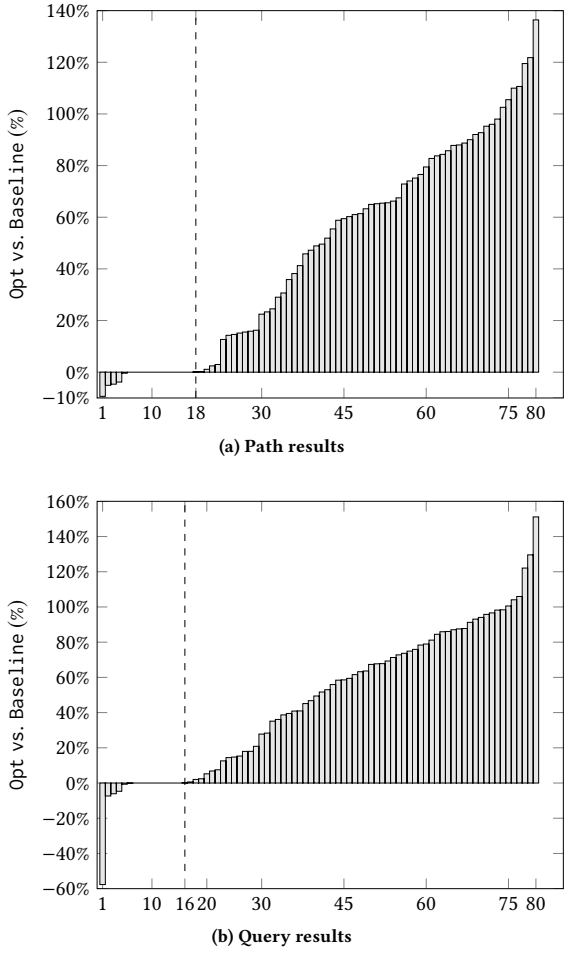

Figure 5: Results of Coreutils.

As shown by Figure 5a, our method can improve the number of explored paths for 63 (78%) programs. On the other hand, there are 5 (6%) programs on which we decrease the number of paths , −5.33% (−9.37%∼−0.4%) on average. Our method can, on average, improves the number of explored paths by 66.11% (−9.37%∼136.41%). These results indicate that our method can improve the effectiveness of symbolic execution.

Besides, as shown in Figure 5b (the first x-value whose y-value is larger than zero is 16), the queries solved during symbolic execution are also increased. Our method can increase the queries for 65 (81%) programs. The relative increase of queries is on average 58.76% (−57.66%∼151.15%). These results also demonstrate the effectiveness of our method. Besides, the results also indicate that there is a correlation between the numbers of queries and paths, which is natural for symbolic execution, *i.e.* more queries often mean more paths. The average synthesis time is 64*s* (50*s*∼120*s*) which indicates that the synthesis is efficient.

Note that the programs represented by the same X-axis value in the two figures may not be the same program. However, the set of the first five programs in Figure 5a, *i.e.* the ones whose path numbers are decreases, is the same as that of Figure 5b. The first program in Figure 5b is `shred` whose symbolic execution is solving intensive. Our method decreases the number of formulas by 53.6%. The reason is that the SMT formulas collected in the first stage are not representative. We collected the SMT formulas generated by one hour's symbolic execution of `shred`. Solving the formulas by the synthesized solving strategy is much slower (about 4x) than solving using the default strategy of Z3. However, solving the formulas generated in `shred`'s first stage by the synthesized strategy is better than that using Z3's default strategy.

Answer to RQ1: *our method is effective to improve symbolic execution's ability of path exploration. On average, our method increases the numbers of paths and queries by* 66.11% *and* 58.76%*, respectively.*

5.2.2 Generalization Ability. We applied our method to Java concolic execution to analyze Java programs for validating the generalization ability. Table 2 shows the detailed experimental results.

Figure 6a shows the results of the relative increase of paths. Our method improves the paths for 24 (70%) programs. On average, the increasing rate is 102.6% (−23.36%∼262.77%). Figure 6b shows the results of the relative increase of total queries. Our method improves the total queries for 24 (70%) programs. On average, the increasing rate is 100.24% (−20.29%∼284.35%). These results indicate that our method has a good generalization ability and can improve Java symbolic executor's ability. The average synthesis time is 48*s* (32*s*∼80*s*).

