Synergizing Symbolic Execution and Fuzzing By Function-level Selective Symbolization

Guofeng Zhang*, Zhenbang Chen*§, Ziqi Shuai*†, Yufeng Zhang‡, and Ji Wang*†§
*College of Computer, National University of Defense Technology, ChangSha, China
†State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China
‡College of Computer Science and Electronic Engineering, Hunan University, Changsha, China
Email: {zhangguofeng16, zbchen, szq}@nudt.edu.cn, yufengzhang@hnu.edu.cn, wj@nudt.edu.cn
§Corresponding author

***Abstract*—Constraint solving and environment modeling are two challenging problems for symbolic execution. When a program contains non-linear expressions, it is difficult for symbolic execution to explore the program's whole path space due to the high complexity of the constraint solving for the non-linear constraints. Besides, when the program uses a third-party library and the source code of the library is not available, the symbolic execution of the program often under-approximates the analysis by concrete execution or over-approximates by introducing new symbolic variables, which may fail to explore the whole path space or introduce false alarms, respectively. This paper proposes FUSE, a framework of synergizing symbolic execution and fuzzing by function-level selective symbolization to tackle these problems. First, FUSE collects the path constraints of each function selectively and introduces symbolic function invocation expressions for the complex or third-party functions. Then, FUSE combines SMT solving and fuzzing to solve the path constraints. We have implemented FUSE on the start-of-the-art symbolic execution engine KLEE. The experimental results demonstrate that FUSE effectively and efficiently improves the code coverage. Compared with the state-of-the-art, FUSE achieves 6.6x speedups for achieving the same code coverage.**

***Index Terms*—Symbolic Execution, Constraint Solving, Fuzzing, Environment Modeling**

I. INTRODUCTION

Symbolic execution [1]–[4] provides a precise method for analyzing programs. Symbolic execution analyzes a program by executing the program in terms of symbolic values and checking the feasibility of paths by constraint solving [5], [6]. Thanks to the recent advancements in constraint solving [5], [6], symbolic execution has been successfully applied to tackle many challenging software engineering problems, such as automatic software testing [7], [8] and automatic bug detection [9], [10], to name a few. On the other hand, symbolic execution itself has also evolved to be more effective and efficient for these software engineering problems.

Although there exist many successful applications, symbolic execution still faces many challenging problems [4], including path explosion, constraint solving, environment modeling, *etc*. These problems limit the further applications and advancements of symbolic execution. Constraint solving dominates the time of symbolic execution [11], [12]. Especially when the program under analysis contains non-linear expressions, *e.g.*, non-linear floating-point computations, the solving of the constraints related to these expressions is very difficult or even impossible [5]. If the constraint solving fails, symbolic execution may fail to explore some program paths, which causes the missing of bugs or lower coverage. Environment modeling is an inherent challenge faced by all program analysis techniques [13]. When the program needs to invoke the methods in third-party libraries or system libraries (called environment methods) and the methods are only in binary format or pretty complex (*e.g.*, super-optimized), symbolic execution may fail to collect the path constraints in the methods or fail to solve the complex constraints produced by the methods, which may yield both false positives and false negatives in bug detection.

Existing methods of tackling the challenges of constraint solving and environment modeling in symbolic execution can be divided into two groups: *under-approximation* and *over-approximation*. For under-approximation approaches [14], the main idea is to use concrete values to simplify the complex constraints [15], [16] (*e.g.*, non-linear arithmetic constraints) or execute the environment methods concretely [2], [3]. These approaches may fail to explore some program paths or make symbolic execution diverge [17]. Besides, over-approximation approaches abstract the path constraints by removing the complex constraints [18], [19] or introducing new symbolic variables for environment methods [8] (*e.g.*, a new symbolic variable for the return value of each invocation), which may cause false alarms and also make symbolic execution diverge. In practice, we have observed that these two kinds of approaches need improvement, and a better tradeoff is desirable.

Besides symbolic execution, fuzzing [20], [21] is also effective for testing programs. Fuzzing may outperform SAT/SMT-based constraint solving [6] on complex constraints, *e.g.*, floating-point constraints [22]. We have observed that symbolic execution and fuzzing are good at analyzing different program parts. For example, symbolic execution is preferable for the parts in which only linear expressions exist but is doomed when very complex constraints exist. On the other hand, fuzzing may be better for the parts in which complex constraints exist and the input spaces are large (or many environment methods are used), but fuzzing is challenged by the parts whose input spaces are small (*e.g.*, magic number expression). Hence, it is desirable to synergize symbolic execution and fuzzing to analyze different parts of the program.

Based on this insight, we propose synergizing symbolic execution and fuzzing for analyzing a program to tackle the problem of constraint solving and environment modeling. Our key idea is to separate the program into *easy* and *hard* parts in the granularity of program functions. For the easy parts, *e.g.*, functions with linear expressions, we use symbolic execution and employ SAT/SMT-based constraint solving to analyze them. On the other hand, for the complex parts, *e.g.*, functions with very complex constraints or environment functions, we use fuzzing to analyze them. This synergy makes symbolic execution more scalable with respect to the challenge of constraint solving. Besides, this synergy achieves a better tradeoff for environment modeling compared with the under-approximation and over-approximation approaches.

More specifically, we propose a dynamic symbolic execution (DSE) [2], [3] framework FUSE that tackles the problems of constraint solving and environment modeling in a unified way. FUSE selectively collects the path constraints of the functions in the program. If the function is in the hard part of the program, FUSE does not collect the path constraints but uses a symbolic method invocation expression for representation. We propose a method for on-the-fly determining whether a function is easy or hard with respect to the hardness of constraint solving. Then, FUSE combines SAT/SMT-based constraint solving and fuzzing to solve the path constraints. For the parts without symbolic method invocations, FUSE employs the SMT solver; for the parts with symbolic method invocations, FUSE converts the solving to a fuzzing problem and employs a fuzzer for solving. We have implemented FUSE on KLEE [7] and JFS [22]. The extensive experiments on two different kinds of benchmarks indicate that our method is effective and efficient for improving the code coverage on complex benchmarks. The main contributions are as follows.

- We propose FUSE, a synergistic framework for dynamic symbolic execution (DSE) and fuzzing. FUSE separates the program into two parts at the function level and combines SMT solving and fuzzing at the constraint solving level to improve symbolic execution's efficiency and effectiveness.
- We propose a method that classifies function on-the-fly during symbolic execution to collect the path constraints selectively.
- We propose a method for solving the path constraints containing symbolic method invocations by combining SMT solving and fuzzing.
- We have implemented FUSE on the state-of-the-art symbolic execution engine KLEE. We applied our prototype to two representative benchmarks, *i.e.*, GNU Scientific Library (GSL) [23] and Coreutils [24]. The results demonstrate that: on the GSL benchmark, compared with the state-of-the-art [25], FUSE, on average, achieves 18.1% relative increase in statement coverage and 6.6x speedups for achieving the same code coverage under BFS; on Coreutils, FUSE achieves the same performance as the original DSE engine.

```
 1  #include <assert.h>
 2  #ifdef HAS_SOURCE
 3  double sin(double x) {
 4   ... // Implementation based on taylor series
 5  }
 6  #else
 7  extern double sin(double);
 8  #endif
 9  double foo(double x, double y) {
10    double z, sinx;
11    sinx = sin(x);
12    if (y == 3.0) {
13      z = 100 * sinx;
14      if (z > 99)
15        assert(false);
16    } else
17      z = 10 * sinx;
18    return z;
19  }
```

Fig. 1. A motivation example program.

