# Symbolic Execution of Floating-point Programs: How Far Are We?

Xu Yang[a,b,1], Guofeng Zhang[a,b,1], Ziqi Shuai[a,b], Zhenbang Chen[a,b,*], Ji Wang[a,b,*]

[a]*College of Computer, National University of Defense Technology, Changsha, 410073, China*
[b]*State Key Laboratory of Complex & Critical Software Environment, National University of Defense Technology, Changsha, 410073, China*

**Abstract**

Floating-point programs are challenging for symbolic execution due to the constraint solving problem. This paper empirically studies five existing symbolic execution methods for floating-point programs to evaluate their effectiveness and limitations. We have implemented the existing methods based on the state-of-the-art symbolic execution tool KLEE and constructed a real-world floating-point program benchmark for evaluation. We evaluate the existing methods with respect to statement coverage and the ability to detect floating-point exceptions. The results demonstrate that the existing methods complement each other. Based on the evaluation results, we propose a synergistic approach to improving the efficiency of the symbolic execution for floating-point programs. The experimental results indicate our synergistic method's effectiveness in finding floating-point exceptions.

*Keywords:* Symbolic Execution, Constraint Solving, Fuzzing, Floating Point, Real Arithmetic

## 1. Introduction

As the mainstream representation of real numbers, floating-point numbers are ubiquitous in computer systems. Almost every programming language has floating-point data types. Floating-point operations can often be found in the numerical software of different applications, such as scientific computation [1] and secure multi-party computation [2]. However, manipulating floating-point numbers is not easy. There are some famous software disasters in history related to floating-point operations, such as the destruction of the Ariane 5 rocket [3] and the failure of the Patriot missile [4]. It is important and also challenging to ensure the correctness of floating-point programs.

Symbolic execution [5] is a widely used program analysis technique. Compared with other program analysis techniques, the most significant advantage of symbolic execution is its high analysis accuracy. It is an effective method to explore the program's path space systematically. Symbolic execution has been widely used in many software engineering activities in recent years,

*Corresponding author: Zhenbang Chen and Ji Wang

*Email addresses:* `xuyang369@nudt.edu.cn` (Xu Yang), `zbchen@nudt.edu.cn` (Zhenbang Chen), `wj@nudt.edu.cn` (Ji Wang)

[1]Xu Yang and Guofeng Zhang contributed equally to this work and are co-first authors.

including automatic software testing [6, 7], bug detection [8], bug repair [9, 10], *etc*. Although symbolic execution has been successfully applied in many areas, it still faces two main technical challenges: path explosion and constraint solving [11], which are the bottlenecks of symbolic execution's further application and development.

Symbolic execution of floating-point programs is challenging because of the solving of floating-point constraints. Nowadays, to precisely analyze programs, many existing symbolic execution engines use bit-vector (BV) related SMT theories [12] for program path encoding, such as QF_ABV in KLEE [6] and QF_BV in SymCC [13]. For floating-point programs, quantifier-free bit-vector floating-point (QF_BVFP) can be used to represent the program's floating-point operations precisely. In principle, the decision procedure of BV-related SMT theories is NP-complete [12]. However, the constraint solving of QF_BVFP formulas is time-consuming, especially for non-linear QF_BVFP formulas. For example, Z3 [14] (*i.e.*, a state-of-the-art SMT solver) takes 38 seconds to solve $x^3 = 27$ when $x$ is a 64-bit floating-point variable[2].

The real-world QF_BVFP constraints are more complex than the above one, and the constraint solver usually times out. The reason is that QF_BVFP SMT theory precisely encodes the floating-point constraints into SAT problems [15] with respect to the IEEE floating-point standard [16], which may produce complex SAT problems, especially for the non-linear QF_BVFP constraints. For example, a QF_BVFP formula with a multiplication expression of 64-bit variables may produce a SAT problem with tens of thousands of boolean variables and clauses [12]. In practice, representative floating-point programs usually contain intensive non-linear floating-point computation. For example, a call to a function in `math.h` (*e.g.*, `sin` or `cos`) may produce complex QF_BVFP constraints because the function's implementation uses intensive floating-point computations. Therefore, the constraint solving of the QF_BVFP formulas is time-consuming and often times out, which dooms the symbolic execution of floating-point programs. Current state-of-the-art QF_BVFP solvers (*e.g.*, Z3 [14] and MathSAT5 [17]) are still limited in analyzing real-world floating-point programs.

There are five categories of the existing methods for the symbolic execution of floating-point programs.

- The first one employs a QF_BVFP SMT solver for a precise representation. The existing approaches of this category enjoy the precise representation but may suffer from the scalability problem when solving non-linear QF_BVFP constraints.

- The second category uses a real arithmetic solver to optimize the QF_BVFP SMT solving [18], 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 problem of real arithmetic's unsoundness with respect to floating-point numbers.

- The third category utilizes fuzzing to solve QF_BVFP formulas [19], 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

[2]The CPU is 2.5GHz

power of the existing fuzzing techniques [20] but suffers from the incompleteness problem, *i.e.*, it cannot prove the unsatisfiability of the formula.

- The fourth category on-the-fly converts each floating-point operation into an integer and bit operations implemented function [21] during symbolic execution. Therefore, a floating-point program is converted into the one where only integers exist. The existing symbolic execution tools that do not support floating-point operations can be used. However, this category may suffer from the path explosion problem [11] because each function of a floating-point operation introduces many extra paths.
- The last category solves QF_BVFP formulas in a search-based manner [22, 23], which converts the solving problem into an optimization problem by defining a fitness function [24] for the formula.

Despite the advantages of the existing methods, the effectiveness and efficiency of these methods for the symbolic execution of real-world floating-point programs still need to be evaluated. The reasons are: 1) Many methods are only evaluated on the SMT benchmarks, *e.g.*, SMTLIB2 benchmark of QF_BVFP category [25]; 2) Different methods are evaluated on different platforms respectively, so they lack a unified evaluation platform; 3) The evaluation benchmarks are different. 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 choose GNU Scientific Library (GSL) [26], *i.e.*, a widely used scientific computing library with intensive floating-point operations, as the benchmark. We selected 431 programs from the GSL as the benchmark programs and divided them into three categories. Besides, we have integrated or implemented the existing methods of the five categories on KLEE [6], *i.e.*, a state-of-the-art symbolic executor for C programs. Furthermore, we evaluate the existing symbolic execution methods using three criteria, *i.e.*, statement coverage, branch coverage, and the number of detected floating-point exceptions.

Our empirical study finds: 1) QF_BVFP-based and fuzzing-based methods exhibit superior performance regarding statement coverage, branch coverage, and the number of detected exceptions. 2) The real arithmetic optimization-based method is more efficient in the early analysis stage. However, QF_BVFP-based and fuzzing-based methods are more efficient in the later analysis stage. 3) Methods based on integer simulation have poor stability. 4) Search-based methods have no advantages in terms of effectiveness or efficiency.

We propose an algorithm synergizing the first three categories based on these findings. We can use the QF_BVFP SMT solver to prove the unsatisfiability. Then, we can use the real arithmetic solver and fuzzing-based solver 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 QF_BVFP formula $\varphi$, we first employ the QF_BVFP SMT solver to check the $\varphi$'s *simple* part (*i.e.*, the part without non-linear expressions). If $\varphi$'s *simple* part is unsatisfiable, the $\varphi$ is unsatisfiable. Otherwise, we use the solution $S_1$ produced by the QF_BVFP SMT solver to check whether $S_1$ satisfies $\varphi$. If $S_1$ does not, we use a real arithmetic solver to solve $\varphi$. When the solver returns a rational solution $R$, we convert $R$ into a floating-point solution $S_2$ and check whether $S_2$ satisfies $\varphi$. If $S_2$ does not, we use $S_2$ as a seed and employ the fuzzing-based solver solving the $\varphi$. Specifically, we have implemented our Synergy method based on Z3 [14], dReal [27], and JFS [19] and integrated it into KLEE [6]. The results of the experiments on the GSL benchmark programs demonstrate our method's effectiveness.

The main contributions of this paper are as follows.

- We carry out the first extensive study of the state-of-the-art symbolic execution methods for floating-point programs.
- Based on the study's findings, we propose synergizing QF_BVFP SMT solving, real arithmetic SMT solving, and fuzzing for solving QF_BVFP formulas, improving the efficiency of constraint solving.
- We have implemented or integrated the existing methods and our Synergy method into KLEE. Our prototype is available[3], which can be used for future research on the symbolic execution of floating-point programs.
- We have conducted extensive experiments on the GSL benchmark programs. The results show that our Synergy method can cover more statements and detect more exceptions under depth-first search (DFS) and breadth-first search (BFS) strategies than the state-of-the-art methods.

This paper extends our previous work [28]. The extension includes the following aspects.

- We have integrated three more state-of-the-art floating-point constraint solvers, *i.e.*, Bitwuzla [29], MathSAT5 [17] and CVC5 [30].
- We have added more exception checkers for floating-point operations, including invalid elementary function calculations, inexact floating-point arithmetic operations, *etc*.
- We enlarge our benchmark set. We inspect mostly functionality interfaces in GSL and select 431 benchmark programs divided into three categories, totaling 95,112 lines of code. Furthermore, we have conducted a more extensive evaluation of the existing methods on these benchmarks.
- Based on the evaluation, we have collected the QF_BVFP formulas and constructed a set of benchmarks for QF_BVFP SMT theory [4], with a total of 351,639 cases, which can be used for the future research of SMT algorithms and tools.

The remainder of this paper is organized as follows. Section 2 introduces the background of floating-point numbers, QF_BV SMT solving, and concolic testing. Section 3 presents the empirical study design, implementation, and experimental setup. Section 4 presents the evaluation results. Section 5 presents our Synergy method and its evaluation results. Section 6 is Threats to Validity. Section 7 discusses and compares the related work. The last section is the conclusion.

[3]https://github.com/zbchen/FuSE/tree/FPSE-Journal

[4]https://github.com/zbchen/FP-SMTLIB

## 2. Background

### 2.1. IEEE Standard 754

Real numbers exist in computers as floating-point numbers. IEEE 754 is the most widely used floating-point standard [31]. Next, we briefly introduce the floating-point format and floating-point exceptions in IEEE 754.

#### 2.1.1. Floating-point Format

IEEE 754 defines the specification for floating-point numbers and their associated operations. In essence, this standard defines a floating-point number, denoted as $f$, through three components: sign ($S$), mantissa ($M$), and exponent ($E$). The calculation of the number is described by the following formula:

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

$S \in \{0, 1\}$ 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 mantissa, where $m_0$ is the hidden bit and $m_1m_2...m_n$ is the fraction ($F$). Lastly, $E \overset{\text{def}}{=} e - 2^{p-1} + 1$, where $p$ is the number of exponent bits, and $e$ is the biased exponent. In IEEE 754, a single-precision floating-point number's exponent and fraction parts have 8 and 23 bits, respectively. A double-precision floating-point number's exponent and fraction parts have 11 and 52 bits, respectively. For example, suppose that a single-precision floating-point number's binary number is 00111110001000000000000000000000, where $S = 0$ and $p = 8$. Through calculation, $e = 2^6 + 2^5 + 2^4 + 2^3 + 2^2 = 124$, $M = 1 + 2^{-2} = 1.25$, and $E = 124 - 127 = -3$. So, this binary floating-point number represents the decimal floating-point number $1.25 \times 2^{-3}$.

