The Journal of Systems and Software 219 (2025) 112226

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Symbolic testing of floating-point bugs and exceptions☆

Dongyu Ma [a,1], Zeyu Liang [b,1], Luming Yin [a], Hongliang Liang [a,*]

[a] Beijing University of Posts and Telecommunications, China
[b] University of California San Diego, United States of America

ARTICLE INFO

Keywords:
Numerical software
Floating-point exceptions and bugs
Symbolic execution

ABSTRACT

Numerical software are susceptible to floating-point bugs and exceptions, which may lead to severe threats like denial of service attacks. Static analysis techniques such as symbolic execution are effective in detecting general bugs which often cause memory error or program crash. Unfortunately, these methods do not deal well with numerical code as they do not support floating-point constraints and math functions symbolically. In this paper, we propose a new analysis framework YUSE, which can detect floating-point bugs by constructing constraints and exploring paths which contain floating-point expressions. Specifically, we introduce interval computation and interval constraint propagation in non-relational numerical abstract domains, and symbolically model math functions, to accurately detect floating-point bugs and exceptions. Moreover, we leverage two-phase constraint solving to enhance YUSE's performance. Experimental results show that YUSE outperforms two state-of-the-art tools, Frama-c and Fpse-study, in terms of effectiveness and efficiency, with 1.4× and 7.1× faster than Frama-c and Fpse-study, respectively. Moreover, YUSE found 20 new bugs in real-world software, 12 of which were assigned CVE IDs and 8 of which were confirmed by developers.

1. Introduction

Numerical software plays a critical role in many fields such as scientific computing, smart home appliances, and national defense. However, bugs in these programs can lead to severe consequences. For example, floating-point arithmetic errors caused Patriot missile system failure during the Gulf War of 1991 (Patriot, 1992), and in 2016 a floating-point to integer overflow led to the failure of European Space Agency's Ariane 5 rocket (ARIANE, 1996).

In this paper, we aim to detect those floating-point bugs and exceptions which consist of (1) general bugs, e.g., use-after-free or buffer overflow, guarded by conditions involving floating-point operations or math functions; (2) floating-point exceptions caused by floating-point operations or math functions. Many approaches and tools have been developed to detect software bugs. Yet, they often struggle with accuracy and speed, especially in detecting floating-point bugs and exceptions (Barr et al., 2013; Kirchner et al., 2015; Zhang et al., 2022; Wu et al., 2016; Dinda et al., 2020; Cousot et al., 2005). For instance, Frama-c (Kirchner et al., 2015) can identify potential floating-point bugs and exceptions by utilizing abstract interpretation, however, it fails to distinguish implicit and explicit type conversions, leading to lots of false positives. Additionally, it does not deal well with calls of math functions and complex floating-point expressions, resulting in many false negatives. Moreover, in our preliminary experiment, we found that its constraint solver *eva* is about 5× slower than the built-in range constraint solver (Range, 2023) in Clang static analyzer (CSA for short) (CSA, 2023). Based on symbolic execution and constraint solving, CSA (CSA, 2023) supports only integer and simple floating-point constants and thus is unsuitable for numerical bug detection. Fpse-study (Zhang et al., 2022) aims to analyze floating-point programs by using symbolic execution, but also suffers from calls of math functions and complex floating-point expressions, resulting in both false positives and negatives. Moreover, the Z3 solver (de Moura and Bjørner, 2008; Z3, 2023) used in Fpse-study is about 20× slower than range constraint solver (CSA, 2020).

To mitigate the above problems, we present a novel analysis framework, YUSE, which is capable of detecting floating-point bugs and exceptions by constructing floating-point-related constraints and exploring floating-point-related paths. Specifically, we first integrate floating-point support into a symbolic execution engine. Second, we introduce interval computation and interval constraint propagation in non-relational numerical abstract domains (Anon, 2017) with the objective of enhancing the precision of detecting floating-point bugs and exceptions. Third, we propose several techniques, i.e., mathematical function

☆ Editor: Professor Yan Cai.

* Corresponding author.
E-mail address: hliang@bupt.edu.cn (H. Liang).

[1] Equal contribution.

https://doi.org/10.1016/j.jss.2024.112226
Received 2 April 2024; Received in revised form 7 September 2024; Accepted 22 September 2024
Available online 2 October 2024

modeling, interval binding and a two-phase constraint solving strategy, to enhance the performance of our analysis framework.

In summary, our work makes the following contributions.

- We propose an analysis framework which can construct floating-point related constraints and explore floating-point related paths to detect floating-point bugs and exceptions.
- We introduce interval computation and interval constraint propagation in non-relational numerical abstract domains, in order to improve the accuracy of detecting floating-point bugs and exceptions.
- We propose several techniques, i.e., math function modeling, interval binding and a two-phase constraint solving strategy, to enhance the performance of our analysis framework.
- We implemented the proposed approach in a tool called YUSE. Evaluation on three real-world software, i.e., GSL, Sox and Mupdf, shows that YUSE is more effective and efficient on floating-point bug detection than two state-of-the-art tools, Frama-c and Fpse-study. Specifically, YUSE is 1.4× and 7.1× faster than Frama-c and Fpse-study, respectively, and found 12 bugs all of which were assigned CVE IDs and 8 bugs all of which were confirmed by corresponding developers. Our tool and benchmarks are publicly available at: https://gitee.com/ma-dongyu/bupt_yuse.

The remainder of this paper is structured as follows. We present the background of static symbolic execution and floating-point data, and the motivating examples in Section 2. We describe the overview and detailed design of YUSE in Sections 3 and 4 respectively. Section 5 provides the evaluation of YUSE in terms of its effectiveness and efficiency. Related work is discussed in Sections 7, and 8 concludes.

2. Background & motivation

In this section, we first describe static symbolic execution and floating-point data briefly. Next, we use two examples to illustrate the limitations of existing static analysis tools in testing numerical programs.

2.1. Static symbolic execution

Symbolic execution (King, 1976; Baldoni et al., 2018) is a popular program analysis technique to test whether certain properties in software can be violated, for example, no division-by-zero is ever performed, no NULL pointer is ever dereferenced, and so on. Symbolic execution can simultaneously explore multiple paths that a program could take under different inputs. Its key idea is to allow a program to take on symbolic input values. Execution is performed by a *symbolic execution engine*, which maintains for each explored control flow path: a first-order Boolean formula that describes the conditions satisfied by the branches taken along that path, and a symbolic memory store that maps variables to symbolic expressions or values. Branch execution updates the formula, while assignments update the symbolic store. A satisfiability modulo theories (SMT) solver is typically used to verify whether there are any violations of the property along each explored path and if the path itself is realizable, i.e., if its formula can be satisfied by some assignment of concrete values to the program's symbolic arguments.

Static symbolic execution is a static analysis technique that uses symbolic values instead of concrete values to simulate program's execution. During simulating, the symbolic execution engine collects semantic information and explores the reachable paths in the program to analyze potential bugs. Compared to those dynamic methods like fuzzing which run the target program with randomly generated inputs, static symbolic execution does not need to execute the program actually, and thus not restricted by the potential complex runtime environment. Besides, in the case of bug detection, search heuristics may help prioritize some interesting paths and hence achieve high efficiency and code coverage (Baldoni et al., 2018).

2.2. Floating-point data and exceptions

ISO/IEC 60559 does not define all aspects of a conforming programming environment. Such behavior should be defined by a programming language definition supporting this standard, if available, and otherwise by a particular implementation. In this work we target the x86_64 family of processors and C programming language, where three primitive floating-point types are available: 32-bit wide single precision (float), 64-bit wide double precision (double), and 80-bit wide double extended precision (long double).

