Robustness Analysis of Floating-Point Programs by Self-Composition

Liqian Chen*, Jiahong Jiang, Banghu Yin, Wei Dong, and Ji Wang

National Laboratory for Parallel and Distributed Processing,
National University of Defense Technology, Changsha 410073, China

Abstract. Robustness is a key property for critical systems that run in uncertain environments, to ensure that small input perturbations can cause only small output changes. Current critical systems often involve lots of floating-point computations which are inexact. Robustness analysis of floating-point programs needs to consider both the uncertain inputs and the inexact computation. In this paper, we propose to leverage the idea of self-composition to transform the robustness analysis problem into a reachability problem, which enables the use of standard reachability analysis techniques such as software model checking and symbolic execution for robustness analysis. To handle floating-point arithmetic, we employ an abstraction that encompasses the effect of rounding and that can encompass all rounding modes. It converts floating-point expressions into linear expressions with interval coefficients in exact real arithmetic. On this basis, we employ interval linear programming to compute the maximum output change or maximum allowed input perturbation for the abstracted programs. Preliminary experimental results of our prototype implementation are encouraging.

1 Introduction

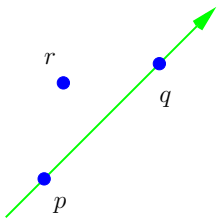
Uncertainty and inexactness in computing have attracted much attention in computer science. In Cyber Physical Systems (CPS), the discrete world of computation is integrated with the continuous world of physical processes. Moreover, CPS run in the open environmental context and thus have to deal with uncertain data which may come from noisy sensor data or approximate computation. Hence, inputs for programs in CPS are of intrinsic uncertainty. On the other hand, due to finite precision on computers, physical values are truncated into digital ones. In modern computers, real numbers are approximated by a finite set of floating-point numbers. Due to the pervasive rounding errors, numerical computation using floating-point arithmetic is not exact. Since many safety-critical CPS systems (such as aircrafts, automobiles and medical devices) often involve lots of numerical computations, there is a great need to ensure that these programs are *robust* with respect to the uncertain input as well as the inexact computation.

Although *robustness* is long known as a standard correctness property for control systems [39], considering the robustness of programs is quite recent [12–14, 34]. Intuitively, robustness of a program means that small input perturbations of the program can

* Corresponding author: lqchen@nudt.edu.cn (Liqian Chen)

cause only small output changes. Most existing work on analyzing robustness of programs assumes that the analyzed program is in exact real arithmetic, although floating-point computation is pervasive in practical applications. This paper targets the analysis of robustness properties of floating-point programs.

A program using floating-point arithmetic often exhibits more robustness issues than that using exact real arithmetic, due to the misunderstandings and non-intuitive behaviors of floating-point semantics. Although floating-point arithmetic is quite different from the exact real arithmetic, most developers of floating-point programs will write programs as if computations were done in exact arithmetic. For the same input, the control flow of the program using floating-point arithmetic can be different from the one that would be taken assuming exact real arithmetic. Similarly, for two inputs whose values are close to each other, the resulting two control flows of the same program can be different (even when following exact real arithmetic). Two different control flows may lead to very large difference in outputs.




```
1 int Orientation(float px, float py, float qx, float qy, float rx, float ry)
2 {
3    float pqx=qx-px, pqy=qy-py;
4    float prx=rx-px, pry=ry-py;
5    float det = pqx * pry - pqy * prx;
6    if (det > 0) return 1;
7    if (det < 0) return -1;
8    return 0;
9 }
```

Fig. 1. Floating-point implementation for orientation test of 2D points

We illustrate the robustness problem due to floating-point computation using a motivating example shown in Fig. 1, which is a "classroom" example of a robustness problem frequently used in the field of geometric computations [30]. The program `Orientation` implements the 2D orientation test that decides whether a point r lies to the left of, to the right of, or on the line $\overrightarrow{pq}$ defined by the 2 points p, q, by evaluating the sign of a determinant `det` which is expressed in terms of the coordinates of the input points. Due to rounding errors, the floating-point computation of the determinant `det` may lead to a wrong result when the true determinant (via exact real arithmetic) is close to zero. From the robustness point of view, for this program, even a very small input perturbation may lead to an output change of 1 or 2. If the rounding modes for the

floating-point operations are not determinate in the program, the output change can be 2 even when there is no perturbation in the inputs (by running the program in different rounding modes). This misinformation may then lead to a failure of a computational geometry application (e.g., crash or not terminate) or produce wrong results [30].