Figure 7 shows the trend of the solving queries in the Java benchmark programs. The X-axis shows the analysis time in seconds. The Y-axis shows the total number of the solved queries in all the programs. As shown by the figure, our method outperforms the baseline method by consistently solving more queries. Suppose we set the task to be solving the queries that the baseline method generates. In that case, our method uses 435s to solve 607379 queries (*i.e.*, the total number of the queries solved by the baseline method), which indicates a 2.07x speedup. Besides, as shown in the figure, the baseline method performs better than our method at the beginning, *i.e.*, before 100 seconds, because our method is synthesizing the solving strategy after exploring 100 paths. After synthesis, our method performs better consistently. In addition, there is one program (*i.e.*, `actson`) on which our method finishes the exploration of the whole path spaces in 7 minutes; whereas, the baseline method does not in 15 minutes.

Answer to RQ2: *our method has a good generalization ability. On average, our method increases the number of queries for Java programs by* 100.24% *and achieves a 2.07x speedup for solving the same amount of queries.*

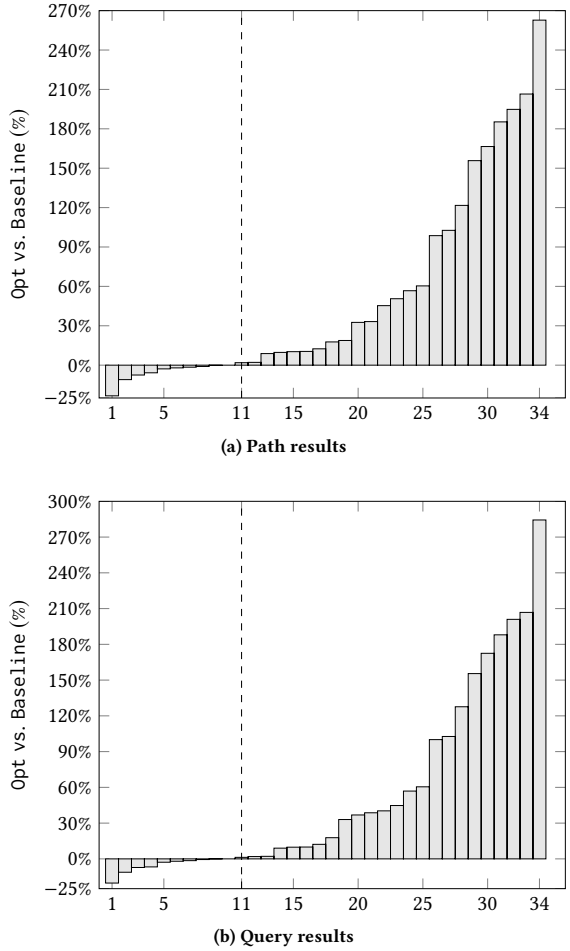

Figure 6: Results of Java programs.

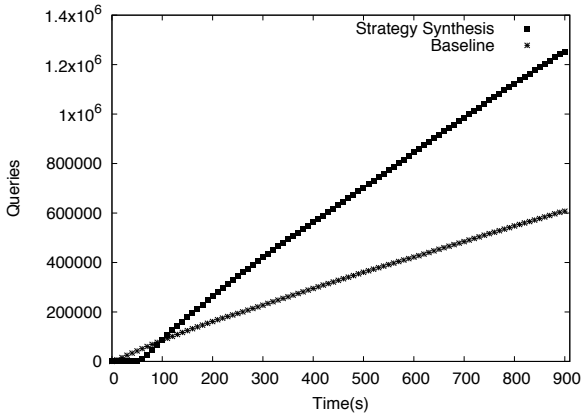

Figure 7: Trend of the solved queries in Java programs.

Table 2: Detailed Results of Java programs. The columns #SAT and #UNSAT show the numbers of solved satisfiable and unsatisfiable queries, respectively. The column #Total shows the total number, *i.e.*, #SAT + #UNSAT. The columns of our method (Strategy Synthesis) also show the relative increasing rate with respect to that of the baseline. The column #Time shows the time for strategy synthesis in seconds. Note that the column #SAT show the numbers of the paths explored by the symbolic executor.

