IFSE: Taming Closed-box Functions in Symbolic Execution via Fuzz Solving

Qichang Wang[1†], Chuyang Chen[2†], Ruiyang Xu[1], Haiying Sun[1*],
Chengcheng Wan[1], Ting Su[1*], Yueling Zhang[1], Geguang Pu[1]
[1]Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China
[2]The Ohio State University, USA
{qichang, xry}@stu.ecnu.edu.cn, chen.13875@osu.edu, {hysun, ccwan, tsu, ylzhang, ggpu}@sei.ecnu.edu.cn

***Abstract*—Modern symbolic execution techniques face the challenge of handling *closed-box (CB)* functions (*e.g.*, system calls, library functions) whose source code is unavailable. One interesting solution in the literature is deferred concretization with fuzz solving. However, no open-source implementation of such techniques exists, and thus it is difficult to evaluate and investigate the effectiveness. In this paper, we present IFSE (Integrating Fuzz Solving into Symbolic Execution), an open-source tool implementing the relevant techniques on top of KLEE to handle the CB functions in symbolic execution. We evaluated IFSE on GNU Coreutils. The results show that IFSE achieves the line and branch code coverage improvement by 28.3% and 12.2% respectively compared to vanilla KLEE. The satisfaction rate of fuzz solver achieves 80.2%, demonstrating its ability to reason CB function related constraints. IFSE is publicly available at *https://github.com/ecnusse/ifse* and a demonstration video is at *https://youtu.be/xMv6_MOlE-I*.**

***Index Terms*—symbolic execution, fuzzing, closed-box function.**

I. INTRODUCTION

Symbolic execution (SE) is a classic program analysis technique to automate test generation [1]. However, when handling real-world programs, SE is difficult to analyze *closed-box (CB) functions* (*e.g.*, system calls, library functions) whose source code is usually not available and can only be accessed by feeding certain inputs and receiving corresponding outputs.

To overcome this difficulty, modern SE engines (*e.g.*, KLEE [2]) employ the *concretization* policy [3]: when encountering a CB function, the policy assigns concrete values to the symbolic arguments of this CB function, natively executes this CB function and binds the concrete output value to the return variable. This policy enables SE to continue the analysis without knowing this CB function's formal semantics, but may unfortunately lead to coverage loss and path divergence [4].

We use the example in Listing 1 to illustrate the *concretization* policy [3] and the consequence. Listing 1 shows a simplified code snippet from the `pwd` program in GNU Coreutils [5], a widely used core tool program collection in Unix-like operating system. It parses the arguments `argc` and `argv` in `parse_option()` (lines 1-9), and invokes different functions (*e.g.*, `logical_getcwd()` at line 25). Assume `strcmp()` at line 5 is a CB function. The concretization policy in KLEE will assign a concrete value "" (an empty string)

*Haiying Sun and Ting Su are the corresponding authors.
†Qichang Wang and Chuyang Chen contributed equally to this research.

```
1  int parse_option(int argc, char **argv) {
2    if (argc == 2) {
3      if (strcmp(argv[0], "--physical") == 0)
4        return 'P';
5      if (strcmp(argv[1], "--logical") == 0)
6        return 'L';
7    }
8    return -1;
9  }
10
11 int main(int argc, char **argv) {
12   make_symbolic(argc);  // Make argc symbolic
13   make_symbolic(argv);  // Make argv symbolic
14   bool logical = false;
15   int c = parse_option (argc, argv);
16   switch (c) {
17     // Set physical mode
18     case 'P': logical = false; break;
19     // Set logical mode
20     case 'L': logical = true; break;
21     // Invalid option, show usage
22     default: usage (EXIT_FAILURE);
23   }
24   // Get logical working directory
25   if (logical) wd = logical_getcwd();
26   else wd = physical_getcwd();
27   // ... Rest of program
28 }
```

Listing 1. An illustrative example on the concretization policy

to the symbolic argument `argv[1]` based on the current path constraint (*i.e.*, `argc==2`). It then natively executes `strcmp()` and binds the concrete output value `-1` to the return variable. Since the return variable of `strcmp()` has been fixed to `-1`, KLEE cannot explore the true branch of line 5, thus missing the chance of exploring function `logical_getcwd()` at line 25. But the true branch is actually satisfiable when `argv[1]=="--logical"`. This consequence is caused by the limitaiton of *concretization* — the constraints between the symbolic arguments and the return variable of CB functions (`strcmp()` in this case) are not maintained.

