2016 23rd Asia-Pacific Software Engineering Conference

The Floating-point Extension of Symbolic Execution Engine for Bug Detection

Xingming Wu[1,3], Zhenbo Xu[4], Dong Yan[2,3], Tianyong Wu[1,3], Jun Yan[1,2,3] and Jian Zhang[1,3]
1. State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
2. Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences
3. University of Chinese Academy of Sciences
4. Qihoo 360 Technology Co. Ltd, China

***Abstract*—Many existing symbolic execution engines for bug detection often ignore floating-point types and operations. That will result in imprecise reasoning about the feasibility of program paths, which in turn leads to false positives and negatives. Recently, there are quite some progress in satisfiability modulo theories (SMT) solving, and some tools are able to support floating-point arithmetic. Nevertheless, naturally extending a symbolic execution engine and directly replacing the back-end with the new SMT solver will not make a good static analyzer for floating-point programs.**

In this paper, we extend an existing symbolic execution engine for `C` program bug finding, so that it can deal with floating-point arithmetic and mathematical functions. For the mathematical functions, we employ an abstract model to keep a balance between overhead and precision. We also introduce a strategy, Lazy-verification, to reduce the number of SMT solver calls. We implemented our approach as a tool called `Canalyze-fp`. Experiments with self-developed benchmarks and non-trivial open source programs show that the proposed approach can effectively avoid the false positives and negatives, without introducing too much overhead.

***Index Terms*—floating-point program, symbolic execution, false positive, false negative.**

I. Introduction

To ensure reliability and security of programs, both static [1] and dynamic [2] program analysis techniques have been proposed. Symbolic execution [3], one of prevalent static analysis techniques, is quite useful for analyzing programs that require high reliability due to its effectiveness in practice. It executes the program with symbolic inputs and uses constraint solving to check the feasibility of the program paths [4]. As the symbolic input can be all possible values in the input domain, symbolic execution is able to analyze the program with high coverage.

However, due to the limitation of constraint solver for floating-point constraints, most of existing symbolic execution engines for bug detection often ignore floating-point types and operations and check the satisfiability of floating-point path conditions with simplified strategies, which will lead to false positives and negatives.

Some tools like `CSA` [5], `Saturn` [6], and `Canalyze` [7] assume that the condition is satisfiable and simplify it as a true boolean expression. Without analyzing floating-point operations, they aim to cover program paths as much as possible. In real-world programs, taking unsatisfiable floating-point conditions as satisfiable may cause an infeasible path treated as feasible, which results in reporting false positives. In the sequel of this paper, we refer to this situation as Tautology Strategy (TS) for simplicity. On the other hand, `KLEE` [8] intends to assign concrete values to operands of the conditional expression and concretely execute the expression. Since the branch condition is evaluated to a concrete value (True or False), there is only one branch that will be explored while the other branch is treated as infeasible and thrown away. Therefore, it is likely to discard plenty of feasible program paths in real-world programs, that leads to false negatives. Similar with TS, we refer to this situation as Concolic Strategy (CS) for simplicity.

With recent progress in SMT solving techniques, state-of-the-art solvers, like Z3 [9], start to support the floating-point theory and be able to reason about the constraint with floating-point arithmetic precisely. Nevertheless, naturally extending a symbolic execution engine and directly replacing the back-end with a floating-point SMT solver are far from efficient for bug detection. There are two reasons accounting for the inefficiency: First, the mathematical function analysis consumes much overhead. Since complex mathematical functions, like `sin()` and `log()`, are quite challenging for SMT solving techniques, the straightforward way is to inline the system implementation of the function and symbolically execute the code. However, this method may heavily increase the number of paths as well as the complexity of path conditions and lead to an excess of overhead. Second, from our experiments, the cost of solving floating-point constraints is hundreds of times that of solving integer constraints. Calling SMT solver for all the branches explored (with some optimizations of constraint solving), which is the commonly used constraint solving strategy in the existing symbolic execution engine either for bug detection [5], [6], [7] or for test case generation [8], will consume astronomical overhead.

In this paper, we first explore the floating-point extension of general symbolic execution engine for bug detection in real-world programs. We aim to avoid effectively and efficiently the false positives and negatives caused by the previous imprecise reasoning about floating-point path conditions. The extension includes floating-point arithmetic and mathematical functions.

1530-1362/16 $31.00 © 2016 IEEE
DOI 10.1109/APSEC.2016.9

For the mathematical functions, we employ an abstract model by characterizing the behavior of function with the manually designed constraints on the return value to support a trade-off of overhead and precision. To reduce the number of SMT solver calls, we introduce a strategy of constraint solving, called *Lazy-verification*. The strategy works in two phases: The first phase, Pruning phase, prunes the bug-free paths in the exploration of program paths; The second phase, Verification phase, employs constraint solving to verify the rest paths, most of which are buggy paths. The insight of this strategy is that we only need to call constraint solving for buggy paths, because bug-free paths have no effect on bug detection.