II. ILLUSTRATION

This section uses an illustrative example to motivate FUSE. The code snippet in Figure 1 gives a simple C function `foo` that contains a bug, *i.e.*, Line 9. `foo` takes two input variables `x` and `y`. In Line 11, the function invokes `sin`, one of the most common trigonometric functions, to compute the input variable `x`'s value. To demonstrate our method, we use a predefined macro `HAS_SOURCE` to control the availability of `sin`'s source code. Then, the following computations rely heavily on the computed result `sinx`.

We use dynamic symbolic execution to analyze the example program since FUSE is implemented on a DSE engine. Note that our method is not limited to DSE. First, we will show that the original DSE engine is hard to cover the buggy line, no matter whether `HAS_SOURCE` is defined, and then elaborate on the power of FUSE.

A. Original Dynamic Symbolic Execution

The original dynamic symbolic execution often starts with an initial concrete input. After symbolizing input variables, the DSE engine executes the program under test on such input both concretely and symbolically. When doing the symbolic execution, the DSE engine will collect symbolic value of the branch being executed in concrete execution at each branch point and construct a path constraint. Then, one of the branch predicates is negated (also named flipped) in the path constraint according to specified search strategy. The negated path constraint is solved by the underlying constraint solver. If satisfiable, the solution will be used as the next concrete input. Otherwise, the DSE engine will try another flip. The process is repeated until the path space is exhausted or some criterion is satisfied.

The major strength of dynamic symbolic execution is that it can utilize concrete values from concrete execution to simplify symbolic expressions whenever the symbolic reasoning is hard (*e.g.*, the symbolic expression is non-linear) or impossible (*e.g.*, the source code is not available), which is considered as an under-approximation approach. The under-approximation approach enables the analysis to proceed at the cost of sacrificing completeness, *i.e.*, some paths are missing. In this example, we first suppose the invoked `sin` is from a third-party binary library, *i.e.*, `HAS_SOURCE` is not defined. In such a case, `sin`'s behaviors can not be fully analyzed. Existing DSE engines usually use the concrete computation result of `sin` to under-approximate `sin`'s behaviors. For example, suppose `foo`'s initial inputs are $\{x \mapsto 0.0, y \mapsto 0.0\}$ and both of x and y are symbolized. During DSE, `sinx` is not tainted by x and only has the concrete value `sin(0.0)`, *i.e.*, 0.0. Hence, when the first execution terminates, the execution's path constraint is as follows.

$$y \neq 3.0$$

Then, the path constraint is flipped and solved to get the next inputs. Since x is a free variable, we get the following inputs $\{x \mapsto \triangle, y \mapsto 3.0\}$, where $\triangle$ represents a random value. It is quite hard for such inputs to cover the buggy line, where the possibility is around 4.5%.

On the other hand, some DSE engines apply over-approximation approaches to tackle external libraries [8]. A typical over-approximation approach uses a new symbolic variable to over-approximate the behaviors of `sin`. For example, suppose that we have $\{x \mapsto 0.0, y \mapsto 3.0\}$ as the initial inputs now. The variable `sinx` is assigned with a new symbolic variable (denoted by sin_x). Then, the collected path constraint is as follows.

$$y = 3.0 \wedge 100 * sin_x \leq 99$$

Suppose we employ DFS search strategy here. Hence, the last branch predicate is flipped and the flipped path constraint is as follows.

$$y = 3.0 \wedge 100 * sin_x > 99$$

Then, the underlying constraint solver is invoked to solve the flipped path constraint and tries to generate test case to reach the bug. Unfortunately, the solution of the flipped constraint is very likely to be $\{sin_x \mapsto \nabla, y \mapsto 3.0\}$, where ∇ represents a value that is greater than 1.0. The value is invalid due to the semantics of `sin`, *i.e.*, the variable `sinx`'s value must be inside the interval $[-1.0, 1.0]$. Hence, the lack of semantics in over-approximation approaches leads to an impossible value.

Even if `HAS_SOURCE` is defined, *i.e.*, the source of `sin` is available, the existing DSE engines still face challenges in analyzing the example program, which is due to the complexity in the implementation of `sin`. Usually, a trigonometric function is implemented using Taylor's approximation series [26], where tons of non-linear operations are involved. Even worse, the computations are based on floating-point values. When the DSE engine delves into the `sin` function, it will collect many non-linear floating-point constraints and pose a huge burden to the underlying constraint solver. The solver may always tend to time out and return `UNKNWON`. As a result, the DSE engine fails to produce the valid inputs to cover the buggy line.

B. Demonstration of FUSE

This paper proposes FUSE, a framework that synergizes symbolic execution and fuzzing by function-level selective symbolization to tackle the above problems. The key of FUSE is to identify *complex functions* in program. A typical complex function is a function with non-linear constraints or an environment function. We propose a method to identify complex functions on-the-fly during symbolic execution, which will be presented in Section III. For the example program, `sin` is a complex function, no matter whether the macro is defined.

During symbolic execution, each time a complex function is invoked, FUSE builds a *symbolic function invocation* expression for it in the path constraint. For example, FUSE builds the following symbolic function invocation expression of `sin` in Line 11 for all paths.

$$sin(x)$$

The symbolic function invocation expression then is propagated to other symbolic expressions. For example, suppose that we have the same initial inputs as the previous over-approximation approach, FUSE will generate the following path constraint that is a bit different as before.

$$y = 3.0 \wedge 100 * sin(x) \leq 99$$

Please note that the symbolic variable sin_x in previous constraint is replaced by the symbolic function invocation expression $sin(x)$, which preserves the semantics of sin. Similarly, under the DFS strategy, we get the following flipped constraint.

$$y = 3.0 \wedge 100 * sin(x) > 99$$

The constraint is then sent to the underlying constraint solver. Although we can consider the named symbolic function invocation as uninterpreted function (UF) at the level of SMT solving, we do not use a UF theory SMT solver to solve the above constraint, where the semantics of complex functions cannot be utilized. Instead, we design a combined solver of SMT solving and fuzzing in FUSE to handle such a constraint. The first step is to split the constraint into easy and complex parts. The criterion is whether there exist symbolic function invocations in the part. Then we get the following partition.