#### 2.1.2. Floating-point Exceptions

Floating-point operations may result in exceptions. Common floating-point exceptions can be divided into five categories: Overflow, Underflow, Divide-By-Zero, Invalid, and Inexact. Table 1 lists the floating-point exceptions and the corresponding checking conditions. In Table 1, $\pm x_{max}$ and $\pm x_{min}$ represent the largest and smallest positive (negative) normalized floating-point numbers that can be represented on the machine, respectively (*i.e.*, $\pm x_{max} = \pm 1.11...11 \times 2^{\mathrm{E_{max}}}$, $\pm x_{min} = \pm 1.0 \times 2^{\mathrm{E_{min}}}$). For example, for 32-bit single-precision floating-point numbers, $E_{max} = 254 - 127 = 127, E_{min} = 1 - 127 = -126$. So $\pm x_{max} = \pm 1.11...11 \times 2^{127}, \pm x_{min} = \pm 1.0 \times 2^{-126}$. The Overflow exception will be reported when the absolute value of the floating-point operation's actual result is greater than the largest representable positive normalized floating-point number; The Underflow exception's checking condition means that the absolute value of the floating-point operation's actual result is greater than 0 and less than the smallest representable positive normalized floating-point number. A Divide-By-Zero exception occurs when a finite non-zero floating-point number is divided by zero. In particular, $log(0)$ also suffers from a Divide-By-Zero exception. Invalid exceptions include cases like the division of zero by zero, a negative operand in a square root function operation ($sqrt$), and a negative operand in a logarithmic function operation ($log$). Since finite floating-point numbers need to represent infinite real numbers, rounding occurs when

Table 1: The floating-point operations that need to be detected and the corresponding checking conditions for these five exceptions. $a$, $b$, and $x$ represent 32-bit or 64-bit normalized floating-point numbers. $\odot$ and $\boxdot$ represent floating-point arithmetic operations (*i.e.*, +, −, ×, or /) with the same finite and infinite precision.

| Exceptions | Operations | Exception checking conditions |
|---|---|---|
| Overflow | $a \odot b$ | $\lvert a \boxdot b \rvert > x_{max}$ |
| Underflow | $a \odot b$ | $0 < \lvert a \boxdot b \rvert < x_{min}$ |
| Divide-By-Zero | $a/b$ | $a \neq 0 \land b = 0$ |
| | $\log x$ | $x = 0$ |
| Invalid | $a/b$ | $a = 0 \land b = 0$ |
| | $\sqrt{x}$ | $x < 0$ |
| | $\log x$ | $x < 0$ |
| Inexact | $a + b$ | $(a + b) - b \neq a \lor (a + b) - a \neq b$ |
| | $a - b$ | $(a - b) + b \neq a \lor a - (a - b) \neq b$ |
| | $a \times b$ | $(b \neq 0 \land (a \times b)/b \neq a) \lor (a \neq 0 \land (a \times b)/a \neq b)$ |
| | $a/b$ | $(b \neq 0 \land (a/b) \times b \neq a) \lor (b \neq 0 \land (a/b) \neq 0 \land a/(a/b) \neq b)$ |

performing arithmetic operations. The last row in Table 1 shows Inexact exceptions due to rounding errors and conditions. It is worth noting that the cases of Invalid and Inexact exceptions are not complete. We only list the ones supported by our implementation.

### 2.2. *Bit-vector SMT solving*

Modern symbolic execution engines [6, 8, 7] usually employ combined theories based on quantifier-free bit-vector (QF_BV) SMT theory to encode programs since the theory is precise to represent the variables and the operations of programs. Existing solving algorithms for bit-vector SMT theory can be divided into two categories: *eager*, the approach reducing the input QF_BV formula to a SAT problem eagerly, and *lazy*, the approach solving a series of gradually refined abstractions of the input QF_BV formula. The former, using efficient SAT solvers, is often considered predominant [32]. Nowadays, state-of-the-art bit-vector solvers usually implement the eager solving algorithm, *e.g.*, STP [33], Z3 [14], and MathSAT5 [17].

The eager approach usually leverages bulks of word-level rewriting rules such as term substitution and arithmetic normalization to simplify the input QF_BV formula, bit-blasts the simplified formula into a SAT problem, and employs a decision procedure for SAT solving. The efficiency of the eager approach relies heavily on the output of bit-blasting [15]. Unfortunately, bit-blasting introduces many boolean variables and clauses to encode bit-vector operations, which may burden the SAT solver heavily. For example, the multiplication of two 64-bit bit-vector variables results in 20,417 Boolean variables and 68,929 clauses [12]. The situation is made even worse in the case of QF_BVFP SMT theory. The encoding is much more complicated because of the requirements in IEEE 754, resulting in more complicated SAT problems. As a result, more efficient constraint solving methods are needed.

**Algorithm 1** Concolic Testing

**Input:** $\mathcal{P}$ is program, $I_0$ is the initial input, and $\mathcal{F}$ is the set of uninterpreted functions

```
1: $\mathcal{W}, \mathcal{V} \leftarrow \emptyset$                                   ▷ branches to be explored
2: $I \leftarrow I_0$
3: repeat
4:     $PC \leftarrow \mathsf{ConcolicExecution}(\mathcal{P}, I, \mathcal{F})$     ▷ $\mathcal{F}$ contains sin, log, etc.
5:     $\mathcal{B} \leftarrow branch(PC, \mathcal{W})$                          ▷ $\mathcal{B}$ is branch
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:        $b \leftarrow \mathsf{Select}(\mathcal{W}, \mathcal{V})$            ▷ select a branch $b$ from the worklist $\mathcal{W}$
12:        $\mathcal{V} \leftarrow \mathcal{V} \cup \{b\}$
13:        $(r, I_b) \leftarrow \mathsf{Solve}(\mathcal{W}[b])$                 ▷ call constraint solver
14:        if $r = \mathtt{SAT}$ then
15:            $I \leftarrow I_b$
16:        end if
17:    until $r = \mathtt{SAT} \vee \forall b \hat{\in} \mathcal{W} \bullet b \in \mathcal{V}$
18: until $\forall b \hat{\in} \mathcal{W} \bullet b \in \mathcal{V}$
```

## *2.3. Concolic Testing*

Algorithm 1 shows the main procedure of concolic testing [34, 35]. The inputs are a program $\mathcal{P}$ and an initial input $I_0$. Concolic testing 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 $\mathcal{W}$. Concolic testing executes $\mathcal{P}$ with input $I_0$ first. Then, the later new input 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 $PC = \bigwedge_{0 \le i \le n} C_i$ and $b_j$ is the branch corresponding to $\neg C_j$.

$$\{b_j \mapsto (\bigwedge_{0 \le i < j} C_i) \wedge \neg C_j \mid 0 \le j \le n\} \tag{2}$$

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

Then, after each execution, concolic testing 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. Concolic testing then solves (Line 13) $b$'s path condition $\mathcal{W}[b]$. If $\mathcal{W}[b]$ is satisfiable, concolic testing uses the solution $I_b$ as the next input to execute $\mathcal{P}$. If $\mathcal{W}[b]$ is unsatisfiable or the solving returns `UNKNOWN`, concolic testing selects the next branch from $\mathcal{W}$. The branch selection (*i.e.*, Select) determines the style of path exploration, *e.g.*, DFS and BFS. Concolic testing continues the iterations to explore $\mathcal{P}$'s path space until all branches have been visited or the time budget is exhausted (omitted for brevity).