To evaluate our approach, we implemented it as a tool, named `Canalyze-fp`, which is based on a previous symbolic execution engine [7] for bug detection in `C` programs. We have applied `Canalyze-fp` to a suite of open source programs (e.g., gnuplot and GSL), each of which has lots of floating-point operations and at least tens of thousands lines of code, and some self-developed floating-point benchmarks. We have also compared our approach with other state-of-the-art symbolic execution engines for bug detection. In these experiments, our approach effectively avoids the false positives and negatives with acceptable time cost.

The rest of this paper is organized as follows. In the next section, after introducing the basics of floating-point operations in `C` programs, we present a motivating example of this paper. Section III gives an overview of our tool. Section IV presents the approach of analyzing floating-point operations. Section V evaluates our approach on self-developed benchmarks and real-world programs. Section VI summarizes the related work, and the last section concludes this paper.

II. Preliminaries

In this section, we first introduce the basics of floating-point operations in `C` programs. Then we present a motivating example.

A. Floating-point operations

For floating-point extension of symbolic execution for bug detection in `C` programs, we analyze floating-point types and operations according to the `C` language standard [10]. In `C` language, floating-point operations consist of two integral parts. The first one is floating-point arithmetic, which is consistent with IEEE-754 standard [11]. The standard defines the formats of floating-point number, the basic arithmetic of floating-point values, like binary operations, the square root operation, and rounding operations. The other one is the use of various floating-point mathematical functions, which are implemented for special functionalities, like trigonometric functions, exponential functions, and nearest integer functions.

B. A motivating example

In this section, we leverage a motivating example to illustrate the occurrences of false positives and negatives. Figure 1 is a contrived example containing floating-point operations. In the rest of this paper, we will make use of this example to demonstrate how our approach eliminates the false positives and negatives.

```
 1  int foo(double dy)
 2  {
 3    int *ip=NULL;
 4    double *dx = (double *) malloc(sizeof(double));
 5    double d1,d2,d3;
 6    int i1 = 0;
 7    int i2 = 0;
 8    *dx = 1.0;
 9    d1 = 6.0;
10    d2 = acos(dy);
11    d3 = *dx+d2;
12    if (dy < 0.0){
13       if (*dx == -2.0)
14         i1 = *ip; /* bug 1 */
15       else
16         i1 = *ip; /* bug 2 */
17    } else {
18       if (d3 > d1)
19         i2 = *ip; /* bug 3 */
20       else
21         i2 = *ip; /* bug 4 */
22    }
23    return i1 + i2;
24  }
```

Fig. 1: The motivating example

In the contrived example, the pointer `ip` is not allocated memory and initialized with NULL, thus, the use of `ip` at line 14, 16, 19, and 21 may trigger the dereference of NULL pointer bug. We traverse the program and find four paths that cover the four lines respectively listed as follows:

P1: $3 \rightarrow 12 \rightarrow 13 \rightarrow 14 \rightarrow 23$;
P2: $3 \rightarrow 12 \rightarrow 13 \rightarrow 15 \rightarrow 16 \rightarrow 23$;
P3: $3 \rightarrow 12 \rightarrow 17 \rightarrow 18 \rightarrow 19 \rightarrow 23$;
P4: $3 \rightarrow 12 \rightarrow 17 \rightarrow 18 \rightarrow 20 \rightarrow 21 \rightarrow 23$.

Then we symbolically execute the four paths and extract their path conditions as follows.

P1: $(dy < 0.0) \ \wedge \ (1.0 = -2.0)$;
P2: $(dy < 0.0) \ \wedge \ (\neg(1.0 = -2.0))$;
P3: $(\neg(dy < 0.0)) \ \wedge \ (1.0 + acos(dy) > 6.0)$;
P4: $(\neg(dy < 0.0)) \ \wedge \ (\neg(1.0 + acos(dy) > 6.0))$.

Since $acos(dy) \in [0, \pi]$ and `dy` can be arbitrary `double` type variable, we can easily infer that P2 and P4 are feasible while P1 and P3 are infeasible. As a result, real bugs only exist in P2 and P4. However, if we use TS and CS to analyze these four paths, we will obtain different results.

Analysis with TS: TS assumes that all the floating-point path conditions are satisfiable. Therefore, the path conditions of the four paths are all evaluated as true and four bugs are reported, that leads to false positives.

Analysis with CS: CS assigns a possible concrete value to the floating-point symbolic input and then symbolically executes the path. For this example, it will assign a value, e.g. 1.0, to `dy` and the path conditions will be instantiated as: $(F \wedge F = F)$, $(F \wedge T = F)$, $(T \wedge F = F)$, and $(T \wedge T = T)$ respectively. Therefore, only P4 is decided to be feasible and reported as a buggy path, that leads to false negative.

In summary, without precisely modeling the floating-point operations, analyzing with TS or CS will result in either false positives or false negatives. This motivates us to adopt a more accurate strategy for modeling the floating-point operations.

III. Overview

In this section we start with introduction of a previous symbolic execution tool `Canalyze` [7]. Then we present an overview of our approach.

`Canalyze` is a static bug-finding tool for `C` programs based on symbolic execution technique. By supplying precise memory management for integer, pointer and `struct` type, it can detect various kinds of bugs, like memory leak, NULL pointer dereference, and the use of undefined value. It supplies summary-based and inline-based methods for the interprocedural analysis. The tool has been applied to plenty of real-world applications and hundreds of new bugs were detected, which have been confirmed by the developers of the applications.