One interesting solution to mitigating the preceding limitation is *deferred concretization* [4] with *fuzz solving* [6]–[8]. *Deferred concretization* preserves the CB function signature and the associated symbolic and concrete values within the path constraints. *Fuzz solving* is employed to solve the constraints related to the CB functions since existing SMT solvers are difficult to solve such constraints. Specifically, the constraints are translated to a program which contains an

`abort()` statement. This `abort()` statement is reachable only when a satisfiable assignment of the constraints is found.

However, to our knowledge, none of these techniques is open-sourced[1]. As a result, it is difficult for researchers and engineers to evaluate and explore their effectiveness. To this end, we took the substantial effort to implement the techniques of *deferred concretization* and *fuzz solving* on top of KLEE. We have made our realization (named IFSE) publicly available to benefit the community in investigating the effectiveness of such techniques to tackle the challenge of closed-box functions in symbolic execution. We compared IFSE and the vanilla KLEE on GNU Coreutils by treating C string library functions as CB functions. The results show that IFSE achieve the line and branch code coverage improvement by 28.3% and 12.2% respectively. It also demonstrates the remarkable ability of our fuzz solver to reason CB function-related constraints and the notable impact of the optimizations we implemented in IFSE.

II. Our tool IFSE

A. Workflow Overview

Figure 1 illustrates the workflow of IFSE, comprising three main components: the symbolic execution (SE) engine, an SMT solver, and a fuzz solver. IFSE also includes several auxiliary components to enhance its performance in different aspects: *filter* (filtering CB functions related constraints), *predictor* (estimating constraint satisfiability), and *splitter* (excluding irrelevant constraints), which will be detailed later.

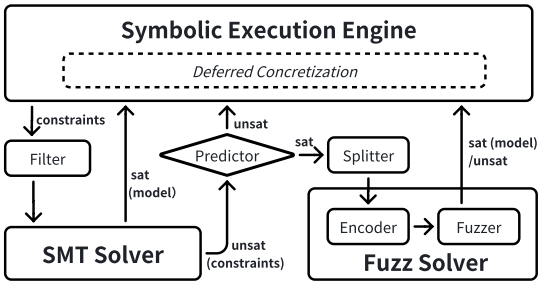

Fig. 1. Workflow of IFSE

Given a program, the SE engine collects path constraints along program paths. Upon a CB function call, it utilizes *deferred concretization* to maintain both the concrete and symbolic information of the CB function within the path constraint. When encountering a branch, IFSE first employs the SMT solver for constraint solving. When the SMT solver returns `UNSAT`, IFSE does not discard the path as regular SE. Instead, IFSE tries to solve the constraint with a *fuzz solver*.

B. Core Algorithm

IFSE integrates various existing techniques [4], [8], [9] to address the CB function challenge. As shown in Algorithm 1, IFSE starts from an initial program state represented as (pc_0, π_0, σ_0): where pc points to the next program instruction; π denotes the path constraint of current path; and σ maps program variables to their symbolic values or concrete values (lines 1-2). IFSE then repetitively picks a state (pc, π, σ) from

[1]We contacted the authors of [4] but they did not plan to release the tool.

