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

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
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 \( P \) by executing \( P \) in a symbolic manner. \( P \)'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 \( P \) under \( I \) results in path p. In the beginning, the PC is true. Then, when a non-branch statement \( S \) in \( P \) is executed, the symbolic computation corresponding to the symbolic values of the variables in \( S \) is carried out. When executing a branch statement \( S_b \) with condition \( b \), symbolic execution calculates \( b \)'s symbolic condition \( C_b \), and checks the feasibility of \( S_b \)'s true and false branches with respect to the current PC. This feasibility checking invokes a constraint solver [17]. If \( PC \land C_b \) is satisfiable [17], the true branch is feasible; if \( PC \land \neg C_b \) 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 \( PC \land C_b \) and \( PC \land \neg C_b \), respectively; otherwise, the infeasible branches are abandoned, i.e., no input can steer \( P \) to this branch. In this way, \( P \)'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 [3], 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. However, if we use a customized solving strategy, e.g., the following one, the solving time is only around 22s.

\[
\text{(check-sat-using (then simplify smt))}
\]

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 \( \Gamma \) be the set of the SMT formulas in different theories and SAT formulas, and \( \Theta \) be the set of solving results, i.e., \( \{ \text{SAT, UNSAT} \} \).

**Definition 2.1.** (Tactic) A tactic \( \Gamma \) is a function \( \Gamma \rightarrow \Gamma \cup \Theta \) that gives the resulted formula or solving result of an input formula.
Tactics also have some parameters that can be used to configure the tactics with integer values. 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.

\[
\neg x_2 \land x_1
\]

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 \( \Theta \) 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 \( I \) 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 \( P \) is a function \( \Gamma \rightarrow I \cup \mathbb{B} \) 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 \( T \) 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 \( T \in \mathbb{T}, c \in \mathbb{C}, P \in \mathbb{P}, n \in \mathbb{N}, \odot \) represents a commonly used numeric relation operator, and \( \beta \) represents a list of \( P \).

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

simplify \odot ITE(num-consts < 3, bit-blast \odot sat, smt) \hspace{1cm} (2)

Given a bit-vector formula \( \phi \), the strategy first applies simplify to \( \phi \) and gets \( \phi_1 \). Then, depending on the number of the constants in \( \phi_1 \), the strategy uses a bit-blasting style of solving in case \( \phi_1 \)’s number is less than three or directly applies smt for solving \( \phi_1 \).

### 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
two paths; otherwise, there are four paths in the false branch. If we use KLEE with Z3 as the SMT solver, we need 1362 seconds 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.

\[
\begin{align*}
& (1) \quad b > 0 \land a = 1.0 \\
& (2) \quad b > 0 \land a \neq 1.0 \\
& (3) \quad b > 0 \land a \neq 1.0 \land a^2 = 2.0 \\
& (4) \quad b > 0 \land a \neq 1.0 \land a^2 \neq 2.0
\end{align*}
\]

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

simplify; bit-blast; smt

After the second step, we will get the following two tactic sequences with parameters, where we use para1 = 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.

simplify(elim_and: true, hoist_mul: true, local_ctx: true, push_ite_bv: true); bit-blast; smt

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.

simplify(elim_and: true, hoist_mul: true, local_ctx: true, push_ite_bv: true); (3) bit-blast; smt

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) \cite{9}. Different choices of solving strategies may produce different results or performances. Hence, we use the MDP with cost to formalize the different tactics choices in the solving procedure.

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

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.
Definition 3.3. A rollout $\zeta \sim \pi$ is a following tuple sequence in which each tuple is in $S \times A \times S \times R$, 
$$\{(s_0, a_0, s_1, r_0, c_0), \ldots, (s_{n-1}, a_{n-1}, s_n, r_{n-1}, c_{n-1})\}$$
and the sequence is randomly constructed as follows.
- Sample an initial state $s_0$ with respect to $\beta_0$.
- Sample an action $a_i$ with respect to $\pi(s_i)$, and sample a transition with respect to $T(s_i, a_i)$ which leads to state $s_{i+1}$.
- The reward $r_i$ is $R(s_i, s_{i+1})$ and the cost $c_i = C(s_i, a_i)$.
- Keep doing the second step until $s_n$ is in $S_f$.

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 \(^1\). Hence, given an MDP $M$ with cost, the optimal policy $\pi^*$ is defined as follows.