$$\underbrace{y = 3.0}_{easy} \wedge \underbrace{100 * sin(x) > 99}_{complex}$$

The easy part is a magic number comparison that is friendly to SMT solver rather than fuzzing, while the complex part is on the other side of the coin. Therefore, FUSE combines the strength of two powerful solving techniques. SMT solving can solve the easy part quickly, and fuzzing can obtain the semantics of complex functions through the binary code, despite the source code's unavailability. In fact, *the solving of the above constraint takes less than **0.2** second in* FUSE*'s implementation*. The solving result is satisfiable, and the solution produced by the combined solver can lead the program execution to Line 9, where FUSE finally finds the bug.

$$
\begin{array}{rcl}
P & ::= & S^* \\
S & ::= & [l:]ST \\
ST & ::= & V \leftarrow E \mid \mathit{if}\ C\ \mathit{goto}\ l \mid V = f([V(,V)^*]) \\
 & & \mid \mathit{return}\ V \mid \mathit{halt} \\
V & ::= & v \mid *v \\
E & ::= & V \mid \&V \mid \mathit{const} \mid V\ \mathit{op}\ V \mid \mathit{input}() \\
 & & (\mathit{op} \in \{+,-,\times,/\ldots\}) \\
C & ::= & V\ \mathit{cmp}\ V\ (\mathit{cmp} \in \{=,\neq,\geq,>,\leq,<\})
\end{array}
$$

Fig. 2. Syntax of a simple language.

III. Function-level Selective Symbolization

A. Notations and Preliminaries

We consider a simple language as defined in Figure 2. A program P is composed of a sequence of statements S that are comprised of a label (if any) and a statement body, *i.e.*, ST. The body ST can be one of the following types.

- an assignment;
- a conditional jump $if\ C\ goto\ l$ where C is the condition and l is the label of the jump target;
- an invocation to a function $V = f([V(,V)^*])$;
- a return stratement $return\ V$ where V is the return value;
- a $halt$ statement.

Each variable V has a type. For brevity, we do not define the type of variables in the language. The definitions of condition C and expression E are the same as that in C language. We assume that the program acquires inputs using $input()$ invocation. Here we can decide whether to attach a symbol to each input. For brevity, the simple language does not support the definition of functions, but our implementation does support real-world C programs.

A concolic execution engine executes the target program both concretely and symbolically simultaneously by maintaining a program state as tuple $(S, \mathcal{M}_c, \mathcal{M}_s, PC)$. Here S is the statement to be executed. $\mathcal{M}_c$ is the concrete memory that maps each program variable to its concrete value, while $\mathcal{M}_s$ is the symbolic memory that maps each variable to its corresponding symbolic expression, if any. The definitions of $\mathcal{M}_c$ and $\mathcal{M}_s$ can be extended to expressions and conditions naturally. For example, $\mathcal{M}_s(V_1 + V_2) = \mathcal{M}_s(V_1) + \mathcal{M}_s(V_2)$ and $\mathcal{M}_s(V_1 \geq V_2) = \mathcal{M}_s(V_1) \geq \mathcal{M}_s(V_2)$. PC is a set of symbolic constraints expressing the condition that makes inputs steer execution to the current location.

B. Framework

Our method is a variant of common concolic execution algorithm. Algorithm 1 shows the framework of Fuse, where our improvements are colored blue. We use a worklist to store all the open branches yet to be covered. The algorithm contains a loop which iteratively explores different paths. We use a set S_{cf} to record complex functions that are identified on-the-fly. For each input I, the algorithm executes the program along the path of I in a concolic manner with selective symbolization (Line 5). The principle is that we will skip the symbolic execution in complex functions and model them as symbolic function invocations. Therefore, identifying complex functions is the key to the success of our method. We discuss

Algorithm 1 Function-level Selective Concolic Execution

```
Fuse($I_0$)
Input: $I_0$ is the initial input
 1: $worklist \leftarrow \emptyset$          //open branches
 2: $S_{cf} \leftarrow \emptyset$            //the set of complex functions
 3: $I \leftarrow I_0$
 4: while true do
 5:     SConcolic($I, S_{cf}$)
 6:     saveOffPathBranches($worklist$)
 7:     if $worklist = \emptyset$ then
 8:         break
 9:     end if
10:     repeat
11:         $b \leftarrow$ select($worklist$)
12:         $s, solution \leftarrow$ mixedSolve($b.pc, S_{cf}$)
13:         if $s =$ SAT then
14:             $I \leftarrow solution$
15:         end if
16:     until $s =$ SAT
17: end while
```

the criteria for identifying complex functions in Subsection III-D. For each input, its execution may pass multiple functions and switch between easy and complex functions for multiple times. We elaborate the details of this part in Subsection III-C. After each execution, we save all off-the-path branches in the current path into $worklist$ (Line 6). Then, one of the branches is selected (Line 11) with regard to the specified search strategy and the related path constraint is solved. Since the path constraint may involve symbolic function invocations, we propose a mixed solving method to handle it (Line 12), which is the key of our approach's performance strength over pure SMT or fuzzing-based solving method. We discuss this mixed solving algorithm in Subsection III-E. Note that our method is orthogonal to existing search strategies.

C. Concolic Execution With Selective Symbolization

Algorithm 2 shows the details of function-level selective symbolization in concolic execution. The execution path of the input I may travel along many functions. We use a tag $mode_c$ to indicate whether the current execution has entered a complex function. Once the execution enters a complex function f, $mode_c$ will be set to $true$ until the execution returns from f (*i.e.*, complex mode). The execution may switch between easy and complex modes for multiple times along one path. In our algorithm, we only symbolically execute the statements under easy mode.

Under easy mode (Line 6), statements are executed symbolically in the way of the original concolic execution [2], [3]. The path condition is collected when conditional statements are met (Line 8). For easy mode, our algorithm differs from common concolic execution in the following three aspects.

1) When we encounter an invocation to a complex function, we switch to complex mode (Line 15 - 18).
2) When an invocation to an external function meets Criterion 2 (Section III-D), we switch to complex mode (Line 19 - 23).
3) For a $return\ V$ statement where V is the return value, we use Criterion 1 (Section III-D) to check whether the symbolic expression of V contains complex operations

Algorithm 2 Concolic execution with selective symbolization

```
SConcolic($I, S_{cf}$)
Input: $I$ is the input, and $S_{cf}$ is the set of complex functions
 1: $stmt \leftarrow initialStatement$
 2: $PC \leftarrow \emptyset$                //$PC$ is the path condition
 3: $entry_c \leftarrow null$         //Entry of complex function
 4: $mode_c \leftarrow false$
 5: while $stmt$ is not $halt$ do
 6:     if $\neg mode_c$ then
 7:         /*not in complex mode*/
 8:         if $stmt$ is $if\ C\ goto\ l$ then
 9:             if $\mathcal{M}_c(C)$ then
10:                 $PC = PC \cup \mathcal{M}_s(C)$
11:             else
12:                 $PC = PC \cup \neg\mathcal{M}_s(C)$
13:             end if
14:         else if $stmt$ is $V = f([V(,V)^*]))$ then
15:             if $f \in S_{cf}$ then
16:                 $mode_c \leftarrow true$
17:                 $entry_c \leftarrow f$
18:             end if
19:             if complexCriterion2($f$) then
20:                 $S_{cf} \leftarrow S_{cf} \cup \{f\}$
21:                 $mode_c \leftarrow true$
22:                 $entry_c \leftarrow f$
23:             end if
24:         else if $stmt$ is $return\ V$ then
25:             if complexCriterion1($\mathcal{M}_s(V)$) then
26:                 $S_{cf} \leftarrow S_{cf} \cup \{currentFunction\}$
27:                 deleteConstraints($PC$, $f$)
28:                 setExpression($f, V$)
29:             end if
30:         end if
31:         concolicExecute($stmt$) /*concolic execution as usual*/
32:     else
33:         /*in complex mode*/
34:         if $stmt$ is $return\ V \wedge currentFunction = entry_c$ then
35:             $mode_c \leftarrow false$
36:             $entry_c \leftarrow null$
37:             setExpression($f, V$)
38:         end if
39:         concreteExecute($stmt$)
40:     end if
41:     $stmt \leftarrow$ nextStatement()
42: end while
```