Figure 2 demonstrates an overview of our tool, where the dash rectangle part comes from `Canalyze`, the rest parts denote our extension. The framework treats as input source files and outputs bug reports if any. During symbolic analysis, the `Memory management` module is used for maintaining the program state, the `Constraint solving` module for reasoning about the feasibility of the path, and `Interprocedural analysis` module for analyzing function calls.

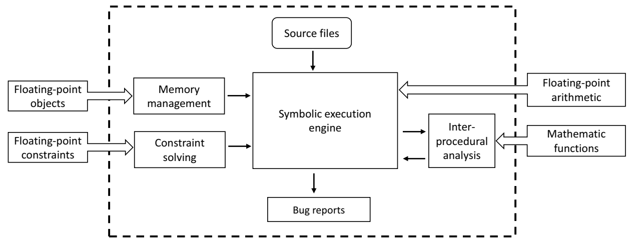
Source files
Floating-point objects
Memory management
Symbolic execution engine
Floating-point arithmetic
Floating-point constraints
Constraint solving
Inter-procedural analysis
Mathematic functions
Bug reports

Fig. 2: Overview of our approach

The floating-point extension includes the following four parts:

1) Floating-point values: memory management of floating-point constants, variables, and pointers.
2) Floating-point arithmetic: symbolic execution of floating-point arithmetic.
3) Mathematical functions: analysis of the mathematical function calls.
4) Floating-point constraints: solving the floating-point constraints.

For maintaining floating-point values, we naturally extend the previous memory model, which can manage precisely the integer values. With the extended memory model, `Canalyze-fp` can analyze `float`, `double` and `long double` types. In the next section, we will present the other three parts.

IV. Approach

In this section, we present the details of floating-point extension, including analysis of floating-point arithmetic and mathematical function calls. We also present the *Lazy-verification* strategy for constraint solving. To facilitate formal discussion, we first define the syntax of floating-point expressions.

A. Syntax of floating-point expressions

Figure 3 presents the syntax of floating-point expressions. The expression $(\tau)e$ is defined as type conversion operations including conversions between different floating-point and other types. The expression $e \otimes e$ and $e \ominus e$ denote binary arithmetic operations (such as $+$ and $\times$) and relational operations (such as $>$ and $\leq$) respectively. The expression $e \odot e$ is defined as binary logic operations (such as logic OR and logic AND). The expression $libs(e)$ denotes the mathematical function calls.

(Expr) e	::=	$x\,\vert(e)\vert(\tau)e\vert e\otimes e\vert e\ominus e\vert e\odot e\vert libs(e)$
(Aop) $\otimes$	::=	$+\vert - \vert \times \vert \div$ ...
(Rop) $\ominus$	::=	$> \vert < \vert \leq \vert \geq$ ...
(Lop) $\odot$	::=	$\vee \mid \wedge$
(Func) $libs$	::=	sin | cos | arcsin | exp |...
(Type) τ	::=	int | float | double | long double
(Operand) x	::=	number | id

Fig. 3: Syntax of floating-point expressions

To simplify the discussion, the unary operations including `+`, `-`, `!`, `++`, and `--` are not addressed in the syntax. The first three operations can be translated into binary operations, e.g., the logical negation operation expression `(!x)` can be converted to (x != 0.0). The latter two expressions have complex semantics with side effects, that will be discussed further in Section IV-D.

With the syntax of floating-point expressions, we will present the modeling rules for the floating-point expressions in the next two subsections.

B. Analysis of floating-point arithmetic

As the SMT solver for constraints with floating-point arithmetic is available, most of the floating-point arithmetic can be analyzed similarly with the integer arithmetic. Nevertheless, there are two points to which we should pay attention.

First is type promotion. When the operands of different types participate in expressions, type promotion operations happen automatically. For example, to evaluate the value of $a_{int} + b_{float}$, we first need to promote the `int` type of a to $float$. Second is the binary logic expressions. For binary logic expressions, each operand e_i will be directly translated to a relational expression $e_i \neq 0.0$. For example, $-0.3 \wedge 2.0$ will be translated to $(-0.3 \neq 0.0) \wedge (2.0 \neq 0.0)$.

To discuss type promotion rules, we introduce a partial relation $\prec$ on type set $\Phi = \{$int, float, double, long double$\}$, where int $\prec$ float $\prec$ double $\prec$ long double. Then we define the type promotion operator $\sqcap : \Phi \times \Phi \rightarrow \Phi$ as

$$t1 \sqcap t2 = ITE(t1 \prec t2, t2, t1),$$

where abbreviation ITE stands for if-then-else and $t1, t2 \in \Phi$.