Algorithm 1 Core Algorithm of IFSE

```
1: pc_0 = l_0, π_0 = true, σ_0 = ∅
2: W ← (pc_0, π_0, σ_0)
3: while W ≠ ∅ do
4:     (pc, π, σ) ← W
5:     instr = getInstruction(pc)
6:     if instr is a CB function call (r = f(a_1, a_2, ...)) then
7:         c_1, c_2, ... = getConcrete(π, σ, a_1, a_2, ...)
8:         c = execute(f, c_1, c_2, ...)
9:         Assume σ[a_1] = s_1, σ[a_2] = c_2, ...
10:        σ' ← (σ[r → ⟨s, c⟩, a_1 → ⟨s_1, c_1⟩, a_2 → c_2, ...])
11:        φ = (⟨s, c⟩ == f(⟨s_1, c_1⟩, c_2, ...))
12:        W ← (next(pc), π ∧ φ, σ')
13:    else if instr is a branch on condition p then
14:        if SMTSolve(filter(π ∧ p)) then          ▷ true branch
15:            W ← (next(pc), (π ∧ p), σ)
16:        else
17:            π' = optimize(π, p)
18:            (res, τ) = FuzzSolve(π' ∧ p)
19:            if res == SAT then
20:                W ← (next(pc), (π ∧ p), σ[τ])
21:            end if
22:        end if
23:        if SMTSolve(filter(π ∧ ¬p)) then         ▷ false branch
24:            W ← (next(pc), (π ∧ ¬p), σ)
25:        else
26:            π' = optimize(π, ¬p)
27:            (res, τ) = FuzzSolve(π' ∧ ¬p)
28:            if res == SAT then
29:                W ← (next(pc), (π ∧ ¬p), σ[τ])
30:            end if
31:        end if
32:    else
33:        ... ▷ handle other types of instructions as traditional SE
34:    end if
35: end while
```

the worklist (lines 3-5) and handles the state as traditional SE. However, IFSE executes differently when encountering CB function calls and branch statements.

When encountering a CB function call $r = f(a_1, a_2, ...)$, IFSE provides concrete values c_i for each argument a_i through solving path constraint π, natively executes f with these concrete values and gets a concrete result c (lines 6-8) as traditional SE does. To address the problems caused by *concretization*, instead of directly assigning c to variable r in σ, IFSE introduces a new symbolic value s for r and retains s and c simultaneously in a tuple $\langle s, c\rangle$, which is proposed as *symcrete* value in *deferred concretization* [4]; it also updates symbolic arguments of f to *symcrete* values, thereby recording all concrete values of arguments used to natively execute f (lines 9-10); finally, it creates a CB function expression ϕ which retains the function signature of f with related *symcrete* values and inserts ϕ into π (lines 11-12).

When encountering a branch on condition *p*, IFSE attempts to explore both arms of the branch (lines 13-31). To keep performance, IFSE first attempts to satisfy the path constraint $\pi \wedge p$ with existing concrete values by the SMT solver. This involves replacing all *symcrete* values with their concrete values and removing all CB function expressions ϕ in $\pi \wedge p$ by the `filter()`. If the SMT solver deems the filtered constraint satisfiable, it indicates existing concrete values satisfy $\pi \wedge p$