(Line 25). If yes, the current function is identified as a complex one on-the-fly (Line 26); otherwise, the function is executed symbolically as usual. Next, we delete the constraints collected inside f from current PC (Line 27), and replace the symbolic expression of V (*i.e.*, $\mathcal{M}_s(V)$) by $f(args)$ (Line 28). Here $f(args)$ represents how V is calculated and will be leveraged by underlying fuzzing-based constraint solver. We also call $f(args)$ as a *symbolic function invocation expression*.

Before the execution switches to the complex mode, we need to record the complex function as the entry $entry_c$ (Line 17 and Line 22). Since a complex function may invoke the other complex functions, we only need to build symbolic function invocation expression for the top complex function of the call chain, *i.e.*, $entry_c$. In complex mode, all statements will be executed concretely (Line 39) with one exception. Once we meet the return statement of the entry complex function, we switch back to easy mode (Line 34). Similarly, we also replace the value of V by $f(args)$ (Line 37).

Algorithm 3 Combine SMT and Fuzzing solvers

```
mixedSolve($PC, S_{cf}$)
Input: $PC$ is a constraint set.
Output: $res$ is a solution set.
 1: $res \leftarrow \emptyset$
 2: $PC_e, PC_c \leftarrow$ split($PC, S_{cf}$)
 3: $s, res_e \leftarrow$ invokeSMT($PC_e$)
 4: if $s$ is SAT then
 5:     $PC_c \leftarrow$ simplify($res_e, PC_c$)
 6:     $s, res_c \leftarrow$invokeFuzzing($PC_c$)
 7:     if $s$ is SAT then
 8:         $res \leftarrow$ merge($res_e, res_c$)
 9:     end if
10: end if
11: return $s, res$
```

D. Automatic Identification of Complex Functions

The most important question in our approach is the automatic identification of complex functions. We expect that it is beneficial to handle constraints involving symbolic function invocation expressions with fuzzing-based solving method rather than SMT solver. In our approach, we identify complex functions according to the following criteria. Note that these criteria can be refined or improved in the future.

- **Criterion 1**: *If the function contains complex operations that bring burden to SMT solver, we classify such function as a complex function*. Currently, we consider the non-linear multiplication, division and other more complex operations on floating point expressions as complex operations. Existing research has demonstrated that the state-of-the-art SMT solver is not good at handling these operations. On the contrary, fuzzing-based solvers (*e.g.*, JFS [22]) are more efficient on these types of constraints. Therefore, it is appropriate to label the functions with these complex operations as complex ones. In our algorithm, we implement this criterion when the return statement under easy mode is met.
- **Criterion 2**: *External library functions whose parameters and return values are of basic types are classified as complex functions*. As demonstrated by the motivation example, it is natural to classify the external library functions without source code to be complex functions, which can be executed by the fuzzer when solving the path constraints. Besides, we require that parameters and return values are all basic types because state-of-the-art fuzzing-based constraint solving method does not support the complex data types such as pointer.

E. Mixed solving

Algorithm 3 shows the combination of fuzzing and SMT solver for solving the constraints that contain symbolic function invocation expressions. The first step is to divide the path condition into easy and complex parts using the identified complex function set S_{cf} (Line 2), denoted by PC_e and PC_c respectively. Each branch predicate is classified into the different parts using the following criteria.

- If a branch predicate contains symbolic function invocation expressions, the branch predicate is pushed into PC_c;
- If a symbolic variable b is involved in the above branch predicate, then any branch predicate involving b is pushed into PC_c;
- All of the other branch predicates are pushed into PC_e.

After splitting, we invoke SMT solver to solve PC_e first (Line 3). The solution (if any) of PC_e is plugged into PC_c to simplify constraints (Line 5). Then we invoke fuzzing-based constraint solver to find solution for PC_c (Line 6). When both these two solvers find solutions, we combine the solutions and return (Line 8). Otherwise, we return UNKNOW, which is omitted for the sake of space.

IV. Implementation and Evaluation

A. Implementation

We implement FUSE in our customized concolic exeuction engine based on KLEE-2.3 (commit 71c1c45) [7]. We use Z3 [6] version 4.6.2 as the backend SMT solver because Z3 supports multiple theories including uninterpreted function, bit-vector and floating-point theory, *etc*. We use JFS [22] to solve constraints with symbolic function invocation, *i.e.*, the complex part. JFS is a constraint solver based on coverage-guided fuzzing technique. We extend JFS to support *select* and *store* operations in array theory that is widely employed by KLEE. Besides, we link the third-party libraries involved in our evaluation to JFS's runtime library to support symbolic function invocations.

B. Research Questions

We conduct extensive experiments to answer the following research questions:

- **RQ1**: **Effectiveness**. How effective is FUSE on exploring program paths compared with the baselines? We use the number of covered lines and coverage of analyzed function to measure the effectiveness.
- **RQ2**: **Efficiency**. How efficient is FUSE to achieve the same coverage compared with the baselines?

C. Experimental Setup

1) Benchmarks: We use the following two real-world benchmarks for evaluating FUSE. (a) GNU Scientific Library (GSL) [23] is a widely employed numerical library in scientific computing. Hence, GSL contains lots of complex mathematical functions, such as trigonometric, exponential and logarithmic functions, which use many floating-point and non-linear operations. In addition, GSL is also a common benchmark in the studies on floating-point program analysis. We choose the programs under the *specfunc* directory in GSL-2.7 as our benchmark. There are 106 programs in total in GSL benchmark. (b) GNU Coreutils [24] is a core tool collection that provides the basic operation utilities of UNIX operating system. Coreutils is the standard benchmark of KLEE-based symbolic execution research. Typically, the programs in Coreutils do not produce many complex constraints. There are 99 programs in Coreutils benchmark.

2) Setup: **Baselines**. We have the following three baselines.