With the type promotion operator, we give the modeling rules for floating-point expressions in Figure 4. The e, e_1, e_2 are expressions, and $e.t$, $e.v$ are the type and value of expression e respectively. The function $s_c(f, t, v)$ describes the type conversion that converts the value v of type f to type t and returns the value of new type. The function $zero(\tau)$ is a function that returns the zero value of type τ, which is defined for the binary logic expressions. For the binary arithmetic and relational expressions, we first perform type promotion.

$$\begin{array}{lcl}
e ::= x & : & e.t \leftarrow x.t; e.v \leftarrow x.v \\
e ::= (e_1) & : & e.t \leftarrow e_1.t; e.v \leftarrow e_1.v \\
e ::= e_1 \otimes e_2 & : & e.t \leftarrow e_1.t \sqcap e_2.t \\
 & & e.v \leftarrow s_c(e_1.t, e.t, e_1.v) \otimes s_c(e_2.t, e.t, e_2.v) \\
e ::= e_1 \ominus e_2 & : & e.t \leftarrow \texttt{int} \\
 & & tt \leftarrow e_1.t \sqcap e_2.t \\
 & & e.v \leftarrow s_c(e_1.t, tt, e_1.v) \ominus s_c(e_2.t, tt, e_2.v) \\
e ::= e_1 \odot e_2 & : & e.t \leftarrow \texttt{int} \\
 & & e.v \leftarrow (e_1 \neq zero(e_1.t)) \odot (e_2 \neq zero(e_2.t)) \\
e ::= (\tau)e_1 & : & e.t \leftarrow \tau \\
 & & e.v \leftarrow s_c(e_1.t, \tau, e_1.v)
\end{array}$$

Fig. 4: Modeling rules for floating-point expressions

C. Abstraction for mathematical function calls

With the consideration of the mathematical functions which are hard to be processed by SMT solvers containing no side effect (e.g., `sin()`, `exp()`), we adopt a light-weight abstraction model to simplify some of the mathematical function calls. The abstraction model is to use the range of return value to simulate function behaviours.

We construct the abstraction model in three steps. First, we manually collect the mathematical functions that are hard to be handled by constraint solvers. Then we classify these functions into several sets according to their ranges, where the functions in the same set have the same range. At last, we assign a symbol to the function call as well as the constraints on this symbol to restrict its range. For example, the function call $sin(x)$ is replaced with a symbol S_{sin_x} and a constraint $(-1.0 \leq S_{sin\_x} \leq 1.0)$ is added to the path conditions.

We have collected 37 functions, including: trigonometric functions, exponential and logarithmic functions, nearest integer functions, reminder functions, classification functions (like `isnan()`), and so on, from the GNU C Library [12], and figured out 18 of them that are hard for SMT solvers. These 18 functions as well as the ranges are listed in Table I. Other functions such as `fabs()` and `sqrt()` are relatively easy to solve and are directly passed to the constraint solvers.

TABLE I: Abstraction Model for Some Mathematical Functions

Function	Range
`sin`, `cos`	$[-1.0, 1.0]$
`asin`, `atan`	$[-\pi/2, \pi/2]$
`acos`	$[0, \pi]$
`exp`, `exp2`, `exp10`, `expm1`, `pow10`	$[0, +\inf]$
`tan`, `fmod`, `drem`, `remainder`, `log`, `log2`, `log10`, `log1p`	$[-\inf, +\inf]$

D. Complex structure analysis

The previous parts of this section discuss the floating-point operations on the scalar variables. We can further extend the modeling rules to complex floating-point objects, like pointer. To store different kinds of memory objects, we employ the existing memory model presented in our previous work [13] to simulate the management of memory regions. With the help of this model, we can statically analyze the floating-point operations not only on scalar variables, but also on the variables whose types are pointer, struct, and array. This model also supports the analysis of the expressions with side effects, such as unary operation ++. We analyze them in the same way as for integer type expressions in the previous work.

E. Constraint solving

The *Lazy-verification* strategy consists of two phases : Pruning phase and Verification phase. Before symbolic execution, we first define plenty of illegal operation patterns, like NULL pointer dereference, according to the C language standard. We mainly detect three kinds of bugs: 1) Use of undefined values; 2) NULL pointer dereference; 3) Memory management problems, including memory leak, use after free and double free.

Pruning phase: In the exploration of each program path, we symbolically execute the instructions one by one. At the same time, we collect the path conditions, but, do not call constraint solving for them and treat all the branches explored as feasible. If we find one illegal operation in the instructions, the path is labeled as a potential buggy path. Otherwise, it indicates that is a bug-free path, we prune it without reasoning about its feasibility, because it has no effect on bug detection, and continue the analysis of next path.

Verification phase: For the potential buggy paths, we reason about them with constraint solving, which will invoke the SMT solver. If the paths are feasible, that indicates a bug is found, we will report the bug.

On one hand, as the number of buggy paths, in the delivered software, is much less than that of bug-free paths, this strategy will reduce significantly both the number of SMT solver calls and the overhead. On the other hand, generally speaking, the infeasible paths take a small portion in the real-world programs, this strategy will not introduce too much unnecessary overhead to analyze the paths which are steered by an infeasible branch, that is treated as feasible at the Pruning phase. With this strategy, our tool can analyze tens of thousands lines of code within tens of minutes.

F. Elimination of the false positives and negatives

In this subsection, we will illustrate how the proposed approach eliminates the false positives and negatives brought by analyzing the example of Section II-B with TS and CS.