Analyzing robustness of floating-point programs is more challenging than analyzing programs assuming exact real arithmetic, since besides the input perturbations, we need to consider also the inexactness of floating-point computation. The floating-point program itself acts as if inputs were perturbated due to the pervasive rounding errors or non-determinate rounding modes. There exist a few known pitfalls of analyzing and verifying floating-point programs [37].

In this paper, we present a robustness analysis method for floating-point programs. The key idea is to leverage the self-composition technique from the field of secure information flow to transform the robustness analysis problem into a reachability (safety) problem. Then we use standard rechability analysis techniques such as software model checking and symbolic execution to analyze the self-composed programs. To cope with floating-point arithmetic, we utilize a rounding mode insensitive abstraction method to abstract floating-point expressions into linear expressions with interval coefficients in the field of reals. On this basis, we use interval linear programming to compute the maximum output change (when given the input perturbation) or the maximum allowed input perturbation (when given the output change) for the abstracted programs. The preliminary experimental results are promising on benchmark programs.

The rest of the paper is organized as follows. Section 2 reviews the IEEE 754 floating-point arithmetic and the basic theory of interval linear systems as well as interval linear programming. Section 3 presents the robustness analysis approach via self-composition for programs (that assume exact real arithmetic). Section 4 presents the techniques to handle floating-point arithmetic. Section 5 presents our prototype implementation together with preliminary experimental results. Section 6 discusses some related work before Section 7 concludes.

2 Preliminaries

In this section, we briefly provide the background on the IEEE 754 floating-point arithmetic and the basic theory on interval arithmetics as well as interval linear programming.

2.1 The IEEE 754 floating-point arithmetic

A digital computer cannot represent all possible real numbers in mathematics exactly. In computing, floating-point numbers provides an approach to represent a finite subset of the real numbers. In this paper, we focus on analyzing programs with respect to the binary formats of the IEEE 754 floating-point standard [43] which is the most commonly used floating-point representation and is followed by almost all modern computers.

In the IEEE 754 standard, the binary representation of a floating-point number x can be described as $x = (-1)^S \times M \times 2^E$, where

– S is the 1-bit *sign* of x, which represents that x is positive (when $S = 0$) or negative (when $S = 1$);
– $E = e - \mathbf{bias}$ is called the *exponent*, where e is a biased $\mathbf{e}$-bit unsigned integer and $\mathbf{bias} = 2^{\mathbf{e}-1} - 1$;
– $M = m_0.m_1m_2 \dots m_{\mathbf{p}}$ is called the *significand*, where $f = .m_1m_2 \dots m_{\mathbf{p}}$ represents a $\mathbf{p}$-bit fraction and m_0 is the hidden bit without need of storage.

The values of $\mathbf{e}, \mathbf{bias}, \mathbf{p}$ depend on the floating-point formats. The IEEE 754 standard supports several formats, among which the basic formats include:

– 32-bit single-precision format, where $\mathbf{e} = 8$ (and thus $\mathbf{bias} = 127$), $\mathbf{p} = 23$;
– 64-bit double-precision format, where $\mathbf{e} = 11$ (and thus $\mathbf{bias} = 1023$), $\mathbf{p} = 52$.

According to the value of e, the floating-point numbers can be divided into the following categories:

– *normalized* number $(-1)^S \times 1.f \times 2^{e-\mathbf{bias}}$, when $1 \le e \le 2^{\mathbf{e}} - 2$;
– *denormalized* number $(-1)^S \times 0.f \times 2^{1-\mathbf{bias}}$, when $e = 0$ and $f \neq 0$;
– $+0$ or -0, when $e = 0$ and $f = 0$;
– $+\infty$ or $-\infty$, when $e = 2^{\mathbf{e}} - 1$ and $f = 0$;
– *NaN* (Not a Number), when $e = 2^{\mathbf{e}} - 1$ and $f \neq 0$.

Let $\mathbf{F}$ be the set of floating-point formats. For each $\mathbf{f} \in \mathbf{F}$, we define

– $mf_{\mathbf{f}} \stackrel{\text{def}}{=} 2^{1-\mathbf{bias}-\mathbf{p}}$, the smallest non-zero positive floating-point number;
– $Mf_{\mathbf{f}} \stackrel{\text{def}}{=} (2 - 2^{-\mathbf{p}})2^{2^{\mathbf{e}}-\mathbf{bias}-2}$, the largest non-infinity floating-point number.