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int foo(float a, float b, float c) {
5    if (cos(a) > log(b)) {
6      if(sin(a) < log(b)) {
7        c = c - 1.0;
8
9        if (c == 1.1)
10         printf("Never reach here!\n"); // unreachable code
11     }
12   }
13
14   return 0;
15 }
16
```

Figure 1: A simple program.

## 2.4. *Illustration Example*

Figure 1 shows a simple program illustrating concolic testing. 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`). The implementations of these functions employ numerical methods (*e.g.*, Taylor expansion [36]) to approximate these mathematical functions, which involve complex floating-point operations. Therefore, concolic testing will generate complex floating-point constraints when analyzing the implementation of these functions. When the QF_BVFP SMT solver solves these floating-point constraints, the solver may generate SAT problems that are too complex for the SAT solver or result in a significant time cost, which makes the concolic testing get stuck and fail to analyze some parts of the program. For example, concolic testing cannot even enter the true branch of the branch statement at Line 5 for the example program in Figure 1.

Many methods have been proposed to address the above problem. Besides improving the decision procedure directly, some methods employ real arithmetic constraint solving or fuzzing [18, 19]. They 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. Concolic testing will generate the following path conditions:

$$\begin{array}{ll} 1) & cos(a) > log(b), \\ 2) & cos(a) > log(b) \wedge sin(a) < log(b), \\ 3) & cos(a) > log(b) \wedge sin(a) < log(b) \wedge c - 1.0 = 1.1. \end{array} \tag{3}$$

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

Here, QF_BVFP SMT solvers do not support the solving of these three constraints. The real arithmetic or fuzzing-based solvers may solve the first two constraints, but they cannot prove the *unsatisfiability* of the third constraint due to the last sub-constraint $c - 1.0 = 1.1$.The sub-constraint's unsatisfiability can be proven by a QF_BVFP SMT solver. Hence, it is desirable to synergize the existing methods to improve the effectiveness of floating-point constraint solving further.

## 3. Study Design

As introduced in Section 1, there are five categories of existing methods for the symbolic execution of floating-point programs. Here, we briefly overview each category and the enhancements we have applied to our empirical study.

### *3.1. Methods*

#### *3.1.1. QF_BVFP SMT Theory Based Method (denoted by **QF_BVFP**)*

This category precisely represents floating-point operations in the program using QF_BVFP SMT theory [12]. The advancement of QF_BVFP solving [17, 29] contributes to the effectiveness of symbolic execution. Nevertheless, existing QF_BVFP solvers still need to be improved in handling real-world programs. As shown in Figure 1, the elementary functions in `math.h` (*e.g.*, `sin` or `cos`) result in highly complex QF_BVFP constraints. The underlying QF_BVFP SMT theory employs the bit-blasting method to convert SMT formulas into SAT formulas, which are solved using a SAT solver. However, when dealing with highly complex QF_BVFP constraints, this transformation can result in intricate SAT formulas, which may present challenges for the SAT solver to solve within a limited time budget. Nevertheless, it is essential to note that methods in this category are both sound and complete.

For this category, we have selected Z3 [14], Bitwuzla [29], and MathSAT5 [17] as the underlying solver for symbolic execution. It is worth noting that the solver selected for this classification supports the QF_ABVFP theory and can be directly integrated into KLEE. However, to unify floating-point SMT theory with other solving methods, we only use them as the QF_BVFP solver, and we refer to this type of method as QF_BVFP for short. In addition, we use ackermannization [37] in implementation to eliminate all array terms in path constraints, *i.e.*, converting a QF_ABVFP formula into a QF_BVFP one.

#### *3.1.2. Real Arithmetic Solving Based Optimization Method (denoted by **RSO**)*

This category employs real arithmetic SMT solving [18] to enhance QF_BVFP constraint solving. It treats QF_BVFP formulas as real arithmetic formulas and utilizes real arithmetic solvers to find the solutions in rational numbers, which are then converted to floating-point values. These methods leverage the efficiency of real arithmetic solvers for certain types of equations. For example, a real arithmetic solver like Z3 can quickly solve the equation $x^3 = 27$ in 0.15 seconds, where x is a real number variable. However, it is important to note that real arithmetic solvers are not sound for QF_BVFP formulas. If the floating-point values derived from the rational solution do not satisfy the QF_BVFP formula, the existing methods resort to searching for the nearby floating-point values, which may be inefficient in practice. On the other hand, if the real arithmetic solver determines that the formula is unsatisfiable, it does not imply that the QF_BVFP formula is also unsatisfiable. For example, properties like associativity may hold for real numbers but not for floating-point numbers. Moreover, scalability is a problem for real arithmetic solvers when dealing with complex non-linear formulas. Additionally, this category is unsuitable for handling non-arithmetic operations, such as bit operations, due to the limitations of real arithmetic solvers. We use RSO as the abbreviation for the methods of this category.

In our study, we implemented the RSO method from [18] and enhanced it using dReal [27] and CVC5 (which supports real number theory) as real arithmetic solvers. This enhanced approach is highly efficient and supports most elementary functions in `math.h` library. We also utilize uninterpreted functions supported by dReal and CVC5 to collect path conditions. Additionally, our enhanced method can handle formulas with multiple variables and employ fuzzing [19] to aid in the search for solutions when the solution derived from the real number solution does not satisfy the formula.

As an example, we consider the second path condition in Formula 3. The real arithmetic solver may provide a rational solution like $\{a \mapsto -5.99231e + 307, b \mapsto 1.93481\}$. However, the corresponding floating-point solution does not satisfy the formula. In such cases, we search $k$ floating-point numbers closest to the rational solution of each variable. In our evaluation, we set the value of $k$ to 11, resulting in $121(11 \times 11)$ possible assignments. Finally, if none of these assignments satisfy the formula, it leads to a substantial overhead in satisfiability checking and produces `UNKNOWN`.

### *3.1.3. Fuzzing Based Method (denoted by **FUZZ**)*

This category employs fuzzing for solving QF_BVFP formulas [19]. The methods of this line convert a solving problem into a fuzzing problem of a program and ensure that the input crashing the program satisfies the QF_BVFP formulas. The input parameters of the program are the variables in the formula, and the output is a fitness value (such as coverage, which is used to guide fuzzing). When the input is an assignment that satisfies the SMT formula, the program can go to a specific location, indicating `SAT`. Fuzzing is a process of searching through iterative feedback, and there may be cases where no solution is found. The result of an unsatisfiable SMT formula may be `UNKNOWN`, so the methods in this category are incomplete.

Fuzzing is suitable for solving non-linear floating-point constraints. For example, using the fuzzing method to solve the formula $x^3 > 27$ can be solved in less than one second. Nevertheless, the performance of fuzzing will be greatly reduced when dealing with smaller solution spaces. For example, when solving the formula $x = 1 \land y = 1$, fuzzing takes more than a minute. In our study, we selected JFS [19] as the tool for this category.

### *3.1.4. Integer Simulation Based Conversion Method (denoted by **ISC**)*

This category transforms a floating-point program into one that only contains integer operations. To achieve this, they substitute each floating-point operation with a method call where the method simulates the operation using integer and bit operations. For instance, consider the subtraction operation $a-b$ involving two 32-bit floating-point numbers. The replacement function $\mathcal{F}_{sub}$ takes two integer inputs with binary values identical to those of $a$ and $b$. Then, $\mathcal{F}_{sub}$ implements the subtraction following the IEEE 754 requirements through bit operations, including mantissa alignment and normalization. In our study, we adopt the method in [21], which is implemented based on the KLEE framework. Consequently, the replacements of floating-point operations are performed one-the-fly during symbolic execution. More specifically, we employ SoftFloat [38] as the simulation library, a choice supported by the experimental results in [21].

### *3.1.5. Search Based Method (denoted by* ***Search****)*

This category encompasses various methods, all sharing a common concept: defining a fitness function $\mathcal{F}$ [24] based on the satisfiability of the constraint. The inputs of function $\mathcal{F}$ are the variables in the constraint, and the roots of the equation $\mathcal{F} = 0$ correspond to the solutions of the constraint. Consequently, the satisfiability problem is transformed into a root calculation problem that can be addressed using the existing optimization techniques [39, 40]. In our study, we employ the method in XSat [22] and its implementation goSAT [41]. Given a path condition $PC = \bigwedge_{0\leq i\leq n} C_i$, where $C_i = e_i \bowtie_i e_i'$, the fitness function $\mathcal{F}(\vec{x})$ for $PC$ is defined as follows [41].

$$\mathcal{F}(\vec{x}) \overset{\text{def}}{=} \sum_{0\leq i\leq n} d\left(\bowtie_i, e_i, e_i'\right), \tag{4}$$

where,

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