- **DSE+SMT**: our DSE engine only using SMT solver as the underlying constraint solver;
- **DSE+JFS**: our DSE engine only using JFS as the underlying constraint solver;
- **MCS** [25]: dynamic symbolic execution with Mixed Concrete-Symbolic (**MCS**) solving [25]. **MCS** splits the path constraint into easy and complex parts like FUSE. However, it applies incremental solving in SMT solver to obtain multiple solutions from the easy part and uses theses solutions to simplify the complex part. Finally, the simplified complex part is solved by SMT solver again. The major difference between **Fuse** and **MCS** is that **MCS** does not employ a function-level symbolization and hence cannot leverage the strength of fuzzing. Since **MCS** is built on Symbolic PathFinder [18], a symbolic execution tool for Java bytecode, we implement the core algorithm of **MCS** in our own DSE engine.

We create a symbolization driver for each program in GSL benchmark. The driver provides initial concrete input for the program's entry function and symbolizes each byte of the input to perform symbolic execution. FUSE analyzes each program for 30 minutes. The timeout threshold of constraint solver is 30 seconds. To eliminate the randomness, we use deterministic search strategies in analysis, *i.e.*, BFS and DFS. Finally, we performed all experiments on a 16-core server with Intel(R) Xeon(R) Platinum 8269CY CPU at 2.50GHz CPU and 32GB of memory. The operating system is Ubuntu 18.04. Each experiment is carried out for 3 times and all experimental results are average values.

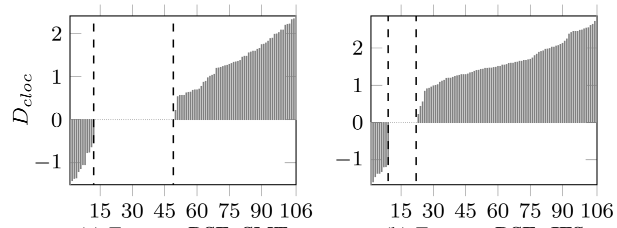
D. Experimental Results

Here, we use the number of covered lines of code (**#CLoC**) and coverage of analyzed function (**Cov**) to evaluate the FUSE's effectiveness. Note that **#CLoC** are measured at the level of C code. The two indicators focus on different points. **#CLoC** directly shows the ability of path exploration, while **Cov** suggests whether the method is easily getting trapped in complex functions and fails to explore the remaining space. A larger **Cov** demonstrates a more powerful ability of path exploration.

1) Effectiveness: Figure 3 and Figure 4 show the results of FUSE compared with baselines on GSL benchmark programs under BFS search strategy. In Figure 3, the X-axis shows the program numbers and the Y-axis shows a decorated relative increase of **#CLoC**. The decorated value (*i.e.*, D_{cloc}) orders the programs on X-axis. Note that we use the decorated value to smooth the big gaps between the relative increases in different programs. To be specific, D_{cloc} is calculated as follows, where L_{fun} represents **#CLoC** in FUSE, and $L_{baseline}$ represents **#CLoC** in baseline.

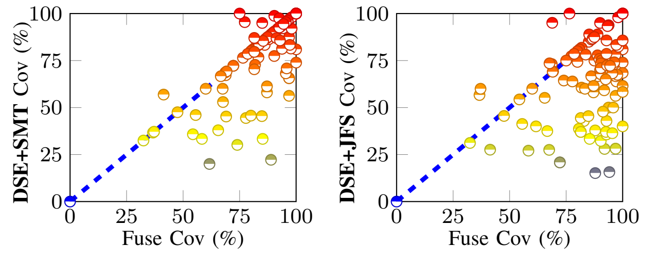
$$r = \frac{L_{fun} - L_{baseline}}{L_{baseline}} \tag{1}$$

$$D_{cloc} = \begin{cases} lg(r \times 100 + 1) & r > 0 \\ 0 & r = 0 \\ -lg(-r \times 100 + 1) & r < 0 \end{cases} \tag{2}$$



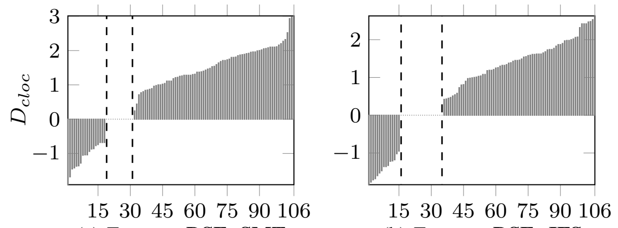
(a) Fuse vs. **DSE+SMT** (b) Fuse vs. **DSE+JFS**

Fig. 3. Results of D_{cloc} on GSL benchmark under BFS.



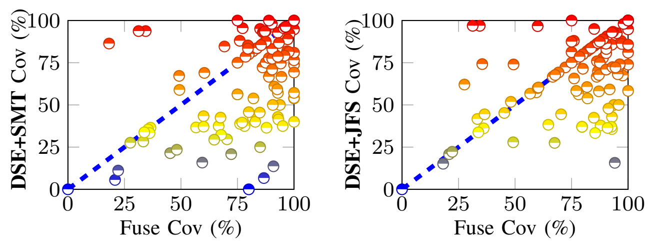
(a) Fuse vs. **DSE+SMT** (b) Fuse vs. **DSE+JFS**

Fig. 4. Results of **Cov** on GSL benchmark under BFS.



(a) Fuse vs. **DSE+SMT** (b) Fuse vs. **DSE+JFS**

Fig. 5. Results of D_{cloc} on GSL benchmark under DFS.



(a) Fuse vs. **DSE+SMT** (b) Fuse vs. **DSE+JFS**

Fig. 6. Results of **Cov** on GSL benchmark under DFS.

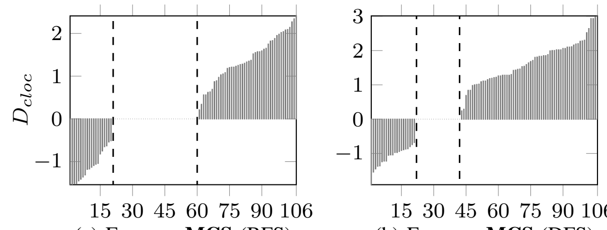
As shown by the equations, there is no increase when D_{cloc} is zero, which maps to the programs between two vertical dotted lines in the figures. This is because these programs are fully explored by both configurations. The actual relative increases are 10% and 100% when D_{cloc} are 1 and 2, respectively. We first give the results under BFS search strategy. Compared with **DSE+SMT**, Fuse covers more lines for $57(53.8\%)$ programs and fewer lines for $12(11.3\%)$ programs. On average, the relative increasing value for the number of covered lines is $24.0\%(-31.6\%\sim256.9\%)$. Compared with **DSE+JFS**, Fuse covers more lines for $84(79.2\%)$ programs and fewer lines for $9(8.5\%)$ programs. On average, the relative increasing value for the number of covered lines is $72.3\%(-45.9\%\sim718.2\%)$. The decrease in **#CLoC** is due to coarse granularity of the function-level symbolization. An identified complex function may contain not only complex non-linear constraints but also a bulk of easy constraints, where the aforementioned two configurations covered lots of lines quickly. The situation can be avoided by designing more precise automatic identification algorithm of complex functions.

