Symbolic Execution of Floating-point Programs: How far are we?

Guofeng Zhang[*], Zhenbang Chen[*§], and Ziqi Shuai[*†]
[*]College of Computer, National University of Defense Technology, ChangSha, China
[†]State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China
Email: {zhangguofeng16, zbchen, szq}@nudt.edu.cn
[§]Corresponding author

***Abstract*—Floating-point programs are challenging for symbolic execution due to the constraint solving problem. To investigate the effectiveness and limitations of the existing methods, we conduct the first empirical study in this paper on five existing symbolic execution methods for floating-point programs. We have implemented the existing methods on the state-of-the-art symbolic execution KLEE and use the real-world representative floating-point programs as the benchmarks, which are used to evaluate the existing methods with respect to code coverage and bug finding. The results indicate that the existing methods complement each other in bug finding. Based on the findings of the experimental results, we propose synergizing the existing methods to improve symbolic execution's effectiveness. The experimental results demonstrate that our synergic method can detect more bugs.**

***Index Terms*—Symbolic Execution, Constraint Solving, Fuzzing, Real Arithmetic, Floating Point**

I. Introduction

As the mainstream representation of real numbers, floating-point numbers are frequently used in different numerical softwares, such as scientific computing [1] and aerospace controllers [2]. However, manipulating floating-point numbers is not easy. There are some famous software disasters caused by floating point operations in history, such as Ariane 5 rocket's self-destruction [2] and the Patriot missile's failure [3]. Therefore, improving the quality of floating-point programs is quite important and challenging.

Symbolic execution provides an effective method for systematically exploring a program's path space. Recently, symbolic execution has been successfully applied in many software engineering activities, including automatic software testing [4], [5], bug detection [6], and bug repair [7], [8], *etc*. However, despite the successful applications, symbolic execution is still challenged by two main technical problems [9], *i.e.*, path explosion and constraint solving. These two problems may doom the further applications and development of symbolic execution. The analysis of numeric programs with floating-point operations is a well-known problem for symbolic execution because of the constraint solving challenge. Nowadays, the existing symbolic executors mainly use bit-vector (BV) SMT theory [10] for a precise program representation. If there are floating-point operations, the existing symbolic executors often use BV floating-point (BVFP) SMT theory [10] for representing path conditions. Although the complexity of solving BV formulas (including BVFP formulas) is NP-complete [11], the solving procedure is time-consuming, especially for non-linear BVFP formulas. For example, Z3 [12] (*i.e.*, a state-of-the-art SMT solver) needs 77 seconds for solving the following BVFP formula in which x is a 64-bit floating-point variable[1].

$$x^3 = 27.0 \tag{1}$$

The real-world BVFP constraints are more complex than the above one, and Z3 usually produces timeout. The reason is that BVFP SMT theory precisely encodes the floating-point constraints into SAT problems [13] with respect to the IEEE 754 standard [14], which may produce complex SAT problems that are pretty hard for the underlying SAT solver, especially for the non-linear BVFP constraints. For example, a BVFP formula with a multiplication expression of 64-bit BV variables may produce a SAT problem with tens of thousands of boolean variables and clauses. However, there are usually intensive non-linear floating-point computations in numeric programs. Therefore, the constraint solving of the BVFP formulas produced by symbolic execution is time-consuming and often times out, making the symbolic execution of the numeric programs with intensive floating-point computations infeasible.

There are five categories of the existing approaches for the symbolic execution of floating-point programs. The first one is to employ BVFP SMT solvers for a precise representation and improve the decision procedure of BVFP SMT solving [15], [16]. The approaches of this line enjoy the precise representation but may suffer from the scalability problem when solving non-linear constraints. The second category uses a real arithmetic solver to optimize the floating-point solving [17], in which the variables are considered as real number variables, and the formula is solved as a real arithmetic formula. This category enjoys the efficiency of real arithmetic SMT solving but suffers from the unsoundness with respect to floating-point numbers. The third category utilizes fuzzing to solve BVFP formulas [18], in which the constraint solving problem is converted to a fuzzing problem by generating a program according to the formula. The input crashing the program satisfies the formula. This category leverages the

[1]The CPU is 2.9GHz

power of the existing fuzzing techniques but suffers from the incompleteness problem, especially when the formula is UNSAT. The fourth category on-the-fly converts each floating-point operation during symbolic execution into a function implemented by integers and bit operations [19]. Therefore, a floating-point program is converted into one in which only integers exist, and the existing symbolic execution tools not supporting floating-point operations can be used. However, this category may suffer from the path explosion problem. The last category solves BVFP formulas in a search-based manner [20], [21], which converts the solving problem into an optimization problem by defining a fitness function [22] for the formula.

Despite the advantages of the existing methods, the effectiveness and the efficiency of these approaches for the symbolic execution of real-world floating-point programs are still unclear. The reasons are: 1) Many approaches are only evaluated on the SMT benchmarks, *e.g.*, SMTLIB2 benchmark of BVFP category [23] [24]; 2) Different approaches are evaluated on different tools, but they lack a common platform for evaluation; 3) The benchmark programs in evaluation are different, which is related to the second problem. Therefore, to further understand the effectiveness and limitations of the existing methods, we design an empirical study in this paper to evaluate the existing methods' effectiveness for real-world floating-point programs. We select 277 programs in GNU scientific library (GSL) [25] , a widely used scientific computing library that contains intensive floating-point operations. We have integrated or implemented the existing methods on KLEE [4] (*i.e.*, a state-of-the-art symbolic executor for C programs) and evaluate the existing methods by employing them for analyzing the benchmark GSL programs with respect to the two critical abilities of symbolic execution, *i.e.*, code coverage and bug finding. To the best of our knowledge, our study is the first empirical study for the symbolic execution of floating-point programs.

Our empirical study finds that: 1) fuzzing-based method achieves the best result of code coverage on real-world floating-point programs, and the results of the methods of the first three categories are closer and stable despite the search strategies; 2) fuzzing-based method and real arithmetic optimization-based method are more efficient than SMT solving when the constraints are complex; 3) the first three categories complement each other for bug finding. Besides, in the last category, *i.e.*, search-based approaches are ineffective for analyzing real-world floating-point programs.

Based on the findings, we propose an algorithm synergizing the first three categories. We can use BVFP solving to prove the unsatisfiability. Besides, real arithmetic solver and fuzzing can be used to improve scalability. Based on this observation, we propose synergizing SMT solving and fuzzing to improve symbolic execution's effectiveness on floating-point programs. Specifically, when solving a BVFP formula PC, we first employ BVFP SMT solving to check PC's *simple part* (*i.e.*, the part without non-linear expressions). If PC's *simple part* is unsatisfiable, PC is unsatisfiable. Otherwise, we use the solution S_1 produced by the BVFP solver to check whether S_1 satisfies PC. If S_1 does not, we use a real arithmetic solver to solve PC. When the solver returns a rational solution R, we convert R into a floating-point solution S_2 and check whether S_2 satisfies PC. If S_2 does not, we use S_2 as a seed and employ fuzzing for solving PC. The results of the experiments on the GSL benchmark programs demonstrate our method's effectiveness in improving symbolic execution's ability of bug finding.