goSAT supports formulas with only floating-point variables. However, practical path conditions may include both QF_BVFP and QF_BV constraints. As a result, we have extended the definition of the fitness function to support QF_BV constraints as well. However, the optimization problem itself is challenging. Therefore, there might be cases where finding the optimal value is not feasible, which introduces a potential limitation to the completeness of this classification method.

## *3.2. Benchmark Construction*

The GNU Scientific Library (GSL) [26] serves as our benchmark[5] for evaluation. GSL is a widely recognized numerical library implemented in C, offering a broad range of functionalities encompassing basic mathematical operations, differentiation and integration operations, numerical computation, and so on. It is extensively integrated into various real-world scientific computing applications like QtiPlot [42] and LabPlot [43]. GSL's core functionality heavily relies on floating-point operations, particularly non-linear operations. For instance, the trigonometric functions in GSL are computed using Taylor approximation series [36], involving numerous non-linear floating-point computations. Consequently, performing symbolic execution on GSL code poses significant challenges. Furthermore, GSL has frequently been employed as a benchmark in

[5]GSL's version is 2.7

Table 2: Benchmarks from GSL and the classification.

| Benchmarks | Number of Programs | Total Lines of Code |
|---|---|---|
| Elementary Functions (EF) | 82 | 3792 |
| Mathematical Algorithm (MA) | 131 | 57715 |
| Special Functions (SF) | 218 | 33605 |
| Total | 431 | 95112 |

many prior studies concerning the symbolic execution of floating-point programs [44, 45]. Hence, GSL is a representative benchmark that can effectively evaluate various methods' capabilities.

We inspected the source code of all the APIs in GSL and selected those that met our experimentation criteria. The criteria for identifying eligible APIs are as follows: the function must have floating-point operations, and the source code must contain branch statements controlled by floating-point variables. Most APIs have at least one parameter of a primitive data type that can be directly symbolized. A few APIs have at least one data structure that contains the members with primitive data types, and these member variables can also be symbolized. Consequently, we manually constructed a driver for each API to enable symbolic execution. These drivers assign the initial concrete values to the API parameters, symbolize the parameters, and invoke the API. In total, we collected 431 programs and divided them into three groups based on the functionalities, totaling 95112 lines of code. Table 2 shows details of each benchmark, including its name and abbreviation, the number of programs, and the total lines of code. The test programs in each benchmark in the table implement similar mathematical functions.

– Elementary Functions (EF): the programs that implement basic mathematical functions, such as Trigonometric, Exponential, and Power functions.
– Mathematical Algorithm (MA): the implementations for basic mathematical algorithms, such as differentiation, integration, equation solving, and numerical calculations.
– Special Functions (SF): the functions that implement special mathematical functions, such as Airy, Clausen, and Jacobi elliptic.

## *3.3. Research Questions*

We designed the following research questions to evaluate the five methods:

- **RQ1**: How effective are the above methods in improving symbolic execution's ability to explore paths?
- **RQ2**: How efficient are the above methods for path exploration?
- **RQ3**: How many floating-point exceptions are found by each method?

## *3.4. Implementation*

We have implemented a concolic testing engine based on KLEE [6] version 2.3, and the implementation's idea is inspired by KLEE-Zesti[6]. We encapsulate the solution of a path constraint as a seed and use the seed as the initial concrete value for concolic execution. When the engine executes branch instructions, it uses the concrete values to determine the direction of the branch under the current path and then collects the opposite path condition. When the current execution is completed, the collected opposite branch conditions and their previous constraints are concatenated into a set of constraints and added to the queue to be explored. Then, based on the search strategy, the next branch to be explored is selected, and the solver is called to generate a new seed for the corresponding constraint set. The new seed will be used for the subsequent concolic executions. Besides, we have implemented support for floating-point numbers and eliminated the array constraints through ackermannization. As described in Section 3.1, we have integrated different backend constraint solvers for the above five methods.

The engine uses the GNU Multiple Precision Arithmetic Library (GMP) to extend 32-bit or 64-bit floating-point numbers to 128-bit floating-point numbers to detect Overflow and Underflow exceptions by on-the-fly checking the judgment criterias with respect to the concrete values. Both of these judgment criteria are approximate implementations of the actual criterion. For the exceptions of the remaining three types, the engine detects the exceptions on-the-fly by constructing the constraints corresponding to the conditions of exceptions (*c.f.*, Table 1), which generates new paths to check the possible exceptions of each floating-point operation. For example, for the program $if(b! = 0)$ $ret = a/b$; , the engine will construct two exception constraints: $b \neq 0 \wedge a \neq 0 \wedge b = 0$ (Divide-By-Zero) and $b \neq 0 \wedge a = 0 \wedge b = 0$ (Invalid). If these two constraints are satisfiable, Divide-By-Zero and Invalid exceptions are found. These exceptions will not occur, but if $if(b! = 0)$ is removed, they may occur.

## *3.5. Experimental Setup*

We conducted comparative experiments on five existing methods for symbolic execution of floating-point programs, as well as our proposed Synergy method. The QF_BVFP method encompasses three solvers: Z3 4.13.0, Bitwuzla 0.1.0, and MathSAT5 5.6.11. The RSO method includes two solvers: dReal 4.21.06.2 and CVC5 1.0.5. The ISC method employs Z3 4.13.0 as the backend solver. The FUZZ method utilizes JFS [19], while the Search method uses goSAT [41]. To answer **RQ1** and **RQ2**, we conducted the experiments using two deterministic search strategies, *i.e.*, DFS and BFS. 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.

All the experiments are conducted on a server with Intel(R) Xeon(R) Platinum 8269CY 80-Core CPU@2.50GHz and 192GB of memory. The operating system is Ubuntu 18.04 LTS. Some methods use solvers with internal randomness. For example, the generation of seeds in the FUZZ method and the use of random optimization algorithms in the Search method. This randomness cannot be eliminated. To reduce the impact of randomness on the experimental results, we ran each method 20 times on each benchmark.

[6]https://srg.doc.ic.ac.uk/projects/zesti

Table 3: The mean and confidence interval with a 99% confidence level for the number of covered statements and the number of covered branches for each method. Black bold indicates that the corresponding method is significantly better than other methods *except Synergy*. Gray shading indicates that the corresponding method is significantly better than other methods.

| | | | **QF_BVFP** (Z3) | **QF_BVFP** (Bitwuzla) | **QF_BVFP** (MathSAT5) | **RSO** (dReal) | **RSO** (CVC5) | **ISC** (Z3) | **FUZZ** (JFS) | **Search** (goSAT) | **Synergy** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| NoCS | EF | BFS | 2649.40 (19.93) | 2518.33 (0.54) | 1470.67 (4.34) | 2352.67 (0.27) | 2227.33 (4.22) | 2466.20 (11.83) | **2712.67 (3.53)** | 2185.00 (11.42) | 2749.67 (0.54) |
| | | DFS | 2529.50 (4.64) | 2507.33 (3.92) | 1467.00 (8.20) | 2285.00 (0.81) | 2268.00 (2.44) | 1762.00 (0.48) | **2571.67 (1.09)** | 2193.00 (5.09) | 2616.00 (5.70) |
| | MA | BFS | **7183.15 (11.19)** | **7193.67 (21.05)** | 4602.33 (1.18) | 5683.00 (0.81) | 5359.33 (1.78) | 6597.85 (9.37) | 6794.33 (5.16) | 4526.67 (25.30) | 7109.67 (0.54) |
| | | DFS | 6921.55 (10.67) | **7064.67 (8.90)** | 4663.33 (33.02) | 5669.00 (8.14) | 5275.67 (2.59) | 5148.65 (72.27) | 6499.67 (6.26) | 4541.67 (18.09) | 6972.67 (0.54) |
| | SF | BFS | 14304.65 (25.97) | 15019.33 (41.68) | 8766.00 (17.11) | 13273.33 (2.12) | 12992.00 (4.91) | 13054.25 (70.38) | **15027.67 (40.44)** | 8325.67 (90.39) | 15733.00 (76.73) |
| | | DFS | 12707.25 (18.53) | **13612.33 (10.44)** | 9006.67 (6.39) | 12494.00 (3.26) | 12711.33 (12.85) | 8147.10 (29.54) | 13481.67 (19.87) | 8333.00 (32.71) | 14929.33 (9.28) |
| NoCB | EF | BFS | **842.80 (4.64)** | 803.67 (0.27) | 328.67 (1.09) | 669.67 (0.54) | 665.33 (0.27) | 785.15 (1.89) | **843.67 (1.09)** | 630.67 (4.05) | 802.67 (1.12) |
| | | DFS | 787.55 (1.52) | 795.00 (0.94) | 330.00 (1.88) | 656.67 (0.54) | 664.67 (0.27) | 486.30 (0.55) | **808.67 (0.27)** | 629.67 (4.90) | 775.67 (0.54) |
| | MA | BFS | 2708.65 (9.28) | **2724.33 (0.72)** | 1430.00 (1.24) | 1942.67 (0.27) | 1798.00 (2.15) | 2324.40 (5.81) | 2490.00 (0.94) | 1651.00 (8.63) | 2646.67 (0.27) |
| | | DFS | 2602.35 (6.63) | **2629.00 (4.49)** | 1499.00 (12.80) | 1939.00 (4.89) | 1750.33 (1.90) | 1688.30 (25.08) | 2341.00 (2.62) | 1659.67 (9.17) | 2600.33 (0.27) |
| | SF | BFS | 4918.50 (8.47) | 5076.00 (12.36) | 2965.00 (4.91) | 4613.67 (2.59) | 4457.67 (4.45) | 4601.50 (23.39) | **5176.00 (8.93)** | 2676.67 (29.27) | 5365.33 (21.00) |
| | | DFS | 4444.25 (4.63) | **4629.00 (7.39)** | 3111.33 (9.12) | 4213.33 (1.36) | 4347.67 (2.71) | 2831.45 (8.28) | 4600.33 (6.12) | 2667.67 (6.93) | 5000.67 (6.31) |

## 4. Experimental Results

This section presents the experimental results of the empirical study and analyzes the results in detail. Please first ignore the results of Synergy in this section, which will be explained later in Section 5. For all the results mentioned, we select the best outcome from the same family of methods for comparison with other methods. The number of covered statements (NoCS) and the number of covered branches (NoCB) serve as two key metrics for the evaluation of the overall experiment.