In Figure 4, each point shows results of **Cov** on a GSL program, where x-value and y-value give the results of Fuse and baseline respectively. Below the diagonal line, Fuse outperforms the baseline; on the other side, Fuse is defeated. Fuse achieves higher coverage of analyzed function on $41(38.7\%)$ programs and $76(71.7\%)$ programs compared with **DSE+SMT** and **DSE+JFS**, respectively. The average coverage of analyzed function are $84.2\%(32.5\%\sim100\%)$, $77.2\%(20\%\sim100\%)$ and $62.7\%(15.2\%\sim100\%)$ in the three configurations, respectively. The results show that Fuse is less possible to get trapped in complex code and able to explore more path space.

The results under DFS are shown in Figures 5&6. In Figure 5, compared with **DSE+SMT**, Fuse increases and decreases the number of covered lines on $75(70.8\%)$ programs and $18(17.0\%)$ programs, respectively. The averaged increasing value is $70.0\%(-81.1\%\sim1025\%)$. Compared with **DSE+JFS**, the increases and decreases are shown on $71(67.0\%)$ programs and $15(14.1\%)$ programs, respectively. The average relative increasing value is $38.0\%(-67.8\%\sim430\%)$. In Figure 6, compared with **DSE+SMT** and **DSE+JFS**, Fuse achieves higher coverage of analyzed function on $66(62.3\%)$ programs and $55(51.9\%)$ programs, respectively. The average coverage results of analyzed function are $78.4\%(18.2\%\sim100\%)$, $63.1\%(0\%\sim100\%)$ and $68.2\%(15.2\%\sim100\%)$ in the three configurations, respectively.

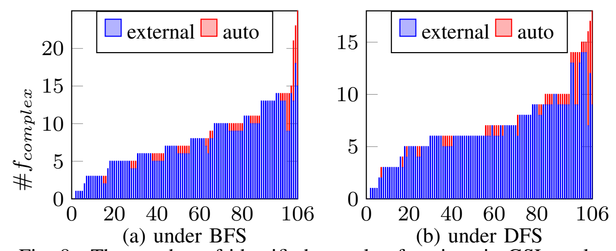
These results indicate that Fuse can combine the strength of SMT solving and fuzzing to explore the path space more effectively. However, we observe that search strategy influences the effectiveness differently with regard to baseline configurations. Compared with **DSE+SMT**, Fuse is more effective for DFS rather than BFS. This is because DFS strategy always tends to generate long and complex path constraints that hinder the SMT solver (**DSE+SMT**). But Fuse is able to abstract the complex semantics in code. Compared with **DSE+JFS**, the observation is opposite. Fuse is more effective for BFS rather than DFS, which is the result of awkwardness JFS suffers when solving simple constraints. Unfortunately, BFS always tends to generate such constraints like simple bit-vector constraints without floating-point in our evaluation.

Besides, we also compared Fuse with the most related work, *i.e.*, **MCS** on the number of covered lines in Figure 7. Fuse covers more lines for $46(43.4\%)$ programs and fewer lines for $20(18.9\%)$ programs under BFS. The values are $64(60.4\%)$ and $21(19.8\%)$ under DFS. On average, the relative increasing values for the number of covered lines are $18.1\%(-45.9\%\sim718.2\%)$ and $70.1\%(-81.1\%\sim1025\%)$ under BFS and DFS, respectively. The results demonstrate the effectiveness of Fuse over **MCS**. The combination of SMT solving and fuzzing in the mixed solving method of Fuse



(a) Fuse vs. **MCS** (BFS) (b) Fuse vs. **MCS** (DFS)

Fig. 7. Results of D_{cloc} on GSL benchmark compare with **MCS**.



(a) under BFS (b) under DFS

Fig. 8. The number of identified complex functions in GSL evaluation.

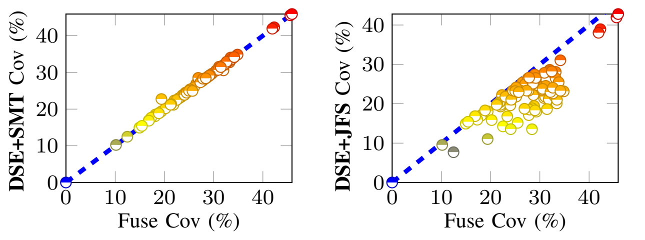
is superior to the sole dependence on SMT solving in **MCS**, even if **MCS** employs many powerful heuristics to simplify and accelerate the solving.

Here, we report the results of automatic identified functions in GSL evaluation. Figure 8 shows the number of complex functions ($\#f_{complex}$) in the analysis of each program under BFS and DFS strategies. Complex functions in GSL evaluation are composed of two types of functions: predefined external library functions (denoted by the blue part of the bar) and automatic identified functions with complex operations (denoted by the red part of the bar). On average, FUSE identifies 8.1 and 7.1 complex functions per program under BFS and DFS strategies, respectively. The results indicate that path constraints under BFS encounter more complex functions than DFS, which is quite intuitive.

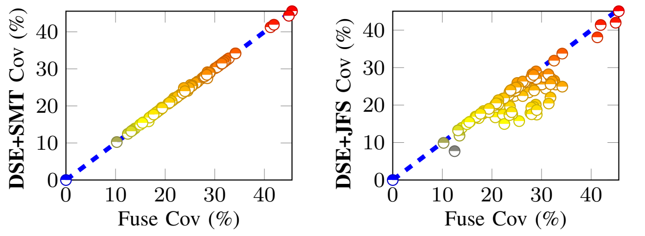
At last, we conducted experiments on Coreutils benchmark to evaluate FUSE on programs that mainly produce easy-to-solve constraints. Figure 9 shows the coverage of analyzed functions on Coreutils benchmark. As shown in the figure, the performance of FUSE is very close to **DSE+SMT** because the combined solver in FUSE degrades to SMT solver when facing simple constraints. However, compared with **DSE+JFS**, FUSE shows a better performance on nearly all programs. The reason is that JFS is not good at solving simple bit-vector constraints because it introduces some unexpected overhead such as compiling and linking overhead.

Answer to RQ1: FUSE is effective to improve the code coverage of GSL benchmark programs. Compared with **MCS**, FUSE, on average, increases the statement coverage by $24.0\%(-31.6\%{\sim}256.9\%)$ and $70.0\%(-81.1\%{\sim}1025\%)$ under BFS and DFS, respectively. FUSE performs the same as pure SMT solving when dealing with simple constraints.

2) Efficiency: In the following, we evaluate the efficiency of FUSE through the time budget to cover the same number of lines. Figure 10 shows the trends of the covered lines during analysis. The X-axis shows the elapsed time in seconds.



(a) Fuse vs. **DSE+SMT** (BFS) (b) Fuse vs. **DSE+JFS** (BFS)



(c) Fuse vs. **DSE+SMT** (DFS) (d) Fuse vs. **DSE+JFS** (DFS)

Fig. 9. Results of **Cov** on Coreutils benchmark.