For a floating-point number, its binary representation consists of sign bits, exponent bits, and mantissa bits. We can have addition, subtraction, multiplication, division, fused multiply add, square root, compare, and other operations on it. Floating-point exceptions are usually caused by illegal or incorrect operations on floating-point number, which include division by zero, overflow, underflow, invalid operation, and inexactness. Readers of interest can refer to ISO/IEC 60559 (ISO, 2020).

In general, a floating-point number f is represented as $\pm d_0.d_1d_2\ldots d_{p-1}\times\beta^e$, where $d_0.d_1d_2\ldots d_{p-1}$ is the significand with p digits. e is the exponent. Precisely, f is defined as follows.

$$f=\pm(d_0+d_1\beta^{-1}+\cdots+d_{p-1}\beta^{-(p-1)})\times\beta^e,\quad 0<d_i<\beta.$$

For a math function of f, the deviation between its finite-bit output and the exact result can be quantified in "units in last place" or ULPs (Goldberg, 1991).

Different computing architectures, such as x86 CPU and NVIDA GPU, have distinct instruction sets for floating-point computations. For instance, Intel provided x87, SSE, AVX and AVX-512 instruction sets which can perform 80-bit, 128-bit, 256-bit and 512-bit floating-point operations. By contrast, NVIDIA GPUs, e.g., Tesla Kxx and GTX 9xx, support both single and double precision with ISO/IEC 60559 precision. Differ from the x86 architecture in that rounding modes are encoded within each floating-point instruction instead of dynamically using a floating-point control word. Trap handlers for floating-point exceptions are not supported, and on the GPU there is no status flag to indicate when calculations have overflowed, underflowed, or have involved inexact arithmetic. Note that our proposed approach is based on the ISO/IEC 60559 standard and thus can be applied to multiple architectures whose implementations conform to the standard. However, YUSE can currently analyze C/C++ programs involving floating-point data as it is implemented on LLVM and Clang.

2.3. Motivating examples

False positives caused by false assumption of floating-point path condition

Prior static analysis tools, e.g., Frama-c (Kirchner et al., 2015) and Fpse-study (Zhang et al., 2022), reason incorrectly about conditions related to floating-point data, leading to false positives. When encountering conditions related to floating-point data, these tools assume these conditions as true to allow the forward execution, in order to explore as many program paths as possible. As a result, the above tools have lots of false positives when analyzing programs with floating-point data.

As shown in `func1` in Fig. 1(a), the path "1-3-4-5" is not reachable and thus the null pointer dereference in line 8 would never occur. However, Frama-c falsely detected this "bug", leading to false positives. The reason behind is that, when Frama-c encounters an unknown condition, it would make more "assumptions" and thus take the conditions at lines 4 and 5 as true. Moreover, Frama-c found a division-by-zero exception at line 8, which is also a false positive, because its constraint solver takes that the value of $sin(z)-1.0$ may be zero. However, it is always greater than zero because of $1.0<z<1.5$.

```
1  float func1(float x, float y,
       float z)
2  {
3    int t1 , *p = NULL;
4    if (x + y > 10.0) {
5      if (x + y < 5.0)
6        t1 = *p;
7      if ((z > 1.0) && (z < 1.5))
8        return 1.0 / (sin(z)-1.0);
9    }
10 }
```

(a)

```
1  int
       gsl_sf_bessel_Knu_scaled_asympx_e
       (const double nu, const double x,
        gsl_sf_result * result)
2  {
3    double mu  = 4.0 / nu;
4    // ...
5    double pre = sqrt(M_PI / (2.0*x));
6    // ...
7    return GSL_SUCCESS;
8  }
```

(b)

```
1  int gsl_sf_zeta_e(const double s
       , gsl_sf_result * result )
2  {
3    // ...
4    const int n = floor((-s)/10.0);
5    // ...
6    return
         GSL_ERROR_SELECT_2
         (stat_g, stat_zoms);
7  }
```

(c)

```
1  int gsl_sf_gamma_inc_e(double a,
       double x, gsl_sf_result* result)
2  {
3    double alpha = a - floor(a);
4    do {
5      // ...
6      alpha -= 1.0;
7    } while(alpha > a);
8    // ...
9  }
```

(d)

Fig. 1. Examples of floating-point bugs and exceptions.

False negatives in detecting floating-point exceptions

When a floating-point expression does not satisfy the specifications defined in the ISO/IEC 60559 (ISO, 2020) standard, a floating-point bug may occur. Specifically, there are two ways to trigger these exceptions, direct operations on floating-point data and operations on the return values of math functions which use floating-point data. As shown in Fig. 1(b), in function `gsl_sf_bessel_Knu_scaled_asympx_e` from GNU Scientific Library (GSL, 2023), if the value of *nu* is very large, the resulted *mu* may be less than the minimum positive value of floating-point data type and hence cause an underflow. When the argument *x* is a negative value, the *sqrt* function at line 5 will cause an invalid operation, which may lead to denial of service attacks. Frama-c does not provide the support for detecting floating-point underflow exception and thus results in false negatives. Let us see function `gsl_sf_zeta_e` also from GNU Scientific Library (GSL, 2023), shown in Fig. 1(c). If *s* takes a very small negative value, the resulted *n* may exceed the maximum value of integer type. Hence this implicit type conversion causes an overflow. Fpse-study cannot detect overflow exceptions caused by implicit type conversion and thus results in false negatives.

3. Overview

YUSE works in two phases as shown in Fig. 2. For a given program under test (PUT for short), in the preprocessing phase, YUSE first parses the abstract syntax tree (AST for short) of the PUT in the AST Parser, and then constructs the control flow graph (CFG for short) in the CFG Constructor.

During the symbolic execution phase, YUSE symbolically executes the PUT based on its CFG and AST with symbolic expressions. The math function modeler symbolizes the math function calls in the PUT. Then, the symbolic executor symbolically executes each path in the PUT, constructs the constraints for variables and expressions along the path, and leverages the constraint solver to judge whether a path is reachable or not. When arriving a predefined hook location, YUSE calls the bug checkers mounted on the execution engine to examine whether a bug is triggered or not, if yes, it generates a bug report. Inter-procedural analysis module uses the caller–callee relationship among procedures and thus enables more precise analysis, e.g., how variables flow across different functions, which functions read or modify global or static variables. Taint analysis module marks the distrusted or sensitive data, e.g., user input, tracks their propagation during program execution, and determines whether they cause potential bugs.

Specifically, we equip our framework with the capability to analyze floating-point data, including symbolic values, symbolic expressions, memory management, constraint solving, and bug checkers. Therefore, constraints relate to floating-point data can be constructed, paths relate to floating-point data can be explored, and exceptions relate to floating-point data can be detected. Besides, we enhance the symbolic engine with interval computation and interval constraint propagation in non-relational numerical abstract domains in order to improve the accuracy of detecting floating-point bugs and exceptions.

4. Approach

4.1. Symbolic support for floating-point data and operations

To symbolically execute a floating-point program, we enhance a traditional symbolic executor (e.g., that of CSA) as follows.

- add symbolic value representations of floating-point constants, variables and pointers.
- construct symbolic expression for operations on floating-point data, e.g., addition, subtraction, multiplication, division operations, math operations, comparison operations, logical operations, rounding operations.
- add solving support for floating-point constraints.

