Synthesizing Smart Solving Strategy for Symbolic Execution

Zehua Chen[1], Zhenbang Chen[1], Ziqi Shuai[1,2], Yufeng Zhang[3], Weiyu Pan[1]
[1]College of Computer, National University of Defense Technology, Changsha, China
[2]State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China
[3]College 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

ACM Reference Format:
Zehua Chen, Zhenbang Chen, Ziqi Shuai, Yufeng Zhang, Weiyu Pan. 2020. Synthesizing Smart Solving Strategy for Symbolic Execution. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3324884.3418904

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.

(`check-sat-using` (**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, `simplify` and `smt` are tactics, and **then** 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 $(\mathcal{E}(\varphi), \mathcal{E}(T_s), t, p)$. $\mathcal{E}(\varphi)$ denotes the embedding of formula φ. T_s is the applied tactic sequence that generates φ from the original formula, and $\mathcal{E}(T_s)$ is

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

ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3418904

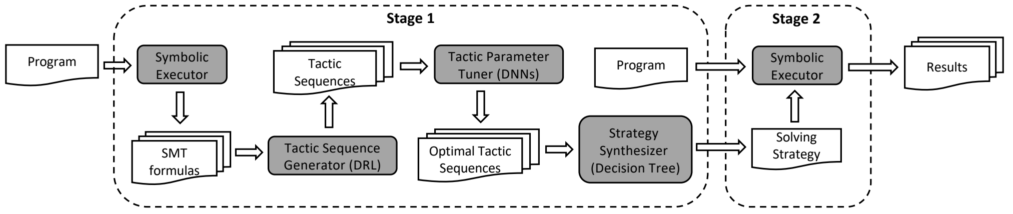

Figure 1: The two-stage procedure of symbolic execution.

the embedding of T_s. t and p are the next tactic and its probability, respectively. We generate the training data of the DRL model from the existing SMT benchmarks. The generation searches the possible strategies of an SMT formula and records the next tactic and its probability *w.r.t.* the consumed resources for applying the tactic. We use the bag-of-words (BOW) method for the embedding of an SMT formula.

Tactic parameter tuning. Tactic parameters also have significant effects on the performance of solving. A tactic without parameter configuration uses the default configuration of the parameters. After getting the tactic sequences (denoted by TS_o) for the formulas in S_t, we randomly generate the parameter configurations for the tactics in the strategy sequences. We use TS_c to denote the tactic sequences with parameter configurations. We predicate the performance of the tactic sequences in $TS_o \cup TS_c$ *w.r.t.* S_t by pre-trained deep neural network (DNN) models. Then, we select top N_1 tactic sequences *w.r.t.* the number of the predicatively solved formulas in S_t. For these N_1 tactic sequences, we configure them to the solver to solve the formulas in $S_t \cup S_v$, and select the top N_2 sequences *w.r.t.* the number of the solved formulas in $S_t \cup S_v$. These top N_2 sequences (denoted by TS_{N_2}) will be used in the next step for synthesizing the strategy.

Strategy synthesis. Based on the top N_2 tactic sequences, we use decision tree [9] to synthesize a solving strategy. We train a decision tree *w.r.t.* TS_{N_2} and $S_t \cup S_v$ to recommend a tactic sequence in TS_{N_2} to an SMT formula. The trained decision tree ensures that the recommended tactic sequence can solve the formulas in $S_t \cup S_v$ that are classified into the class of the tactic sequence as many as possible. Based on the decision tree, we can construct the solving strategy by composing the tactic sequences by the *if-then-else* operator in the strategy DSL. The conditions are the learned predicates in the decision tree.

Discussion. Solving strategy synthesis introduces overhead. It is challenging to achieve a tradeoff between the solving performance and the extra synthesis overhead. To reduce the overhead of synthesis, we use the offline trained DNN models in the first and second steps to avoid online training. Besides, we observe that the SMT formulas generated during the symbolic execution of a program are similar. Hence, we do not need many SMT formulas for synthesis, which also reduces the overhead.

3 PRELIMINARY RESULTS AND OUR PLAN

We have implemented our method on KLEE with Z3 as the backend solver [2]. 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% ($-56.8\% \sim 70.7\%$). The average synthesis time is 87s (54s $\sim$ 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

[1] Mislav Balunovic, Pavol Bielik, and Martin Vechev. 2018. Learning to solve SMT formulas. In *Advances in Neural Information Processing Systems*. 10317–10328.
[2] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *CAV 2011*. 171–177.
[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI 2008*. USENIX Association, USA, 209–224.
[4] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.
[5] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.
[6] 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*. 15–44.
[7] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252
[8] Nicolás Gálvez Ramírez, Youssef Hamadi, Eric Monfroy, and Frédéric Saubion. 2016. Evolving smt strategies. In *ICTAI 2016*. IEEE, 247–254.
[9] Jiang Su and Harry Zhang. 2006. A fast decision tree learning algorithm. In *AAAI*, Vol. 6. 500–505.
[10] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In *FSE 2012*. Association for Computing Machinery, New York, NY, USA, 11.
[11] Yufeng Zhang, Zhenbang Chen, and Ji Wang. 2012. Speculative Symbolic Execution. In *ISSRE 2012*. 101–110.

[2]KLEE's version is 2.1-pre, and Z3's version is 4.6.2