The main contributions of this paper are as follows.

- We carry out the first extensive study for the state-of-the-art symbolic execution methods of floating-point programs.
- Based on the study's findings, we propose synergizing BVFP SMT solving, real arithmetic solving, and fuzzing for solving BVFP formulas, which can improve the efficiency of BVFP constraint solving.
- We have implemented or integrated the existing methods and our method into KLEE. Our prototype is available[2], which can be used for future research on the symbolic execution of floating-point programs.
- Compared with the state-of-the-art methods, our synergic method detects more bugs under both DFS and BFS.

The remainder of this paper is organized as follows. Section II introduces the background of floating-point numbers, BV SMT solving, and dynamic symbolic execution. Section III shows the details of our study's design. Section IV presents the implementation and the evaluation results. Section VI presents our synergy method and its evaluation result. The related work is discussed and compared in Section VII. The last section concludes.

II. Background

A. Floating-point Representation

IEEE 754 [14] defines the standard of floating-point numbers and their operations. In principle, the standard defines a floating-point number f by three parts: sign (S), mantissa (M), and exponent (E), and the number is calculated by the three parts as follows.

$$(-1)^S \times M \times 2^E \tag{2}$$

$S \in \{0, 1\}$ is f's first bit that denotes f's sign, where 0 and 1 represent that f is positive and negative, respectively. $M \overset{\text{def}}{=} m_0.m_1m_2...m_n$ is the significand, where m_0 is the hidden bit and $m_1m_2...m_n$ is the fraction. Lastly, $E \overset{\text{def}}{=} e - 2^{p-1} + 1$, where p is the number of exponent bits, and e is the biased exponent. For example, single-precision floating-point number's exponent and fraction components have 8 and 23 bits, respectively.

B. Bit-vector SMT solving

Modern symbolic execution engines [4]–[6] widely employ combined theories based on bit-vector (BV) SMT theory to model program behaviors, since it is natural for the theory to

[2]https://github.com/zbchen/FPSE_study

represent the variables and operations of programs. Existing solving algorithms for bit-vector SMT theory can be divided into two categories: being *eager*, the approach reducing the input BV formula to a SAT problem eagerly, and being *lazy*, the approach solving a series of gradually refined abstractions of the input BV formula. The former, making good use of efficient SAT solvers, is often considered to be predominant [26]. Nowadays, the state-of-the-art bit-vector solvers usually implement the eager solving algorithm, e.g., STP [27], Z3 [12], CVC4 [28], to name a few.

Specifically, an eager approach usually leverages bulks of word-level rewriting rules such as term substitution and arithmetic normalization to simplify the input BV formula, bit-blasts the simplified formula into a SAT problem, and employs a SAT-based decision procedure for solving. Clearly, the efficiency of the eager approach relies heavily on the output of bit-blasting. Unfortunately, just as the name implies, *bit-blasting* will introduce a sea of boolean variables and clauses to encode even single bit-vector operations, which imposes a heavy burden on the SAT solver. For example, the multiplication of two 64-bit bit-vector variables brings 20,417 boolean variables and 68,929 clauses [11]. The situation is made even worse in the case of BVFP SMT theory, where the standard BVFP SMT solvers also use bit-blasting to handle floating-point operations. The encoding is much harder in that it must take into account the restrictions of the IEEE 754 standard, rendering more complicated SAT problems that hinder the SAT solver. As a result, people have to develop some incomplete solving techniques for more efficient solving, as we have clarified in Section I.

C. Dynamic Symbolic Execution

Algorithm 1 shows the main procedure of concolic testing, which is also denoted as dynamic symbolic execution (DSE) [29], [30]. The inputs are a program $\mathcal{P}$ and an initial input I_0. DSE is often implemented in a worklist manner and uses a map $\mathcal{W}$ to store the branches to be explored. We use $b\hat{\in}\mathcal{W}$ to represent that branch b has a value in the map $\mathcal{W}$. DSE executes $\mathcal{P}$ with input I_0 first. Then, the later new inputs will lead the program to different paths. When concolic testing executes $\mathcal{P}$ with input I, it also collects the path condition PC of the current path. Based on PC, the branches to be explored along the current path (denoted by $branch(PC,\mathcal{W})$) will be added to $\mathcal{W}$. $branch(PC,\mathcal{W})$ is defined as follows, where $PC = \bigwedge_{0\leq i\leq n} C_i$ and b_j is the branch corresponding to $\neg C_j$.

$$\{b_j \mapsto (\bigwedge_{0\leq i<j} C_i) \wedge \neg C_j \mid 0 \leq j \leq n\} \tag{3}$$

Note that only the branches not in $\mathcal{W}$ (Line 6) are added.

Then, after each execution, DSE selects (Line 11) a branch b from $\mathcal{W}$ for generating the next input, which is expected to steer $\mathcal{P}$ to the execution along b. We use $\mathcal{V}$ to record the branches that have been selected. DSE then solves (Line 13) b's path condition $\mathcal{W}[b]$. If $\mathcal{W}[b]$ is satisfiable, DSE uses the solution I_b as the next input to execute $\mathcal{P}$. If $\mathcal{W}[b]$ is

Algorithm 1 Dynamic Symbolic Execution

```
DSE($\mathcal{P}, I_0$)
Input: $\mathcal{P}$ is program, and $I_0$ is the initial input
 1: $\mathcal{W}, \mathcal{V} \leftarrow \emptyset$        //branches to be explored
 2: $I \leftarrow I_0$
 3: repeat
 4:   $PC \leftarrow$ ConcolicExecution($\mathcal{P}, I, \mathcal{F}$)
 5:   $\mathcal{B} \leftarrow branch(PC, \mathcal{W})$
 6:   $\mathcal{W} \leftarrow \mathcal{W} \cup \{b_i \mapsto PC_i \mid b_i \mapsto PC_i \in \mathcal{B} \wedge \neg(b_i\hat{\in}\mathcal{W})\}$
 7:   if $\forall b\hat{\in}\mathcal{W} \bullet b \in \mathcal{V}$ then
 8:     break
 9:   end if
10:   repeat
11:     $b \leftarrow$ select($\mathcal{W}, \mathcal{V}$)
12:     $\mathcal{V} \leftarrow \mathcal{V} \cup \{b\}$
13:     $(r, I_b) \leftarrow$ Solve($\mathcal{W}[b]$)
14:     if $r = \texttt{SAT}$ then
15:       $I \leftarrow I_b$
16:     end if
17:   until $r = \texttt{SAT} \vee \forall b\hat{\in}\mathcal{W} \bullet b \in \mathcal{V}$
18: until $\forall b\hat{\in}\mathcal{W} \bullet b \in \mathcal{V}$
```

```c
#include <stdio.h>
#include <math.h>

int foo(float a, float b, float c) {
 if (cos(a) > log(b)) {
  if(sin(a) < log(b)) {
   c = c - 1.0;

   if (c == 1.1)
    printf("Never reach here!\n");
  }
 }

 return 0;
}
```

Fig. 1. A simple numerical program.

unsatisfiable or the solving returns `UNKNOWN`, DSE selects the next branch from $\mathcal{W}$. The branch selection (*i.e.*, select) determines the style of path exploration, *e.g*, depth-first search (DFS) and breadth-first search (BFS). DSE continues the iterations to explore $\mathcal{P}$'s path space until all the branches have been visited or timeout (omitted for the sake of brevity).