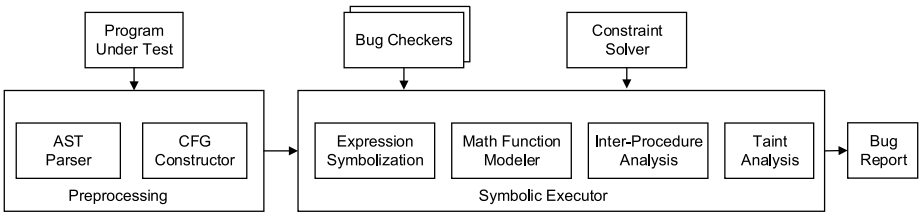

Fig. 2. The architecture of YUSE.

- symbolically model math functions, e.g., trigonometric functions, exponential functions, logarithmic functions, power functions, by adding symbolic representations for function arguments and return values and binding intervals to them.
- construct constraints for floating-point exceptions defined in the ISO/IEC 60559 standard, and design checkers to detect these exceptions.

With the help of these enhancements, YUSE can explore the paths with floating-point expressions, detect those bugs guarded by such expressions and exceptions caused by floating-point operations. For instance, with its symbolic support for floating-point data and operations, YUSE can determine that the path "1-3-4-5" in `func1` in Fig. 1(a) is not reachable.

4.2. Interval computation and interval constraint propagation in non-relational numerical abstract domains

In order to calculate a variable's interval accurately, we introduce interval computation and interval constraint propagation in non-relational numerical abstract domains. The non-relational numerical abstract domain is an abstract representation method to describe the numerical information of program variables. Specifically, we first abstract each variable as an interval, i.e., its possible interval which consists of an upper bound and a lower bound.

Then we specify interval computation rules for common computations, e.g., addition, subtraction, multiplication, division, comparison, intersection, union, and add these rules into the symbolic engine. As a result, the engine can apply these rules when collecting and merging the variables' values. For example, considering two variables a and b, and their intervals (a_l, a_u) and (b_l, b_u), respectively, the rules for their intersection, addition and greater than are as follows.

$$a \cap b = \begin{cases} \emptyset & \text{if } a_l > b_u \text{ or } a_u < b_l \\ (\text{Max}(a_l, b_l), \text{Min}(a_u, b_u)) & \text{Otherwise} \end{cases} \tag{1}$$

$$a + b = (\text{Max}(\text{FLT\_MIN}, a_l + b_l), \text{Min}(\text{FLT\_MAX}, a_u + b_u)) \tag{2}$$

$$a > b = \begin{cases} \text{True} & \text{if } a_l > b_u \\ \text{False} & \text{if } a_u < b_l \\ (-\infty, +\infty) & \text{Otherwise} \end{cases} \tag{3}$$

where FLT_MAX means 3.402823466e+38f and FLT_MIN denotes −3.402823466e+38f.

Moreover, when constructing a symbolic constraint, i.e., meeting a conditional statement, the symbolic engine first calculates the interval of the conditional expression, then updates the corresponding symbolic states, and finally continues the execution with this new interval.

Note that, it is not always successful to merge intervals due to the limitations of constraint solver. For example, it returns true when solving the intersection of two discontinuous intervals in order to help explore more paths.

By introducing interval computation and interval constraint propagation in non-relational numerical abstract domains to the symbolic engine, the symbolic values or constraints of float-point expressions are calculated, merged and propagated in an interval manner, which can improve the accuracy of detecting floating-point bugs and exceptions. Taking the `func1` in Fig. 1(a) as example, YUSE can judge that the correct interval of $sin(z)$ is $(sin(1.0), sin(1.5))$ and hence the value of $sin(z) - 1.0$ is always greater than zero. By contrast, Fpse-study took it as a division-by-zero bug mistakenly.

4.3. Constraint construction and optimization for floating-point exceptions

Constraint construction for floating-point exceptions

Floating-point exceptions include division by zero, overflow, underflow, type conversion, invalid operation, and inexactness, according to the floating-point ISO/IEC 60559 standard. In order to detect these exceptions, we construct constraints for each kind of exception, as shown in Table 1. Taking inexactness exception as an example, we illustrate how to construct constraint for floating-point exceptions. Inexactness exception is mainly caused by two cases: (1) two very close floating-point numbers are subtracted, whose constraint is denoted as $x - y = 0 \mid x < \text{FLT\_EPSILON} + y$; (2) two floating-point numbers with significant differences are subtracted, whose constraint is denoted as $x - y = x \mid x > \text{FLT\_INTERVAL} + y$. As shown in Fig. 1(d), function `gsl_sf_gamma_inc_e` in GSL exposes an inexactness exception. Actually, when a is a significant negative value, $alpha - 1.0 > a$ is almost always true, causing an almost endless loop.

Constraint optimization for floating-point exceptions

In order to improve the accuracy and efficiency of path constraint solving, during the symbolic execution, YUSE rewrites the constraints on overflow and underflow. For example, the constraint on overflow caused by *addition* $L + R >$ FLT_MAX, is rewritten as $(R > 0) \wedge (L > (\text{FLT\_MAX} - R))$ or $(R < 0) \wedge (L > (\text{FLT\_MAX} - R))$; $L + R <$ FLT_MIN is rewritten as $(R > 0) \wedge (L < (\text{FLT\_MIN} - R))$ or $(R < 0) \wedge (L < (\text{FLT\_MIN} - R))$.

Similarly, the constraint on underflow caused by *multiplication* $L * R >$ FLT_MAX_NEG $\wedge$ $L * R <$ FLT_MIN_POS is rewritten to $(R > 0) \wedge (L > (\text{FLT\_MAX\_NEG}/R)) \wedge (L < (\text{FLT\_MIN\_POS}/R))$ or $(R < 0) \wedge (L > (\text{FLT\_MIN\_POS}/R)) \wedge (L < (\text{FLT\_MAX\_NEG}/R))$.

Here L and R denotes the left and right value of a binary operator respectively, FLT_MIN_POS is 1.17549435e−38f, FLT_MAX_NEG is −1.17549435e−38f.

For example, when YUSE meets the function gsl_sf_bessel_Knu_scaled_asympx_e in GSL, as shown in Fig. 1(b), it first identifies the math function `sqrt` at line 5 and gets the trigger condition from Table 1, i.e., $x < 0$. Then it checks whether the parameter x's symbolic value, i.e., $(-\infty, +\infty)$ here, meets the trigger condition via its constraint solver, which returns true here. Therefore YUSE reports an invalid operation exception while Fpse-study and Frama-c cannot.

4.4. Modeling math functions

To detect floating-point bugs and exceptions caused by math functions, the math function modeler identifies math functions and accurately reasons about the paths which goes through such functions. As a result, bug checkers can trigger or detect floating-point bugs and exceptions caused by these math functions. Then, we classify the math functions, and bind different constraint intervals according to

Table 1
Floating-point exceptions, Triggering condition and result.

