FDSE v2: Variable Importance Guided Hybrid Fuzzing
(Competition Contribution)

Guofeng Zhang [1], Zhenbang Chen [1] *, and Ji Wang [1]

College of Computer Science and Technology, National University of Defense Technology, Changsha, China
{zhangguofeng16, zbchen, wj}@nudt.edu.cn

Abstract. `FDSE v2` is a hybrid fuzzing tool that automatically generates high-coverage test suites for C programs. In prior work, we introduced an initial version of `FDSE` that used fuzzing-based pre-analysis to enhance symbolic execution. This paper presents `FDSE v2`, featuring a novel constraint tree component that more effectively coordinates the fuzzer and concolic execution engine, thereby reducing ineffective search in fuzzing. The constraint tree models the space of path constraints already explored by concolic execution and enables fine-grained partitioning of input variables based on their semantic relevance. During seed mutation, the fuzzer focuses only on the most important variable groups identified through this analysis. Concolic execution excels at navigating complex path conditions involving arithmetic or logical guards, while the fuzzer leverages the resulting high-quality seeds as starting points for deeper exploration under extended time budgets. By reducing redundant efforts between the two engines, `FDSE v2` achieves significant improvements in both code coverage and vulnerability detection effectiveness.

Keywords: Hybrid Fuzzing · Concolic Execution · Testcase Generation.

1 Overview

Software testing remains one of the most labor-intensive phases in the software development lifecycle [1]. Automated test-case generation is therefore essential to reduce manual effort and improve software reliability. By integrating complementary automated testing techniques, such approaches can achieve both higher effectiveness and greater efficiency.

This year, we present `FDSE v2`, which represents a redesign and strategic shift from its predecessor `FDSE` [11]. While `FDSE` was primarily a symbolic execution tool, achieved 4th place overall in the Test-Comp 2025. `FDSE v2` is now repositioned as a hybrid fuzzing framework [10,7,9] that tightly integrates symbolic execution with coverage-guided fuzzing. In Test-Comp 2026 [2], FDSE v2

[1] Also with the affiliation: State Key Laboratory of Complex & Critical Software Environment, National University of Defense Technology, Changsha, China

* Jury Member and Corresponding Author

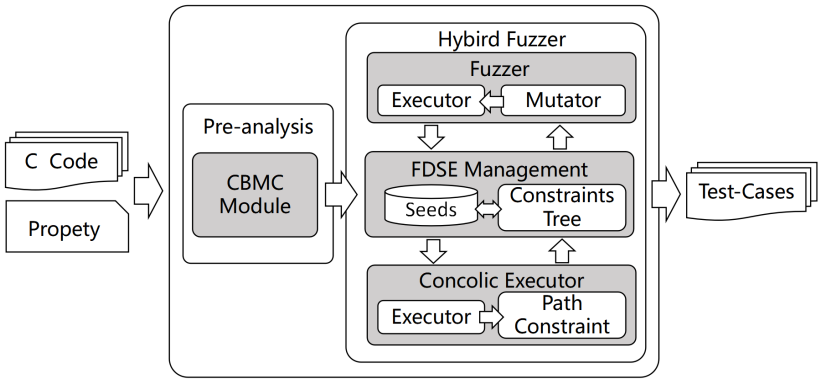

Fig. 1: `FDSE v2`'s workflow in Test-Comp.

earned a total score of 9,319, securing 2nd place overall, and scored 8,121 in the Cover-Branches category, also ranking 2nd.

`FDSE v2` demonstrates further improvements in code coverage over its predecessor. A key contribution of this work is the use of a constraint tree generated by concolic execution to identify groups of input variables that are semantically relevant to uncovered program regions, thereby guiding the seed mutation strategy of the fuzzer. During concolic execution [8], every program operation is precisely modeled, and at each branching point, an SMT solver is invoked to generate inputs that satisfy the negated branch condition, thereby producing high-quality test inputs that reach deep program states.