### *4.1. Results of RQ1*

Table 3's first column shows the evaluation metrics. The second column shows the three benchmarks. Columns 4-6, 7-8, 9, 10, 11, and 12 are the results of QF_BVFP, RSO, ISC, FUZZ, Search, and Synergy, respectively. The NoCS rows in Table 3 show the mean and confidence interval[7] of 99% confidence level[8] for the total number of covered statements by each method under BFS and DFS. NoCB rows show the results of branch coverage. It is worth noting that the analysis and discussion of Synergy are temporarily omitted in this section.

As shown by the table, FUZZ achieves the best results on the EF benchmark in covering statements and branches under both BFS and DFS. This is attributed to the numerous invocations of basic mathematical functions in the EF benchmark test programs. The fuzzing-based method solves directly on the original constraint formula without the need to approximate the function into a formula with complex non-linear computations, achieving a better result. One exception is under BFS, the confidence intervals of the mean number of the covered branches by QF_BVFP (Z3) and FUZZ overlap, *i.e.*, $(838.60, 848.06) \cap (842.58, 844.76) \neq \emptyset$. The reason for this is that the path constraints are relatively simpler under BFS.

On MA benchmark programs, QF_BVFP (Bitwuzla) and QF_BVFP (Z3) cover a larger number of statements and branches under both BFS and DFS. This is because the MA benchmark programs

[7]The normal confidence interval of $1 - \alpha$ confidence level can be expressed as $\bar{x}\ (\pm z_{\alpha/2} \frac{\sigma}{\sqrt{n}})$.

[8]The significance level $\alpha$ is 0.01.

implement basic mathematical algorithms and involve numerous equality constraints. Many of these constraints have a small solution space, and the QF_BVFP SMT solvers are better for solving these constraints than the fuzzing-based one.

For the SF benchmark programs, FUZZ covers more statements and branches under BFS, while QF_BVFP (Bitwuzla) exhibits superior performance under DFS. The reason for this is that the path constraints in DFS are more complex and longer, which results in a smaller solution space. Therefore, fuzzing-based methods do not have an advantage.

Overall, RSO's mediocre performance is due to the fact that many real solutions obtained by RSO do not result in floating-point solutions satisfying the constraints, and the subsequent searches also fail. The Search method covers the fewest statements and branches across all benchmark tests. The reason is that the search space of search-based methods only includes normalized floating-point numbers but ignores the denormalized floating-point numbers. Furthermore, the performance of QF_BVFP (MathSAT5) can be ascribed to MathSAT5's limit in solving floating-point constraints, especially when dealing with constraints including elementary functions.

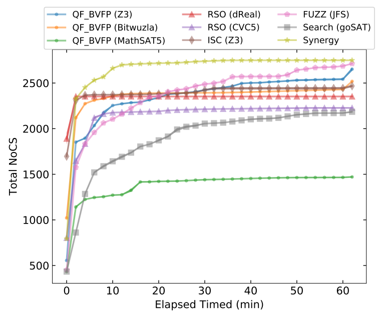
**Finding 1:** *Overall,* ***FUZZ*** *demonstrates the highest performance on EF, whereas* ***QF_BVFP*** *excels on MA. Regarding SF,* ***FUZZ*** *marginally outperforms* ***QF_BVFP*** *under BFS, while* ***QF_BVFP*** *exhibits a slight advantage over* ***FUZZ*** *under DFS.*

**Finding 2:** ***RSO****'s performance is moderate, while* ***Search****'s performance is the least satisfactory.* ***ISC****'s performance is notably superior under BFS compared to DFS.*

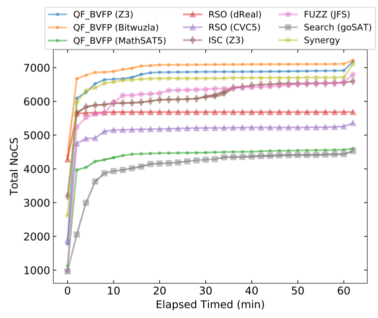
## *4.2. Results of RQ2*

We also evaluate the efficiency of the methods. We conducted a comparative analysis of the trends in the mean number of the covered statements and the mean number of the covered branches over time. *Same as before, please note that Synergy is not considered in this analysis.* Figure 2 shows the trend results with respect to the number of covered statements. Figure 3 displays the trend results of braches. The solid line in the figure represents the mean of the results of multiple runs. The shaded area represents the confidence interval with the 99% confidence level.

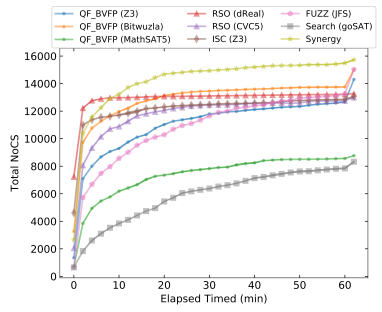
As shown by the figures, on the EF benchmark, QF_BVFP (Bitwuzla) has a clear efficiency advantage in the early stages under BFS, while FUZZ becomes competitive in the later stages. This is because the path constraints in the later stages of BFS become more complex and involve the invocations of basic mathematical functions. However, under DFS, QF_BVFP (Bitwuzla) with propagation-based local search [46] has a significant efficiency advantage. On the MA benchmark, QF_BVFP (Bitwuzla) is more effective under both BFS and DFS. The reason is that Bitwuzla can perform early pruning to reduce the search space. For the SF benchmark, RSO has better efficiency performance. The corresponding floating-point solutions of the real solutions produced by the RSO solver can often satisfy the formula. In particular, RSO (dReal) shows better efficiency than RSO (CVC5), which is due to dReal's interval solving technology. However, due to its unsoundness, the final covered statements and branches are not as many as the FUZZ and QF_BVFP methods.



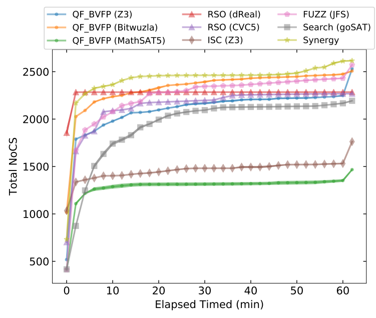
(a) EF(BFS)



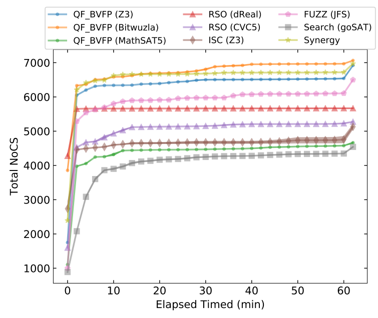
(b) MA(BFS)



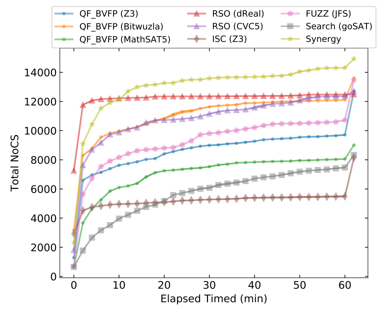
(c) SF(BFS)



(d) EF(DFS)



(e) MA(DFS)



(f) SF(DFS)

Figure 2: A trend of the number of covered statements by each method under DFS and BFS for the three category benchmarks.

Overall, RSO demonstrates a global efficiency advantage in the early stage. QF_BVFP also shows competitiveness across all benchmarks, but its performance is contingent on the employed solver. By observing the slope of the FUZZ line segment, we can see that the efficiency of FUZZ shows a gradual increase. Besides, ISC exhibits the poorest stability, primarily because the simulation of floating-point operations introduces additional paths that exceed KLEE's default maximum memory capacity. Consequently, KLEE resorts to randomly terminating states with a higher probability. The poor efficiency of the Search method can be attributed to its tendency to fall into local search during the search process. Lastly, QF_BVFP (MathSAT5) exhibits the lowest efficiency due to the limitations of MathSAT5's solving capabilities. Additionally, there is a sharp rise in the growth trend of NoCS and NoCB after 60 minutes. The reason is that KLEE generates the corresponding test cases for the remaining unexplored states after timeout (*i.e.*, after 60 minutes). Therefore, according to the time generated by each test case, Figures 2 and 3 show a clear climb after KLEE timeouts. Finally, we draw the following conclusions.

**Finding 3:** *In general, **RSO** is more efficient in the early stage. **QF_BVFP** and **FUZZ** are more efficient in the later stage. The efficiency of **ISC** is medium and unstable. **Search**'s efficiency is the worst.*

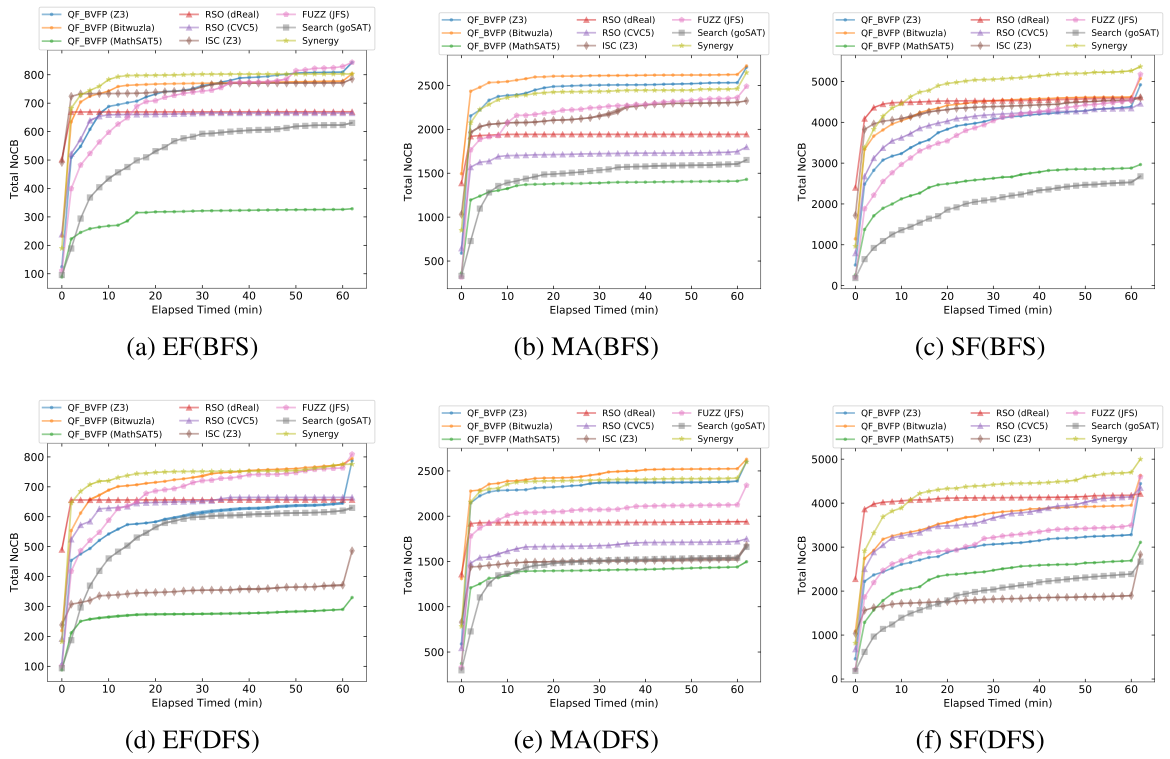


Figure 3: A trend of the number of covered branches by each method under DFS and BFS for the three category benchmarks.

### 4.3. *Results of RQ3*

Table 4 shows the results of the five methods for detecting the floating-point exceptions specified in Table 1. Table 4 shows the mean and confidence interval with 99% confidence of the number of exceptions detected by each method after 20 runs. The table presents the mean value, along with the margin of error indicated in parentheses, which constitutes the confidence interval. The first three columns represent three types of benchmarks, five types of methods, and the employed solvers, respectively. Columns 4-8 are the five types of floating-point exceptions listed in Table 1. The last column is the total number of exceptions detected by each method. Again, the Synergy method is ignored here.

As shown by Table 4, the number of Inexact exceptions is the highest, followed by Underflow and Overflow exceptions. The numbers of Invalid and Divide-By-Zero exceptions are the least. The reason is that Inexact exceptions have not received attention during the development, and the developers care more about Invalid and Divide-By-Zero exceptions, so there are only a small number of Invalid and Divide-By-Zero exceptions detected. The Divide-By-Zero and Invalid exceptions happen at relatively shallow places in the program, so they can be found almost in each run. Therefore, the margin error of the results for these two exceptions is small or 0.

On both the EF and MA benchmarks, FUZZ has a clear advantage in the total number of exceptions detected, especially in detecting Overflow, Underflow, and Inexact exceptions. The reason is that fuzzing usually gives priority to special number seeds (*e.g.*, maximum/minimum normalized

Table 4: The means and confidence intervals at 99% confidence level for the total number of exceptions detected by different methods under BFS and DFS. Black bold indicates that the corresponding method is significantly better than other methods *except Synergy*. Gray shading indicates that the corresponding method is significantly better than other methods.

| Benchmark | Methods | Solvers | Overflow | | Underflow | | Divide-By-Zero | | Invalid | | Inexact | | All | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | BFS | DFS | BFS | DFS | BFS | DFS | BFS | DFS | BFS | DFS | BFS | DFS |
| EF | **QF_BVFP** | Z3 | 23.75 (0.25) | 16.25 (0.25) | 96.00 (7.02) | 73.00 (4.99) | 5.00 (0.00) | 3.00 (0.00) | **9.00 (0.00)** | 4.00 (0.00) | 188.25 (2.35) | 157.25 (2.65) | 322.00 (4.94) | 253.50 (2.87) |
| | | Bitwuzla | 26.75 (0.25) | 24.75 (0.25) | 106.00 (7.98) | 103.50 (6.48) | **8.00 (0.00)** | 7.00 (0.00) | **9.00 (0.00)** | 7.00 (0.00) | 187.50 (2.49) | 194.75 (3.44) | 337.25 (5.74) | 337.00 (3.36) |
| | | MathSAT5 | 5.00 (0.00) | 5.00 (0.00) | 14.25 (0.25) | 14.50 (0.50) | 5.00 (0.00) | 5.00 (0.00) | 5.00 (0.00) | 3.00 (0.00) | 51.75 (1.43) | 51.75 (1.88) | 81.00 (1.68) | 79.25 (2.35) |