D. Illustration Example

Figure 1 shows a simple example program for illustration. The program is a numerical program that contains three symbolic floating-point variables and many invocations of complex mathematical functions, *e.g.*, the natural logarithm function (`log`) and the trigonometric functions (`sin` and `cos`). DSE usually needs to provide environment models for the exploited mathematical functions to analyze the program. Generally, the computations of these functions are implemented by Taylor's approximation series [31], where non-linear floating-point computations exist extensively. Therefore, when exploring these functions, the standard concolic testing engine issues many complicated BVFP formulas that are quite challenging for the state-of-the-art BVFP solvers (*e.g.*, Z3 and CVC4). Finally, the analysis gets stuck and fails to explore the interesting part of the program. In fact, the analysis cannot cover even the first branch in Line 5.

Many approaches have been proposed to address the above problem. Besides improving the decision procedure directly, the remaining approaches try to resort to real arithmetic constraint solving or fuzzing. Both approaches use uninterpreted functions to abstract the behaviors of mathematical functions when collecting path conditions. For example, suppose the initial concrete values of symbolic variables are $\{a \mapsto 2.0, b \mapsto 2.0, c \mapsto 0.0\}$ and the search strategy is DFS, DSE will generate the following path conditions.

(1) $cos(a) > log(b)$
(2) $cos(a) > log(b) \wedge sin(a) < log(b)$
(3) $cos(a) > log(b) \wedge sin(a) < log(b) \wedge c - 1.0 = 1.1$

where a, b and c are 32-bit floating-point variables.

III. Study Design

A. Selected Methods

As introduced in Section I, there are five categories of the existing methods for the symbolic execution of floating-point programs. We briefly introduce each category's selected method and the enhancements we have done for it for our empirical study.

*1) **BVFP SMT theory based method (denoted by BVFP)**:* This category precisely represents the floating-point operations in the program by BVFP SMT theory [11]. The advancement of BVFP solving [15], [16] directly improve the symbolic execution's effectiveness. However, the existing BVFP solvers are still limited for the symbolic execution of real-world programs. As shown by the illustration program in Figure 1, a call to an elementary function in `math.h` (*e.g.*, `sin` or `cos`) may produce very complex BVFP constraints because the function's implementation uses intensive floating-point computations. We select Z3 [12] as the solver for this category.

*2) **Real arithmetic solving based optimization (denoted by RSO)**:* This category employs real arithmetic constraint solving [17] to optimize BVFP constraint solving, which considers the BVFP formula as a real arithmetic formula and employs a real arithmetic solver to get a solution in rational numbers. Then, the solution is converted to floating-point values. The approaches of this line utilize the ability of real arithmetic solver to improve the solving's efficiency, *e.g.*, the real arithmetic formula $x^3 = 27$ can be solved in 0.15 seconds by Z3 (Z3 also supports real arithmetic formulas). However, the real arithmetic solver is not sound for BVFP formulas. In case the floating-point values of the rational solution do not satisfy the floating-point formula, the existing approaches search the nearby floating-point values for the solution, which may be inefficient in practice. On the other hand, if the real solver proves the unsatisfiability of the formula, the BVFP formula may be satisfiable, *e.g.*, associativity does not hold for floating-point numbers but holds for real numbers. Besides, the scalability problem still exists when solving complex non-linear real arithmetic formulas and `UNSAT` formulas. Furthermore, these approaches can not support the non-arithmetic operations (*e.g.*, bit operations) due to the ability of the real arithmetic solver.

In our study, we have implemented the RSO method in [17] and enhanced it by employing dReal [32] as the real arithmetic solver, which is efficient and supports most elementary functions in `math.h`. Moreover, we collect the path conditions with uninterpreted functions supported by dReal. Besides, our enhancement supports the solving of the formulas with multiple variables. We also employ fuzzing to facilitate the solution searching in case the real number solution does not satisfy the formula.

*3) **Fuzzing based method (denoted by FUZZ)**:* This category employs fuzzing for solving BVFP formulas [18]. The approaches of this line convert a solving problem into a fuzzing problem of a program and ensure that the input crashing the program satisfies the BVFP constraints. Fuzzing is effective for solving the constraints with a large solution space. For example, the following BVFP formula can be solved by fuzzing in 1 second.

$$x^3 \geq 27 \tag{4}$$

However, fuzzing's performance becomes poor when the solution space is small. Besides, when the BVFP formula is larger and more complex, the fuzzing may also fail to find the solution. We use JFS [18] in our study.

*4) **Integer simulation based conversion method (denoted by ISC)**:* This category's key idea is to convert a floating-point program into one with only integer operations. Specifically, they replace each floating-point operation with a method invocation where the method simulates the operation by integer and bit operations. For example, for the subtraction operation $a - b$ of two 32-bit floating-point numbers, the replacing function $\mathcal{F}_{sub}$ takes two integer inputs whose binary values are the same as those of a and b. Then, $\mathcal{F}_{add}$ implements the subtraction's requirements in IEEE 754 standard by bit operations, *e.g.*, mantissa alignment, and normalization. We select the method in [19] for the study, whose implementation is based on KLEE. Hence, the replacements of the floating-point operations are carried out on-the-fly. More specifically, we use SoftFloat [33] as the simulation library, which is also suggested by the experimental results in [19].