Type	Expression examples	Triggering condition	Result
Overflow	$x \triangle y \mid \triangle \in \{+,-,*,/\},\ x,y \in$ {floating-point data or math functions}	$x \triangle y >$ FLT_MAX $x \triangle y <$ FLT_MIN	$+\infty$ $-\infty$
	$x = y$ (double cast to float) $x = y$ (float cast to int)	$y >$ FLT_MAX $\vee$ $y <$ FLT_MIN $y >$ INT_MAX $\vee$ $y <$ INT_MIN	$\pm\infty$
Underflow	$x \triangle y \mid \triangle \in \{*,/,*=,/=\},\ x,y \in$ {floating-point data or math functions}	$0 < x \triangle y <$ FLT_MIN_POS $\vee$ FLT_MAX_NEG $< x \triangle y < 0$	Subnormal
	$x = y$ (double cast to float)	$0 < y <$ FLT_MIN_POS $\vee$ FLT_MAX_NEG$< y < 0$	
Division by zero	$x/y, x\%y$, fmod(x,y), drem(x,y), remainder(x,y)	$(x \neq 0) \wedge (y = 0)$	$+\infty$
Inexact	$x - y \mid x \gg y$ $x - y \mid x \approx y$	$x >$ FLT_INTERVAL $+ y$ $x <$ FLT_EPSILON $+ y$	x 0
Invalid Ops.	x/y $\sqrt{x}, \log(x)$ $\arcsin(x), \arccos(x)$ $\tan(x)$ $\text{pow}(x,y)$ Operations on invalid values like: Inf+x, floor(Inf), x/Inf, x/NaN, 0*NaN	$(x = 0) \wedge (y = 0)$ $x < 0$ $-1 < x < 1$ $x = \frac{\pi}{2} + k\pi$ $x = 0 \wedge y \leq 0$ Invalid values are judged based on the above results.	NaN

different classifications to the math functions and finally bind to the program state. Specifically, for a math function F, we initiate the process by assigning a symbolic value to F's return value. According to its monotonicity, periodicity, and the interval of its arguments, we calculate the interval of F's output and bind the resulted interval to the symbolic value, which forms a new state on the exploded graph.[2] During analyzing the program, as a conditional expression is encountered and the interval of any variable in the expression changes, a new state with the updated interval will be added to the exploded graph. The states on the exploded graph can help reason about paths and solve constraints. Modeling math functions helps YUSE collect the intervals more accurately and thus analyze the program more accurately.

Symbolic call of math functions

We first select dozens of functions from GNU scientific library (GSL, 2023), such as sin, cos, tan, asin, acos, atan, cosh, tanh, acosh, atanh, sqrt, log, exp, pow, gamma, fmod, remainder, remquo etc, and different floating-point type representations of these functions, and model them in symbolic executor to help checkers detect bugs. When meeting a math function F, YUSE leverages taint analysis to track F's data propagation, and then create symbolic values and intervals for F. Therefore, when exploring a path along with F, YUSE can detect those floating-point bugs and exceptions caused by F.

Interval binding

When modeling a math function, we bind and assign a precise interval to it in order to accurately reason about the path along with the function, which we call as *interval binding*. According to a math function's characteristics, such as monotonicity, periodicity, arguments, and return values, etc, we divide math functions into four categories as follows and use different interval binding method for each category, as shown in Algorithm 1.

Algorithm 1 is used for interval binding. The input is a call expression of a math function (X for convenience) in the PUT. When meeting a call to X function, YUSE first creates for X an empty symbolic value in order to store X's final interval later. Then we process each math function according to different categories.

Category 1: Functions with monotonicity. For such a function like $logx$, we calculate the output interval based on input arguments' intervals. In addition, taking into account the rounding error, we change the output interval R(f(x)) to [f(x) - δ, f(x) + δ], where δ is set as 2^{20} ULP (Kong et al., 2013), and bind the output interval to the state of the exploded graph. Specifically, we obtain the calling expression of the math function based on the abstract syntax tree, and the current code position LCtx from the CheckerContext (Line 1). The CheckerContext acts as a bridge between a checker and the symbolic executor, and provides the checker with the program context and operational interfaces for detecting bugs, e.g., the current program state and code location. Then, we construct a new symbolic value and bind the symbolic value with the help of CheckerContext to model the program state corresponding to the function's return value (Lines 2–3). Next we get the arguments' intervals (Lines 4–7, 10, 11), call X function to get its output interval (Line 8, Lines 12–15), bind the output interval to the program state (Line 9, 16, 17), and update the exploded graph (Lines 35–36). In this way, we get a relatively accurate return interval of the function.

Category 2: Periodic functions with upper and lower bounds. If this function is a periodic function, and if there is a monotonic part in the function, such as $cosx, sinx, asinx, acosx$, we calculate and bind the interval to it in the same way as that for the Category 1 function (Lines 18–20). If the function is not monotonic, we directly update the interval with the upper and lower bounds of the function's period bounds (Lines 21–23, Lines 25–28), bind them to the program state (Line 24, 29, 30), and update the exploded graph (Lines 35–36).

Category 3: Function expression resulted with a constant value. For such a expression whose result is always equal to a fixed value, such as $sin^2x + cos^2x = 1$, we directly get a fixed interval (Lines 31–32), such as [1, 1], and bind it to the function's symbolic return value (Lines 33–34). The specific binding process is the same as above.

Category 4: Other cases. Other math functions that do not belong in the above three categories, such as discontinuous functions, discrete functions, combinations of irregular math functions, e.g., $pow(x), x * sinx$, are difficult or even impossible to model symbolically.

Furthermore, when operating on the results of math function calls, interval operations may also be involved, like $sinx * logx$. In this way, we can get a relatively accurate return interval of the function and add it to the program state of the exploded graph for subsequent analysis. The combination of math function modeling, interval binding, and interval operations in non-relational numerical abstract domain can further improve the accuracy of YUSE. For `func1` in Fig. 1(a), Fpse-study and CSA identified a division-by-zero exception at line 8, which is a false positive. By contrast, thanks to modeling math functions, YUSE accurately determines the interval for z at line 7, then calls the sin function to obtain its output interval, i.e., (sin(1.0), sin(1.5)), and finally infers that the value of sin(z) - 1.0 at line 8 is always greater than zero.

[2] https://clang-analyzer.llvm.org/checker_dev_manual.html.

Algorithm 1: Modeling a math function X symbolically

```
Input: CE: X's call expression; C: CheckerContext; plbSval/pubSval:
       lower/upper bound symbolic value of periodic function;
1  LCtx = getLocationContext(C);
2  retSval = makeRetSval(C, CE);
3  retSvalState = bindExpr(CE,C,retSval);
4  arg = CE.getArgument();
5  argSymVal = getSymSVal(arg);
6  if CE is monotonic function then
7      if argSymVal.isConcreteFloat then
8          returnConcreteSval = processConcreteFloatSval(argSymVal);
9          retSvalState = bindExpr(CE, LCtx,
             adjustInterval(returnConcreteSval, δ)) ;
10     if argSymVal.isSymbolVal then
11         argIntervalSet = getConstraintInterval(argSymVal);
12         resultLowerSval,resultUpperSval =
             processSymbolVal(argIntervalSet);
13         R_GE_LOWER_SVAL = evalBinOp(BinaryOperator::GE, retSval,
             resultLowerSval);
14         R_LE_UPPER_SVAL = evalBinOp(BinaryOperator::LE, retSval,
             resultUpperSval);
15         returnSymSval = evalBinOp(BinaryOperator::AND,
             R_GE_LOWER_SVAL, R_LE_UPPER_SVAL);
16         solve(returnSymSval);
17         retSvalState = bindExpr(CE, LCtx,
             adjustInterval(returnSymSval, δ));
18 if CE is periodic function with upper and lower binds then
19     if CE's part is monotonic function then
20         execute same instructions as lines 7-17;
21     else
22         if argSymVal.isConcreteFloat then
23             returnConcreteSval =
                 processConcreteFloatSval(argSymVal);
24             retSvalState = bindExpr(CE, LCtx,
                 adjustInterval(returnConcreteSval, δ));
25         if argSymVal.isSymbolVal then
26             R_GE_LOWER_SVAL = evalBinOp(BinaryOperator::GE,
                 retSval,plbSval);
27             R_LE_UPPER_SVAL = evalBinOp(BinaryOperator::LE,
                 retSval, pubSval);
28             returnSymSval = evalBinOp(BinaryOperator::AND,
                 R_GE_LOWER_SVAL, R_LE_UPPER_SVAL);
29             solve(returnSymSval);
30             retSvalState = bindExpr(CE, LCtx,
                 adjustInterval(returnSymSval, δ));
31 if the result of CE is a constant value C then
32     returnSymSval = evalBinOp(BinaryOperator::EQ, retSval, C) ;
33     solve(returnSymSval);
34     retSvalState = bindExpr(CE, LCtx, adjustInterval(returnSymSval,
         δ));
35 if retSvalState is established then
36     addTransition(retSvalState);
```

4.5. Two-phase constraint solving

To solve path constraints, using a range constraint solver like the one in CSA may bring many false positives; while using an SMT solver (CSA, 2020) may be more accurate but is about 20× slower than range solver (Range, 2023). In order to enhance the performance of our analysis framework and to maintain accuracy, we propose a two-phase constraint solving strategy, i.e., use range solver first and then SMT solver. Specifically, when solving a symbolic constraint, we first use range solver for solving, and then send the solved results to SMT solver for verifying. This hybrid method can enhance the performance of our analysis framework and reduce time overhead significantly while ensuring accuracy (see Section 5.5).