| | **RSO** | dReal | 26.75 (0.75) | 24.50 (0.50) | 39.00 (3.99) | 37.75 (4.24) | 5.00 (0.00) | 3.00 (0.00) | 7.00 (0.00) | 7.00 (0.00) | 99.25 (0.25) | 93.75 (0.25) | 177.00 (2.99) | 166.00 (3.99) |
| | | CVC5 | 11.50 (0.29) | 9.50 (0.29) | 0.00 (0.00) | 0.00 (0.00) | 6.00 (0.00) | 4.00 (0.00) | 4.00 (0.00) | 4.00 (0.00) | 57.25 (0.25) | 56.75 (0.25) | 78.75 (0.48) | 74.25 (0.25) |
| | **ISC** | Z3 | 2.00 (0.00) | 2.00 (0.00) | 15.50 (0.50) | 8.50 (0.29) | 5.00 (0.00) | 3.00 (0.00) | 6.00 (0.00) | 3.00 (0.00) | 106.50 (0.86) | 28.25 (0.63) | 135.00 (1.35) | 44.75 (0.63) |
| | **FUZZ** | JFS | **56.75 (0.25)** | **55.75 (0.25)** | **134.50 (0.29)** | **127.25 (0.25)** | **8.00 (0.00)** | **8.00 (0.00)** | 8.00 (0.00) | **8.00 (0.00)** | **252.75 (0.25)** | **235.25 (0.25)** | **460.00 (0.41)** | **434.25 (0.25)** |
| | **Search** | goSAT | 9.75 (1.49) | 7.50 (0.86) | 9.75 (0.25) | 9.00 (1.22) | 5.00 (0.00) | 4.00 (0.00) | 6.00 (0.00) | 6.00 (0.00) | 128.25 (1.43) | 137.50 (4.78) | 158.75 (3.08) | 164.00 (6.62) |
| | **Synergy** | −− | 65.75 (1.25) | 55.25 (0.48) | 163.75 (2.24) | 146.50 (5.16) | 7.00 (0.00) | 5.00 (0.00) | 10.00 (0.00) | 9.00 (0.00) | 256.50 (0.86) | 240.00 (1.68) | 503.00 (4.33) | 455.75 (6.96) |
| MA | **QF_BVFP** | Z3 | 169.25 (2.59) | 145.50 (2.17) | 274.50 (16.21) | 247.50 (16.80) | 33.25 (0.25) | 21.75 (0.25) | 49.50 (0.50) | 41.25 (0.25) | 384.25 (10.60) | 347.00 (11.00) | 910.75 (3.61) | 803.00 (3.78) |
| | | Bitwuzla | 202.25 (2.24) | 194.00 (2.34) | 294.25 (28.68) | 281.25 (27.35) | 31.50 (0.50) | 24.25 (0.25) | 43.75 (0.25) | 37.75 (0.25) | 444.25 (4.30) | 430.50 (6.88) | 1016.00 (21.96) | 967.75 (18.26) |
| | | MathSAT5 | 3.00 (0.00) | 2.00 (0.00) | 26.25 (0.63) | 26.25 (0.63) | 2.00 (0.00) | 2.00 (0.00) | 27.75 (0.25) | 27.75 (0.25) | 113.50 (2.84) | 112.00 (2.34) | 172.50 (3.19) | 170.00 (2.70) |
| | **RSO** | dReal | 112.75 (0.25) | 113.00 (0.00) | 53.00 (7.98) | 54.00 (7.98) | 19.75 (0.25) | 19.75 (0.25) | 39.75 (0.25) | 39.75 (0.25) | 160.25 (0.25) | 162.25 (0.25) | 385.50 (8.48) | 388.75 (8.23) |
| | | CVC5 | 54.75 (0.25) | 52.75 (0.25) | 0.00 (0.00) | 0.00 (0.00) | 29.75 (0.25) | 26.75 (0.25) | 43.75 (0.25) | 43.75 (0.25) | 83.25 (0.63) | 88.00 (0.00) | 211.50 (0.50) | 211.25 (0.48) |
| | **ISC** | Z3 | 4.75 (0.75) | 3.00 (0.58) | 50.50 (1.89) | 12.50 (2.06) | 14.25 (0.48) | 9.00 (1.15) | 39.75 (0.25) | 26.25 (0.25) | 208.50 (3.56) | 48.75 (4.18) | 317.75 (4.86) | 99.50 (7.78) |
| | **FUZZ** | JFS | **206.50 (3.49)** | **190.75 (1.75)** | **341.25 (0.94)** | **319.75 (0.48)** | **34.75 (0.25)** | **27.75 (0.25)** | **59.00 (1.00)** | **51.00 (1.00)** | **497.50 (0.50)** | **485.50 (1.50)** | **1139.00 (3.67)** | **1074.75 (2.42)** |
| | **Search** | goSAT | 49.50 (0.96) | 49.75 (0.63) | 15.25 (0.75) | 14.00 (0.58) | 26.50 (0.86) | 16.75 (0.63) | 39.75 (0.94) | 40.00 (1.00) | 273.25 (5.21) | 277.25 (9.21) | 404.25 (8.56) | 397.75 (11.67) |
| | **Synergy** | −− | 247.25 (1.97) | 222.50 (1.84) | 404.00 (1.00) | 394.00 (1.00) | 35.75 (0.25) | 29.75 (0.25) | 56.25 (0.25) | 56.25 (0.75) | 561.00 (1.00) | 573.25 (1.11) | 1304.25 (3.94) | 1275.75 (0.75) |
| SF | **QF_BVFP** | Z3 | 149.00 (0.41) | 128.00 (0.58) | 335.25 (31.01) | 260.00 (20.62) | 6.25 (0.25) | 6.75 (0.25) | 19.75 (1.25) | 14.50 (0.50) | 761.25 (4.58) | 648.00 (2.99) | 1271.50 (27.12) | 1057.25 (18.40) |
| | | Bitwuzla | 188.75 (1.75) | 180.00 (0.81) | 400.75 (41.49) | **364.75 (37.19)** | **12.50 (1.19)** | **10.50 (0.29)** | **25.75 (0.25)** | **21.75 (0.25)** | 900.75 (16.58) | **862.75 (2.17)** | 1528.50 (22.62) | **1439.75 (37.04)** |
| | | MathSAT5 | 8.75 (0.75) | 13.75 (0.75) | 72.00 (3.07) | 70.75 (1.43) | 5.50 (0.50) | 6.25 (0.25) | 8.50 (0.29) | 7.75 (0.25) | 319.25 (6.90) | 315.50 (6.48) | 414.00 (11.37) | 414.00 (8.65) |
| | **RSO** | dReal | 155.75 (0.25) | 157.75 (0.25) | 200.50 (11.47) | 184.75 (12.22) | 7.00 (0.00) | 7.00 (0.00) | 13.00 (0.00) | 11.00 (0.00) | 513.75 (0.25) | 485.50 (0.29) | 890.00 (11.97) | 846.00 (12.64) |
| | | CVC5 | 110.50 (0.50) | 110.25 (0.48) | 1.00 (0.00) | 1.00 (0.00) | 9.00 (0.00) | 9.00 (0.00) | 10.00 (0.00) | 9.00 (0.00) | 299.00 (2.70) | 307.25 (2.59) | 429.50 (3.19) | 436.50 (2.21) |
| | **ISC** | Z3 | 5.00 (0.58) | 3.00 (0.58) | 54.00 (1.00) | 23.25 (1.03) | 8.00 (0.00) | 4.00 (0.00) | 15.50 (0.29) | 3.00 (0.00) | 502.25 (3.32) | 142.25 (1.84) | 584.75 (3.44) | 175.50 (2.49) |
| | **FUZZ** | JFS | **257.25 (4.24)** | **192.75 (5.74)** | **488.00 (2.37)** | **340.00 (0.71)** | 10.50 (0.29) | 7.25 (0.25) | 20.75 (2.24) | 16.75 (1.25) | **949.75 (4.41)** | 829.75 (3.44) | **1726.25 (8.94)** | 1386.50 (8.82) |
| | **Search** | goSAT | 61.75 (2.35) | 63.25 (1.88) | 20.25 (1.18) | 15.00 (0.58) | 4.25 (0.25) | 4.25 (0.25) | 3.75 (0.25) | 4.50 (0.29) | 366.00 (8.47) | 361.00 (6.74) | 456.00 (11.65) | 448.00 (8.73) |
| | **Synergy** | −− | 256.75 (2.62) | 233.75 (2.24) | 502.00 (1.22) | 468.25 (4.74) | 10.25 (0.25) | 9.25 (0.25) | 14.75 (0.25) | 13.75 (0.25) | 1149.25 (12.66) | 1195.50 (20.12) | 1933.00 (9.41) | 1920.50 (13.14) |

floating-point numbers.). On the SF benchmark, QF_BVFP detects more total exceptions than FUZZ under DFS, which indicates that Bitwuzla's propagation-based local search is better than fuzzing's pure search method when the exception constraints are longer. In terms of detecting Divide-By-Zero and Invalid exceptions, FUZZ and QF_BVFP have their own advantages on different benchmarks. For the EF benchmark, the results are almost the same, while the ones with better results on MA and SF are FUZZ and QF_BVFP, respectively. The reason is that there are more composite expressions in the MA benchmark, *e.g.*, $log(A)$ or $1/A$, where $A$ is a non-linear expression with multiple symbolic variables, on which FUZZ has more advantages. In contrast, there are fewer such scenarios in the SF benchmark. In addition, by comparing QF_BVFP (Z3) and QF_BVFP (Bitwuzla), we know that the same method uses different solvers, and the ability to find exceptions is very different. RSO, ISC, and Search have poor ability to detect exceptions. The main reason is that their ability to explore paths is not as good as QF_BVFP and FUZZ, resulting in them only detecting the exceptions in shallow code.

It is worth noting that we employ an under-constrained way [47] to perform symbolic execution. We built a driver for each analyzed function and symbolized all arguments in the drive. The pre-conditions of the analyzed functions are ignored. So, the exceptions we discovered may not be real bugs. However, this is sufficient to evaluate the ability of the five existing methods to detect exceptions. In summary, we have the following conclusions.

**Finding 4:** *Under BFS and DFS,* ***QF_BVFP*** *and* ***FUZZ*** *can detect more exceptions.* ***RSO, ISC*** *and* ***Search*** *show a poor perfomance on exception detection. Besides,* ***QF_BVFP*** *and*

***FUZZ** are complementary for exception detection.*

## 5. Synergy Method

Our empirical study has yielded several key findings. Notably, QF_BVFP and FUZZ exhibit the highest statement coverage, and RSO performs well in the early stage. Additionally, QF_BVFP and FUZZ demonstrate superior efficiency in the later stages, and these two methods are particularly effective at exception detection. In light of these findings, we propose a synergistic approach involving the integration of the three methods, *i.e.*, QF_BVFP (Z3), RSO(dReal), and FUZZ(JFS). Algorithm 2 shows the details of our synergistic algorithm. This algorithm takes a path constraint $PC$ as the input and produces an output $(r, S)$, where $r$ is `SAT`, `UNSAT`, or `UNKNOWN`. If $r$ is `SAT`, the output is a solution mapping $S$ that provides a value for each variable in $PC$. Notably, $PC$ represents a bit-vector formula containing floating-point and integer expressions. Symbolic method invocations may also exist when using the function in `math.h`, *e.g.*, `sin(x)`. The central idea of this algorithm is the Synergy of QF_BVFP SMT solving, real arithmetic SMT solving, and fuzzing-based solving techniques to enhance the efficiency of constraint solving.

The algorithm initially extracts *simple* atomic constraints from the input path constraint $PC$. An atomic constraint $C_i$ is considered *simple* (denoted by $\mathsf{simple}(C_i)$) if it adheres to the following two conditions.

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

Besides, non-linear integer operations are considered simple operations. Following this, the algorithm employs a QF_BVFP solver to solve the simplified path constraint $PC_s$ (Lines 1-2). If the solver proves that $PC_s$ is unsatisfiable, it implies the unsatisfiability of the original $PC$. The algorithm returns `UNSAT` (Line 4). If $PC_s$ is satisfiable, the algorithm checks whether the solution $S_s$ satisfies the original $PC$ (denoted by $S_s \models PC$). When $S_s$ satisfies $PC$, a solution is identified and returned (Line 7); Otherwise, the solving procedure continues.

Afterwards, the algorithm proceeds to partition the $PC$ into two components (Line 9): $PC_i$ and $PC_f$. $PC_i$ exclusively consists of atomic constraints containing only integer expressions, and the variables found within $PC_i$ are absent from $PC_f$. Consequently, $PC_i$ is an integral part of $PC_s$, and $S_s$ effectively satisfies the constraints within $PC_i$. Consequently, the algorithm's focus narrows down to solving $PC_f$ exclusively. The fundamental idea involves treating $PC_f$ as a real number formula and solving it using a real arithmetic solver. However, due to the real solver's limitations, particularly its limitation in certain elementary arithmetic functions, we need to abstract $PC_f$ (Line 10). This abstraction process (*i.e.*, $\mathsf{Abstract}(PC_f)$) encompasses the following scenarios.

- `ceil(x)` is abstracted by introducing a new real number variable $y$ and applying the constraints $x \leq y \leq x + 1$.
- Similar to the abstraction of `ceil(x)`, `floor(x)` is abstracted as $x - 1 \leq y \leq x$.

**Algorithm 2** Constraint Solving

