Efficient Multiplex Symbolic Execution with Adaptive Search Strategy

Tianqi Zhang[1], Yufeng Zhang[2], Zhenbang Chen[1], Ziqi Shuai[1], Ji Wang[1,3]
[1]College of Computer, National University of Defense Technology, Changsha, China
[2]College of Computer Science and Electronic Engineering, Hunan University, Changsha, China
[3]State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China
zhangtianqi18@nudt.edu.cn,yufengzhang@hnu.edu.cn,{zbchen,szq,wj}@nudt.edu.cn

ABSTRACT

Symbolic execution is still facing the scalability problem caused by path explosion and constraint solving overhead. The recently proposed MuSE framework supports exploring multiple paths by generating partial solutions in one time of solving. In this work, we improve MuSE from two aspects. Firstly, we use a light-weight check to reduce redundant partial solutions for avoiding the redundant executions having the same results. Secondly, we introduce online learning to devise an adaptive search strategy for the target programs. The preliminary experimental results indicate the promising of the proposed methods.

CCS CONCEPTS

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

KEYWORDS

symbolic execution, search strategy, machine learning

ACM Reference Format:
Tianqi Zhang, Yufeng Zhang, Zhenbang Chen, Ziqi Shuai, Ji Wang. 2020. Efficient Multiplex Symbolic Execution with Adaptive Search Strategy. 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/10.1145/3324884.3418902

1 INTRODUCTION

Symbolic execution has been widely applied in many areas [1], including test case generation, program repair, *etc.* In symbolic execution, The values of variables are maintained as symbolic expressions. A constraint solver is invoked to determine the feasibility of the path on each branch. When applied to large-scale programs, symbolic execution is still facing the scalability problem caused by path explosion and constraint solving. Firstly, the number of paths may grow exponentially with the number of conditional statements. Therefore, it is infeasible to iterate all the paths of a large-scale program. Secondly, constraint solving is notoriously hard [5]. On each path, the constraint solver is invoked to determine the satisfiability of the corresponding path constraints. In practice, constraint solving may dominate the procedure of symbolic execution [1].

Multiplex symbolic execution (MuSE) [9] is one of the methods [1] tackling the scalability problem. MuSE is based on the insight that the solution space searched by the constraint solver and the path space explored by the symbolic execution engine are both from the same input space. Therefore, when solving the path constraints, the constraint solver is essentially searching the path space. The *partial solutions* encountered during the constraint solving procedure can be utilized as useful test cases triggering off-the-path branches on the current path, although the partial solutions cannot satisfy the path constraints. MuSE can achieve the same coverage within orders of magnitude less time budget than vanilla symbolic execution.

We find that MuSE is not efficient enough in two aspects. Firstly, although the constraint solver may produce hundreds of partial solutions during one solver call, many partial solutions may trigger the same path. It is wasteful to execute redundant test inputs symbolically. Secondly, MuSE produces orders of magnitude more test inputs within the same times of constraint solving. Therefore, MuSE needs a different strategy to prioritize unexplored branches. In this work, we plan to solve these two problems.

2 METHOD

We propose two methods to improve the efficiency of MuSE. The improved algorithm is shown in Algorithm 1. The procedure starts with an initial input I_0 and executes all the generated inputs in T. B stores all the unexplored branches. A strategy s detemines which branch in B is prioritized. The algorithm uses a constraint solver supporting partial solutions. After solving, both solution (if SAT) and partial solutions are stored into T. This procedure repeats until all branches are explored, or some stop criterion (*e.g.*, time budget) is satisfied. In the following, we discuss how we improve MuSE.

2.1 Reducing Redundant Inputs

In MuSE, the constraint solver may produce hundreds of partial solutions in one time of solving. We observe that many partial solutions trigger the same path or have the same coverage. Therefore, we devise a light-weight *filter* (line 15) mechanism to reduce redundant test inputs. Firstly, we maintain the execution tree representing all the paths that have been covered as well as the open branches yet to be explored. For each covered path and unexplored branch, we also store the corresponding path condition. We filter out partial solutions which satisfy the same path condition with a covered path. We have conducted preliminary experiments on 10 real-world Java programs using our extended Simplex-based QF_LIA theory solving algorithm with MuSE on JPF [9]. Results show that such

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/10.1145/3324884.3418902

a fast check can reduce 7.4% of redundant partial solutions. Secondly, since one loop statement may correspond to multiple paths in the execution tree, inputs corresponding to different branches are not meant to cover different statements. Therefore, we also use a standby concrete execution further to filter inputs that do not lead to new coverage. In our preliminary experiments, we find that a total of 99% redundant partial solutions can be reduced. Note that we use statement coverage to determine whether a partial solution is redundant. Another criterion may lead to different results.