Recall path P3, in which a false positive is reported with TS, its path conditions are $(\neg(dy < 0)) \wedge (1.0 + acos(dy) > 6.0)$, which is treated as feasible. According to the abstract model in Table I, the function call `acos(dy)` is simulated by the symbol S_{acos_dy}, the path conditions are transformed into $(\neg(dy < 0)) \wedge ((1.0 + S_{acos\_dy}) > 6.0) \wedge (S_{acos\_dy} \geq 0.0) \wedge (S_{acos\_dy} \leq \pi)$. Obviously, the two conditions $(1.0 + S_{acos\_dy}) > 6.0)$ and $(S_{acos\_dy} \leq \pi)$ lead to contradiction and thus the path is infeasible. Therefore, compared with the TS strategy, this simple technique can eliminate some of the false positives.

On the other hand, for path P2, which is discarded with CS, its path condition is $(dy < 0.0) \wedge (\neg(1.0 == -2.0))$. We can easily decide the constraints are satisfiable with a SMT solver that supports the floating-point theory. Therefore, we will not miss the bug in this path.

V. Evaluation

We now present our evaluation. We first present the overall implementation and then come with the experimental setup. Next, by comparing with other tools on various benchmarks, we demonstrate the effectiveness and efficiency of the proposed approach on eliminating false positives and negatives.

A. Implementation

We implemented our method on top of symbolic execution engine of the previous work `Canalyze` [7], named `Canalyze-fp`. To solve the constraints with floating-point arithmetic, we adopt the SMT solver Z3 [9], that supports the floating-point theory, as the back-end. We compile our extension and the original tool as a stand-alone tool in Linux operation system. In the following parts, all our experiments are executed on Ubuntu 12.04-64 virtual machine with 3GB memory, and the host is running at 3.4 GHz Intel Core7 processor with 8GB memory on Windows 7.

B. Experimental setup

To illustrate the efficiency and effectiveness of our approach, we first set up four research questions, which cover the evaluation of the proposed approach, comparing with the related tools, and the performance on the real instances.

- RQ1: how effective is the abstraction model of mathematical functions in detecting program bugs?
- RQ2: how effective is the proposed approach in the elimination of false positives and negatives?
- RQ3: how effective is the *Lazy-verification* strategy in analyzing real-world programs?
- RQ4: how effective is the proposed approach applied to real-world programs?

In the following subsections, we will present the related tools, the benchmarks used in our evaluations, and the organization of experiments.

1) Static analysis tools

We chose four start-of-the-art tools: `Saturn` [6], Clang Static Analyzer (`CSA`) [5], `KLEE` [8] and `Canalyze` [7]. Table II presents the four tools as well as the strategies (TS and CS, referred to Section I) for processing the floating-point path conditions. The tool `KLEE` requires the user to manually specify the program inputs as symbolic variables, or they will be assigned with concrete values. The other three tools are pure static analysis tools, that can analyze the program automatically.

TABLE II: Static Analysis Tools for Comparison

Tool	Strategy	Negative effect
`Saturn`	TS	False positives
`CSA`		
`Canalyze`		
`KLEE`	CS	False negatives

2) Benchmarks

The benchmarks contain self-developed benchmarks and real-world programs.

a) Self-developed benchmarks: We developed 28 benchmarks in `C` language following the floating-point syntax, each of which consists of a single function that has no function calls except the mathematical functions and is initiated with NULL pointer dereference bugs, which can be detected as all the selected tools declared. To evaluate the false positives of the tools, some paths of the benchmarks are designed to be infeasible. The characteristics of the benchmarks are presented in Table III. The benchmarks include two categories of floating-point operations, without and with mathematical function call. The former category contains 15 benchmarks and 29 bugs while the latter category contains 13 benchmarks and 26 bugs. The benchmarks are available online: http://lcs.ios.ac.cn/~zj/Benchmarks/Canalyze-fp.html.

TABLE III: Self-developed Benchmarks

Benchmark type	Floating-point operations	#Bench	#Bug
Arithmetic operation	$+, -, \div, \times, ++, --$, type conversion, ...	15	29
Function call	`sin()`, `acos()`, `tan()`, `log()`, `exp()`, ...	13	26

b) Real-world programs: We also applied our tool to three real-world programs, which are listed in Table IV. All the benchmarks are written in `C` and can be downloaded from the open source site. They are chosen specifically because they contain lots of floating-point operations and at least tens of thousands of lines of code.

TABLE IV: Details of the Real-world Programs

Program	Version	KLOC	Remark
gnuplot	5.1	87.2	a portable command-line driven graphing utility for multiple platforms
GSL	2.1	135.8	GNU Scientific Library, a widely-used numerical library
TTF2PT1	3.4.4	15.5	a tool for converting True Type Font to Postscript Type 1
Total	-	238.5	-

3) Organization of experiments

With the help of the two sets of benchmarks, we can answer the research questions. Our experiment is split into two parts.

- Experiments with self-developed benchmarks: In Section V-C, we first compare the proposed abstraction model with the inline-based method for mathematical functions to display the effectiveness of the model and answer RQ1. Then, we present the experiment results of our

tool and the other tools on the benchmarks to show the effectiveness of our approach and answer RQ2.
- Experiments with real-world programs: We make use of three real-world programs to demonstrate the scalability of our tool and answer RQ3 and RQ4 in Section V-D.