```
Solve($PC$)
Input: $PC = \bigwedge_{0 \le i \le n} C_i$ is a path constraint.
Output: $(r, S)$, $r$ is the solving result, and $S$ is the solution if $r$ is SAT.
1:  $PC_s \leftarrow \bigwedge\{C_i \mid 0 \le i \le n \wedge \mathsf{simple}(C_i)\}$
2:  $(r_s, S_s) \leftarrow \mathsf{QF\_BVFP\_Solve}(PC_s)$                ▷ call Z3 solver
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:  $(PC_i, PC_f) \leftarrow \mathsf{Separate}(PC)$          ▷ $PC$ is separated into $PC_i$ and $PC_f$
10: $PC_a \leftarrow \mathsf{Abstract}(PC_f)$       ▷ For ceil, floor functions and integer variables
11: $(r_r, S_r) \leftarrow \mathsf{REAL\_Solve}(PC_a)$                 ▷ call dReal solver
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: $(r_f, S_f) \leftarrow \mathsf{Fuzzing\_Solve}(PC_f, seed)$            ▷ call JFS solver
20: return $(r_f, S_s \downarrow PC_i \cup S_f)$
```

- Every variable within $PC_f$ is treated as a real number variable. Add the constraints specifying the variable's maximum and minimum values of the data type. For instance, for an integer variable $x$, we add the following constraints: $\mathtt{INT\_MIN} \le x \le \mathtt{INT\_MAX}$.

For the abstract formula $PC_a$, we utilize 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 $S_r$ into the corresponding floating-point solution, denoted as $\mathsf{FP}(S_r)$. Subsequently, if $\mathsf{FP}(S_r)$ complies with the constraints of $PC_f$, we have found a solution and merged it with the solution for $PC_i$ (Line 13). Here, $S_s \downarrow PC_i$ denotes the part of $S_s$ that gives values to the variables in $PC_i$. However, if $\mathsf{FP}(S_r)$ does not satisfy the conditions of $PC_f$, we employ a fuzzing-based solver and use $\mathsf{FP}(S_r)$ as the initial seed (Line 17). The rationale behind this approach is that the fuzzier will likely be more efficient in discovering the solution. Moreover, in situations where the real arithmetic solver returns UNSAT, we also resort to the fuzzing-based solver (Line 19) due to the real arithmetic solver's unsoundness.

For example, for the path condition $cos(a) > log(b) \wedge sin(a) < log(b)$ of Figure 1's illustration example, our algorithm utilizes dReal to get a real number solution. However, its floating-point solution does not satisfy the formula. Then, our algorithm can successfully obtain the solution by adopting JFS and using dReal's solution as a seed. In addition, 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,

$$\underbrace{cos(a) > log(b) \wedge sin(a) < log(b)}_{complex} \wedge \underbrace{c - 1.0 = 1.1}_{simple}. \tag{5}$$

Synergy invokes an SMT solver to decide the satisfiability of the *simple* part. In this case, the *simple* part is unsatisfiable, which can be proved by the SMT solver efficiently. Note that all formula variables have the type of `Float32`. The loss of precision during the 32-bit floating-point computation leads to unsatisfiability. Therefore, we can prove that the whole formula is unsatisfiable.

## *5.1. Results Analysis*

The basic information of Table 3 is introduced in detail in Section 4.1. The gray shading in the table indicates that the corresponding method is significantly better than other methods. For the EF benchmark, Synergy performs better in the number of covered statements but is inferior to FUZZ in the number of covered branches. On the MA benchmark, QF_BVFP (Bitwuzla) outperforms our method on both metrics. The reason is that the Synergy method is unable to effectively separate the constraints of this benchmark. In most cases, the entire constraint falls into either *simple* or *complex* category. For the SF benchmark, Synergy demonstrates the advantage in both statement and branch coverage.

As evidenced by Figures 2 and 3, Synergy demonstrates clear advantages in execution efficiency across the EF and SF benchmarks, attributed to dReal's rapid solving speed and its boosting to fuzzing by initial seeds. In the MA benchmark, Synergy is inferior to QF_BVFP (Bitwuzla) and QF_BVFP (Z3) under BFS and inferior to QF_BVFP (Bitwuzla) under DFS.

As shown in Table 4, Synergy detects the highest total number of exceptions. However, in terms of detecting Divide-By-Zero and Invalid exceptions, Synergy is not always superior to other methods. The reason is that the complexity of exception constraints determines whether the exceptions can be discovered. For instance, when detecting the Invalid exception of $sqrt(A)$, if $A$ is a complex expression involving multiple symbolic variables, the Synergy method holds an advantage. Conversely, when $A$ is a single symbolic variable, the QF_BVFP method is comparatively more advantageous. Under the DFS of the EF benchmark, the confidence interval of the detected Overflow count by FUZZ overlaps with that of Synergy. We utilize the failed solutions from dReal as the initialization seeds for JFS. Although this idea can enhance the overall performance of Synergy, in some specific scenarios, it does not differ significantly from the original seeds of JFS. A similar situation is observed with the detected Overflow counts by both FUZZ and Synergy under the SF benchmark.

In summary, Synergy demonstrates an advantage in symbolic execution for floating-point programs.

*Compared to other methods, our* ***Synergy*** *exhibits notable advantages regarding statement coverage, branch coverage, and exception detection.*

### *5.2. 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 $PC_f$. On the other hand, our solving algorithm is incomplete and may produce `UNKNOWN` due to the limits of the real arithmetic solver and the fuzzing-based solving, especially when the constraints are too complex.

The real arithmetic solver is critical for our method. First, the solver's scalability impacts the efficiency of our method. Second, the solver's support of elementary arithmetic functions determines the selection of complex functions (*i.e.*, $\mathcal{F}$ of Algorithm 2). In principle, we set the elementary arithmetic functions supported by the solver as complex functions. Because of the same reason, the abstraction before invoking the real arithmetic solver (*i.e.*, $\mathsf{Abstract}(PC_f)$) also depends on the ability of the real arithmetic solver. Consequently, for operations and complex functions unsupported by the solver, the abstraction of these elements becomes imperative.

## 6. Threats to Validity

There are internal and external threats to validity. One of the internal threats is the possible path divergence in our concolic testing 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 asked two senior developers to review the source code to reduce the threat. Besides, we have tested our implementation extensively on hundreds of programs.

There are four aspects to external threats. One is a single symbolic execution tool. Although KLEE is a representative symbolic execution tool, there is no guarantee that these floating-point solvers will have the same experimental results when integrated into other symbolic execution tools. The second is limited representative benchmarks. Although we have constructed many benchmarks from real-world floating-point computing libraries (*i.e.*, GSL), they may still not be representative. Likewise, real-world floating-point programs may not fit into any type of the three benchmarks. The third is limited representative baselines. Although we compared the most advanced floating-point constraint solvers, we did not include all the available solvers. Besides, we do not compare with baseline using array-based SMT theory for encoding the program (*i.e.*, using QF_ABVFP SMT theory). In addition, the differences between different solvers will affect the final performance of the method. For example, QF_BVFP (Z3), QF_BVFP (Bitwuzla), and QF_BVFP (MathSAT5) give very different results. Finally, in terms of exception detection, while we detect the vast majority of common floating-point exceptions, we do not fully cover all floating-point exceptions in IEEE 754 (*e.g.*, there are other operations that can cause Invalid and Inexact exceptions). The Overflow and Underflow exception checking conditions may have false positives, for example, the result of $x_{max} + 1$ will be rounded to $x_{max}$. Obviously, an Overflow exception does not occur in the example given, yet our method might flag it as such. In general, our exception checking conditions are sound but incomplete.

## 7. Related Work

Our work is closely related to the symbolic execution of floating-point programs. Lakhotia *et al.* [48] propose a search-based method for solving floating-point constraints in the symbolic execution engine Pex [7]. The key idea is to create a fitness function for each constraint. The constraint solving is converted to the root search problem of the fitness function. This method's critical aspect is the fitness function's design, which may be limited to non-arithmetic operations. Besides, this method still faces scalability problems when the constraint becomes complex. Romano [21] proposes to replace floating-point operations with the corresponding integer-implemented versions for analyzing floating-point programs. Then, the program with floating-point operations is converted to one with only integer operations. The path explosion problem poses a significant challenge to this method, as it requires that each floating-point operation be replaced by a corresponding function call. This method becomes infeasible for real-world numeric programs with intensive floating-point operations (*e.g.*, GSL benchmark programs). Barr *et al.* [18] propose Ariadne, which utilizes real arithmetic solving to improve symbolic execution's floating-point constraint solving. When finding a rational solution, Ariadne converts it to a floating-point solution and tests its validity. If the floating-point solution does not satisfy the constraint, Ariadne searches the solution's nearby values for the targets that satisfy the constraint. Ariadne inspires our method. Compared with Ariadne, we have the following advantages. First, Ariadne only supports solving the constraints with a single floating-point variable. If there are multiple variables, Ariadne simplifies the constraint by concretization, which may cause the failure to find solutions. Our method supports solving the constraints with multiple variables. Second, Ariadne abstracts the elementary arithmetic functions (*e.g.*, `sin` and `cos`) with fresh symbolic variables, which makes the analysis imprecise. Leveraged by dReal, our method does not abstract these functions and has better precision. Third, Ariadne only searches the nearby floating-point values if the floating-point solution does not satisfy the constraint. Our method employs fuzzing by seeding the fuzzing procedure with the solution found by the real arithmetic solver, which is expected to be more effective and efficient, as indicated by the experimental results in Section 4. Last, we also utilize QF_BVFP SMT solving to prove the unsatisfiability, a problem for methods like Ariadne.