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

The solving procedure of an SMT formula $\phi$ 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 $\phi$ by an SMT solver is a tuple $(Q, T_p, \beta_p, T_p, \{SUCC, FAIL\}, R_p, C_p)$, where
- $Q$ is the set containing the possible constraints during solving and two special states SUCC and FAIL.
- $T_p$ is the set of parameterized tactics.
- $\beta_p$ is the initial distribution such that $\beta_p(\phi) = 1$.
- $T_p : Q \setminus \{SUCC, FAIL\} \times T_p \rightarrow D$ gives a deterministic transition for a constraint and a tactic, i.e., $T_p(s, T_p(\phi)) = s'$, where $T_p(\phi) \in T_p$.
- 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.
- $R_p$ gives 1 to (SUCC) $\in Q \times Q$ and 0 to the others.
- $C_p : Q \setminus \{SUCC, FAIL\} \times T_p \rightarrow \mathbb{R}$ 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 $\pi_\phi$ for a formula $\phi$ is the one whose distribution satisfies the following condition.

$$\Pr_{\zeta \sim \pi_\phi} (s_n = SUCC) > 0 \quad (5)$$

\(^1\)We do not consider a discount because there are only finite steps in our SMT solving scenario.

\[ \text{Figure 4: MDP for solving } x^3 = (5.0 + 3.0). \]

It means that it is possible to solve $\phi$ by employing the policy $\pi_\phi$. For example, if we only consider the MDP in Figure 4, $(\text{simplify, smt})$ is a successful policy (i.e., $\pi(s_0)(\text{simplify}) = 1$ and $\pi(s_1)(\text{smt}) = 1$) with the successful solving’s probability 1. We use $\Pi_\phi$ 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 $\pi_\phi$ in $\Pi_\phi$ satisfies $\Pr_{\zeta \sim \pi_\phi} (\{s_n = SUCC\}) = 1$. Then, the optimal policy $\pi^*_\phi$ for $\phi$ is defined as follows.

$$\pi^*_\phi = \arg \max_\pi \mathbb{E}_{\zeta \sim \pi} \left[ 1 - \sum_{i=0}^{n-1} c_i \right] \quad (6)$$

For example, $\Pi_\phi$ of the MDP in Figure 4 has two policies, and the optimal policy is $(\text{simplify, smt})$, 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).

\section{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.

\subsection{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 $G$ to record the SMT formulas generated by
Algorithm 1: Symbolic Execution With Strategy Synthesis

\begin{verbatim}
SE(\mathcal{P}, S_1, S_2, Probs)
\end{verbatim}

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 \leftarrow \{s_1\}, 0
3 $\mathcal{G}, S \leftarrow \emptyset$, $\mathcal{DS}$
4 while worklist \neq \emptyset do
5 \textbf{if} stage = 0 then
6 \quad s \leftarrow \text{Select(worklist, } S_1\text{)}
7 \textbf{end}
8 \textbf{else}
9 \quad s \leftarrow \text{Select(worklist, } S_2\text{)}
10 \textbf{end}
11 $Q \leftarrow \text{Execute}(s, S)$
12 $\mathcal{G} \leftarrow \mathcal{G} \cup Q$
13 \textbf{if First stage ends then}
14 \quad $\mathcal{S} \leftarrow \text{Synthesize}(\mathcal{G}, \text{Probs})$
15 \quad stage \leftarrow 1
16 \textbf{end}
17 \textbf{end}
18 return $\mathcal{S}$

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 $S$ 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 $S_t$, $S_o$, and $S_{test}$ from the input set of formulas. $S_t$ is used to generate tactic sequences and probe-based conditions. $S_o$ is used to generate a composed solving strategy, and $S_{test}$ is used to compare the synthesized strategy and the default strategy. Then, we predict 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 $\varphi$. CPV($\varphi$, $ts$, Probs) is $\emptyset$ when $ts$ is empty sequence; otherwise, CPV($\varphi$, (0) $ts$, Probs) is defined as follows.

$\{ \mathcal{P} \mapsto \{ \mathcal{P}(\varphi') \mid \varphi' \in \text{Probs} \} \cup \text{CPV}(\varphi, ts, \text{Probs}) \}$

(7)

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

$M_1 \cup M_2[k] := \begin{cases} M_1[k] \cup M_2[k] & k \in M_1 \land k \in M_2 \\ M_1[k] & k \in M_1 \\ M_2[k] & k \in M_2 \end{cases}$

(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($V$, $S$) represents the cost of solving the SMT formulas in $V$ by employing the solving strategy $S$. We use the default strategy if the synthesized strategy is not better.
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 \((E(\varphi), E(T_s), t, p)\) consisting of four parts: \(E(\varphi)\) is the embedding of the current formula \(\varphi\); \(E(T_s)\) is the embedding of the tactic sequence applied until now to generate \(\varphi\); \(t\) 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. 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 \(T_S\). Then, together with the ones in \(T_s\) (Line 6), we evaluate the effectiveness and efficiency of the tactic sequences in \(T_{Sp}\). 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 \(\varphi\) can be solved by employing the tactic sequence \(t_s\). Then, we select the top \(N_1\) tactic sequences with respect to its value in \(M\).