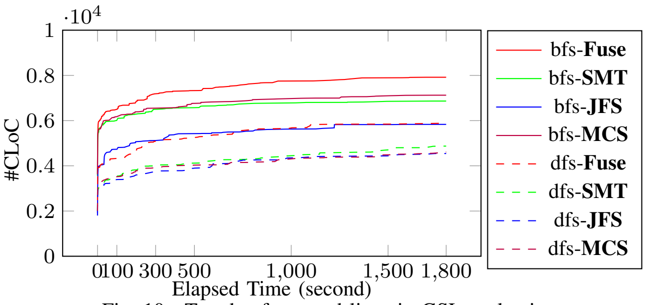


Fig. 10. Trends of covered lines in GSL evaluation.

The Y-axis shows the average number of covered lines of 106 programs in GSL benchmark. The solid lines represent trends under BFS search strategy, while the dotted lines represent trends under DFS search strategy. As shown in the figure, FUSE outperforms all baselines consistently under any search strategy. Under BFS, FUSE achieves 8x, 450x and 6.6x speedups for covering the largest number of lines using **DSE+SMT**, **DSE+JFS**, and **MCS** respectively. Under DFS, the speedups are 7.8x, 9.9x and 8x, respectively. These results indicate that FUSE helps to improve the efficiency of dynamic symbolic execution a lot.

Answer to RQ2: FUSE is efficient to achieve the same code coverage. Compared with the best baseline, FUSE achieves 6.6x and 7.8x times of speedups under BFS and DFS.

3) Threat to Validity: The major threats to the validity of our evaluation are internal and external threats. One internal threat is the randomness in the evaluation. To ensure the reproducibility of our evaluation, we employed deterministic search strategies during the symbolic execution and conducted repeated experiments. However, the inherent randomness in fuzzing-based method is impossible to be fully removed, which might threaten the validity. Another internal threat is our implementation. Although we have found and fixed some bugs in our implementation during the evaluation, there could exist other missing bugs that damage the validity. The external threat is that our benchmark could not represent all kinds of programs. To address the external threat, we chose two popular benchmarks that include diverse widely used programs for evaluation. These benchmarks contain complex non-linear floating-point operations and extensive system calls.

V. Related Work

The mixed solving method plays a vital role in FUSE. Mixed solving refers to combining multiple solving techniques to solve constraints, e.g., the mixed solving method in FUSE combines SMT solving and fuzzing. Similar to FUSE, Colossus [27] designs a fuzz-based constraint solver which is also a combination of SMT solver (named logical solver in Colossus) and fuzz solver. However, the difference is that the combination in Colossus is at the granularity of whole path constraint, *i.e.*, Colossus does not split path constraints, failing to utilize the reasoning ability provided by SMT solver to handle magic number comparison. Hence, the mixed solving method in FUSE is more compact and efficient. Malbug and Fraser [28] employ constraint solver to assist the mutation process in a search-based test generation method, which prevent the search-based method getting stuck in local optima.

FUSE can be also regarded as a constraint solving optimization method of symbolic execution. There are a number of publications on accelerating constraint solving in symbolic execution. KLEE [7] employs two core optimizations called constraint independence and counter-example caching, where the two optimizations collaborate mutually to achieve better efficiency. Shuai *et al.* [19] propose to compute type and interval information of array accesses during the symbolic execution and pass the information to the underlying constraint solving for boosting the array constraint solving. Liu *et al.* [12] compare stack-based incremental solving approach with cache-based approach in symbolic execution and conclude that the former is often more effective. Speculative symbolic execution [29] executes the program speculatively at each branch point and ignores the check of path feasibility. Only when the number of speculated branches reaches a threshold is the path feasibility checked.

Besides those works on optimizing constraint solving, there also exists the work for tackling the path explosion problem [30]. Search strategies are the key to improving the efficiency of path exploration. In principle, there exists no search strategy that is the silver bullet. Besides the traditional search strategies, *i.e.*, DFS and BFS, many search strategies are proposed for different targets, such as quickly improving the code coverage [7], [31], [32], finding a specific type of bugs [13], and finding the paths that reach a specific program point [33] or satisfying a specific typestate property [34]. In addition, besides specifying a target, there also exists work [35], [36] that generates a specific search strategy for the program under symbolic execution. Besides search strategies, there also exists work that merges [37] paths or prunes [38] redundant paths. The approaches of merging paths introduce disjunctions into path conditions and need to do the tradeoff between the complexity of constraint solving and the profits brought by path merging. The methods of pruning paths [38]–[40] often abstract the program with respect to the analysis's specification (*e.g.*, an assertion or a typestate property) and prune the paths that do not violate the specification or are equal to the explored paths. FUSE is designed with respect to constraint solving and orthogonal to the methods for the path explosion problem.

FUSE is also related to fuzzing or optimization-based constraint solving. JFS [22] converts the constraint solving problem of floating-point constraints into a fuzzing problem by generating a program for a constraint. Then, it employs the existing fuzzer (*i.e.*, libFuzzer [41]) for solving, where the generated inputs that crash the program also satisfy the constraint. JFS inspires our work, and our implementation is also based on JFS. FUZZY-SAT [42] employs fuzzing to solve bit-vector formulas and proposes several optimizations with respect to different formula forms. Different from JFS, FUZZY-SAT does not generate a program but mutates the initial value to search for the solution. Compared with them, FUSE combines SMT solving and fuzzing to solve the constraints; Besides, FUSE supports the fuzzing of the constraints in which symbolic method invocations exist. XSAT [43] converts the solving of floating-point constraints into a mathematical optimization (MO) problem by the fitness function [44] defined for the constraint. JIGSAW [45] also generates a program for a constraint and converts the solving problem into a search problem. MLBSE [46] also employs the sampling-based optimization method to handle constraint solving and environment modeling problems. Compared with MLBSE, FUSE synergizes fuzzing and SMT for tackling the two problems. It is interesting to combine MO-based approaches with fuzzing and SMT to improve efficiency further.

FUSE can be also seen as a hybrid of symbolic execution and fuzzing. We use fuzzing in the stage of constraint solving to improve symbolic execution. Note that, FUSE is orthogonal to the recently proposed hybrid testing [16], which combines two complementary techniques: symbolic execution and fuzzing in a top level. Fuzzing has the advantage of high throughput, low-cost and high degree of automation but fails to cover branches guarded by rigorous conditions. Symbolic execution is capable of covering hard-to-reach branches but confined by scalability problem. Hybrid testing employs fuzzing to cover easy-to-cover paths and leaves hard branches to symbolic execution. Up to now, researchers have proposed a number of hybrid testing methods/systems, including Driller [16], Munch [47], libKLUZZER [48], DigFuzz [49], Pangolin [50], SAFL [51], *etc*. FUSE is mainly in the framework of symbolic execution and can be used as a component in hybrid testing systems.