*5) **Search based method (denoted by Search)**:* There exist many approaches in this category. The basic idea is to define a fitness function $\mathcal{F}$ [22] with respect to the satisfiability of the constraint. Function $\mathcal{F}$'s inputs are the variables in the constraint, and the roots of $\mathcal{F} = 0$ are the solutions for the constraint. Hence, the satisfiability problem is converted to a problem of root calculation that the existing optimization techniques can solve [34], [35]. We use the method in XSat [20] and its implementation goSAT [21] in our study. Given a path condition $PC = \bigwedge_{0 \le i \le n} C_i$ and b_j, where $C_i = e_i \bowtie_i e'_i$, PC's fitness function $\mathcal{F}(\vec{x})$ is defined as follows [21].

$$\mathcal{F}(\vec{x}) \overset{\text{def}}{=} \sum_{0 \le i \le n} d\left(\bowtie_i, e_i, e'_i\right) \tag{5}$$

where,

$$\begin{aligned}
d\left(\le, e_1, e_2\right) &\overset{\text{def}}{=} e_1 \le e_2 \ ? \ 0 : |e_1 - e_2| \\
d\left(<, e_1, e_2\right) &\overset{\text{def}}{=} e_1 < e_2 \ ? \ 0 : |e_1 - e_2| + 1 \\
d\left(\ge, e_1, e_2\right) &\overset{\text{def}}{=} e_1 \ge e_2 \ ? \ 0 : |e_1 - e_2| \\
d\left(>, e_1, e_2\right) &\overset{\text{def}}{=} e_1 > e_2 \ ? \ 0 : |e_1 - e_2| + 1 \\
d\left(==, e_1, e_2\right) &\overset{\text{def}}{=} |e_1 - e_2| \\
d\left(\ne, e_1, e_2\right) &\overset{\text{def}}{=} e_1 \ne e_2 \ ? \ 0 : 1
\end{aligned}$$

goSAT supports the formula with only floating-point variables. However, in practice, path conditions may contain both BVFP and BV constraints. Hence, we also extend the fitness function definition to BV constraints.

B. Benchmark Construction

We use GNU Scientific Library (GSL) [25] as benchmark in the evaluation. The version is 2.7. On the one hand, GSL is a well-known numerical library written in C, providing various useful facilities, including basic mathematical functions, statistics, and sorting. As a result, GSL is widely integrated by many real-world scientific computing applications such as QtiPlot [36], and LabPlot [37]. Intuitively, the implementations of GSL's facilities rely heavily on floating-point operations, especially non-linear operations like floating-point multiplication. For example, trigonometric functions in GSL are implemented by Taylor's approximation series [31], bringing many non-linear floating-point computations. Thereby, it is challenging for symbolic execution to analyze GSL's code. On the other hand, many existing studies on floating-point program analysis also take GSL as their benchmark, including some research targeted at symbolic execution [38], [39]. Hence, we believe GSL is representative for our evaluation, which can reveal the abilities of different approaches.

In our evaluation, we only analyzed the public APIs under the `specfunc` folder, in which the majority of APIs have at least one parameter with a primitive type. The reason is that the state-of-the-art symbolic execution tools usually concretize variables with non-primitive type, *i.e.*, pointer variables. Therefore, we manually construct a driver for each API to enable the dynamic symbolic execution. The driver assigns initial concrete values to the API's parameters, symbolizes them, and invokes the API. In total, we collected 277 programs to construct the benchmark. The total number of lines of code is 30,397.

C. Research Question

We design the following research questions to evaluate different methods:

- **RQ1**: How effective are the five methods mentioned above in improving symbolic execution's ability of path exploration? The number of covered lines can evaluate the effectiveness.
- **RQ2**: How efficient are the five methods achieving the same number of covered lines?
- **RQ3**: How many bugs are found by each method?

D. Experimental Setup

To answer **RQ1** and **RQ2**, we use two deterministic search strategies (*i.e.*, DFS and BFS) to conduct the experiments. For each program in the benchmark, the analysis time is 1 hour. The timeout for constraint solving is 30 seconds. We believe such a relatively large timeout is reasonable for the evaluation. To answer **RQ3**, we further add an online exception check for each floating-point operation. The symbolic execution tool will generate a new path to check the possible exception at each floating-point operation. In this paper, we focus on the following four common exceptions of floating-point programs:

- **Overflow**: If the absolute value of the computed result is greater than the largest representable floating-point value, then overflow occurs.
- **Underflow**: If the absolute value of the computed result is smaller than the smallest representable floating-point value, then underflow occurs.
- **Divide-by-Zero**: If a non-zero floating-point number is divided by zero, then divide-by-zero exception occurs.
- **Invalid**: If zero is divided by zero, it is invalid.

We suggest the readers refer to the IEEE 754 standard [14] for more detailed definitions. Finally, we conducted experiments on all five baselines.

All experiments were conducted on a 104-core server with Intel(R) Xeon(R) Platinum 8269CY CPU @ 2.50GHz CPU and 194GB of memory. The operating system is Ubuntu 18.04. We conducted all the experiments three times to eliminate the randomness, and the experimental results are average values.

IV. Implementation and Results

A. Implementation

We have built a customized concolic execution engine based on KLEE [4] version 2.3. We wrap the solutions of path constraints as seeds and use the seeds as the initial concrete values of concolic execution. No other modifications were performed. We have integrated different backend constraint solvers for the above five methods, as clarified in Sec III. Specifically, the version of Z3 is 4.6.2.

B. Experimental Results

This section gives the experimental results of our evaluation. The answers to the above research questions are as follows.

1) Results of RQ1: The number of covered lines directly reflects the ability of path exploration, which indicates the effectiveness of symbolic execution. Figure 2 and Figure 3 show the global results of covered lines under BFS and DFS, respectively. Let us omit **Synergy** for the moment in this section. The X-axis displays the names of studied methods, and the Y-axis shows the number of the covered lines of code (cLoC). The figure shows that the average results of the

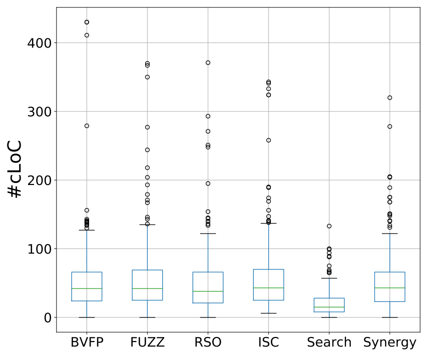

Fig. 2. The results of code coverage under BFS.

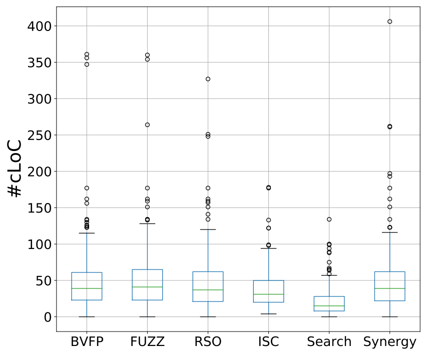

Fig. 3. The results of code coverage under DFS.

first four methods are similar under BFS. However, **Search**'s results are worse. On average, **BVFP**, **FUZZ**, **RSO**, **ISC** and **Search** cover 54, 55, 51, 56, and 21 lines of code (LoC), respectively. Besides, **FUZZ** has more exception results, which is caused by **FUZZ**'s randomness. Under DFS, the results of the first three methods are similar and better than the last two.

Because some benchmarks are in the same source code file, we collect the covered lines of each file to inspect the results in more detail. Figure 4 and Figure 5 show the results. The X-axis shows files located in the `specfunc` folder, where each file contains a collection of APIs with the same functionality. For instance, `log` (*i.e.*, the abbreviation of `gsl_sf_log.h`) is consisted of 5 APIs and implements many complex logarithmic functions. In total, there are 30 files with various functionalities. The Y-axis shows the log value of the number of cLoC.