4.6. Mitigating path explosion

Path explosion is one of the main challenges of symbolic execution. To mitigate this issue, first, YUSE invokes the constraint solver at each branch, if the solver can prove that the logical formula given by the path constraints of a branch is not satisfiable, this path can be safely discarded by the symbolic engine without affecting soundness. This can remove as many statements as possible while preserving unsatisfiability. Second, YUSE limits the number of times a loop is unrolled and the depth of recursive function calls to prevent the generation of a vast number of paths due to deep loops or recursive functions. This helps control the number of paths considered during the analysis process. Third, YUSE generates and uses summaries for frequently called functions, which allows the executor to effectively reuse prior analysis results, instead of re-analyzing the functions. This not only reduces the workload of the analysis but also helps control the growth of paths. Finally, in the case of solver timeout, YUSE takes both the true and false branches, adds a lazy constraint to the path conditions and continues the execution. When the exploration reaches an interest state, e.g., a bug is found, YUSE will check whether the path is unreachable, and suppress it if yes. This helps avoid exploring those paths that are unlikely to reveal new bugs. Overall, the above methods enable YUSE to effectively mitigate the path explosion problem when facing large and complex program code, improving the accuracy and efficiency of the analysis, as demonstrated in Section 5.

5. Evaluation

In this section, we aim to answer the following research questions through empirical evaluation.

RQ1: How effective is YUSE in detecting floating-point bugs and exceptions compared to state-of-the-art tools?
RQ2: How efficient is YUSE in constraint solving?
RQ3: Can YUSE find real bugs in real-world software?

5.1. Implementation and experiment setup

YUSE is implemented on top of Clang version 12.0.0 and supports two static code analyzers, i.e., clang-tidy and clang static analyzer. YUSE supports floating-point arithmetic, branch conditions, math functions, and it can perform both path-sensitive and path-insensitive analysis on programs under test. Note that YUSE is affected by compiler optimizations like -fast-math. For a program under test, YUSE constructs its abstract syntax tree and control flow graph for symbolic analysis while compiler optimizations usually are performed before the preprocess stage. As a result, aggressive compiler optimizations like -fast-math may hinder an exception which otherwise would be detected by YUSE.

All experiments were conducted on Ubuntu 20.04, with 64-core CPU (Intel(R) Xeon(R) Gold 6226R CPU @ 2.90 GHz and 251.0 GB of memory.

5.2. Benchmarks

Benchmarks used in the experiments include synthetic benchmarks and real-world benchmarks. We first wrote synthetic benchmarks and then injected them to three real-world software to construct real-world benchmarks in order to measure the effectiveness Frama-c, Fpse-study and YUSE in finding bugs. Finally we adopt the real-world benchmarks in our experiments. For example, as shown in Fig. 3, a synthetic function `func1()` is injected to the software Sox to construct a real-world benchmark.

```
1  // gsm_add is a function in Sox project
2  word gsm_add (word a, word b){
3    double c;   // injected
4    func1(c);   // injected
5    longword sum = (longword)a + (longword)b;
6    return saturate(sum);
7  }
8  // func1 is a synthetic function.
9  void func1(double a){
10   double b;
11   if (a > 0.0 && a < 5.0)
12     b = a * a;
13   else
14     b = a + a;  // An overflow exception maybe occurs.
15 }
```

Fig. 3. An example code snippet of real-world benchmark.

Table 2
Details of the real-world benchmarks.

Program	Vers.	KLOC	Summary
GSL	5.0.1	387	A scientific computing library under the GNU project, providing lots of functions for scientific computing
Sox	14.4.2	103	A cross-platform audio editing software
Mupdf	2.5.0	175	A lightweight open source software framework for viewing and converting PDF, XPS, and E-book documents

Synthetic Benchmarks (60 correct, 60 incorrect) According to floating-point operations and exception types, we developed dozens of programs[3] which contain the following floating-point bugs or exceptions: (1) bugs guarded by floating-point conditions. These bugs include common types of bugs (e.g., division-by-zero and out-of-bound) guarded by branch conditions which are composed of floating-point operations (e.g., Addition/subtraction/multiplication/division). (2) floating-point exceptions caused by floating-point operations. These exceptions contain overflow/underflow/division-by-zero/inexact/invalid operations caused by floating-point operations. and (3) floating-point exceptions caused by math functions. These exceptions consist of overflow/underflow/division-by-zero/inexact/invalid operations resulted from calling math functions. Therefore, for each category, the synthetic benchmarks contain 20 programs expected to be correct and 20 ones expected to be incorrect, respectively. These programs (120 in total) have between 1 and 301 (median 8) branches and request between 0 and 128 (median 8) symbolic bytes. Moreover, their lines of code in C programming language are between 14 and 26 (median 17) lines.

Real-world Benchmarks (60 correct, 60 incorrect) We evaluated our tool using three real-world numerical software, i.e., GNU Scientific Library (GSL) (GSL, 2023), Sound eXchange (Sox) (SOX, 2023) and Mupdf (mupdf, 2023), whose details are listed in Table 2. They come from numerical computing software libraries, audio software and image processing software, respectively. They contain lots of floating-point variables, conditions and operations. These benchmarks have between 6 and 254 (median 67) branches and request between 4 and 48 (median 16) symbolic bytes.

5.3. Baseline tools

We compare YUSE with two state-of-the-art static tools Frama-c (Kirchner et al., 2015) and Fpse-study (Zhang et al., 2022) to validate the effectiveness of YUSE (RQ1).

[3] YUSE and all benchmarks are available at https://gitee.com/ma-dongyu/bupt_yuse.

We did not choose Astrée and Ariadne as comparison tools because they are not open-source. We did not compare YUSE with Klee-float because KLEE-float does not support IEEE-754 exceptions and flags.

Frama-c is a well-known static analysis tool for C programs. It can find multiple kinds of potential bugs such as null pointer dereference, integer overflow, buffer overflow, division-by-zero, and precision loss exceptions, using abstract interpretation technique.

Fpse-study, which was developed on top of KLEE, can analyze floating-point programs using symbolic execution. It can detect floating-point exceptions and those bugs guarded by a floating-point condition.

5.4. RQ1: Effectiveness in detecting floating-point bugs and exceptions

To answer RQ1, we evaluated three tools against the real-world benchmarks, and compared the bugs detected by them. Table 3 shows the true positive(TP), true negative(TN), false positive(FP), false negative(FN) detection results of three tools for each kind of float-point bugs and exceptions in three real-world software. We can see that (1) the accuracy of Frama-c, Fpse-study and YUSE is 52.5%, 63.3%, and 92.5% respectively; (2) the recall of Frama-c, Fpse-study, and YUSE is 51.7%, 35.0%, and 91.6% respectively. YUSE outperforms two state-of-the-art tools, Frama-c and Fpse-study, in both accuracy and recall of detecting floating-point bugs and exceptions. Moreover, YUSE is 1.4× and 7.1× faster than Frama-c and Fpse-study, respectively.