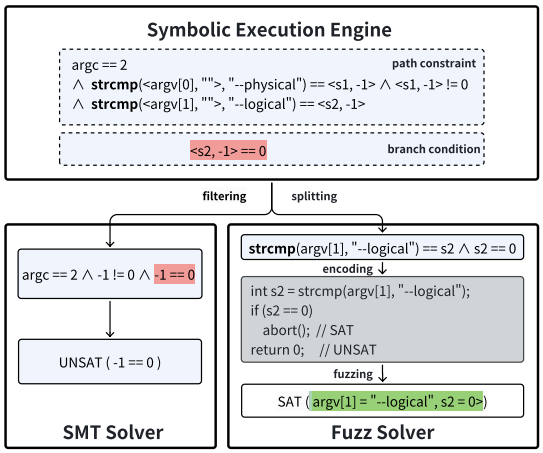

Fig. 2. Process of IFSE in handling the path constraint at line 5 in Listing 1

(lines 14-15). Otherwise, IFSE invokes `optimize()` to optimize π to π' (line 17) by *predictor* and *splitter* successively, detailed in section II-C. IFSE then solves $\pi' \wedge p$ using the fuzz solver (line 18). The fuzz solver translates $\pi' \wedge p$ into a program containing an `abort()` statement, which is reachable only when the program input corresponds to a satisfying assignment of the path constraint. It then keeps mutating the input until satisfying $\pi' \wedge p$ or reaching timeout. If the fuzz solver returns `SAT`, IFSE will create a new program state and update the satisfying assignment τ in σ (lines 19-21). The other arm of the branch is handled similarly (lines 23-31).

Exmaple Explanation. Now we explain how IFSE handles the CB function `strcmp()` at line 5 in Listing 1. As shown in Figure 2, when encountering `strcmp()` in line 5, IFSE provides a concrete value "" (an empty string) for the symbolic argument `argv[1]`. It then natively executes `strcmp()` using "" and gets -1 as the result. As mentioned before, IFSE will introduce a fresh symbolic value `s2` for the return variable of `strcmp()`, retain them in a *symcrete* value ⟨`s2,-1`⟩ and update symbolic argument `argv[1]` as ⟨`argv[1],""`⟩. As a result, the CB function expression `strcmp(`⟨`argv[1],""`⟩`, "--logical") ==` ⟨`s2,-1`⟩ containing both function signature and related *symcrete* values is added into path constraint.

After the above process, IFSE encounters a branch with condition ⟨`s2,-1`⟩ `== 0` in line 5. The `filter()` first replaces all *symcrete* values with their concrete values. For example, ⟨`argv[1],""`⟩ is replaced by "" and ⟨`s2,-1`⟩ is replaced by -1. Then all CB function expressions will be removed, with the remaining constraint as `(argc == 2 ∧ -1 != 0 ∧ -1 == 0)`. The SMT solver returns `UNSAT` because the concrete value -1 of `s2` (highlighted in red) cannot fulfill the constraint.

Before invoking the fuzz solver, IFSE tries to reduce the scale of path constraint by only including relevant constraints of branch condition. As shown in Figure 2, IFSE splits the path constraint and only retains `strcmp(`⟨`argv[1],""`⟩`, "--logical") ==` ⟨`s2,-1`⟩ which is related to the branch condition ⟨`s2, -1`⟩ `== 0`. Then IFSE discards those old concrete values by replacing *symcrete* values to their symbolic values. The fuzz solver then encodes the optimized constraints into a program (the grey part of Figure 2) with `argv[1]` as input and keeps mutating `argv[1]` until encountering the `abort()` statement or reaching timeout. If the former occurs, it returns `SAT` with the corresponding assignments of `argv[1]` and `s2` as a satisfiable assignment of the constraint (highlighted in green); otherwise, it returns `UNSAT`.

C. Optimizations

We have adopted various optimizations to enhance the practical viability of IFSE:

Constraint Splitter. Many constraint terms are irrelevant to the current branch condition as they lack variables that determine its satisfiability and thus do not affect constraint satisfiability [8], [9]. Hence, We simplify the path constraint by excluding those terms irrelevant with current branch condition.

Unsat Predictor. If the fuzz solver fails to solve a path constraint at a certain branch multiple times, the constraint is likely unsatisfiable. To avoid unnecessary solving, we implement the *unsat predictor* mentioned in *deferred concretization* [4] to track historical results of the fuzz solver at each branch and directly return `UNSAT` at a branch if the fuzz solver consistently returns `UNSAT` before.

Memory Tracker. The program generated by constraints translation runs independently within its own memory space which is isolated from SE engine. Thus, correctly collecting and transferring the memory-related data in the SE engine to the program is a considerable challenge. To address this, IFSE tracks memory-related data and injects memory manipulating statements with the data into the program. This enables IFSE to handle CB functions with complex data types like pointers.