Program	Baseline			Strategy Synthesis			
	#SAT	#UNSAT	#Total	#SAT (Inc%)	#UNSAT (Inc%)	#Total (Inc%)	Time(s)
commons-csv	20372	18310	38682	73903 (262.77%)	74772 (308.37%)	148675 (284.35%)	54
super-csv	33688	1088	34776	103252 (206.49%)	3440 (216.18%)	106692 (206.80%)	52
nanoxml	29829	9014	38843	87940 (194.81%)	28966 (221.34%)	116906 (200.97%)	46
argo	21285	3258	24543	60721 (185.28%)	9957 (205.62%)	70678 (187.98%)	52
fastcsv	29740	1595	31335	79260 (166.51%)	6123 (283.89%)	85383 (172.48%)	51
simple-csv	42237	673	42910	108027 (155.76%)	1590 (136.26%)	109617 (155.46%)	38
univocity	20841	11856	32697	46200 (121.68%)	28227 (138.08%)	74427 (127.63%)	48
jsoniter	30101	0	30101	61006 (102.67%)	10 (1000.00%)	61016 (102.70%)	47
markdown4j	22278	8552	30830	44252 (98.64%)	17427 (103.78%)	61679 (100.06%)	49
jcsv	22051	188	22239	35364 (60.37%)	321 (70.74%)	35685 (60.46%)	55
pobs	13138	11650	24788	20586 (56.69%)	18303 (57.11%)	38889 (56.89%)	63
commonmark	23780	1410	25190	34559 (45.33%)	1905 (35.11%)	36464 (44.76%)	40
jtidy	488	279	767	647 (32.58%)	429 (53.76%)	1076 (40.29%)	59
Antlr	12452	11130	23582	18753 (50.60%)	13949 (25.33%)	32702 (38.67%)	42
actson	3446	41452	44898	4094 (18.80%)	57344 (38.34%)	61438 (36.84%)	36
xml	17345	2856	20201	23102 (33.19%)	3768 (31.93%)	26870 (33.01%)	41
txtmark	12421	3389	15810	14619 (17.70%)	4001 (18.06%)	18620 (17.77%)	48
jericho	8106	3205	11311	8955 (10.47%)	3736 (16.57%)	12691 (12.20%)	51
foxykeep	1177	605	1782	1298 (10.28%)	661 (9.26%)	1959 (9.93%)	39
htmlparser	6623	3474	10097	7267 (9.72%)	3821 (9.99%)	11088 (9.81%)	52
minimal-json	19911	8038	27949	21668 (8.82%)	8796 (9.43%)	30464 (9.00%)	45
jmp123	16122	606	16728	16465 (2.13%)	619 (2.15%)	17084 (2.13%)	49
url-detector	4679	1689	6368	4767 (1.88%)	1725 (2.13%)	6492 (1.95%)	49
rhino	12395	3802	16197	13938 (12.45%)	2467 (-35.11%)	16405 (1.28%)	45
toba	2945	284	3229	2945 (0.00%)	284 (0.00%)	3229 (0.00%)	54
javaparser	2877	1105	3982	2876 (-0.03%)	1105 (0.00%)	3981 (-0.03%)	48
jsqlparser	1910	894	2804	1891 (-0.99%)	897 (0.34%)	2788 (-0.57%)	49
jaad	4196	167	4363	4131 (-1.55%)	167 (0.00%)	4298 (-1.49%)	56
nanojson	5949	792	6741	5826 (-2.07%)	780 (-1.52%)	6606 (-2.00%)	39
html5parser	1306	7	1313	1269 (-2.83%)	7 (0.00%)	1276 (-2.82%)	32
htmlcleaner	275	155	430	259 (-5.82%)	142 (-8.39%)	401 (-6.74%)	44
jsoup	466	87	553	431 (-7.51%)	82 (-5.75%)	513 (-7.23%)	48
fastjson-dev	7336	1315	8651	6528 (-11.01%)	1152 (-12.40%)	7680 (-11.22%)	48
barcode	1216	1238	2454	932 (-23.36%)	1024 (-17.29%)	1956 (-20.29%)	80

5.3 Threats to Validity

Our experimental results are mainly threatened by external validity. The number and types of the benchmark programs may be limited. We plan to evaluate our two prototypes on more benchmarks in the next step. Besides, our method is also threatened by the generalization ability of the machine learning models. Although we have used the models to carry out the experiments on a different engine and the programs of a different language, the models may perform poorly on new benchmarks. We plan to use more SMT benchmarks to offline train our models to improve their effectiveness and generalization ability further.

6 RELATED WORK

As far as we know, we are the first to synthesize a program-specific solving strategy under the background of symbolic execution. Our work is related to the following research topics: the optimization of constraint solving in symbolic execution, optimizing solving strategy of SMT solving, *etc.* Next, we review the related work and compare our work with them.

