Synthesizing Smart Solving Strategy for Symbolic Execution

Zehua Chen1, Zhenbang Chen1, Ziqi Shuai1,2, Yufeng Zhang3, Weiyu Pan1
1College of Computer, National University of Defense Technology, Changsha, China
2State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China
3College of Information Science and Engineering, Hunan University, Changsha, China
{zbchen,szq,panweiyu}@nudt.edu.cn,yufengzhang@hnu.edu.cn

ABSTRACT
Constraint solving is one of the challenges for symbolic execution. Modern SMT solvers allow users to customize the internal solving procedure by solving strategies. In this extended abstract, we report our recent progress in synthesizing a program-specific solving strategy for the symbolic execution of a program. We propose a two-stage procedure for symbolic execution. At the first stage, we synthesize a solving strategy by utilizing deep learning techniques. Then, the strategy will be used in the second stage to improve the performance of constraint solving. The preliminary experimental results indicate the promising of our method.

CCS CONCEPTS
• Software and its engineering → Software verification and validation;

KEYWORDS
Symbolic Execution, SMT Solving Strategy, Synthesis

1 INTRODUCTION AND MOTIVATION
Symbolic execution [7] is an SMT-based program analysis method that can systematically explore the path space of a program. Symbolic execution has been successfully applied to many software engineering activities, such as automatic testing, bug finding, and program repair. For symbolic execution, one of the main bottlenecks to its scalability is constraint solving [4].

Existing approaches for optimizing the constraint solving in symbolic execution include caching and reusing [3, 10], simplification of the constraints before solving [3], incremental solving [11], etc. All the existing approaches consider the SMT solver as a black-box. Actually, modern SMT solvers (e.g., Z3 [5] and CVC4 [2]) provide mechanisms for the users to control the solving procedure, e.g., solving strategy [6] in Z3. An SMT solving with a different solving strategy may have a different performance. However, most symbolic executors use the default solving strategy of the underlying SMT solver. Customizing a better solving strategy for the SMT solver can improve the solving’s performance in symbolic execution.

For example, consider the following SMT formula in floating-point theory, 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 the following solving strategy, the solving time is only around 22s.

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

Usually, modern SMT solvers provide a domain-specific language (DSL) to specify solving strategies. A solving strategy can be constructed from some tactics in terms of composition operators. For example, in the above example, \text{then simplify smt} are tactics, and \text{check-sat-using} is the sequential composition operator. A tactic may transform an SMT formula in many different ways, such as simplification and translation. Some tactics are special for the final solving in SAT or SMT theory, such as smt and sat. A tactic also has some parameters that can be used to configure the transformation or solving.

In this extended abstract, we propose to synthesize a smart solving strategy for the program under symbolic execution. Our key observation is that a program has its specific SMT formulas during symbolic execution. We need a customized solving strategy for the program during symbolic execution. The key idea is to use the SMT formulas generated at the early stage of symbolic execution to synthesize a strategy that can be used in the later stage. The synthesis utilizes deep learning and decision tree learning techniques to online synthesize a solving strategy during symbolic execution.

2 PROPOSED METHOD
We propose the two-stage framework in Figure 1 for symbolic execution. The first stage of symbolic execution is for synthesizing the solving strategy that will be used in the second stage. We use the SMT formulas generated by the symbolic executor in the first stage to synthesize the solving strategy. The SMT formulas are divided into training and validation sets, denoted by \(S_t\) and \(S_v\), respectively. The synthesis consists of three steps that will be explained next.

Tactic sequence generation. For each formula in \(S_t\), we predicate a tactic sequence by a deep reinforcement learning (DRL) model. We train the DRL model offline. A training data of the DRL model consists of four parts and can be represented by \((E(\varphi), E(T_s), t, p)\). \(E(\varphi)\) denotes the embedding of formula \(\varphi\). \(T_s\) is the applied tactic sequence that generates \(\varphi\) from the original formula, and \(E(T_s)\) is

\(^{1}\)Z3’s version is 4.6.2. The CPU is 2.5GHz.
We have implemented our method on KLEE with Z3 as the back-end solver. We train the DRL model and the DNN models by Pytorch. The synthesis procedure is implemented in Python 3.6. We have carried out the preliminary experiments on GNU coreutils, a commonly used benchmark for KLEE-based symbolic execution methods. Each program is analyzed for 30 minutes. On the 86 coreutils programs, we can improve the number of the explored paths for 49 programs. For the remaining 37 programs, we have no effect on 8 programs but decrease the number of paths for 29 programs. On average, we improve the number of explored paths by 11.4% (−56.8% ~ 70.7%). The average synthesis time is 87s (54s ~ 146s).

For the next step, our plan is as follows: 1) extensive experiments on coreutils and other types of benchmarks, such as floating-point programs; 2) implementation and validation of our method on other types of symbolic execution engines, such as JPF-based engines for Java programs; 3) online adjustment of the DNN models in our method to improve the precision and effectiveness.

### 4 RELATED WORK

As far as we know, we are the first to synthesize a program-specific solving strategy under the background of symbolic execution. There are few existing work of finding optimal solving strategies for SMT solving. In [8], the authors mutate the default solving strategy to search the optimal strategy for a set of SMT formulas. FastSMT [1] employs DNN to learn the optimal strategy for SMT benchmarks. FastSMT inspires our work. However, our work targets the online synthesis of solving strategy for symbolic execution.

### ACKNOWLEDGEMENTS

This research was supported by National Key R&D Program of China (No. 2017YFB1001802) and NSFC Program (No. 61632015, 61690203 and 61532007).

### REFERENCES