VI. Conclusion

This paper proposes FUSE, a framework of synergizing symbolic execution and fuzzing by function-level selective symbolization to address the problems of constraint solving and environment modeling. FUSE determines complex functions on-the-fly with respect to the hardness of constraint solving and the availability of source code. The semantics of complex function in path constraint are represented by symbolic function invocation expressions. At last, the path constraint is partitioned into easy and complex parts, solved by SMT solver and fuzzing respectively. We implemented FUSE on a concolic variant of KLEE, and performed extensive experiments on two real-world benchmarks, i.e., GSL and Coreutils. The experimental results show the effectiveness and efficiency of our approach. In the future, we plan to design a more systematical way to precisely identify complex functions. For example, we can mark a complex function if there is an infinite loop in the function to avoid getting trapped in the loop.

FUSE and our experiments can be accessed at https://github.com/zbchen/FuSE.

ACKNOWLEDGEMENTS

This research was supported by the NSFC Programs (No. 62172429, 62002107 and 62032024). Yufeng Zhang is sponsored by CCF-Huawei Formal Method Innovation Research Plan.

REFERENCES

[1] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.
[2] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI*, 2005, pp. 213–223.
[3] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *SIGSOFT*, 2005, pp. 263–272.
[4] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 50:1–50:39, 2018.
[5] D. Kroening and O. Strichman, *Decision Procedures - An Algorithmic Point of View*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.
[6] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, 2008, pp. 337–340.
[7] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008, pp. 209–224.
[8] N. Tillmann and J. de Halleux, "Pex-white box test generation for .net," in *TAP*, 2008, pp. 134–153.
[9] P. Godefroid, M. Y. Levin, and D. A. Molnar, "SAGE: whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, pp. 40–44, 2012.
[10] P. Jalote, L. C. Briand, and A. van der Hoek, Eds., *ICSE*. ACM, 2014.
[11] H. Palikareva and C. Cadar, "Multi-solver support in symbolic execution," in *CAV*, 2013, pp. 53–68.
[12] T. Liu, M. Araújo, M. d'Amorim, and M. Taghdiri, "A comparative study of incremental constraint solving approaches in symbolic execution," in *HVC*, 2014, pp. 284–299.
[13] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: automatic exploit generation," in *NDSS*, 2011.
[14] Y. Lin, "Symbolic execution with over-approximation," Ph.D. dissertation, University of Melbourne, Parkville, Victoria, Australia, 2017.
[15] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *SOSP*, 2012, pp. 380–394.
[16] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS*, 2016.
[17] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *NDSS*, 2008.
[18] C. S. Pasareanu and N. Rungta, "Symbolic pathfinder: symbolic execution of java bytecode," in *ASE*, 2010, pp. 179–180.
[19] Z. Shuai, Z. Chen, Y. Zhang, J. Sun, and J. Wang, "Type and interval aware array constraint solving for symbolic execution," in *ISSTA*, 2021, pp. 361–373.
[20] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990.
[21] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The Fuzzing Book," in *The Fuzzing Book*. Saarland University, 2019.
[22] D. Liew, C. Cadar, A. F. Donaldson, and J. R. Stinnett, "Just fuzz it: solving floating-point constraints using coverage-guided fuzzing," in *FSE*, 2019, pp. 521–532.
[23] "Gnu scientific library," https://www.gnu.org/software/gsl/.
[24] "Gnu core utilities," https://www.gnu.org/software/coreutils/.
[25] C. S. Pasareanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," in *ISSTA*, 2011, pp. 34–44.
[26] G. Strang, *Calculus*. SIAM, 1991, vol. 1.
[27] A. Pandey, P. R. G. Kotcharlakota, and S. Roy, "Deferred concretization in symbolic execution via fuzzing," in *ISSTA*, 2019, pp. 228–238.
[28] J. Malburg and G. Fraser, "Combining search-based and constraint-based testing," in *ASE*, 2011, pp. 436–439.
[29] Y. Zhang, Z. Chen, and J. Wang, "Speculative symbolic execution," in *ISSRE*, 2012, pp. 101–110.
[30] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.
[31] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *DSN*, 2009, pp. 359–368.
[32] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, pp. 2:1–2:49, 2012.
[33] K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *SAS*, 2011, pp. 95–111.
[34] Y. Zhang, Z. Chen, J. Wang, W. Dong, and Z. Liu, "Regular property guided dynamic symbolic execution," in *ICSE*, 2015, pp. 643–653.
[35] S. Cha, S. Hong, J. Lee, and H. Oh, "Automatically generating search heuristics for concolic testing," in *ICSE*, 2018, pp. 1244–1254.
[36] S. Cha and H. Oh, "Concolic testing with adaptively changing search heuristics," in *FSE*, 2019, pp. 235–245.
[37] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *PLDI*, 2012, pp. 193–204.
[38] H. Yu, Z. Chen, J. Wang, Z. Su, and W. Dong, "Symbolic verification of regular properties," in *ICSE*, 2018, pp. 871–881.
[39] H. Cui, G. Hu, J. Wu, and J. Yang, "Verifying systems rules using rule-directed symbolic execution," in *ASPLOS*, V. Sarkar and R. Bodík, Eds. ACM, 2013, pp. 329–342.
[40] S. Cha and H. Oh, "Making symbolic execution promising by learning aggressive state-pruning strategy," in *ESEC/FSE*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 147–158.
[41] "Libfuzzer," https://llvm.org/docs/LibFuzzer.html.
[42] L. Borzacchiello, E. Coppa, and C. Demetrescu, "Fuzzing symbolic expressions," in *ICSE*. IEEE, 2021, pp. 711–722.
[43] Z. Fu and Z. Su, "Xsat: A fast floating-point satisfiability solver," in *CAV*, ser. Lecture Notes in Computer Science, S. Chaudhuri and A. Farzan, Eds., vol. 9780. Springer, 2016, pp. 187–209.
[44] P. McMinn, "Search-based software test data generation: a survey," *Softw. Test. Verification Reliab.*, vol. 14, no. 2, pp. 105–156, 2004.
[45] J. Chen, J. Wang, C. Song, and H. Yin, "Jigsaw: Efficient and scalable path constraints fuzzing," 2022.
[46] L. Bu, Y. Liang, Z. Xie, H. Qian, Y. Hu, Y. Yu, X. Chen, and X. Li, "Machine learning steered symbolic execution framework for complex software code," *Formal Aspects Comput.*, vol. 33, no. 3, pp. 301–323, 2021.
[47] S. Ognawala, T. Hutzelmann, E. Psallida, and A. Pretschner, "Improving function coverage with munch: a hybrid fuzzing and directed symbolic execution approach," in *SAC*, H. M. Haddad, R. L. Wainwright, and R. Chbeir, Eds. ACM, 2018, pp. 1475–1482.
[48] H. M. Le, "Llvm-based hybrid fuzzing with libkluzzer (competition contribution)," in *FASE*, ser. Lecture Notes in Computer Science, H. Wehrheim and J. Cabot, Eds., vol. 12076. Springer, 2020, pp. 535–539.
[49] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing," in *NDSS*. The Internet Society, 2019.
[50] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction," in *SOSP*. IEEE, 2020, pp. 1613–1627.
[51] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun, "SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing," in *ICSE'18*. ACM, 2018, pp. 61–64.