On average, **BVFP**, **FUZZ**, **RSO**, **ISC** and **Search** cover 494, 510, 473, 515 and 190 LoC per file under BFS, respectively. As show by the figures, **BVFP**, **FUZZ**, **RSO**, **ISC** and **Search** achieve the best result in 11, 12, 9, 13, 0 files under BFS, respectively. As for the DFS strategy in Figure 5, the five methods cover 454, 464, 435, 345, and 193 lines of code per file on average, respectively, and achieve the best results in 12, 13, 10, 4, 0 files, respectively. These results are consistent with the global results shown in Figure 2 and Figure 3.

As indicated by these results, we can observe that the first three methods are stable under both DFS and BFS. **ISC**'s result is better under BFS than that under DFS. **ISC** achieves the best result under BFS but performs worse than the first three under DFS. The reason is that **ISC** generates more inputs because of the conversions for floating-point operations. The path conditions under BFS are simpler than those under DFS. Under BFS, **ISC** can successfully generate more inputs considering more boundary conditions, resulting in better coverage. However, under DFS, **ISC** may be stuck in the floating-point simulation methods.

As shown by these results for code coverage, we have the following findings.

Finding 1: *Under the same time limit, **FUZZ** achieves the global best result for code coverage. Compared with BVFP, the average improvement is not significant. **Search** performs worst for real-world floating-point programs.*

Finding 2: *The results of **BVFP**, **FUZZ** and **RSO** are stable and closer under both DFS and BFS.*

2) Results of RQ2: Besides effectiveness, we want to inspect the efficiency of the methods. The efficiency of a method can be evaluated by the time budget of achieving the same number of covered lines. Figure 6 shows the trends of the number of covered lines under BFS. Similar to the results of *RQ1*, under BFS strategy, **FUZZ** and **ISC** have a better efficiency compared with **BVFP**. **FUZZ** achieves 1.18x speedups for covering the largest number of lines of code under BVFP. In addition, besides **ISC**, **BVFP** is more efficient at the beginning within the first 10 minutes (because the constraints are simpler at the beginning) and becomes slower between 10 and 30 minutes because the constraints become more complex (but still competitive). After 30 minutes, **BVFP** becomes better again and has a similar performance to **FUZZ**, because the constraints under BFS are simpler.

Figure 7 shows the trends of the number of covered lines under DFS. In the beginning, the trends are similar to those under BFS. After 10 minutes, **BVFP**'s efficiency is worse than **FUZZ** and **RSO**. It is because the constraints under BFS become longer, on which BVFP SMT solving is limited. On average, **FUZZ** and **RSO** achieve 4.62x and 3x speedups compared with **BVFP**, respectively.

Finding 3: *Under BFS, **BVFP**'s efficiency is competitive compared with **FUZZ** and **RSO**. Under DFS, **FUZZ** and **RSO** are more efficient than **BVFP**. **Search**'s efficiency is the worst.*

3) Results of RQ3: Besides code coverage, we also inspect the ability of bug finding. Table I and Table II give the results of the bugs found by different methods. Here, we only report the results on four common floating-point exceptions in the

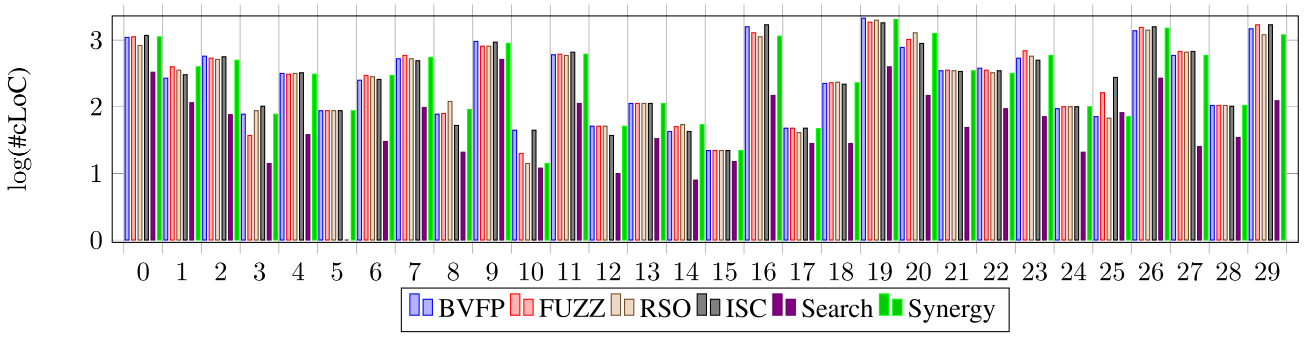

Fig. 4. Results of the covered lines under BFS.

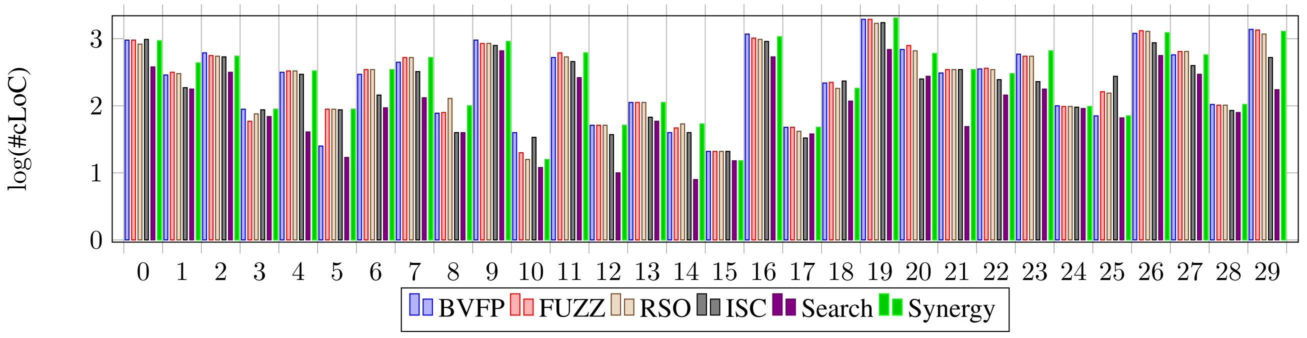

Fig. 5. Results of the covered lines under DFS.

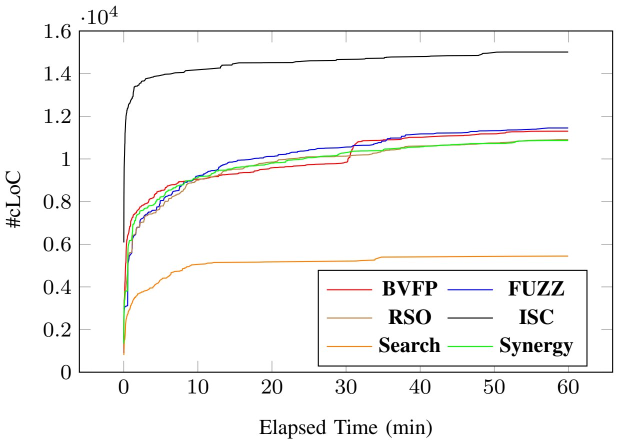

Fig. 6. Trends of covered lines in GSL evaluation under BFS.