### Algorithm 3: Parameter Tuning

**Data: \(Q\) is the set of training SMT formulas, \(T_S\) is a set of tactic sequences.**

1. begin
2. \(T_{Sp} \leftarrow \emptyset\)
3. for each \(t_s \in T_S\) do
4.    \(T_{Sp} \leftarrow T_{Sp} \cup \{\text{ParaGenerate}(t_s)\}\)
5. end
6. \(T_S = T_{Sp} \cup T_S\)
7. for each \(t_s \in T_S\) do
8.    \(M[t_s] \leftarrow \{\{\varphi \mid \varphi \in Q \land \text{Predicate}(\varphi, t_s)\}\}\)
9. end
10. \(T_{SN_1} \leftarrow \text{Top}(M, N_1)\)
11. for each \(t_s \in T_{SN_1}\) do
12.    \(M_1[t_s] \leftarrow \text{Solve}(Q, t_s)\)
13. end
14. return \(\text{Top}(M_1, N_2)\)
15. end

### Algorithm 4: Strategy Generation

**Data: \(S_v\) is the set of validation formulas, \(T_S\) is a set of tactic sequences, and \(C\) is a set of predicates.**

1. begin
2. if \(|S_v| < K\) then
3.    return \(\text{arg min}_{t_s \in T_S} \text{Solve}(S_v, t_s)\)
4. end
5. \(t_s \leftarrow \max_{t \in \{t_s \mid \text{length}(t_s) \leq \text{length}(t)\}}\)
6. \(S'_v \leftarrow \{t_s(\varphi) \mid \varphi \in S_v\}\)
7. \(T'_S \leftarrow \{t_{S_f} \mid t_{S_f} t_s \in T_S\}\)
8. \(C_{min} \leftarrow \min_{c \in C} \text{Cost}(c, S'_v, T'_S)\)
9. \(F_s \leftarrow \{t_1, \ldots, t_n \in T_S'\}\)
10. \(t_{i}^1 \leftarrow \min_{t \in F_s} \sum_{q \in S'_v, C_{min}} \text{min}(\text{Solve}(\{q\}, t_s) \mid t \in T_S')\)
11. \(t_{i}^2 \leftarrow \min_{t \in F_s} \sum_{q \in S'_v, C_{min}} \text{min}(\text{Solve}(\{q\}, t_s) \mid t \in T_S')\)
12. \(S_1 \leftarrow \text{GenStrategy}(S'_v, C_{min}, T'_S)\)
13. \(S_2 \leftarrow \text{GenStrategy}(S'_v, C_{min}, T'_S)\)
14. return \(t_s \vdash \text{ITE}(C_{min}, S_1, S_2)\)
15. end

For the top \(N_1\) tactic sequences in \(T_{SN_1}\), we select the top \(N_2\) ones by using the solver to solve the formulas in \(Q\) (Lines 10-14).

Given an SMT formula \(\varphi\) and a tactic sequence \(t_s\), the timeout predication is carried out as follows, where \(t_s\) is a non-empty sequence.

\[
\text{Predicate}(\varphi, t_s) := \begin{cases} 
\text{Predicate}(t_1(\varphi), t_s) & t_s=(t)^\prime t_s(\wedge t(s) \in \Gamma) \quad \text{true} \\
\text{Predicate}(t_1(\varphi), t_s) & t_s=(t)^\prime t_s(\wedge t(s) \in \Theta) \quad \text{false} \\
\text{NNPredicate}(\varphi) & \text{else} 
\end{cases} 
\]

(9)