Others. We also implemented other optimizations to further improve the performance of IFSE, including constraint simplification and conflict detection.

III. Implementation And Evaluation

We built IFSE on KLEE [2] (version 3.0, commit dfa53ed), used Z3 [10] (version 4.8.15) as the SMT solver and implemented a fuzz solver KRPK based on LibFuzzer [11]. KRPK now supports core, array, bit vector and floating point theories and can also support other fuzzers. The implementation of IFSE includes 4,600 lines of C++ in KLEE and 11,000 lines of Rust for KRPK. We took around ten months to build, optimize and validate the implementation of IFSE.

A. Evaluation Design and Setup

We evaluated IFSE and vanilla KLEE on 79 GNU Coreutils programs [5], a common SE benchmark with many CB function calls. We measured the line and branch coverage to assess IFSE's path exploration ability, and the satisfaction rate ($\frac{\#\text{SAT}}{\#\text{Fuzz Solving}}$) to eavlaute the performance of our fuzz solver. We also investigated the correlation between the satisfaction rate and IFSE's coverage improvement over KLEE, as well as the impact of two major optimizations, *constraint splitter* and *unsat predictor*. We chose C string library functions as CB functions in our experiment and compiled the tested programs with uclibc support while holding back the definitions of these string functions. These string functions are frequently

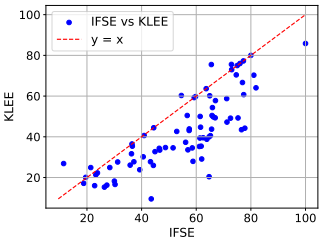

Fig. 3. Comparison between KLEE and IFSE in terms of line coverage

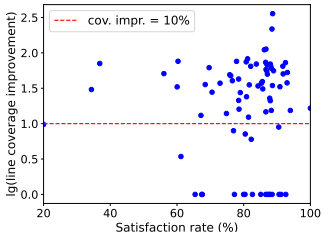

Fig. 4. Correlation between line coverage impr. and satisfaction rate

used in GNU Coreutils and contain complex features like pointers. IFSE and KLEE were set with a 4-hour timeout and the fuzz solver was set with an 8-second timeout per program. We repeated the experiment ten times and averaged the results. The evaluation was performed on 64-bit Ubuntu 22.04 equipped with a 64-core AMD Ryzen Threadripper PRO 3995WX CPU and 128 GB of RAM. Detailed experimental data is available in the IFSE GitHub repository.

B. Evaluation Results

Code Coverage. Figure 3 compares the line coverage achieved by IFSE (x-axis) and KLEE (y-axis) across the 79 tested programs (represented by blue points). The points below the line `y = x` indicate that IFSE outperforms KLEE in line coverage. On 63 programs, IFSE improved line coverage by 0.7∼357.9%. On average, IFSE achieved a line coverage of 54.8%, a 28.3% improvement over KLEE's 42.7%. Meanwhile, IFSE achieved the average branch coverage of 64.2%, while KLEE achieved 57.2%, with 12.2% coverage improvement. These results show the effectiveness of IFSE. We further unioned the achieved line coverage of IFSE and KLEE, and obtained the average line coverage of 55.9%. This union result achieved the line coverage improvement by 30.9% compared to KLEE. It indicates that the majority of new program paths are indeed explored by IFSE rather than KLEE.

Satisfaction Rate. The fuzz solver KRPK in IFSE achieves an average satisfaction rate of 80.2%. Figure 4 shows the correlation between KRPK's satisfaction rate and IFSE's line coverage improvement over KLEE on each tested program. It shows that KRPK's satisfaction rate has a positive impact on the performance of IFSE (most points situate at the upper right corner). We also notice some points at the lower right corner. This is because such programs may not contain branches with string functions on which KLEE has achieved saturated coverage without any room for further exploration by IFSE.