Constraint solving is one of the main challenges of symbolic execution. The advancements in SAT/SMT constraint solving often improves the efficiency of symbolic execution. Now, existing work of optimizing constraint solving under the background of symbolic execution usually does the job before invoking the solver and uses the solver in a black-box manner [5, 15, 30]. KLEE [5] optimizing

the constraint solving as follows: caching the counter-examples to disprove the later constraints, rewriting the constraint into the simpler one by folding constants and simplifying the expression, and separating constraints into independent groups for better reusing. Green [30] proposes to reuse the results of constraint solving across different programs and analysis tasks and provides a canonical constraint representation for better reusing. Jia *et al.* [15] extends Green to support logic implication relation, which improves the reusing further. Speculative symbolic execution (SSE) [34] reduces the number of solver invocations by speculatively executing the program and ignoring the path feasibility. If the speculation succeeds, many times of constraint solving can be saved. KLEE-Array [21] proposes transforming the array constraints into non-array constraints with respect to the array content to simplify the array constraint solving in symbolic execution. Compared with these approaches, our work uses the solver in a white-box and customizes the solver for the program by synthesizing a solving strategy online.

On the other hand, there is also work of using the solver in a white-box manner. Multiplex symbolic execution (MuSE) [33] uses the underlying solver in a white-box manner by collecting partial solutions during solving. Then, MuSE generates multiple program inputs by solving once. Liu *et al.* [18] study the results of employing stack-based or cache-based incremental solving supported by state-of-the-art solvers to improve the efficiency of symbolic execution, and the results indicate that the stack-based one is more effective. Besides, SSE [34] also uses UNSAT core [17] to improve the efficiency of backtracking. Our work falls into the line of these approaches and is complementary to them.

There exists a few existing work for optimizing solving strategies for SMT solvers. In [22], the authors propose to mutate the default solving strategy to search for the optimal strategy for a set of SMT formulas. FastSMT [3] employs DNN to learn the optimal solving strategy for SMT benchmarks and inspires our work. However, our work targets the online synthesis of a solving strategy for symbolic execution. There exits work of applying machine learning techniques in other parts of constraint solving. NeuroSAT [24] is a message-passing neural network trained for predicting the satisfiability of SAT formulas. NeuroCore [23] also trains a neural network to predicate whether a variable will appear in the unsat core [17]. Song *et al.* [27] propose using a learning-based approach to predicate the partition in solving integer linear programming (ILP) problems, which achieves good performance result compared with a commercial ILP solver. NLocalSAT [32] proposes using a neural network to guide the initialization of assignments in stochastic local search-based SAT solving. The application of these approaches under symbolic execution is interesting and left to be the future work.

7 CONCLUSION

Constraint solving challenges symbolic execution. In this paper, we propose to online synthesize a solving strategy for the program under symbolic execution. We propose a two-stage procedure for symbolic execution, and the synthesis is carried out based on the SMT formulas in the first stage. Our approach leverages offline trained machine learning models during the synthesis to predicate the tactic sequences and reduce the synthesis overhead. We have implemented our approach on mainstream symbolic executors and carried out extensive experiments on the symbolic execution of C and Java programs. The experimental results indicate the effectiveness and generalization ability of our approach.

The future work lies in the following aspects: 1) more extensive experiments on other benchmark programs and symbolic execution engines; 2) online adjustment of the deep learning models in our method to improve the precision and effectiveness; 3) applying the idea in different scenarios of constraint solving-based tasks.

ACKNOWLEDGEMENTS

This research was supported by National Key R&D Program of China (No. 2017YFB1001802) and NSFC Program (No. 61632015, 62002107, 62032024 and 61690203).

REFERENCES

[1] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing symbolic execution with veritesting. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 1083–1094. https://doi.org/10.1145/2568225.2568293

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3 (2018), 50:1–50:39. https://doi.org/10.1145/3182657

[3] Mislav Balunovic, Pavol Bielik, and Martin T. Vechev. [n.d.]. Learning to Solve SMT Formulas. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 10338–10349.

[4] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Vol. 6806. Springer, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

[5] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. [n.d.]. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 209–224.

[6] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90. https://doi.org/10.1145/2408776.2408795

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[8] Leonardo Mendonça de Moura and Grant Olney Passmore. 2013. The Strategy Challenge in SMT Solving. In *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune (Lecture Notes in Computer Science)*, Maria Paola Bonacina and Mark E. Stickel (Eds.), Vol. 7788. Springer, 15–44. https://doi.org/10.1007/978-3-642-36675-8_2

[9] Eugene A. Feinberg and Adam Shwartz. 2002. *Handbook of Markov Decision Processes: Methods and Applications* (second ed.). Springer US.