Here, if a formula can be solved before the final tactic, the predication result is \(\text{true}\); 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 (\(\text{NNPredicate}(\varphi)\)). 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 \(T_S\) 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 \(T_S\) for a subset of \(S_v\). At the beginning, we get the longest common prefix \(t_{S_c}\) of all the tactic sequences (if it exists), where \(t \leq t_{S_c}\) represents that \(t\) is a prefix of \(t_{S_c}\). Then, there is a difference between the tactic sequences after \(t_{S_c}\), and the algorithm selects the tactic greedily with respect to the cost of solving the formulas in \(S_v\) after applying \(t_{S_c}\) (\(S'_v\) at Line 6). The best predicate is the one using which to separate the formulas will have a least solving cost, and Cost(\(c, S'_v, T'_S\)) (Line 8) is defined as
We have implemented our method on KLEE to evaluate the effectiveness of our method, we use Coreutils as the validation set. However, the generation introduces more overhead, which may also carry out more decisions. This balance needs to balance the effectiveness and generation overhead. In principle, we can have a strategy that can achieve the best tactic sequence for each SMT formula in the tactic sequences starting with the best tactics (Lines 10&11), where $TS$ represents the sequence subset of $S_T$ whose element starts with $t$, i.e., $\{(t)\in S_T | (t)\in TS\}$. 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 $S_C$’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 SP-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 predicing 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
Table 1: The Java benchmark programs.

<table>
<thead>
<tr>
<th>Subject</th>
<th>SLOC</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>txtmark</td>
<td>4255</td>
<td>A Markdown parser</td>
</tr>
<tr>
<td>barcode</td>
<td>3793</td>
<td>A library for QR code recognition</td>
</tr>
<tr>
<td>javaparser</td>
<td>22286</td>
<td>A Java code parser</td>
</tr>
<tr>
<td>html5parser</td>
<td>14266</td>
<td>A complete HTML5 parser</td>
</tr>
<tr>
<td>jsqparser</td>
<td>30250</td>
<td>A SQL statement parser</td>
</tr>
<tr>
<td>actson</td>
<td>623</td>
<td>A Json parser</td>
</tr>
<tr>
<td>nanoxml</td>
<td>1429</td>
<td>An XML parser</td>
</tr>
<tr>
<td>rhino</td>
<td>20042</td>
<td>A Speech-to-Intent engine</td>
</tr>
<tr>
<td>htmlparser</td>
<td>22231</td>
<td>A Java HTML parser</td>
</tr>
<tr>
<td>toba</td>
<td>6029</td>
<td>A Java bytecode to C compiler</td>
</tr>
<tr>
<td>jericoh</td>
<td>9542</td>
<td>A Java HTML parser</td>
</tr>
<tr>
<td>minimal-json</td>
<td>1859</td>
<td>A Json parser</td>
</tr>
<tr>
<td>xml</td>
<td>3367</td>
<td>An XML parser</td>
</tr>
<tr>
<td>pobs</td>
<td>3024</td>
<td>A Java object parser</td>
</tr>
<tr>
<td>Antlr</td>
<td>27118</td>
<td>A parser generator</td>
</tr>
<tr>
<td>fastjson-dev</td>
<td>19329</td>
<td>Alibaba Json parser</td>
</tr>
<tr>
<td>jmt123</td>
<td>3273</td>
<td>A MP3 decoder</td>
</tr>
<tr>
<td>nanojson</td>
<td>2185</td>
<td>Nano Json parser</td>
</tr>
<tr>
<td>foxykeep</td>
<td>3865</td>
<td>A Java code generator</td>
</tr>
<tr>
<td>jsoniter</td>
<td>13005</td>
<td>A Json parser</td>
</tr>
<tr>
<td>univocity</td>
<td>18263</td>
<td>A Java code parser</td>
</tr>
<tr>
<td>fastcsv</td>
<td>807</td>
<td>A CSV parser</td>
</tr>
<tr>
<td>argo</td>
<td>2687</td>
<td>A Jdom parser</td>
</tr>
<tr>
<td>htmlcleaner</td>
<td>8328</td>
<td>A Java HTML parser</td>
</tr>
<tr>
<td>jsoup</td>
<td>12512</td>
<td>A Java HTML parser</td>
</tr>
<tr>
<td>url-detector</td>
<td>1705</td>
<td>A URL parser</td>
</tr>
<tr>
<td>jcsv</td>
<td>1039</td>
<td>A CSV parser</td>
</tr>
<tr>
<td>commons-csv</td>
<td>1452</td>
<td>A CSV parser</td>
</tr>
<tr>
<td>super-csv</td>
<td>3734</td>
<td>A CSV parser</td>
</tr>
<tr>
<td>markdown4j</td>
<td>3740</td>
<td>A markdown parser</td>
</tr>
<tr>
<td>simple-csv</td>
<td>1583</td>
<td>A simple CSV parser</td>
</tr>
<tr>
<td>jaad</td>
<td>35984</td>
<td>An AAC decoder and MP4 demultiplexer library</td>
</tr>
<tr>
<td>jttidy</td>
<td>18937</td>
<td>A HTML cleaner</td>
</tr>
<tr>
<td>commonmark</td>
<td>4964</td>
<td>A markdown parser</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>327506</td>
<td></td>
</tr>
</tbody>
</table>

