




FDSE: Enhance Symbolic Execution by Fuzzing-based Pre-Analysis (Competition Contribution)

Guofeng Zhang^{1,2,3}, Ziqi Shuai^{1,2,3}, Kelin Ma^{1,2}, Kunlin Liu^{1,2,3},
Zhenbang Chen^{1,2} , and Ji Wang^{1,2,3}

¹ College of Computer, National University of Defense Technology, Changsha, China

² State Key Laboratory of Complex & Critical Software Environment, National University of Defense Technology, Changsha, China

³ State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China

{zhangguofeng16,szq,kelinma,klliu18,zbchen,wj}@nudt.edu.cn

Abstract. FDSE serves as an automatic test generation tool designed for C programs based on symbolic execution. FDSE employs fuzzing-based pre-analysis and combines static symbolic execution and dynamic symbolic execution to improve the effectiveness of test generation. FDSE achieves 5132 scores and is ranked 4th in the branch coverage track of Test-Comp 2024.

Keywords: Symbolic Execution · Fuzzing · Test-Case Generation.

1 Test Generation Approach

Test case design is one of the most labor-intensive tasks in software engineering. Automatic test case generation helps the test case designers reduce labor and improve testing quality. Existing techniques usually accept more than one type of software artifact (*e.g.*, source code and software models) as input. Then, these techniques utilize existing methods (*e.g.*, optimization [10] or program analysis [11]) to generate test cases. Besides, some approaches combine different methods to achieve better effectiveness and efficiency [1].

Symbolic execution (SE) [5] is one of the underlying techniques that can be used for automatic test case generation. Current SE methods can be categorized into static symbolic execution (SSE) and dynamic symbolic execution (DSE). SSE simulates the execution of the program using symbolic inputs. During analysis, SSE maintains many execution states. When encountering a branch statement, SSE forks states to explore both branches. Many SSE engines have been developed, such as KLEE [4] and SPF [9], to name a few. DSE combines symbolic execution and concrete execution to further improve SE's effectiveness and efficiency. Specifically, DSE executes the program using concrete input and collects path constraint of current execution. Then, based on the path constraints, DSE constructs the new constraint for generating new input that steers the program

*Z. Chen—Jury Member.

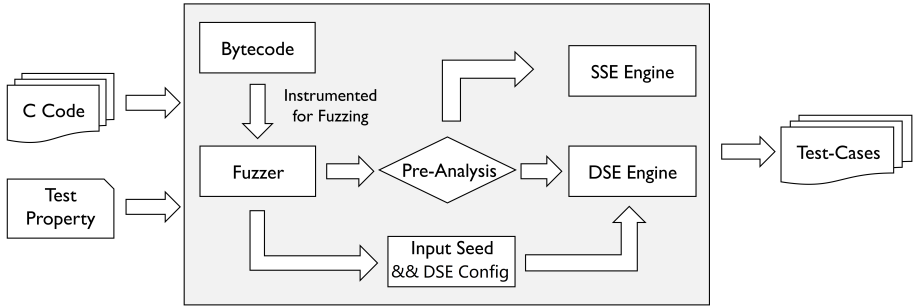


Fig. 1: FDSE’s Workflow in Test-Comp.

to different program path. In principle, SSE and DSE provide different means of systematically exploring the program’s path space.

FDSE is mainly a SE-based test case generator. In most cases, FDSE uses DSE to generate tests. To mitigate DSE’s disadvantage in handling the programs with long-time execution or large symbolic data, *e.g.*, the programs with large symbolic arrays, loops, or many branches, FDSE employs a fuzzing-based pre-analysis and combines SSE to improve DSE’s effectiveness and efficiency of generating tests for the benchmarks of Test-Comp.