Optimization. We studied the impact of two optimizations, *constraint splitter* and *unsat predictor*, on the performance of IFSE. The results show that *splitter* improves the average line coverage by 37.3%, the *predictor* 2.9%, and their combination yields a 43.8% improvement. It indicates that the two optimizations are complementary as *predictor* may assess the satisfiability of large constraints more accurately when these constraints are first scaled down by *splitter*.

IV. Related Work

Pandey *et al.* [4] propose the idea of *deferred concretization* to tackle the challenge of closed-box functions in symbolic execution. Specifically, they use *fuzz solving* [6] to solve the path constraints related to closed-box functions when traditional SMT solvers fail. Unfortunately, their tool COLOSSUS is not publicly available. Our tool IFSE follows their basic idea and has been made publicly available to allow replication and further investigation. IFSE has also been optimized during our evaluation. For example, we optimize fuzz solving by splitting path constraints [8], [9], which is shown to be important in practice. IFSE also allows the evaluation of some new constraint solving techniques like S$\overline{A}$DHAK [8] for closed-box functions, which otherwise is impossible. There are also some other tools (*e.g.*, JFS [6], FUZZY-SAT [7], FUSE [9]) using fuzz solving to build dedicated constraint solvers for symbolic execution. But none of them targets closed-box functions related constraints. For example, JFS [6] targets floating-point related constraints; FUSE [9] targets floating-point and non-linear related constraints.

V. Conclusion

To tackle the challenge of CB functions in symbolic execution, we propose IFSE, an open-source tool following the idea of deferred symbolic execution and fuzz solving. IFSE show its effectiveness in handling CB functions in real-world programs from GNU Coreutils. We have made IFSE publicly available to benefit the community and future research.

Acknowledgment

We thank the anonymous ICSE reviewers. This work was supported in part by National Key Research and Development Program (Grant 2022YFB3104002) and Shanghai Trusted Industry Internet Software Collaborative Innovation Center; National Natural Science Foundation of China for Young Scientists (Grant 62202166, NSFC).

References

[1] C. Cadar and K. Sen, "Symbolic Execution for Software Testing: Three Decades Later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.
[2] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." in *OSDI*, vol. 8, 2008, pp. 209–224.
[3] R. David, S. Bardin, J. Feist, L. Mounier, M.-L. Potet, T. D. Ta, and J.-Y. Marion, "Specification of Concretization and Symbolization Policies in Symbolic Execution," in *ISSTA*, 2016, pp. 36–46.
[4] A. Pandey, P. R. G. Kotcharlakota, and S. Roy, "Deferred Concretization in Symbolic Execution via Fuzzing," in *ISSTA*, 2019, pp. 228–238.
[5] "GNU Coreutils," https://www.gnu.org/software/coreutils/, 2023.
[6] D. Liew, C. Cadar, A. F. Donaldson, and J. R. Stinnett, "Just Fuzz It: Solving Floating-Point Constraints Using Coverage-Guided Fuzzing," in *ESEC/FSE*, 2019, pp. 521–532.
[7] L. Borzacchiello, E. Coppa, and C. Demetrescu, "Fuzzing Symbolic Expressions," in *ICSE*. IEEE, 2021, pp. 711–722.
[8] S. K. Muduli and S. Roy, "Satisfiability Modulo Fuzzing: A Synergistic Combination of SMT Solving and Fuzzing," *PACMPL*, vol. 6, no. OOPSLA, pp. 1236–1263, 2022.
[9] G. Zhang, Z. Chen, Z. Shuai, Y. Zhang, and J. Wang, "Synergizing Symbolic Execution and Fuzzing By Function-level Selective Symbolization," in *APSEC*. IEEE, 2022, pp. 328–337.
[10] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*. Springer, 2008, pp. 337–340.
[11] "LibFuzzer," https://llvm.org/docs/LibFuzzer.html, 2023.