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 \( N_{\text{opt}} \) represents the number of paths or queries after employing our method, and \( N_{\text{baseline}} \) represents the number of original symbolic execution.

\[
\frac{N_{\text{opt}} - N_{\text{baseline}}}{N_{\text{baseline}}} \quad (12)
\]

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, \((-5.33\% \sim -0.4\%) \) on average. Our method can, on average, improves the number of explored paths by 66.11\% \((-9.37\% \sim 136.41\%) \). 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\% \((-57.66\% \sim 151.15\%) \). 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 64s (50s ~ 120s) 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 48s (32s~80s).

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.
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.

<table>
<thead>
<tr>
<th>Program</th>
<th>Baseline</th>
<th>Strategy Synthesis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#SAT</td>
<td>#Total</td>
</tr>
<tr>
<td>commons-csv</td>
<td>20372</td>
<td>18310</td>
</tr>
<tr>
<td>super-csv</td>
<td>33688</td>
<td>1088</td>
</tr>
<tr>
<td>nanoxml</td>
<td>29829</td>
<td>9014</td>
</tr>
<tr>
<td>argo</td>
<td>21285</td>
<td>3258</td>
</tr>
<tr>
<td>fastcsv</td>
<td>29740</td>
<td>1595</td>
</tr>
<tr>
<td>simple-csv</td>
<td>42237</td>
<td>673</td>
</tr>
<tr>
<td>univocity</td>
<td>20841</td>
<td>11856</td>
</tr>
<tr>
<td>markdow4j</td>
<td>30101</td>
<td>0</td>
</tr>
<tr>
<td>jcsv</td>
<td>22278</td>
<td>8552</td>
</tr>
<tr>
<td>pobs</td>
<td>22051</td>
<td>18</td>
</tr>
<tr>
<td>commonmark</td>
<td>23780</td>
<td>1410</td>
</tr>
<tr>
<td>jtidy</td>
<td>488</td>
<td>279</td>
</tr>
<tr>
<td>Antlr</td>
<td>12452</td>
<td>11130</td>
</tr>
<tr>
<td>actson</td>
<td>3446</td>
<td>41452</td>
</tr>
<tr>
<td>xml</td>
<td>17345</td>
<td>2836</td>
</tr>
<tr>
<td>txtmark</td>
<td>12421</td>
<td>3389</td>
</tr>
<tr>
<td>jericho</td>
<td>8106</td>
<td>3205</td>
</tr>
<tr>
<td>foxykeep</td>
<td>1177</td>
<td>605</td>
</tr>
<tr>
<td>htmlparser</td>
<td>6623</td>
<td>3474</td>
</tr>
<tr>
<td>minimal-json</td>
<td>19911</td>
<td>8038</td>
</tr>
<tr>
<td>jmpl23</td>
<td>16122</td>
<td>606</td>
</tr>
<tr>
<td>url-detector</td>
<td>4679</td>
<td>1689</td>
</tr>
<tr>
<td>rhino</td>
<td>12395</td>
<td>3802</td>
</tr>
<tr>
<td>toba</td>
<td>2945</td>
<td>284</td>
</tr>
<tr>
<td>javaparser</td>
<td>2877</td>
<td>1105</td>
</tr>
<tr>
<td>jsqaparser</td>
<td>1910</td>
<td>894</td>
</tr>
<tr>
<td>jaad</td>
<td>4196</td>
<td>167</td>
</tr>
<tr>
<td>nanojson</td>
<td>5949</td>
<td>792</td>
</tr>
<tr>
<td>html5parser</td>
<td>1306</td>
<td>7</td>
</tr>
<tr>
<td>jhtmlcleaner</td>
<td>275</td>
<td>155</td>
</tr>
<tr>
<td>jsoup</td>
<td>466</td>
<td>88</td>
</tr>
<tr>
<td>fastjson-dev</td>
<td>7336</td>
<td>1315</td>
</tr>
<tr>
<td>barcode</td>
<td>1216</td>
<td>1238</td>
</tr>
</tbody>
</table>

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 improve the other constraints, rewrites 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 logical 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 offline synthesis of a solving strategy for symbolic execution. There exits work of applying machine learning techniques in other parts of constraint solving. NeuroSMT [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, 62020107, 62032024 and 61690203).

REFERENCES