2 Framework

Figure 1 illustrates the Test-Comp version of FDSE. Firstly, we compile the C program into bytecode and instrument the bytecode to generate a fuzzer for pre-analysis. During fuzzing, we record the runtime features of the program, such as the number of input variables or branches and the size of allocated arrays. Secondly, we selectively employ DSE or SSE according to the number of static branches, which is calculated by a simple static analysis. If the number exceeds a threshold, *e.g.*, 10,000 in the competition, FDSE employs SSE because DSE may face the challenge of long-time execution. Otherwise, FDSE continues to use DSE. Hence, either DSE or SSE is applied to analyze a benchmark program. Finally, when employing the DSE engine, selective symbolization of the variables is performed based on the information generated by fuzzing, aiming to mitigate the problem of large symbolized arrays. Furthermore, the DSE engine limits the number of loop unfolding times to prevent path explosion. This fuzzing-based pre-analysis is based on the following two observations of the Test-Comp benchmarks.

- When the program utilizes large loops to initialize a large-sized symbolic array⁴, DSE maintains a huge number of symbolic variables internally, which hinders the analysis’s efficiency and frequently exceeds memory limits. To mitigate this, we employ fuzzing for pre-analysis to generate the parameters that restrict the scale for DSE.

⁴ For example, the benchmark `standard_copy2_ground-1.c`

```

#define N 100000
int main() {
    int a1[N], a2[N], a3[N], i;
    for(i=0; i<N; i++) {
        a1[i]=input(); a2[i]=input();
    }
    for(i=0; i<N; i++) a3[i]=a1[i];
    for(i=0; i<N; i++) a3[i]=a2[i];
    for(i=0; i<N; i++)
        assert(a1[i]==a3[i]);
    return 0;
}

```

Fig. 2: standard_copy2_ground-1.c

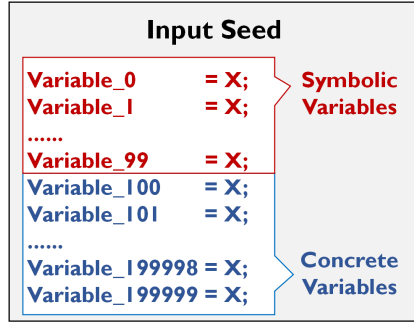


Fig. 3: Selective Symbolization in FDSE

- For programs that contain a large number of static branches⁵, executing a terminated path needs much time, which hinders the overall efficiency of DSE. To tackle this problem, we propose using SSE instead of DSE to analyze such programs, as SSE can perform better in this scenario.

Demonstration. We use a benchmark program in Test-Comp to demonstrate the fuzzing-based pre-analysis. Figure 2 shows an example program that contains four loops with a size of 100,000 and requires 200,000 input variables (*i.e.*, symbolic variables). SE is impractical to explore the path space of this program. The key idea is to employ fuzzing first to generate seed inputs and symbolize a part of input variables during SE, which can improve efficiency while ensuring high coverage. Consider the program in Figure 2. The first step is to employ fuzzing to generate input seeds, as shown in Figure 3. These seeds contain 200,000 variables, each with a random value X . Since only eight static branches exist, FDSE uses the DSE engine. During DSE, FDSE limits the boundary of each loop, allowing the loop body to be unrolled up to a configured number of times. This configuration is determined by the information collected by fuzzing. FDSE unrolls the loop only 50 times if the fuzzer detects that the loop body is executed more than 100 times. Then, DSE reads the input seeds obtained from fuzzing. For this example, DSE only symbolizes the first 100 variables due to the 50 times of loop unrolling. The remaining variables only have concrete values. When generating test cases, the generated values of symbolic variables are concatenated with the values of the subsequent concrete variables in the input seed. Thus, DSE can still generate a complete test case.

3 Result and Discussion

FDSE is optimized and achieves 5132 scores (4th place) in the branch coverage track. Our tool performs well in many sub-categories, such as **Arrays**, **Bit Vectors**, and **Hardness**. Thanks to Test-Comp’s competition, we have identified

⁵ For example, the program `Problem05_label40+token_ring.01.cil-1.c`

several shortcomings in our DSE engine beyond the common challenges (such as path explosion and constraint solving [2]).