C. Experiments on self-developed benchmarks

In this section, we show the experimental results on the self-developed benchmarks.

1) Effectiveness of the abstraction model

As we have introduced in Section I, the straightforward inline method for the mathematical function calls is time-consuming and we proposed an abstraction model (Section IV-C) to speed up the analysis. We compare these two methods on the function call benchmarks (we ignore the arithmetic operation benchmarks because they contain no mathematical functions) to display the pros and cons of our approach. The inline method needs the source codes of the mathematical functions, which are obtained from GNU `C` library [12].

Table V presents the comparison between the abstraction model (AM) and the inline method (IM). The first and second columns are the mathematical function and the number of bugs in each benchmark. In the following columns, we present the number of bugs reported and execution time for each method.

TABLE V: Comparison between AM and IM

Mathematical functions	#Total bugs	AM		IM	
		#Bugs reported	Time (s)	#Bugs reported	Time (s)
`sin()`	2	2	0.31	2	439.53
`cos()`	2	2	0.22	2	503.17
`asin()`	2	2	0.22	2	1617.71
`acos()`	2	2	0.33	2	1516.25
`atan()`	2	2	0.16	2	2273.04
`log()`	2	2	1.91	2	3367.22
`exp()`	2	2	0.80	2	789.23
mixed	12	17	10.03	12	7816.54
Total	26	31	23.88	26	18322.69

From the table, we see that our AM approach reported 31 bugs of which five are the false positive while the IM detected all the bugs without false positives. However, the IM costs on average more thousands times of execution time. The reason is that the abstraction method significantly reduced the number of program paths and simplified the complexity of path conditions.

From Table V, we draw the conclusion that the AM technique with acceptable false positives is necessary for modeling the mathematical functions, or the time cost of analysis will be unacceptable.

2) False positives and negatives avoided

We compare our approach with tools using TS to demonstrate false positives avoided and with tool using CS to display the false negatives avoided.

a) False positives: The results of four tools are shown in Table VI. The #Total bugs column is the number of bugs in the benchmarks. The #Bugs reported column is the number of bugs reported by the tools, including the number of the true positives (TP), the false positives (FP), and the false negatives (FN).

TABLE VI: FPs Reported by Four Tools

Tool	#Total bugs	#Bugs reported		
		TP	FP	FN
`Saturn`	55	55	28	0
`CSA`	55	55	27	0
`Canalyze`	55	55	29	0
`Canalyze-fp`	55	55	5	0

From the table, we see that the tools all detected all of the bugs and had no false negatives. However, tools with TS (the first three rows) regarded some of the infeasible paths that contain floating-point path conditions as feasible, that led to a relatively high rate of false positives, most of which can be recognized by our approach (the last row). We should point out that our tool also reported five false positives, that are brought by our abstract model of mathematical functions. More discussions of that can refer to Section V-E.

b) False negatives: Now we discuss the ability of our approach in eliminating the false negatives. Table VII presents the results of our tool and `KLEE`. The second column is the number of total bugs. The following columns are the number of bugs reported by the tools.

TABLE VII: FNs in `KLEE` and `Canalyze-fp`

Tool	#Total bugs	#Bugs reported		
		TP	FP	FN
`Klee`	55	28	0	27
`Canalyze-fp`	55	55	5	0

From Table VII, we see that `KLEE` reported bugs without false positives. However, with the CS method, it discarded some feasible program paths that have floating-point conditions and led to a relative high rate of false negatives. Since our approach will not omit any explored feasible paths, we can avoid the false negatives brought by the CS method thoroughly.

D. Experiments on real-world programs

We will show the performance of our approach on the real-world programs. First, we present the effectiveness of the *Lazy-verification* strategy. Then, since our tool is an extension of `Canalyze`, we show the difference of performance between our tool and the original one on the three real-world and large scale instances in Table IV.

Lazy-verification: The performance of `Canalzye-fp` with and without *Lazy-verification* strategy are shown in Table VIII. We limit the running time to 50 hours[1] and use the inline-based method [7] for the interprocedural analysis. N/A

[1]This is a test-and-try configuration. The longer timeout can also be experimented, but, this configuration has showed strongly the effect of our method.

means the analysis runs out of the time limited. #branch column is the total number of the branches explored in the benchmarks, which is collected from the running with *Lazy-verification* strategy. #SMT column and `Time` columns are the times of SMT solver calls and the time overhead (in second). The third and fourth columns present the performance of the tool when the strategy is not used. As the analysis of all the benchmarks run out of the limited time, we are unable to count the times of SMT solver calls and denote with "-". The fifth and sixth columns present the performance of the tool with *Lazy-verification* strategy. As the potential buggy paths, which contain the illegal operations and each of which will invoke one SMT solver call, take a small portion in the real-world programs (#SMT column), the overhead is significantly reduced.