In general, the result of a floating-point operation may not be exactly representable in the floating-point representation, and thus the result needs to be rounded into a floating-point number. The IEEE 754 standard supports four rounding modes: toward nearest, toward $+\infty$, toward $-\infty$, and toward zero. In this paper, in order to distinguish floating-point arithmetic operations from exact real arithmetic ones, we introduce additional notations. As usual, $\{+, -, \times, /\}$ are used as exact rational arithmetic operations. The corresponding floating-point operations are denoted by $\{\oplus_{\mathbf{f},r}, \ominus_{\mathbf{f},r}, \otimes_{\mathbf{f},r}, \oslash_{\mathbf{f},r}\}$, tagged with a floating-point format $\mathbf{f} \in \mathbf{F}$ and a rounding mode $r \in \{+\infty, -\infty, 0, n\}$ (n representing rounding to nearest). We also use ? to denote arbitrary rounding mode.

Due to rounding errors, many well-known algebraic properties (such as associativity and distributivity) over the reals do not hold for floating-point arithmetic.

Example 1. Consider the following expressions in the 32-bit single-precision floating-point arithmetic.

$$(2^{24} \oplus_{32,?} -2^{24}) \oplus_{32,?} 1 = 1$$
$$(2^{24} \oplus_{32,-\infty} 1) \oplus_{32,-\infty} -2^{24} = 0$$
$$(2^{24} \oplus_{32,+\infty} 1) \oplus_{32,+\infty} -2^{24} = 2$$

Note that in the 32-bit single-precision format, the significand is $M = m_0.m_1m_2 \dots m_{\mathbf{23}}$. However, to represent the exact result of $2^{24} + 1$ over the reals, we need one more bit for the significand M (say $m_{\mathbf{24}}$). Hence, rounding happens. $2^{24} \oplus_{32,-\infty} 1$ will result in 2^{24}, while $2^{24} \oplus_{32,+\infty} 1$ will result in $(1 + 2^{-23}) \times 2^{24}$.

2.2 Interval Linear Systems and Interval Linear Programming

Let $\underline{A}, \overline{A} \in \mathbb{R}^{m\times n}$ be two matrices with $\underline{A} \le \overline{A}$, where comparison operators are defined element-wise, then the set of matrices $\mathbf{A} \in \mathbb{IR}^{m\times n}$ defined by

$$\mathbf{A} = [\underline{A}, \overline{A}] = \{A \in \mathbb{R}^{m\times n} : \underline{A} \le A \le \overline{A}\}$$

is called an *interval matrix*, and the matrices $\underline{A}, \overline{A}$ are called its bounds. Let us define the *center matrix* of $\mathbf{A}$ as $A_c = \frac{1}{2}(\underline{A} + \overline{A})$ and the *radius matrix* as $\triangle_A = \frac{1}{2}(\overline{A} - \underline{A})$. Then, $\mathbf{A} = [\underline{A}, \overline{A}] = [A_c - \triangle_A, A_c + \triangle_A]$. An *interval vector* is a one-column interval matrix $\mathbf{d} = [\underline{d}, \overline{d}] = \{d \in \mathbb{R}^m : \underline{d} \le d \le \overline{d}\}$, where $\underline{d}, \overline{d} \in \mathbb{R}^m$ and $\underline{d} \le \overline{d}$.

Let $\mathbf{A}$ be an $m \times n$ interval matrix and b be a vector of size m. The following system of interval linear inequalities

$$\mathbf{A}x \le b$$

denotes an *interval linear system*, that is, the *family* of all systems of linear inequalities $Ax \le b$ such that $A \in \mathbf{A}$.

Definition 1 (Weak solution). *A vector $x \in \mathbb{R}^n$ is called a* weak solution *of the interval linear system $\mathbf{A}x \le b$, if it satisfies $Ax \le b$ for some $A \in \mathbf{A}$. Furthermore, the set*

$$\Sigma_\exists(\mathbf{A}, b) = \{x \in \mathbb{R}^n : \exists A \in \mathbf{A},\ Ax \le b\}$$

is said to be the weak solution set *of the system $\mathbf{A}x \le b$.*

The weak solution set of an interval linear system is characterized by the following theorem [40].