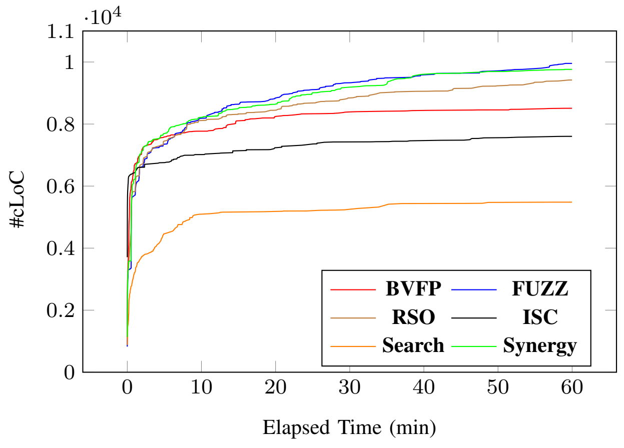

Fig. 7. Trends of covered lines in GSL evaluation under DFS.

TABLE I
NUMBER OF BUGS FOUND BY DIFFERENT METHODS UNDER BFS.

Methods	Overflow	Underflow	Invalid	Div-Zero	All
BVFP	303	955	4	13	1275
FUZZ	459	1488	6	9	1962
RSO	522	1324	4	7	1857
ISC	262	271	5	17	555
Search	1	108	0	8	117
Synergy	557	1576	4	13	2150

TABLE II
NUMBER OF BUGS FOUND BY DIFFERENT METHODS UNDER DFS.

Methods	Overflow	Underflow	Invalid	Div-Zero	All
BVFP	257	594	3	11	865
FUZZ	399	1097	5	12	1513
RSO	436	994	4	12	1446
ISC	75	110	0	9	194
Search	2	108	0	8	118
Synergy	429	1209	5	8	1651

benchmark programs. The last columns of the tables show the total number of bugs found.

As shown by the tables, **FUZZ** detects most bugs under both DFS and BFS. **RSO** ranks second. Although **ISC** achieves the best code coverage under BFS, its effectiveness on bug finding is limited, whose reason is that **ISC** has a larger path space caused by the conversion. Besides, the effectiveness of different types of bugs is different. For example, **RSO** is better for detecting overflows. The reason is that the real arithmetic based approach is more natural for checking overflow exceptions. In addition, **Search** is very limited for bug detection. Note that the detected bugs may be false positives due to no assumption of the calling context.

Finding 4: *FUZZ and **RSO** detect more bugs under both DFS and BFS. Besides, **BVFP**, **FUZZ** and **RSO** are complementary for bug finding.*

C. Threats to Validity

There are internal and external threats to validity. One of the internal threats is the possible path divergence in our dynamic symbolic execution engine. For example, when analyzing real-world programs, paths may diverge due to concretizations and changing environments. Although we have chosen deterministic search strategies during the symbolic execution, path divergence indeed brings some randomness, which might threaten the validity of our work. Another internal threat is our implementation. We ask two senior developers to review the source code to reduce the threat. Besides, we have tested our implementation extensively on hundreds of programs.

The external threats are two folds. One is the representativeness of our benchmark. Although we have constructed a benchmark from a real-world floating-point computation library, the benchmark programs may be limited. The other one is the selected tools of the five existing methods. For example, we use Z3 as the solver for the **BVFP** method and JFS for the **FUZZ** method. We plan to conduct more extensive evaluations of more representative tools (*e.g.*, Bitwuzla [16]) on more benchmark programs in the future.

V. Synergy Method

Based on the findings of our empirical study, we propose synergizing the first three methods. Algorithm 2 gives the details of our synergic constraint solving. The input is a path constraint PC. The output is (r, S) where r can be `SAT`, `UNSAT` or `UNKNOWN`. If r is `SAT`, S is the map that gives the solution value for each variable in PC. Note that PC is a bit-vector formula in which both floating-point expressions and integer expressions may exist; Besides, symbolic method invocations may exist, *e.g.*, `sin(x)`. The algorithm's key idea is to combine BVFP SMT solving, real arithmetic SMT solving, and fuzzing-based solving to improve the efficiency of solving path condition PC.

The algorithm first extracts the *simple* atomic constraints from PC, where an atomic constraint is simple (*i.e.*, $\mathsf{simple}(C_i)$) if C_i satisfies the following two conditions.

- C_i *does not* contain non-linear *floating-point* operations.
- C_i *does not* contain any symbolic method invocations.

Note that here we consider non-linear integer operations as simple operations. Then, the algorithm uses a BVFP solver to solve PC_s. If PC_s is unsatisfiable, it implies that PC is unsatisfiable and the algorithm returns `UNSAT` (Line 4); Otherwise, the algorithm checks whether the solution S_s satisfies PC (denoted by $S_s \models PC$). If S_s satisfies PC, we

Algorithm 2 Constraint Solving

$\mathsf{Solve}(PC)$
Input: $PC = \bigwedge_{0 \le i \le n} C_i$ is a path constraint.
Output: (r, S), where r is the solving result and S is the solution map if r is `SAT`.

```
1:  $PC_s \leftarrow \bigwedge\{C_i \mid 0 \le i \le n \wedge \mathsf{simple}(C_i)\}$
2:  $(r_s, S_s) \leftarrow \mathsf{BVFP\_Solve}(PC_s)$
3:  if $r_s = \mathtt{UNSAT}$ then
4:      return $(\mathtt{UNSAT}, \emptyset)$
5:  end if
6:  if $S_s \models PC$ then
7:      return $(\mathtt{SAT}, S_s)$
8:  end if
9:  $(PC_i, PC_f) \leftarrow \mathsf{Separate}(PC)$
10: $PC_a \leftarrow \mathsf{Abstract}(PC_f)$
11: $(r_r, S_r) \leftarrow \mathsf{REAL\_Solve}(PC_a)$
12: if $r_r = \mathtt{SAT} \wedge \mathsf{FP}(S_r) \models PC_f$ then
13:     return $(\mathtt{SAT}, S_s \downarrow PC_i \cup \mathsf{FP}(S_r))$
14: end if
15: $seed \leftarrow 0$
16: if $r_r = \mathtt{SAT} \wedge \mathsf{FP}(S_r) \not\models PC_f$ then
17:     $seed \leftarrow \mathsf{FP}(S_r)$
18: end if
19: $(r_f, S_f) \leftarrow \mathsf{Fuzzing\_Solve}(PC_f, seed)$
20: return $(r_f, S_s \downarrow PC_i \cup S_f)$
```

find a solution and return (Line 7); Otherwise, the solving procedure continues.