TABLE VIII: Performance of using and not using *Lazy-verification* strategy

Benchmark	#branch	Constraint solving		Constraint solving + *Lazy-verification*	
		#SMT	Time(s)	#SMT	Time(s)
Gnuplot	1182735	-	N/A	35	1380
GSL	605185	-	N/A	58	2519
TTF2PT1	41334	-	N/A	9	733

Canalyze VS Canalyze-fp: As `Canalyze-fp` employs the *Lazy-verification* strategy to accelerate the analysis, to present the overhead brought by the floating-point extension, we also implemented the strategy in `Canalyze`. Table IX presents the overall results of `Canalyze-fp` and `Canalyze`. The second and third columns are the number of bugs reported and time cost of `Canalyze`, then comes with `Canalyze-fp`. In the `#bug` column, the number inside the parenthesis indicates the number of false positives which are brought by the imprecise analysis of the floating-point path conditions, the number outside the parenthesis is the total number of bugs reported. The last two columns show the differences of these two tools including the false positives avoided and time cost (in second) increased.

TABLE IX: Performance on the Real-world Instances

Benchmark	`Canalyze`		`Canalyze-fp`		Difference	
	#bug	Time (s)	#bug	Time (s)	#bug	Time (s)
Gnuplot	25(4)	1339	21(0)	1380	4	41
GSL	23(2)	1404	21(0)	2519	2	1115
TTF2PT1	8(1)	80	7(0)	733	1	653

From Table IX, we see that our approach eliminated all the false positives in analyzing the real-world instances with `Canalyze`. Since solving constraints with floating-point operations consumes much time, the analysis time of our tool inevitably increases. However, from the results, the extra overhead is acceptable compared to the size of real-world instances. In a word, our approach is scalable to the large programs.

E. *Threats to validity*

As the model of mathematical functions is not precise, our proposed approach will also report false positives. For example, a constraint $sin^2(x) + cos^2(x) > 1$, which is unsatisfiable, will be abstracted as constraint $((S^2_{sin\_x}) + (S^2_{cos\_x}) > 1) \wedge (-1 \leq S_{sin\_x} \leq 1) \wedge (-1 \leq S_{cos\_x} \leq 1)$, where S_{sin_x} and S_{cos_x} are the floating-point symbols. After constraint solving, the new constraints will be solved as satisfiable. Thus, the infeasible path will be treated as feasible, that will lead to a false positive. To further reduce the threat, we plan to explore a more precise model as future work.

VI. Related work

We discussed the most closely related work in section I, where we focused on the symbolic execution engine for bug detection. In this section, we look a bit further into other previous work on floating-point program analysis and on constraint solving strategies.

A. *Symbolic execution*

Despite the lack of the SMT solver for floating-point constraints, symbolic execution technique has also been employed to deal with specific issues of floating-point programs.

Botella et al. [14] introduced the interval computing based method to solve the constraints with floating-point arithmetic in symbolic execution. Search-based techniques have also been employed to solve the floating-point constraints in `FloPSy` [15] and `PathCrawler` [16] for test case generation.

`KLEE-FP` [17], which was based on `KLEE`, supported symbolic reasoning on the equivalence between floating-point values. It explored the symbolic execution of floating-point arithmetic, but ignored the mathematical functions. Recently, T. Barr et al. [18] attempted to automatically detect floating-point exceptions in GSL [19]. They also adopted the range of the mathematical functions to model their behaviours. However, without an appropriate constraint solver, they solve the floating-point constraints as real constraints and iterate the results to find a solution for the floating-point constraints. Ramachandran et al. [20] extended a JAVA symbolic execution tool to check the different results of a program with floating-point arithmetic and with real arithmetic by making use of a well-designed solver based on the real theory for solving floating-point constraints. Though these works concern the floating-point types and operations in symbolic execution, they do not aim at a general framework for bug detection.

B. *Analysis of floating-point programs*

One alternative technique is abstract interpretation [21]. Eric Goubault [22] attempted to analyze the errors caused by the imprecision of floating-point arithmetic. `Fluctuat` [23] is a static analyzer for studying the propagation of rounding errors in `C` programs. `FramaC` [24], supporting floating-point arithmetic, is a source code analysis platform that aims at the verification of industrial-size `C` programs, but, it deals with only linear floating-point constraints. `Astreé` [25], which also supports floating-point types and operations, attempts to prove automatically the absence of run time errors. However, abstract interpretation has the inherent limitation that it can not analyze program path-sensitively, and brings false positives.

Another alternative technique is dynamic analysis. Benz et al. [26] presented a dynamic program analysis that supports the programmer in finding accuracy problems. Then, based on the method, Zou et al. [27] proposed a metaheuristic search-based approach to generating test inputs that aim to trigger significant inaccuracies in floating-point programs. Tao Bao and Xiangyu Zhang [28] developed an online technique to monitor and predict floating point program execution instability problems.

Different from these works, our approach focuses on improving the symbolic execution engine for bug detection by modeling floating-point operations more precisely than the existing works.