Theorem 1. *A vector $x \in \mathbb{R}^n$ is a weak solution of $\mathbf{A}x \le b$ iff it satisfies $A_c x - \triangle_A |x| \le b$.*

Let $\mathbf{A} \in \mathbb{IR}^{m\times n}$ be an $m \times n$ interval matrix, $b \in \mathbb{R}^m$ be an m-dimensional vector, and $\mathbf{c} \in \mathbb{IR}^n$ be an n-dimensional interval vector. The *family* of linear programming (LP) problems

$$f(A, b, c) = \max\{c^T x :\ Ax \le b\}$$

with data satisfying

$$A \in \mathbf{A}, c \in \mathbf{c}$$

is called an *interval linear programming (ILP)* problem.

In this paper, we are only interested in computing the upper bound $\overline{f}(\mathbf{A}, b, \mathbf{c}) = \sup\{f(A, b, c) : A \in \mathbf{A}, c \in \mathbf{c}\}$. In general, according to Theorem 1, to compute the exact $\overline{f}(\mathbf{A}, b, \mathbf{c})$, in the worst case up to 2^n LP problems have to be solved, one for each orthant. Recall that a (closed) orthant is one of the 2^n subsets of an n-dimensional Euclidean space defined by constraining each Cartesian coordinate axis to be either nonnegative or nonpositive. In each orthant, we consider the following LP problem:

$$\begin{array}{ll} \max & \sum_{j=1}^n c'_j x_j \\ s.t. & \bigwedge\limits_{0\le i\le m} \sum_{j=1}^n A'_{ij} x_j \le b_i \end{array}$$

where

$$c'_j = \begin{cases} \overline{c}_j & \text{if } x_j \geq 0 \\ \underline{c}_j & \text{if } x_j < 0 \end{cases} \qquad A'_{ij} = \begin{cases} \underline{A}_{ij} & \text{if } x_j \geq 0 \\ \overline{A}_{ij} & \text{if } x_j < 0 \end{cases}$$

And $\overline{f}(\mathbf{A}, b, \mathbf{c})$ will be the the maximum over all the optimal values of the 2^n LP problems with one per each orthant.

3 Robustness analysis via self-composition

3.1 Robustness of programs

In this paper, we follow the definition for robustness of programs used by Majumdar and Saha [34]. Let f be a function with inputs $x_1, \ldots, x_n$ and output y, i.e., $y = f(x_1, \ldots, x_n)$. The function f is said to be (δ, ϵ)-*robust* in the i-th input x_i if a perturbation of at most δ in the input x_i can only cause a change of at most ϵ in the output, i.e.,

$$\forall x_i, x'_i . |x_i - x'_i| \leq \delta \implies |f(x_1, \ldots, x_i, \ldots, x_n) - f(x_1, \ldots, x'_i, \ldots, x_n)| \leq \epsilon$$

where $\delta, \epsilon \in \mathbb{R}$ are non-negative constant parameters specified by users. Recall that we consider the perturbation over only one input at a time while assume that there is no perturbation over all other inputs at the same time.

Moreover, in practice, users may be interested in the maximum output change of y with respect to x_i and δ, i.e.,

$$\overline{\epsilon}_\delta \stackrel{\text{def}}{=} \max_{x,x'} \left\{ |y - y'| \;\middle|\; \begin{array}{l} y = f(x_1, \ldots, x_i, \ldots, x_n) \\ y' = f(x_1, \ldots, x'_i, \ldots, x_n) \\ |x_i - x'_i| \leq \delta \end{array} \right\}$$

Similarly, users may be interested in the maximum input perturbation allowed over x_i with respect to y and ϵ, i.e.,

$$\overline{\delta}_\epsilon \stackrel{\text{def}}{=} \max_{y,y'} \left\{ |x_i - x'_i| \;\middle|\; \begin{array}{l} y = f(x_1, \ldots, x_i, \ldots, x_n) \\ y' = f(x_1, \ldots, x'_i, \ldots, x_n) \\ |y - y'| \leq \epsilon \end{array} \right\}$$

Example 2. Consider the following program which implements a piece-wise linear function. When $x = 1.001$, the two branches give the same result $y = 1002.001$ in exact real arithmetic (assuming floats are reals). It is easy to see that in exact real arithmetic, this program is $(0.1, \epsilon_0)$-*robust* for all $\epsilon_0 \geq 100.1$ but is not $(0.1, \epsilon_1)$-*robust* for all $\epsilon_1 < 100.1$. This can be deduced by observing that in exact real arithmetic, given the input perturbation $\delta = 0.1$, the maximum output change of y is $\overline{\epsilon}_\delta = 100.1$; Given the output change $\epsilon = 100.1$, the maximum input perturbation allowed over x is $\overline{\delta}_\epsilon = 0.1$.

```
1 float piecewise_linear(float x) {
2   float y;
3   if (x < 1.001)
4     y = x + 1001.0;
5   else
6     y = x * 1001.0;
7   return y;
8 }
```