Our work is related to the constraint solving of floating-point formulas. Significant advancements have already been made to the existing QF_BVFP SMT solvers, including Z3 [14], MathSAT [17], Bitwuzla [49], *etc.* However, these solvers are still limited for the symbolic execution of real-world floating-point programs because the QF_BVFP constraints are too complex for these solvers to solve. On the other hand, JFS [19] proposes to employ fuzzing to solve the floating-point constraints and achieve a better result than state-of-the-art solvers on floating-point related SMTLIB benchmarks [50, 25]. JFS generates a program from a constraint and ensures that the inputs crashing the program satisfy the constraint. Then, JFS employs the existing fuzzing tool to fuzz the program to find the solution. JFS is effective for the constraints with a large space of solutions but limited for the constraints with small solution spaces. Besides, when the constraints are pretty complex, the fuzzing procedure also faces challenges. Our method utilizes the solution generated by the real arithmetic solver to improve the fuzzing procedure. It shows a better result than the pure fuzzing method (as demonstrated by the results in the evaluation). Furthermore, our work is related to real arithmetic SMT solving. Our implementation is based on dReal

[27], which supports solving the real number constraints with non-linear arithmetic expressions and many elementary arithmetic functions. XSat [22] 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 can then be solved by the existing sampling methods, such as MCMC [39, 40]. A similar idea has also been adopted in MLBSE [51][23]. Our method provides a framework for synergizing these different constraint solving methods for better symbolic execution of floating-point programs.

Our work also involves optimizing constraint solving during symbolic execution. KLEE [6] uses simplification and cache to reduce the constraint's complexity and solving times, respectively. Green [52] also suggests using cache for optimization and proposes to share the solving results across different programs and analysis tasks. Liu *et al.* [53] propose utilizing the mechanism of incremental constraint solving during symbolic execution and carry out an empirical study that suggests employing stack-based incremental constraint solving during symbolic execution. KLEE-Array [54] 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 the symbolic executor and the constraint solver more tightly. Zhang *et al.* [44] propose using partial solutions during constraint solving to generate multiple test inputs by solving once. Chen *et al.* [55] propose synthesizing a solving strategy for the program under symbolic execution to improve the efficiency of solving. Shuai *et al.* [56] 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 methods tackling the path explosion problem of symbolic execution. There are two lines of the existing methods. The first line's methods propose different search strategies for symbolic execution under different backgrounds, including improving code coverage [57], reaching a program location [58], exploring specific paths [59], *etc.* All these methods 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 methods. The methods in this line line abstract the states of symbolic execution with respect to different properties, such as reachability [58] and regular properties [60]. Then, the redundant paths classified by the abstraction, *e.g.*, non-reachable or non-violating paths, can be pruned safely, directly improving symbolic execution's scalability.

## 8. Conclusion

We conduct the first empirical study of the symbolic execution of floating-point programs. Our research subjects are five state-of-the-art methods: QF_BVFP, RSO, ISC, FUZZ, and Search. To make the research results credible, we collected up to 431 benchmarks and divided them into three groups. Comparative experiments are conducted on three sets of benchmarks, and the code coverage and exception detection results are discussed and analyzed in detail. Finally, it is found that QF_BVFP and FUZZ have better code coverage abilities, RSO, QF_BVFP, and FUZZ have better efficiency for code coverage, and QF_BVFP and FUZZ complement each other in exception detection. Based on this finding, we propose the Synergy method to improve the symbolic execution of floating-point further. Experimental results show that our Synergy method has better code coverage and can detect more exceptions.

There are three aspects for future work: 1) enlarge the scope of benchmark programs by incorporating more representative floating-point programs; 2) investigate more advanced constraint separation to enhance the Synergy method further; 3) integrate bit-blasting and fuzzing-based techniques in a more synergistic way to further improve the performance of solving floating-point constraints.

## Acknowledgments

This research is supported by National Key R&D Program of China (No. 2022YFB4501903) and the NSFC Programs (No. 62172429 and 62032024).

## References

[1] D. H. Bailey, High-precision floating-point arithmetic in scientific computation, Comput. Sci. Eng. 7 (3) (2005) 54–61.

[2] Y.-C. Liu, Y.-T. Chiang, T.-S. Hsu, C.-J. Liau, D.-W. Wang, Floating point arithmetic protocols for constructing secure data analysis application, Procedia Computer Science 22 (2013) 152–161.

[3] J.-L. Lions, L. Luebeck, J.-L. Fauquembergue, G. Kahn, W. Kubbat, S. Levedag, L. Mazzini, D. Merle, C. O'Halloran, Ariane 5 flight 501 failure report by the inquiry board (1996).

[4] G. A. O. W. D. I. MANAGEMENT, T. DIV, Patriot missile defense: Software problem led to system failure at dhahran, saudi arabia (1992).

[5] J. C. King, Symbolic execution and program testing, Communications of the ACM 19 (7) (1976) 385–394.

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

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

[8] P. Godefroid, M. Y. Levin, D. A. Molnar, SAGE: whitebox fuzzing for security testing, Commun. ACM 55 (3) (2012) 40–44.

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

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

[11] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, I. Finocchi, A survey of symbolic execution techniques, ACM Comput. Surv. 51 (3) (2018) 50:1–50:39.

[12] D. Kroening, O. Strichman, Decision Procedures - An Algorithmic Point of View, Springer, 2008.

[13] S. Poeplau, A. Francillon, Symbolic execution with SymCC: Don't interpret, compile!, in: 29th USENIX Security Symposium (USENIX Security 20), USENIX Association, 2020, pp. 181–198.
URL `https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau`

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

[15] M. Brain, F. Schanda, Y. Sun, Building better bit-blasting for floating-point problems, in: TACAS'19, Vol. 11427, Springer, 2019, pp. 79–98.

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

[17] A. Cimatti, A. Griggio, B. J. Schaafsma, R. Sebastiani, The MathSAT5 SMT solver, in: TACAS'13, Vol. 7795, Springer, 2013, pp. 93–107.

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

[19] D. Liew, C. Cadar, A. F. Donaldson, J. R. Stinnett, Just fuzz it: solving floating-point constraints using coverage-guided fuzzing, in: ESEC/FSE'19, 2019, pp. 521–532.
[20] LibFuzzer, 2015, Online, Available: `http://llvm.org/docs/LibFuzzer.html`.
[21] A. Romano, Practical floating-point tests with integer code, in: International Conference on Verification, Model Checking, and Abstract Interpretation, Springer, 2014, pp. 337–356.
[22] Z. Fu, Z. Su, XSat: A fast floating-point satisfiability solver, in: S. Chaudhuri, A. Farzan (Eds.), CAV'16, Vol. 9780, Springer, 2016, pp. 187–209.
[23] X. Li, Y. Liang, H. Qian, Y. Hu, L. Bu, Y. Yu, X. Chen, X. Li, Symbolic execution of complex program driven by machine learning based constraint solving, in: ASE'16, ACM, 2016, pp. 554–559.
[24] P. McMinn, Search-based software test data generation: a survey, Softw. Test. Verification Reliab. 14 (2) (2004) 105–156.
[25] Smt-lib qf_bvfp benchmark, `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BVFP/`, accessed: 2022-07-07.
[26] Gnu scientific library, `https://www.gnu.org/software/gsl/`, accessed: 2022-07-07.
[27] S. Gao, S. Kong, E. M. Clarke, dReal: An SMT solver for nonlinear theories over the reals, in: CADE'13, Vol. 7898, Springer, 2013, pp. 208–214.
[28] G. Zhang, Z. Chen, Z. Shuai, Symbolic execution of floating-point programs: How far are we?, in: 2022 29th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2022, pp. 179–188.
[29] A. Niemetz, M. Preiner, Bitwuzla, in: C. Enea, A. Lal (Eds.), Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II, Vol. 13965 of Lecture Notes in Computer Science, Springer, 2023, pp. 3–17.
[30] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, Y. Zohar, CVC5: A versatile and industrial-strength SMT solver, in: D. Fisman, G. Rosu (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I, Vol. 13243 of Lecture Notes in Computer Science, Springer, 2022, pp. 415–442.
[31] D. Goldberg, What every computer scientist should know about floating-point arithmetic, ACM computing surveys (CSUR) 23 (1) (1991) 5–48.
[32] L. Hadarean, K. Bansal, D. Jovanovic, C. W. Barrett, C. Tinelli, A tale of two solvers: Eager and lazy approaches to bit-vectors, in: CAV'14, Vol. 8559, Springer, 2014, pp. 680–695.
[33] V. Ganesh, D. L. Dill, A decision procedure for bit-vectors and arrays, in: CAV'07, Vol. 4590, Springer, 2007, pp. 519–531.
[34] P. Godefroid, N. Klarlund, K. Sen, DART: directed automated random testing, in: PLDI'05, ACM, 2005, pp. 213–223.
[35] K. Sen, D. Marinov, G. Agha, CUTE: a concolic unit testing engine for C, in: ESEC/FSE'05, ACM, 2005, pp. 263–272.
[36] G. Strang, Calculus, Vol. 1, SIAM, 1991.
[37] D. M. Perry, A. Mattavelli, X. Zhang, C. Cadar, Accelerating array constraints in symbolic execution, in: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2017, pp. 68–78.
[38] Berkeley softfloat, `http://www.jhauser.us/arithmetic/SoftFloat.html/`, accessed: 2022-07-07.
[39] C. Andrieu, N. de Freitas, A. Doucet, M. I. Jordan, An introduction to MCMC for machine learning, Mach. Learn. 50 (1-2) (2003) 5–43.
[40] C. P. Robert, G. Casella, Monte Carlo Statistical Methods, Springer, 2004.
[41] M. A. Ben Khadra, D. Stoffel, W. Kunz, goSAT: Floating-point satisfiability as global optimization, in: FMCAD'17, IEEE, 2017, pp. 11–14.
[42] Qtiplot - data analysis and scientific visualisation, `https://www.qtiplot.com/`, accessed: 2022-07-07.
[43] Labplot - scientific plotting and data analysis, `https://labplot.kde.org/`, accessed: 2022-07-07.
[44] Y. Zhang, Z. Chen, Z. Shuai, T. Zhang, K. Li, J. Wang, Multiplex symbolic execution: Exploring multiple paths by solving once, in: ASE'20, IEEE, 2020, pp. 846–857.

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

[46] A. Niemetz, M. Preiner, Ternary propagation-based local search for more bit-precise reasoning, in: # PLACEHOLDER_PARENT_METADATA_VALUE#, Vol. 1, TU Wien Academic Press, 2020, pp. 214–224.

[47] D. A. Ramos, D. Engler, {Under-Constrained} symbolic execution: Correctness checking for real code, in: 24th USENIX Security Symposium (USENIX Security 15), 2015, pp. 49–64.

[48] K. Lakhotia, N. Tillmann, M. Harman, J. de Halleux, Flopsy - search-based floating point constraint solving for symbolic execution, in: ICTSS'10, Vol. 6435, Springer, 2010, pp. 142–157.

[49] A. Niemetz, M. Preiner, Bitwuzla at the SMT-COMP 2020, CoRR abs/2006.01621 (2020).

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

[51] L. Bu, Y. Liang, Z. Xie, H. Qian, Y. Hu, Y. Yu, X. Chen, X. Li, Machine learning steered symbolic execution framework for complex software code, Formal Aspects Comput. 33 (3) (2021) 301–323.

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

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

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

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

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

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

[58] K. Ma, Y. P. Khoo, J. S. Foster, M. Hicks, Directed symbolic execution, in: E. Yahav (Ed.), SAS '11, Vol. 6887, Springer, 2011, pp. 95–111.

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

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