[10] Michele Fratello and Roberto Tagliaferri. 2019. Decision Trees and Random Forests. Elsevier, 374–383.

[11] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. 213–223. https://doi.org/10.1007/978-3-642-19237-1_4

[12] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40–44. https://doi.org/10.1145/2093548.2093564

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[14] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. 2016. *Deep Learning*. MIT Press.

[15] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. 2015. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proceedings of the 2015 International*

Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015, Michal Young and Tao Xie (Eds.). ACM, 177–187. https://doi.org/10.1145/2771783.2771806

[16] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[17] Daniel Kroening and Ofer Strichman. 2008. *Decision Procedures: An Algorithmic Point of View*. https://doi.org/10.1007/978-3-540-74105-3

[18] Tianhai Liu, Mateus Araújo, Marcelo d'Amorim, and Mana Taghdiri. 2014. A Comparative Study of Incremental Constraint Solving Approaches in Symbolic Execution. In *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings*. 284–299. https://doi.org/10.1007/978-3-319-13338-6_21

[19] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 691–701. https://doi.org/10.1145/2884781.2884807

[20] Corina S. Pasareanu and Neha Rungta. 2010. Symbolic PathFinder: symbolic execution of Java bytecode. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto (Eds.). ACM, 179–180. https://doi.org/10.1145/1858996.1859035

[21] David Mitchel Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. 2017. Accelerating array constraints in symbolic execution. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, Tevfik Bultan and Koushik Sen (Eds.). ACM, 68–78. https://doi.org/10.1145/3092703.3092728

[22] Nicolás Gálvez Ramírez, Youssef Hamadi, Eric Monfroy, and Frédéric Saubion. 2016. Evolving SMT Strategies. In *28th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2016, San Jose, CA, USA, November 6-8, 2016*. IEEE Computer Society, 247–254. https://doi.org/10.1109/ICTAI.2016.0046

[23] Daniel Selsam and Nikolaj Bjørner. 2019. Guiding High-Performance SAT Solvers with Unsat-Core Predictions. In *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings (Lecture Notes in Computer Science)*, Mikolás Janota and Inês Lynce (Eds.), Vol. 11628. Springer, 336–353. https://doi.org/10.1007/978-3-030-24258-9_24

[24] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. [n.d.]. Learning a SAT Solver from Single-Bit Supervision. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

[25] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*. 263–272. https://doi.org/10.1145/1081706.1081750

[26] SMT-LIB2 Website. 2021. http://smtlib.cs.uiowa.edu/benchmarks.shtml.

[27] Jialin Song, Ravi Lanka, Yisong Yue, and Bistra Dilkina. 2020. A General Large Neighborhood Search Framework for Solving Integer Linear Programs. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

[28] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (second ed.). The MIT Press.

[29] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex–White Box Test Generation for .NET. In *Tests and Proofs*, Bernhard Beckert and Reiner Hähnle (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 134–153. https://doi.org/10.1007/978-3-540-79124-9_10

[30] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: reducing, reusing and recycling constraints in program analysis. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, Will Tracz, Martin P. Robillard, and Tevfik Bultan (Eds.). ACM, 58. https://doi.org/10.1145/2393596.2393665

[31] Hengbiao Yu, Zhenbang Chen, Yufeng Zhang, Ji Wang, and Wei Dong. 2017. RGSE: a regular property guided symbolic executor for Java. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 954–958. https://doi.org/10.1145/3106237.3122830

[32] Wenjie Zhang, Zeyu Sun, Qihao Zhu, Ge Li, Shaowei Cai, Yingfei Xiong, and Lu Zhang. 2020. NLocalSAT: Boosting Local Search with Solution Prediction. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, Christian Bessiere (Ed.). ijcai.org, 1177–1183. https://doi.org/10.24963/ijcai.2020/164

[33] Yufeng Zhang, Zhenbang Chen, Ziqi Shuai, Tianqi Zhang, Kenli Li, and Ji Wang. [n.d.]. Multiplex Symbolic Execution: Exploring Multiple Paths by Solving Once. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 846–857. https://doi.org/10.1145/3324884.3416645

[34] Yufeng Zhang, Zhenbang Chen, and Ji Wang. 2012. Speculative Symbolic Execution. In *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012*. IEEE Computer Society, 101–110. https://doi.org/10.1109/ISSRE.2012.8

[35] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. 2010. Understanding bag-of-words model: a statistical framework. *Int. J. Mach. Learn. Cybern.* 1, 1-4 (2010), 43–52. https://doi.org/10.1007/s13042-010-0001-0