- Our DSE engine does not apply any simplification rule to reduce symbolic expressions, which results in redundant expressions and makes the tool crash on some **Hardware** benchmarks due to exceeding memory limits.
- Our DSE engine is limited in environment modeling, *e.g.*, the common system libraries. When programs call these system libraries, the relevant path constraints are lost, making it difficult to improve coverage, particularly in the tasks in **BusyBox**, **DeviceDriverLinux64**, and **AWS-C-Common**.
- Our DSE engine is still limited in handling large symbolic arrays. Restricting the number of symbolic variables limits the path exploration ability, which may fail to cover deep branches.
- We do not prioritize or minimize the generated tests, which results in redundant test cases and leads to validator timeout. For example, in the **Combinations** category, over 20% of tests were not executed.
- FDSE is only optimized for branch coverage track. Smarter SE search strategies for branch and error coverage are expected.

4 Software Project and Data Available

The DSE engine’s implementation of FDSE is based on SymCC [8]. The SSE engine is KLEE [4]. The fuzzing component is implemented in C++ and based on LLVM⁶[6]. The employed constraint solver of DSE is Z3 [7]. The command line interface is implemented in Python.

In Test-Comp 2024, FDSE participated in **coverage-branches** and **coverage-error** categories, where we only optimize FDSE for **coverage-branches**. The benchexec tool information module is **fdse.py**, and the benchmark description is **fdse.xml**. To use our tool script, the parameters of the property file, time budget, and benchmark path must be set as follows:

```
fdse -testcomp -property-file=<.> -max-time=<.> -single-file-name=<.>
```

Our symbolic execution engine treats each benchmark as running on a 64-bit architecture and always tries to maximize code coverage. The test suite generated is written to the directory **fdse_output/test-suite**. According to the definition of Test-Comp rules, the test suite includes a metadata XML file and a test-case XML file that follows the required format.

FDSE, developed by the National University of Defense Technology, can be found at <https://github.com/zbchen/fdse-test-comp>. FDSE is accessible for download as a binary artifact on Zenodo, and the specific version available for download is **testcomp24**⁷, and it is publicly accessible under the Apache-2.0 license terms. Moreover, Test-Comp 2024 [3]⁸ provides users with scripts, benchmarks, and FDSE binaries to facilitate the replication of competition results.

⁶ LLVM’s version is 10.0.1.

⁷ <https://doi.org/10.5281/zenodo.10203198>

⁸ <https://test-comp.sosy-lab.org/2024>

Acknowledgement This research was supported by National Key R&D Program of China (No. 2022YFB4501903) and the NSFC Programs (No. 62172429 and 62002107).

References

1. Anand, S., Burke, E.K., Chen, T.Y., Clark, J.A., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P.: An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.* **86**, 1978–2001 (2013)
2. Baldoni, R., Coppola, E., D’Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* **51**, 1–39 (2016)
3. Beyer, D.: Automatic testing of C programs: Test-Comp 2024. Springer (2024)
4. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8–10, 2008, San Diego, California, USA, Proceedings. pp. 209–224. USENIX Association
5. King, J.C.: Symbolic execution and program testing. *Commun. ACM* **19**, 385–394 (1976)
6. LLVM: <https://llvm.org>
7. de Moura, L.M., Bjørner, N.S.: Z3: An efficient smt solver. In: International Conference on Tools and Algorithms for Construction and Analysis of Systems (2008)
8. Poeplau, S., Francillon, A.: Symbolic execution with symcc: Don’t interpret, compile! In: USENIX Security Symposium (2020)
9. Păsăreanu, C.S., Rungta, N.: Symbolic pathfinder: symbolic execution of java bytecode. Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (2010)
10. Shahbazi, A., Miller, J.: Black-box string test case generation through a multi-objective optimization. *IEEE Transactions on Software Engineering* **42**, 361–378 (2016)
11. Tillmann, N., de Halleux, J.: Pex-white box test generation for .net. In: Tests and Proofs. pp. 134–153 (2008)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