YUSE has 5 false negatives. Through manual analysis, we found these five bugs occur either in deep loop statements or in math function calls, which cause path explosion and thus cannot be successfully detected by YUSE. YUSE has 4 false positives because its solver timed out when solving some complex floating-point constraints. Frama-c and Fpse-study have much false positives because they cannot deal well with type conversions, reason about symbol intervals, inaccurately model math functions, and solve complex constraints. Moreover, their false negatives mainly come from: bugs initiated in deep loop statement and math function calls, which causes path explosion, insufficient cross-translation unit capabilities, lack of detection logic for underflow and inexact floating-point exceptions, lack of modeling some math functions (such as log1p, logb, fmod, lgamma, remquo, etc.).

5.5. RQ2: Efficiency in constraint solving

To answer RQ2, we employed three different solver strategies: range constraint solver, SMT constraint solver, and our two-phase constraint solving approach. Our results, shown in Table 4, indicate that the range constraint solver had the shortest solving time, while the SMT solver took the longest, approximately 20 × longer than the range solver. The two-phase strategy showed a moderate analysis time, about 7 × longer than the range solver. This strategy effectively enhances the performance of the constraint solver while maintaining an acceptable number of solver calls. Further, we set a one-hour for bug detection analysis, as shown in Table 5. Within this time limit, the range constraint solver resolved nearly all constraints in the snippets, albeit with a 67.1% accuracy. The SMT solver, while resolving the least number of bugs, achieved the highest accuracy at 95.5%. The two-phase strategy balanced both aspects, solving most bugs with a 93.8% accuracy rate. The range constraint solver, despite being sound, struggled with complex constraints, often resulting in false positives and negatives due to its simplistic approach and sensitivity to path depth. In contrast, the SMT solver excelled in handling complex constraints with higher accuracy but required longer solving and analysis times. Overall, the two-phase constraint solving strategy demonstrated superior performance compared to the other two methods. It utilized the strengths of the SMT solver to minimize the range solver's false positives, thereby enhancing accuracy, and also mitigated the SMT's time overhead to achieve a balance between analysis time and accuracy.

Table 3
Effectiveness on detecting floating-point(FP) bugs and exceptions.

Tool	GSL					Sox					Mupdf				
	Bugs guarded by FP conditions					Exceptions caused by FP operations					Exceptions caused by FP math functions				
	TP	TN	FP	FN	Time (s)	TP	TN	FP	FN	Time (s)	TP	TN	FP	FN	Time (s)
Frama-c	17	7	13	3	12193	12	11	9	8	3302	2	14	6	18	5548
Fpse-study	12	16	4	8	60895	8	20	0	12	16603	1	19	1	19	27895
YUSE	17	18	2	3	8661	19	20	0	1	2308	19	18	2	1	3916

Table 4
Efficiency when using different solving strategies.

Benchmark	YUSE-RS		YUSE-SMT		YUSE-RSSMT		
	Constraints	Time (s)	Constraints	Time (s)	Constraints		Time (s)
	RS		SMT		RS	SMT	
GSL	30571	1219	30571	23882	30571	346	8661
Sox	8532	336	8532	6719	8532	183	2308
Mupdf	13822	585	13822	12189	13822	221	3916

```
1  static real flowTrigger(sox_effect_t * effp, sox_sample_t const * ibuf,
       sox_sample_t * obuf, real * ilen, real * olen){
2      priv_t *p = (priv_t *)effp->priv;
3      sox_bool hasTriggered = sox_false;
4      real i, idone = 0.0;
5      // ...
6      idone--;
7      while (idone < *ilen && !hasTriggered) {
8          if (p->measureTcMult != p->triggerMeasTcMult)
9              p->samplesIndex_ns = (p->samplesIndex_ns + p->
                   flushedLen_ns) / p->samplesLen_ns;
10     }
11     // ...
12 }
```

Fig. 4. A bug found by YUSE which was assigned CVE-2023-47483.

Table 5
Effectiveness when using different solving strategies.

Benchmark	YUSE-RS				YUSE-SMT				YUSE-RSSMT			
	TP	TN	FP	FN	TP	TN	FP	FN	TP	TN	FP	FN
GSL	17	10	10	3	4	20	0	16	9	19	1	11
Sox	19	12	8	1	11	20	0	9	19	20	0	1
Mupdf	19	11	9	1	6	19	1	14	18	18	2	2

5.6. RQ3: Ability to find real bugs

To address RQ3, we tested three real-world software, i.e., GSL, Sox and Mupdf, using YUSE, Frama-c and Fpse-study respectively, in order to evaluate their capabilities of finding real bugs. YUSE found 1601, 111 and 134 potential bugs in GSL, Sox and Mupdf respectively, from which we selected 30 randomly and validated them manually. We uncovered 20 new bugs, 12 of which were assigned CVE IDs and 8 of which were confirmed by developers, as shown in Table 6. The 2nd column describes the buggy functions, and the 3rd column denotes the CVE (Common Vulnerabilities and Exposures) number or issue recognition confirmed by corresponding developers. We further tested these 20 validated bugs using two baseline tools, Frama-c and Fpse-study. The results, presented in Columns 4 and 5 of Table 6, reveal that neither Frama-c nor Fpse-study could detect 12 bugs in the Sox and Mupdf benchmarks. Frama-c identified 5 bugs in GSL, while Fpse-study detected a single bug in the same benchmark.

Taking CVE-2023-47483 as an example, in `flowTrigger` function shown in Fig. 4, YUSE finds the path "1-2-3-4-6-7-8-9" is reachable and reports a division-by-zero bug on line 9. The reason behind this is that during the analysis of Sox, (1) YUSE finds a path along which `p->samplesLen_ns` was assigned with zero before entering `flowTrigger` function; (2) Using inter-procedural analysis and taint analysis techniques, YUSE explores a path along which the variable's value keeps unmodified. Therefore, combined with its interval collection capability, YUSE can find this bug. By comparison, Frama-c and Fpse-study cannot discover this bug and result in false negatives, due to their insufficient support for float-point data and operations.

6. Limitations

When modeling a math function, we bind and assign a precise interval to its return value in order to accurately reason about the paths through the function, however, some complex math functions, such as discontinuous functions, discrete functions, or combinations of irregular math functions, e.g., $pow(x), x * sinx$, are difficult or even impossible to model symbolically.

Moreover, YUSE is affected by compiler optimizations like `-fast-math`. For a program under test, YUSE constructs its abstract syntax tree and control flow graph for symbolic analysis while compiler optimizations usually are performed before the preprocess stage. As a result, aggressive compiler optimizations may hinder an exception which otherwise would be detected by YUSE.

Table 6
Real bugs detected by YUSE. FS: Fpse-study; FC: Frama-C.