However, most existing hybrid fuzzing approaches interact with the concolic execution engine only superficially, for example by exchanging seeds or basic coverage information, and consequently discard the rich set of path constraints accumulated during analysis [3]. In contrast, `FDSE v2` systematically collects all constraints and their corresponding concrete inputs generated across multiple runs of concolic execution. Within the central control process, these constraints are reconstructed into a persistent constraint binary tree. This structure enables precise extraction of variable groups associated with uncovered code regions, which in turn directs hybrid fuzzing toward unexplored parts of the program state space.

This architectural innovation enables `FDSE v2` to mitigate redundant exploration between its constituent engines, leading to more efficient coverage growth and improved bug-finding capability.

2 Test Generation Approach

Figure 1 shows the workflow of `FDSE v2` in Test-Comp. The tool takes a C program and test properties as input. The program is compiled into bytecode and instrumented separately for fuzzing and concolic execution. A key enhancement is the integration of the CBMC [4] (Bounded Model Checker) module. During

```c
#define N 100
int main() {
  int a1[N], a2[N], i;
  for(i=0; i<N; i++) {
    a1[i]=input();
    a2[i]=input();
  }
  assume(a1[0]+a2[0]>a2[1]);
  assume(a1[2]+a2[2]>a2[3]);
  ......
  if(a1[98]+a2[98]>a2[1]){
    // Uncoverd
  }
  return 0;
}
```

Fig. 2: motivation.c

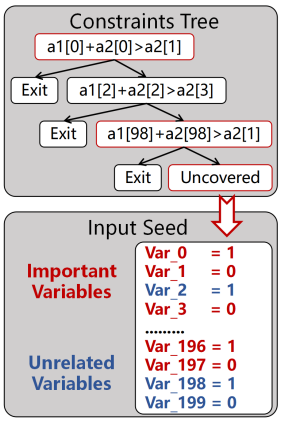

Fig. 3: Constraint tree identification of important variables

pre-analysis, CBMC unrolls loops up to a fixed bound to explore part of the program's state space and generate initial seeds that reach non-trivial paths.

The `FDSE v2` manager first launches the fuzzer to collect seeds achieving new edge coverage. When progress stalls, it switches to the concolic executor, which re-executes recent high-coverage seeds to extract path constraints. These are stored in a persistent constraint binary tree maintained by the manager. The manager then identifies uncovered edges, locates corresponding nodes in the tree, and uses an SMT solver to generate high-quality seeds satisfying the target conditions. These seeds are returned to the fuzzer, which performs selective byte-level mutations guided by the key variables in the constraints. All seeds yielding new coverage are exported as test cases throughout execution.

Demonstration. We illustrate how `FDSE v2` identifies important variable groups to guide fuzzing mutations through a motivating example. Figure 2 shows a program that accepts 200 integer inputs. Suppose `FDSE v2` has only covered the false branch at line 11 and seeks an input that triggers the true branch. The manager invokes an SMT solver, which successfully generates a seed satisfying the condition for the true branch. To further explore the program space beyond line 12, randomly mutating this seed across all 200 input variables would be highly inefficient. Instead, `FDSE v2` analyzes the constraint tree: starting from the condition at line 11, it recursively identifies semantically related constraints. For instance, the variable $a2[1]$ involved in the condition at line 8 overlaps with the condition at line 11. This important variable set $\{a1[0], a2[0], a2[1], a1[98], a2[98]\}$ of related constraints defines a reduced solution space involving only a small subset of relevant variables. The fuzzer then performs selective mutations on this improtant variable group within the subspace, using the SMT-generated seed as a starting point, as illustrated in Figure 3. This strategy avoids wasting computational effort on irrelevant inputs and enables rapid convergence toward uncovered code regions.