Then, the algorithm separates PC into two parts (Line 9): PC_i and PC_f, where PC_i contains the atomic constrains with only integer expressions and the variables in PC_i do not appear in PC_f. Hence, PC_i is a part of PC_s and S_s satisfies PC_i. The algorithm only needs to solve PC_f. The key idea is to consider PC_f as a real number formula and solve PC_f by a real arithmetic solver. However, because the real solver does not support some elementary arithmetic functions, we need to abstract PC_f first (Line 9). Specifically, the abstraction (*i.e.*, $\mathsf{Abstract}(PC_f)$) in our implementation considers the following cases.

- We abstract `ceil(x)` by introducing a new real number variable y and the constraints $x \le y \le x + 1$.
- Similar to `ceil(x)`, `floor(x)` is abstracted as follows.
$$x - 1 \le y \le x$$
- We consider each variable in PC_f as a real number variable and add the constraints of the maximum and minimum values for the variable's data type. For example, we add the following constraints for an integer variable x.
$$\mathtt{INT\_MIN} \le x \le \mathtt{INT\_MAX}$$

For the abstract formula PC_a, we employ the real arithmetic solver ($\mathsf{REAL\_Solve}(PC_a)$ at Line 11) to solve it. If PC_a is satisfiable, we convert its real number solution S_r to the floating-point solution (denoted by $\mathsf{FP}(S_r)$). Then, if $\mathsf{FP}(S_r)$ satisfies PC_f, we find a solution to merge it with the solution for PC_i (Line 13), where $S_s \downarrow PC_i$ represents S_s's part of the variables in PC_i. If $\mathsf{FP}(S_r)$ does not satisfy PC_f, we employ fuzzing-based solver and use $\mathsf{FP}(S_r)$ as the seed for solving PC_f (Line 19). Here, the intuition is that the fuzzer would be more efficient in finding the solution by starting from $\mathsf{FP}(S_r)$. Besides, if the real arithmetic solver returns `UNSAT`, we also use the fuzzing-based solver to solve PC_f because of the real arithmetic solver's unsoundness.

For example, for the formula $cos(a) > log(b) \wedge sin(a) < log(b)$ of the illustration example, our algorithm utilizes dReal to get a real number solution, but its floating-point solution does not satisfy the formula; Then, based on JFS by seeding the floating-point solution, our algorithm can successfully get a solution. Besides, for the unsatisfiable formula (3) of the illustration example, our algorithm avoids the fruitless search of fuzzing with the help of the SMT solving technique. We get the following partition in the first step.

$$\underbrace{cos(a) > log(b) \wedge sin(a) < log(b)}_{complex} \wedge \underbrace{c - 1.0 = 1.1}_{simple}$$

Firstly, our approach invokes an SMT solver to decide the satisfiability of the simple part. In this case, the simple part is unsatisfiable, which can be decided by the SMT solver efficiently. Note that all variables in the formulas have the type of `Float32`. The loss of precision during the 32-bit floating-point computation leads to unsatisfiability. Therefore, we can determine that the whole formula is unsatisfiable as well.

Experimental Results. As shown in the figures and tables in Section IV, **Synergy** achieves competitive code coverage results under both DFS and BFS. For bug finding, **Synergy** achieves the best result under both DFS and BFS.

Our synergy algorithm is competitive on code coverage and can detect more bugs than the existing methods.

Discussion. Our solving algorithm is sound. To tackle the problem of the real arithmetic solver's unsoundness, we employ the fuzzing-based solver in any case of the real arithmetic solver's result, and the fuzzing-based solver is sound. Besides, although we abstract the floating-point formula (Line 10 in Algorithm 2), we check the validity of the solution with respect to the original path condition PC_f, and the fuzzing-based solving is also for PC_f. On the other hand, our solving algorithm may produce `UNKNOWN` due to the limits of the real arithmetic solver and the fuzzing-based solving, especially when the constraints are too complex.

VI. Related Work

Our work is related to the constraint solving of floating-point formulas. There are already significant advancements for the existing BVFP SMT solvers, including Z3 [12], MathSAT [15] and Bitwuzla [16], *etc*. However, these solvers are still limited for the symbolic execution of real-world floating-point programs because the BVFP constraints are too complex for these solvers to solve. Furthermore, our work is related to the work of real arithmetic SMT solving. Our implementation is based on dReal [32], which supports solving the real number constraints with non-linear arithmetic expressions and many elementary arithmetic functions. XSat [20] also provides a method for solving floating-point constraints in a search-based manner, *i.e.*, converting the solving problem to a mathematical optimization problem, which then can be solved by the existing sampling methods, such as MCMC [34], [35]. Similar idea is also adopted in MLBSE [40] [21] . Our method provides a framework for synergizing these different constraint solving approaches for better symbolic execution of floating-point programs.

Our work also involves optimizing constraint solving during symbolic execution. KLEE [4] uses simplification and cache to reduce the constraint's complexity and solving times, respectively. Green [41] also suggests using cache for optimization and proposes to share the solving results across different programs and analysis tasks. Liu *et al.* [42] proposes utilizing the mechanism of incremental constraint solving during symbolic execution and carries out an empirical study, which suggests employing stack-based incremental constraint solving during symbolic execution. KLEE-Array [43] eliminates array constraints produced by symbolic execution by representing the array operations of the program with non-array expressions. Hence, the constraints issued by symbolic execution do not have array expressions, which improves the constraint solving's efficiency. Another optimization line is to couple symbolic executor and constraint solver more tightly. Zhang *et al.* [38] propose to use the partial solutions during constraint solving to generate multiple test inputs by solving once. Chen *et al.* [44] propose to online synthesize a solving strategy for the program under symbolic execution to improve the solving's efficiency. Shuai *et al.* [45] collect the information of array operations during symbolic execution and pass the information to the array constraint solver to remove the redundant array axioms during constraint solving, which improves the efficiency of array constraint solving.

Our work is orthogonal to the approaches tackling the path explosion problem of symbolic execution. There are two lines of the existing approaches. The first line's approaches propose different search strategies for symbolic execution under different backgrounds, including improving code coverage [46], reaching a program location [47], exploring specific paths [48], *etc*. All these approaches utilize the information calculated by dynamic or static analysis to guide symbolic execution towards finishing the specific tasks in different backgrounds as soon as possible. The other line is to prune the redundant paths of symbolic execution, which complements the first line's approaches. The approaches of this line abstract the states of symbolic execution with respect to different properties, such as reachability [47] and regular properties [49]. Then, the redundant paths classified by the abstraction, *e.g.*, nonreachable or non-violating paths, can be pruned safely, directly improving symbolic execution's scalability.

VII. Conclusion

We have conducted the first empirical study for the symbolic execution of floating-point programs. We study five state-of-the-art methods. The study's results indicate that SMT, fuzzing, and real arithmetic solving-based methods complement each other in bug finding. Based on the finding, we propose a synergic method to further improve the symbolic execution of floating-point. Furthermore, the experimental results indicate that our synergic method detects more bugs.

Our artifact and implementation are available at https://github.com/zbchen/FPSE_study.