	Function	CVE/Issues	FS	FC
Sox	read_samples()	CVE-2023-47481	–	–
	sox_ladspa_flow()	CVE-2023-47482	–	–
	flowTrigger()	CVE-2023-47483	–	–
	lsx_offset_seek()	CVE-2023-47484	–	–
	seek()	CVE-2023-47485	–	–
	interleave()	CVE-2023-47486	–	–
Mupdf	mark_line()	CVE-2023-47487	–	–
	bmp_decompress_rle4()	CVE-2023-51103	–	–
	fz_new_pixmap_from_float_data()	CVE-2023-51104	–	–
	compute_color()	CVE-2023-51105	–	–
	pnm_binary_read_image()	CVE-2023-51106	–	–
	pnm_binary_read_image()	CVE-2023-51107	–	–
GSL	gsl_sf_conicalP_xlt1_large_neg_mu_e()	issue 1	–	√
	gsl_cdf_laplace_Qinv()	issue 2	–	√
	gsl_sf_bessel_Jnu_asympx_e()	issue 3	√	–
	gsl_ran_gamma_knuth()	issue 4	–	√
	hyperg_1F1_1()	issue 5	–	–
	gsl_s_bessel_Knu_scaled_asympx_e()	issue 6	–	√
	gsl_sf_conicalP_xlt1_large_neg_mu_e()	issue 7	–	√
	gsl_sf_bessel_Knu_scaled_asymp_unif_e()	issue 8	–	–

7. Related work

Recent years have seen significant advancements in the field of numerical software analysis, leading to the development of various research methods and tools. Broadly, these approaches are mainly based on three techniques: symbolic execution, fuzzing and abstract interpretation.

7.1. Detecting float-point bugs and exceptions via symbolic execution

Recent research efforts in software testing and program verification have extensively utilized symbolic execution. KLEE-fp (Collingbourne et al., 2011a) extends the KLEE symbolic execution framework to cross-check SIMD/SSE implementations against their corresponding scalar versions, in order to prove their bounded equivalence or find their inconsistencies. KLEE-cl (Collingbourne et al., 2011b) supports symbolic reasoning on the equivalence between symbolic values for crosschecking a C or C++ program against an accelerated OpenCL version, and thus can detect data race bugs in OpenCL programs.

Ariadne (Barr et al., 2013) adapts symbolic execution to detect floating-point exceptions. It uses a real arithmetic solver to solve floating-point constraints, however, compared to real numbers, floating-point numbers have limited precision and representational interval, converting the solving on floating-point constraints to the solving on real constraints will result in inaccuracies.

Similarly, Klee-float Liew et al. (2017), while supporting floating-point arithmetic, lacks specific logic for detecting floating-point exceptions and cannot adequately model math functions. FPGen (Guo and Rubio-González, 2020) formulates the problem of generating high error-inducing floating-point inputs as a code coverage maximization problem solved using symbolic execution, in order to maximize the numerical error. However, it cannot detect floating-point exceptions.

SeVR-fpe (Wu et al., 2017) uses value-range analysis to accelerate symbolic execution for floating-point exception detection. It takes advantage of interval arithmetic and relational arithmetic to identify value ranges and then enumerates the value in ranges to find a solution. In comparison, YUSE specifies *interval computation rules* for common computations, e.g., addition, multiplication, comparison, and adds these rules into the symbolic engine. The engine can apply these rules when collecting and merging the variables' values during symbolic execution, thereby making YUSE more efficient. Besides, YUSE can deal well with the results of math function calls which contain interval operations. The combination of math function modeling and interval binding further improves the accuracy of YUSE.

Zhang et al. (2022) have conducted the first empirical study on five existing symbolic execution methods for floating-point programs. They have shown different capacities of different methods for solving bit-vector floating-point formulas. Their study's result indicates that SMT, fuzzing, and real arithmetic solving-based methods complement each other in bug finding. Based on the finding, they implement a tool named Fpse-study synergizing the existing methods to improve symbolic execution's effectiveness. However, it cannot accurately collect floating-point constraints and model math functions.

Despite these advancements, existing symbolic execution tools are generally constrained by floating-point type conditions and lack specific logic for detecting floating-type bugs. They often struggle to accurately reason about program paths and detect floating-point bugs and exceptions, leaving much room to improve accuracy and efficiency.

7.2. Detecting float-point bugs and exceptions via dynamic analysis

Dynamic program analysis has proven to be an effective and practical approach for automatically identifying vulnerabilities in software. FP-Analysis (Zou et al., 2019) is a search-based approach to find inputs that trigger the largest atomic condition on each atomic operation, in order to search large floating-point errors in numerical code. FPED (Xia et al., 2021) is an inspector of floating-point errors for arithmetic expressions which can pick a suitable benchmark generation approach by analyzing the distribution of the expression of a floating-point operation, thereby minimizing the possibilities of underreporting floating-point errors.

NumFUZZ (Ma et al., 2022) leverages greybox fuzzing technique, informed by floating-point format-aware coverage, to identify floating-point exceptions in numerical programs. Additionally, RADE (Wang et al., 2022) combines ranking analysis and search algorithm to detect substantial floating-point errors in numerical programs. FPChecker (Laguna et al., 2022) is a dynamic analysis tool to detect floating-point errors in HPC applications and supports multiple HPC models like MPI, OpenMP, and CUDA. GPU-FPX (Li et al., 2023) can detect floating-point exceptions in NVIDIA GPUs by leveraging binary instrumentation and hence offers the occurrence location and control flow for the identified exception.

Despite their effectiveness, these methods are constrained by their reliance on the run-time environment or code instrumentation, which often results in high resource consumption, low performance and many false negatives.

7.3. Detecting float-point bugs and exceptions via abstract interpretation

Abstract interpretation is a prominent technique in the analysis of floating-point programs. Astrée (Cousot et al., 2005) uses abstract interpretation to prove the correctness of C code, including floating-point computations. It has been notably used in the verification of flight control software for Airbus aircraft. Frama-c (Kirchner et al., 2015), a comprehensive source code analysis platform for C programs, supports floating-point computations and can identify potential float-point exceptions and bugs like overflows and precision loss. PRECiSA (Moscato et al., 2017) is an abstract interpretation framework to analyze round-off errors in floating-point programs.

Despite their strengths, these abstract interpretation methods do not perform path-sensitive analysis and thus result in lots of false positives.

YUSE differs from the above methods by effectively addressing floating-point types and operations. It can accurately deal with math function operations, construct floating-point path conditions, and detect floating-point bugs and exceptions. YUSE boasts high coverage, low overhead, and minimal false positives. Furthermore, YUSE employs a two-phase solving strategy which significantly balances effectiveness and efficiency.

8. Conclusion

This paper proposes an analysis framework YUSE, which can construct floating-point related constraints and explore floating-point paths to detect floating-point bugs and exceptions in numerical code. Interval computation and interval constraint propagation in non-relational numerical abstract domains are also introduced to improve the analysis accuracy. Moreover, YUSE symbolically models math functions and leverages a two-phase constraint solving strategy to enhance its performance. Evaluation results on three real-world software, i.e., GSL, Sox and Mupdf, show that YUSE is more effective and efficient in detecting floating-point bugs and exceptions than two state-of-the-art tools, i.e., Frama-c and Fpse-study. Specifically, YUSE is 1.4× and 7.1× faster than Frama-c and Fpse-study, respectively, and found 20 new bugs, 12 of which were assigned CVE IDs and 8 of which were confirmed by corresponding developers.

CRediT authorship contribution statement

Dongyu Ma: Writing – review & editing, Writing – original draft, Validation, Software, Resources, Methodology, Investigation, Data curation, Conceptualization. **Zeyu Liang:** Writing – review & editing, Visualization, Validation, Methodology, Formal analysis. **Luming Yin:** Visualization, Software, Resources, Methodology. **Hongliang Liang:** Writing – review & editing, Validation, Supervision, Project administration, Methodology, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

References

Anon, Non-relational abstract domain https://www-apr.lip6.fr/~mine/enseignement/mpri/2017-2018/03-nonrel_ok.pdf.

ARIANE 5: Flight 501 failure http://sunnyday.mit.edu/nasa-class/Ariane5-report.html.