Algorithm 1 Improved MuSE

```
1:  Input: Program $P$, open branches set $B$, search strategy $s$
2:  $T = \{I_0\}$ //test cases to be executed
3:  do
4:      for $I$ in $T$ do //execute all generated inputs
5:          $p = concolicExecute(P, I)$ //$p$ is the current path
6:          $saveUnexploredBranches(p, B)$
7:      end for
8:      $b = select(B, s)$
9:      $\phi = pathCondition(b)$
10:     $(res, solution, partial\text{-}solutions) = solving(\phi)$
11:     if $res$ = SAT then
12:         $T = T \cup \{solution\}$
13:     end if
14:     /*save partial-solutions whether SAT or not, if any*/
15:     $T = T \cup filter(partial\text{-}solutions)$ //reduce redundant inputs
16: while $B \neq \emptyset \wedge \neg$ stopCriterion()
```

2.2 Adaptive Search Strategy for MuSE

Search strategy is important to the performance of symbolic execution [1]. In MuSE, the search strategy not only decides which branch is to be explored but also affects the efficiency and effectiveness of the generated partial solutions. For example, the path condition of a deeper branch tends to contain more constraints and may lead to timeout in solver for vanilla symbolic execution. However, in MuSE, solving more complex path condition tends to generate more partial solutions, which are supposed to cover more program paths. Nevertheless, it is hard to predict whether a partial solution will improve the program coverage. Thus, it is difficult to devise an appropriate search strategy for MuSE. Due to the complexity of programs, there is no silver bullet strategy for all target programs. Recently, Sooyoung *et al.* propose using online learning to devise an adaptive search strategy for target programs [3]. We introduce such a learning paradigm to improve the search strategy of MuSE.

Table 1: Features of unexplored branch b

1	whether b is from a loop statement
2	the depth of b (normalized)
3	the number of unexplored branches after all the off-path-branches along the path prefix of b
4	the sum of the depths of unexplored branches after all the off-path-branches along the path prefix of b (normalized)
5	whether b lies in the top 10% part of the most deepest path
6	whether b lies in the bottom 10% depth of the most deepest path
7	whether b is from a statement covered by last runs
8	whether b is a direct descendant of some branch b' which is solved yet

In our algorithm, a strategy s is a function defined on the set of branch $s(b) = w \cdot feat(b)$, where w is the parameter vector and $feat(b) = (feat_1(b), \cdots, feat_n(b))$ is a vector representing the features of b. As shown in Table 1, each feature describes the static or dynamic information of b. For example, feature 1 determines whether b is from a loop statement. If yes, we want to assign b a lower priority. Feature 2 prioritizes deeper branches because they may lead to more partial solutions.

Algorithm 2 shows the procedure of the online learning of the search strategy. The overall procedure is a genetic algorithm [7] searching the optimal parameter vector fitting the target program best. Initially, we use a set of randomly chosen strategy set S to direct MuSE. For each strategy, we run MuSE for a fixed number of iterations. Next, we evaluate the coverage information of each strategy and choose the good ones G. Then we generate offsprings from G and kill bad ones from S. The procedure iterates until the execution tree is completed or some stop criterion is satisfied. The strategy search procedure is performed online along with the symbolic execution. During this procedure, bad strategies will be abandoned and good strategies will be found.

Algorithm 2 Online Learning Adaptive Search Strategy

```
1:  Input: Program $P$, initial input seed $I_0$
2:  $S = \{s_1, \ldots, s_k\}$ //initial search strategies
3:  $B = \emptyset$ //open branches to be explored
4:  do
5:      for $s_i$ in $S$ do
6:          $C_i$ = MuSE($P, B, s_i$) // return coverage information
7:      end for
8:      $G = choose(S)$ // choose better strategies w.r.t. $C_i$
9:      $S = S \cup offSpring(G) \setminus kill(S)$
10: while $B \neq \emptyset \wedge \neg$ stopCriterion()
```

3 RELATED WORK AND RESEARCH PLAN

Many works have been proposed to improve the search strategy of symbolic execution [1]. The CFDS [2] method chooses branch whose opposite branch is the nearest to uncovered statement. The CGS [6] method excludes branch whose context has been met before. Generational search [4] prioritizes all the open branches on the current path. Xie *et al.* use fitness function to devise search strategy [8]. Recently, Sooyoung *et al.* propose using learning to devise adaptive search strategy for target program [2, 3].

Nextly, we plan to investigate more features and conduct large-scale experiments to evaluate our online strategy learning method.

ACKNOWLEDGEMENT

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

REFERENCES

[1] 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, Article 50 (May 2018), 39 pages.
[2] J. Burnim and K. Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *ASE 2008*. IEEE Computer Society, USA, 443–446.
[3] Sooyoung Cha and Hakjoo Oh. 2019. Concolic Testing with Adaptively Changing Search Heuristics. In *ESEC/FSE 2019* (Tallinn, Estonia). ACM, 235–245.
[4] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1 (Jan. 2012), 20–27.
[5] Daniel Kroening and Ofer Strichman. 2008. *Decision Procedures: An Algorithmic Point of View*. https://doi.org/10.1007/978-3-540-74105-3
[6] Hyunmin Seo and Sunghun Kim. 2014. How We Get There: A Context-Guided Search Strategy in Concolic Testing. In *FSE 2014* (Hong Kong, China). 413–424.
[7] M. Srinivas and L. M. Patnaik. 1994. Genetic algorithms: a survey. *Computer* 27, 6 (1994), 17–26.
[8] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *2009 IEEE/IFIP DSN*. IEEE Computer Society, Los Alamitos, CA, USA.
[9] Yufeng Zhang, Zhenbang Chen, Ziqi Shuai, Tianqi Zhang, Kenli Li, and Ji Wang. 2020. Multiplex Symbolic Execution: Exploring Multiple Paths by Solving Once. In *ASE 2020*. IEEE Computer Society, USA.