C. Constraint solving

As constraint solving costs tremendous overhead in symbolic execution, several strategies of constraint solving have been explored. Hristina Palikareva and Cristian Cadar [29] introduced multiple-solver techniques, which run in parallel multiple SMT solvers and return the fastest result for the best-performance. Xu et al. [7] employed a hybrid of range solver and SMT solver to accelerate the constraint solving. For the branch explored, it is first solved by the range solver. Then, if the result is undecided (like being `unknown`), the SMT solver is invoked. Since the SMT solver is much slower than the range solver, the hybrid strategy improves two to three times performance of constraint solving. Different from these works, which will call constraint solving for all the branches explored, our *Lazy-verification* strategy reasons about only the potential buggy paths and prune the bug-free paths directly, which significantly reduces the overhead of constraint solving.

VII. Conclusion and future work

In this paper, we extend an existing symbolic execution engine to support floating-point types and operations for avoiding false positives and negatives in detecting bugs in real-world `C` programs. We utilize a practical abstraction model for the mathematical functions to ease the path analysis and constraint solving, which are usually time-consuming. Besides, we introduce a *Lazy-verification* strategy of constraint solving to reduce the number of SMT solver calls. The evaluation on the self-developed and real-world benchmarks showed that the proposed approach can effectively and efficiently eliminate the false positives and negatives brought by the imprecise modeling of floating-point path conditions.

We believe that our approach is also a promising technique for improving the other existing tools, which will be our future work. In addition, we hope to extend the approach to effectively and efficiently detect floating-point errors, such as floating-point exceptions, in our future work.

VIII. Acknowlegement

The authors would like to thank Qiong Lu, Feifei Ma, and the anonymous reviewers for their helpful comments. This work is supported by the National Basic Research Program (973 Program) of China under Grant No. 2014CB340701, the National Natural Science Foundation of China under Grant No. 91418206, and the CAS/SAFEA International Partnership Program for Creative Research Teams.

References

[1] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 1999.
[2] T. Ball, "The concept of dynamic analysis," in *ESEC/FSE*, 1999, pp. 216–234.
[3] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.
[4] J. Zhang, "Constraint solving and symbolic execution," in *VSTTE*, 2005, pp. 539–544.
[5] Clang static analyzer homepage. [Online]. Available: http://clang-analyzer.llvm.org/index.html
[6] Y. Xie and A. Aiken, "Saturn: A scalable framework for error detection using boolean satisfiability," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 3, 2007.
[7] Z. Xu, J. Zhang, Z. Xu, and J. Wang, "Canalyze: a static bug-finding tool for C programs," in *ISSTA*, 2014, pp. 425–428.
[8] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008, pp. 209–224.
[9] Z3 homepage. [Online]. Available: https://github.com/Z3Prover/z3
[10] I. O. for Standardization, "ISO/IEC 9899:1999 - Programming languages - C," 1999.
[11] "IEEE Standard for Binary Floating-Point Arithmetic," *ANSI/IEEE Std 754-1985*, 1985.
[12] GNU C library homepage. [Online]. Available: https://www.gnu.org/software/libc/
[13] Z. Xu, T. Kremenek, and J. Zhang, "A memory model for static analysis of C programs," in *ISoLA*, 2010, pp. 535–548.
[14] B. Botella, A. Gotlieb, and C. Michel, "Symbolic execution of floating-point computations," *STVR.*, vol. 16, no. 2, pp. 97–121, 2006.
[15] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux, "Flopsy - search-based floating point constraint solving for symbolic execution," in *ICTSS*, 2010, pp. 142–157.
[16] N. Williams, B. Marre, P. Mouy, and M. Roger, "Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis," in *EDCC*, 2005, pp. 281–292.
[17] P. Collingbourne, C. Cadar, and P. H. J. Kelly, "Symbolic crosschecking of floating-point and SIMD code," in *EuroSys*, 2011, pp. 315–328.
[18] E. T. Barr, T. Vo, V. Le, and Z. Su, "Automatic detection of floating-point exceptions," in *POPL*, 2013, pp. 549–560.
[19] GSL homepage. [Online]. Available: http://www.gnu.org/software/gsl/
[20] J. Ramachandran, C. S. Pasareanu, and T. Wahl, "Symbolic execution for checking the accuracy of floating-point programs," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 1, pp. 1–5, 2015.
[21] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, 1977, pp. 238–252.
[22] E. Goubault, "Static analyses of the precision of floating-point operations," in *SAS*, 2001, pp. 234–259.
[23] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine, "Towards an industrial use of FLUCTUAT on safety-critical avionics software," in *FMICS*, 2009, pp. 53–69.
[24] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c - A software analysis perspective," in *SEFM*, 2012, pp. 233–247.
[25] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The astreé analyzer," in *ESOP*, 2005, pp. 21–30.
[26] F. Benz, A. Hildebrandt, and S. Hack, "A dynamic program analysis to find floating-point accuracy problems," in *PLDI*, 2012, pp. 453–462.
[27] D. Zou, R. Wang, Y. Xiong, L. Zhang, Z. Su, and H. Mei, "A genetic algorithm for detecting significant floating-point inaccuracies," in *ICSE*, 2015, pp. 529–539.
[28] T. Bao and X. Zhang, "On-the-fly detection of instability problems in floating-point program execution," in *OOPSLA*, 2013, pp. 817–832.
[29] H. Palikareva and C. Cadar, "Multi-solver support in symbolic execution," in *CAV*, 2013, pp. 53–68.