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

Barr, E.T., Vo, T., Le, V., Su, Z., 2013. Automatic detection of floating-point exceptions. ACM Sigplan Not. 48 (1), 549–560.

Collingbourne, P., Cadar, C., Kelly, P.H., 2011a. Symbolic crosschecking of floating-point and SIMD code. In: Proceedings of the Sixth Conference on Computer Systems. pp. 315–328.

Collingbourne, P., Cadar, C., Kelly, P.H.J., 2011b. Symbolic testing of OpenCL code. In: Eder, K., ao Lourenço, J., Shehory, O. (Eds.), Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers. In: Lecture Notes in Computer Science, vol. 7261, Springer, pp. 203–218. http://dx.doi.org/10.1007/978-3-642-34188-5_18.

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X., 2005. The ASTRÉE analyzer. In: Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005, Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings 14. Springer, pp. 21–30.

Using CSA to find bugs https://llvm.org/devmtg/2020-09/slides/Using_the_clang_static_ananalyzer_to_find_bugs.pdf.

CSA homepage. [Online]. Available: http://clang-analyzer.llvm.org/index.html.

de Moura, L., Bjørner, N., 2008. Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 337–340.

Dinda, P., Bernat, A., Hetland, C., 2020. Spying on the floating point behavior of existing, unmodified scientific applications. In: Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing. HPDC '20, Association for Computing Machinery, New York, NY, USA, pp. 5–16. http://dx.doi.org/10.1145/3369583.3392673.

Goldberg, D., 1991. What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. 23 (1), 5–48. http://dx.doi.org/10.1145/103162.103163.

GSL homepage. [Online]. Available: http://www.gnu.org/software/gsl/.

Guo, H., Rubio-González, C., 2020. Efficient generation of error-inducing floating-point inputs via symbolic execution. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 1261–1272.

2020. ISO/IEC/IEEE international standard - Floating-point arithmetic. In: ISO/IEC 60559:2020(E) IEEE Std 754-2019. pp. 1–86. http://dx.doi.org/10.1109/IEEESTD.2020.9091348.

King, J.C., 1976. Symbolic execution and program testing. Commun. ACM 19 (7), 385–394. http://dx.doi.org/10.1145/360248.360252.

Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B., 2015. Frama-C: A software analysis perspective. Form. Asp. Comput. 27 (3), 573–609. http://dx.doi.org/10.1007/s00165-014-0326-7.

Kong, S., Gao, S., Clarke, E.M., 2013. Floating-Point Bugs in Embedded GNU C Library. CMU School of Computer Science Technical Report, CMU-CS-13-130, Tech. Rep.

Laguna, I., Tirpankar, T., Li, X., Gopalakrishnan, G., 2022. Fpchecker: Floating-point exception detection tool and benchmark for parallel and distributed HPC. In: IEEE International Symposium on Workload Characterization, IISWC 2022, Austin, TX, USA, November 6-8, 2022. IEEE, pp. 39–50. http://dx.doi.org/10.1109/IISWC55918.2022.00014.

Li, X., Laguna, I., Fang, B., Swirydowicz, K., Li, A., Gopalakrishnan, G., 2023. Design and evaluation of GPU-FPX: A low-overhead tool for floating-point exception detection in NVIDIA GPUs. In: Butt, A.R., Mi, N., Chard, K. (Eds.), Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2023, Orlando, FL, USA, June 16-23, 2023. ACM, pp. 59–71. http://dx.doi.org/10.1145/3588195.3592991.

Liew, D., Cadar, C., Donaldson, A., KLEE floating-point extensions team imperial. https://github.com/srg-imperial/klee-float.

Ma, C., Chen, L., Yi, X., Fan, G., Wang, J., 2022. NuMFUZZ: A floating-point format aware fuzzer for numerical programs. In: 2022 29th Asia-Pacific Software Engineering Conference. APSEC, pp. 338–347. http://dx.doi.org/10.1109/APSEC57359.2022.00046.

Moscato, M., Titolo, L., Dutle, A., Munoz, C.A., 2017. Automatic estimation of verified floating-point round-off errors via static analysis. In: Computer Safety, Reliability, and Security: 36th International Conference, SAFECOMP 2017, Trento, Italy, September 13-15, 2017, Proceedings 36. Springer, pp. 213–229.

mupdf https://github.com/ArtifexSoftware/mupdf.

Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia https://www.gao.gov/assets/imtec-92-26.pdf.

Range constraint solver in CSA https://clang.llvm.org/doxygen//RangeConstraintManager_8cpp_source.html.

SOX homepage. [Online]. Available: https://sourceforge.net/projects/sox/.

Wang, Z., Yi, X., Yu, H., Yin, B., 2022. Detecting high floating-point errors via ranking analysis. In: 2022 29th Asia-Pacific Software Engineering Conference. APSEC, IEEE, pp. 397–406.

Wu, X., Li, L., Zhang, J., 2017. Symbolic execution with value-range analysis for floating-point exception detection. In: Lv, J., Zhang, H.J., Hinchey, M., Liu, X. (Eds.), 24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017. IEEE Computer Society, pp. 1–10. http://dx.doi.org/10.1109/APSEC.2017.6.

Wu, X., Xu, Z., Yan, D., Wu, T., Yan, J., Zhang, J., 2016. The floating-point extension of symbolic execution engine for bug detection. In: Potanin, A., Murphy, G.C., Reeves, S., Dietrich, J. (Eds.), 23rd Asia-Pacific Software Engineering Conference, APSEC 2016, Hamilton, New Zealand, December 6-9, 2016. IEEE Computer Society, pp. 265–272. http://dx.doi.org/10.1109/APSEC.2016.045.

Xia, Y., Guo, S., Hao, J., Liu, D., Xu, J., 2021. Error detection of arithmetic expressions. J. Supercomput. 77 (6), 5492–5509.

Z3 solver https://github.com/Z3Prover/z3.

Zhang, G., Chen, Z., Shuai, Z., 2022. Symbolic execution of floating-point programs: How far are we? In: 29th Asia-Pacific Software Engineering Conference, APSEC 2022, Virtual Event, Japan, December 6-9, 2022. IEEE, pp. 179–188. http://dx.doi.org/10.1109/APSEC57359.2022.00030.

Zou, D., Zeng, M., Xiong, Y., Fu, Z., Zhang, L., Su, Z., 2019. Detecting floating-point errors via atomic conditions. Proc. ACM Program. Lang. 4 (POPL), 1–27.

Dongyu Ma received her B.Sc. degree in Software Engineering from Dalian JiaoTong University, China in 2021. She is currently a M.Sc. student at Beijing University of Posts and Telecommunications, China, working on program analysis and software testing.

Zeyu Liang received his B.Sc. degree in Mathematics from University California San Diego, USA in 2024. He is currently a research assistant at Beijing University of Posts and Telecommunications, China, working on deep learning and software testing.

Luming Yin received his B.Sc. degree in Computer Science from GuangXi University, China in 2022. He is currently a M.Sc. student at Beijing University of Posts and Telecommunications, China, working on program analysis and software testing.

Hongliang Liang received his Ph.D. degree in computer science from the University of Chinese Academy of Sciences, China in 2002. He leads the Trusted Software and Intelligent System research group at Beijing University of Posts and Telecommunications, China. His main research interests include system software, trustworthy software, and artificial intelligence. He has published more than 40 articles in high impact journals and blind peer-reviewed conferences. He is a member of the IEEE and ACM, and serves as a reviewer for some prestigious journals, including the IEEE TSE, TPAMI, TRel, TIST, FGCS, COSE and JSS, etc.