ACKNOWLEDGEMENTS

This research was supported by the NSFC Programs (No. 62172429 and 62032024).

References

[1] D. H. Bailey, "High-precision floating-point arithmetic in scientific computation," *Comput. Sci. Eng.*, vol. 7, no. 3, pp. 54–61, 2005.

[2] M. Dowson, "The ariane 5 software failure," *ACM SIGSOFT'97*, vol. 22, no. 2, p. 84, 1997.

[3] M. Zhivich and R. K. Cunningham, "The real cost of software errors," *IEEE Secur. Priv.*, vol. 7, no. 2, pp. 87–90, 2009.

[4] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI'08*, 2008, pp. 209–224.

[5] N. Tillmann and J. de Halleux, "Pex-white box test generation for .net," in *TAP'08*, 2008, pp. 134–153.

[6] 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.

[7] R. S. Shariffdeen, Y. Noller, L. Grunske, and A. Roychoudhury, "Concolic program repair," in *PLDI '21*. ACM, 2021, pp. 390–405.

[8] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: scalable multiline program patch synthesis via symbolic analysis," in *ICSE'16*. ACM, 2016, pp. 691–701.

[9] 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.

[10] "Smt-lib standard logics," https://smtlib.cs.uiowa.edu/logics.shtml/, accessed: 2022-07-07.

[11] D. Kroening and O. Strichman, *Decision Procedures - An Algorithmic Point of View*.

[12] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS'08*, 2008, pp. 337–340.

[13] M. Brain, F. Schanda, and Y. Sun, "Building better bit-blasting for floating-point problems," in *TACAS'19*.

[14] *IEEE standard for binary floating-point arithmetic - IEEE standard 754-1985*. Beuth, 1985.

[15] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The mathsat5 SMT solver," in *TACAS'13*.

[16] A. Niemetz and M. Preiner, "Bitwuzla at the SMT-COMP 2020," *CoRR*, vol. abs/2006.01621, 2020.

[17] E. T. Barr, T. Vo, V. Le, and Z. Su, "Automatic detection of floating-point exceptions," in *POPL '13*. ACM, 2013, pp. 549–560.

[18] 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'19*, 2019, pp. 521–532.

[19] A. Romano, "Practical floating-point tests with integer code," in *VMCAI'14*, K. L. McMillan and X. Rival, Eds., vol. 8318. Springer, pp. 337–356.

[20] Z. Fu and Z. Su, "Xsat: A fast floating-point satisfiability solver," in *CAV'16*, S. Chaudhuri and A. Farzan, Eds., vol. 9780. Springer, 2016, pp. 187–209.

[21] X. Li, Y. Liang, H. Qian, Y. Hu, L. Bu, Y. Yu, X. Chen, and X. Li, "Symbolic execution of complex program driven by machine learning based constraint solving," in *ASE'16*. ACM, 2016, pp. 554–559.

[22] P. McMinn, "Search-based software test data generation: a survey," *Softw. Test. Verification Reliab.*, vol. 14, no. 2, pp. 105–156, 2004.

[23] "Smt-lib qf_fp benchmark," https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_FP/, accessed: 2022-07-07.

[24] "Smt-lib qf_bvfp benchmark," https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BVFP/, accessed: 2022-07-07.

[25] "Gnu scientific library," https://www.gnu.org/software/gsl/, accessed: 2022-07-07.

[26] L. Hadarean, K. Bansal, D. Jovanovic, C. W. Barrett, and C. Tinelli, "A tale of two solvers: Eager and lazy approaches to bit-vectors," in *CAV'14*.

[27] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *CAV'07*.

[28] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *CAV'11*.

[29] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI'05*. ACM, 2005, pp. 213–223.

[30] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ESEC/FSE'05*. ACM, 2005, pp. 263–272. [Online]. Available: https://doi.org/10.1145/1081706.1081750

[31] G. Strang, *Calculus*. SIAM, 1991, vol. 1.

[32] S. Gao, S. Kong, and E. M. Clarke, "dreal: An SMT solver for nonlinear theories over the reals," in *CADE'13*, vol. 7898. Springer, 2013, pp. 208–214.

[33] "Berkeley softfloat," http://www.jhauser.us/arithmetic/SoftFloat.html/, accessed: 2022-07-07.

[34] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan, "An introduction to MCMC for machine learning," *Mach. Learn.*, vol. 50, no. 1-2, pp. 5–43, 2003.

[35] C. P. Robert and G. Casella, *Monte Carlo Statistical Methods*. Springer, 2004.

[36] "Qtiplot - data analysis and scientific visualisation," https://www.qtiplot.com/, accessed: 2022-07-07.

[37] "Labplot - scientific plotting and data analysis," https://labplot.kde.org/, accessed: 2022-07-07.

[38] Y. Zhang, Z. Chen, Z. Shuai, T. Zhang, K. Li, and J. Wang, "Multiplex symbolic execution: Exploring multiple paths by solving once," in *ASE'20*. IEEE, 2020, pp. 846–857.

[39] D. Liew, D. Schemmel, C. Cadar, A. F. Donaldson, R. Zähl, and K. Wehrle, "Floating-point symbolic execution: a case study in n-version programming," in *ASE'17*. IEEE Computer Society, 2017, pp. 601–612.

[40] 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.

[41] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: reducing, reusing and recycling constraints in program analysis," in *FSE'12*. ACM, 2012, p. 58.

[42] T. Liu, M. Araújo, M. d'Amorim, and M. Taghdiri, "A comparative study of incremental constraint solving approaches in symbolic execution," in *HVC'14*, vol. 8855. Springer, 2014, pp. 284–299.

[43] D. M. Perry, A. Mattavelli, X. Zhang, and C. Cadar, "Accelerating array constraints in symbolic execution," in *ISSTA'17*, T. Bultan and K. Sen, Eds. ACM, 2017, pp. 68–78.

[44] Z. Chen, Z. Chen, Z. Shuai, G. Zhang, W. Pan, Y. Zhang, and J. Wang, "Synthesize solving strategy for symbolic execution," in *ISSTA '21*. ACM, 2021, pp. 348–360.

[45] Z. Shuai, Z. Chen, Y. Zhang, J. Sun, and J. Wang, "Type and interval aware array constraint solving for symbolic execution," in *ISSTA '21*. ACM, 2021, pp. 361–373.

[46] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *DSN '09*. IEEE Computer Society, 2009, pp. 359–368.

[47] K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *SAS '11*, E. Yahav, Ed., vol. 6887. Springer, 2011, pp. 95–111.

[48] Y. Zhang, Z. Chen, J. Wang, W. Dong, and Z. Liu, "Regular property guided dynamic symbolic execution," in *ICSE '15*. IEEE Computer Society, 2015, pp. 643–653.

[49] H. Yu, Z. Chen, J. Wang, Z. Su, and W. Dong, "Symbolic verification of regular properties," in *ICSE '18*. ACM, 2018, pp. 871–881.