3 Strengths and Weaknesses

Compared to prior approaches, `FDSE v2` offers three main advantages. First, its constraint tree eliminats redundant exploration between fuzzing and symbolic execution and significantly improving coverage efficiency, especially on programs with deep or numerically complex guards that pure fuzzing struggle to penetrate. Second, by analyzing data dependencies, `FDSE v2` identifies input variables relevant to uncovered branches and guides mutations toward semantically meaningful regions of the input space. Third, unlike `FDSE`, which suffered from explosive constraint growth on some programs and placed heavy demands on the SMT solver, `FDSE v2` leverages the fuzzer's high-throughput execution. After extracting important variables from the constraint tree, `FDSE v2` performs targeted mutations. This often achieves the same coverage faster than computing exact solutions via the SMT solver, effectively prioritizing rapid guided exploration over costly precise solving.

However, `FDSE v2` has several limitations. First, constructing and maintaining the constraint tree incurs substantial memory and computational overhead, particularly for programs with extensive branching or long execution paths, which can hinder scalability. Second, the effectiveness of guided mutation relies heavily on the accuracy of constraint collection. In programs involving pointer arithmetic or calls to uninstrumented external libraries, path constraints may be lost or incomplete, preventing full identification of improtant variables and thereby reducing mutation efficiency. Finally, `FDSE v2` does not design a specialized search strategy over the constraint tree for bug-finding tasks, which limits its performance in the **Cover-Error** category.

4 Software Project

`FDSE v2` is developed by the National University of Defense Technology. More infomation is publicly available at https://github.com/zbchen/fdse-test-comp. The tool integrates two core components: a concolic executor built on `SymCC` [8], and a fuzzing engine implemented in C++ using LLVM 10.0.1 [5]. Coordination between these engines is managed by a central module that employs the Z3 SMT solver [6], while the command-line interface is implemented in Python.

`FDSE v2` participated in the **Cover-Branches** and **Cover-Error** categories in Test-Comp 2026. Integration with BenchExec is provided through the module `fdse.py`, with benchmark settings defined in `fdse.xml`. Users can invoke the tool as follows:

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

During execution, all benchmarks are treated as 64-bit programs, and the engine prioritizes maximizing code coverage. Generated test suites are written to the directory `fdseoutput/test-suite`, with each suite containing a metadata XML file and one or more test-case XML files conforming to the Test-Comp format.

5 Data Available

The artifact of `FDSE v2` used for the Test-Comp 2026 submission is archived on Zenodo[1] and distributed under the Apache-2.0 license. To support full reproducibility, the official Test-Comp 2026 website[2] provides access to the competition benchmarks, execution environment, validation scripts, and the `FDSE v2` binaries employed during evaluation. All data necessary to replicate the reported results, including tool configuration, benchmark instances, and output formats, are publicly accessible through these resources.

This research was supported by National Key R&D Program of China (No. 2024YFF0908003) and the NSFC Program (No. 62172429, 62032024).

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. Beyer, D.: Evaluating tools for automatic software testing: Test-Comp 2026. In: Proc. FASE. LNCS 16504, Springer (2026)
3. Jiang, L., Yuan, H., Wu, M., Zhang, L., Zhang, Y.: Evaluating and improving hybrid fuzzing. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) (2023)
4. Kroening, D., Schrammel, P., Tautschnig, M.: Cbmc: The c bounded model checker. ArXiv **abs/2302.02384** (2023)
5. LLVM: Https://llvm.org
6. 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)
7. Peng, C., Hao, C.: Angora: Efficient fuzzing by principled search. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 711–725 (2018)
8. Poeplau, S., Francillon, A.: Symbolic execution with symcc: Don't interpret, compile! In: USENIX Security Symposium (2020)
9. Sen, K.: Concolic testing. In: Proceedings of the 23th IEEE/ACM international conference on Automated software engineering (2007)
10. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: Qsym : A practical concolic execution engine tailored for hybrid fuzzing. In: USENIX Security Symposium (2018)
11. Zhang, G., Shuai, Z., Ma, K., Liu, K., Chen, Z., Wang, J.: FDSE: Enhance symbolic execution by fuzzing-based pre-analysis (competition contribution). In: Proc. FASE. pp. 304–308. LNCS 14573, Springer (2024). https://doi.org/10.1007/978-3-031-57259-3_16

[1] https://zenodo.org/records/17900875
[2] https://test-comp.sosy-lab.org/2026