3.2 Self-composition

The idea of self-composition is firstly used in the field of secure information flow [4, 20] to characterize non-interference. Let P be a program and P' be a copy of P with each variable x in P replaced by a fresh variable x'. Using Hoare triples, non-interference can be characterized as:

$$\{L = L'\}\ \ P; P'\ \ \{L = L'\}$$

where L denotes low-security variables. In other words, it requires that running two instances of the same program with equal low-security values and arbitrary high-security values results in equal low-security values. Hence, via self-composition, a secure information flow property of P reduces to a reachability property over single program executions of the program $P; P'$.

In this paper, we would like to leverage the idea of self-composition to reduce the robustness problem of a program P into an equivalent reachability problem over $P; P'$. Assume that program P has n input variables $x_1, \ldots, x_n$ and an output variable y. Similarly, using Hoare triples, the (δ, ϵ)-*robustness* of program P over the i-th input x_i can be characterized as:

$$\{|x_i - x_i'| \leq \delta \wedge \bigwedge_{1 \leq j \leq n, j \neq i} x_j = x_j'\}\ \ P; P'\ \ \{|y - y'| \leq \epsilon\}$$

Example 3. Consider again the program `piecewise_linear` in Example 1. The self-composition of the function body is shown below. To express the robustness property, we add the assumption $|x - x'| \leq \delta$ as a precondition at the beginning of the self-composed program and add an assertion $|y - y'| \leq \epsilon$ as a postcondition at the end.

```
1  assume(−δ ≤x-x′ ≤ δ)
2
3  if (x < 1.001)
4     y = x + 1001.0;
5  else
6     y = x * 1001.0;
7
8  if (x′ < 1.001)
9     y′ = x′ + 1001.0;
10 else
11    y′ = x′ * 1001.0;
12
13 assert(−ϵ ≤y-y'≤ ϵ)
```

Essentially, the copied program P' has the same program code as P but uses variables with different initial values. Hence, there exists inherent symmetry and redundancy in the self-composed programs. In order to make the following analysis and verification process for self-composed programs easier, program transformations can be used to optimize the self-composed programs. In the field of secure information flow analysis, Terauchi and Aiken [44] proposed type directed transformation to improve self-composition. The main idea of type-directed transformation is not to self-compose

branch (or loop) statements when the branch (or loop) condition is only dependent on the values of low-security variables. In addition, for an assignment statement $\{x := e;\}$, when the right-hand expression e is only dependent on the values of low-security variables, its self-composition is simplified as $\{x := e; x' := x;\}$.

With respect to robustness, a similar transformation can be applied. Intuitively, we could consider the perturbed input variable x_i as a high-security variable and all other input variables x_j's as low-security variables where $j \neq i$. Hence, similarly to type directed transformation, we do not self-compose branch (or loop) statements when the branch (or loop) condition is not dependent on the values of perturbed input variables. For an assignment statement $\{x := e;\}$, when the right-hand expression e is not dependent on the values of perturbed input variables, its self-composition is simplified as $\{x := e; x' := x;\}$.

Example 4. Consider the following function *min_plus*1 which implements $\min(x+1, y)$ by adding 0.1 to x ten times. The optimized self-composition result of the function body after applying transformation is given below, when we consider the perturbation over the input variable x (while assuming no perturbation over y). More specifically, since the loop condition $i < 10$ in the original program is not dependent on the value of the perturbed input variable x, we do not self-compose the loop statement and thus there is only one loop in the transformed resulting self-composed program.

```
1  float min_plus1(float x, float y){
2    float z;
3    int i;
4    i = 0;
5    while (i < 10) {
6      x = x + 0.1;
7      i = i + 1;
8    }
9    if (y <= x)  z = y;
10   else  z = x;
11   return z;
12 }
```

```
1  assume(−δ ≤x-x′ ≤ δ and y=y′)
2
3  i = 0; i′ = i;
4  while (i < 10) {
5    x = x + 0.1; x′ = x′ + 0.1;
6    i = i + 1; i′ = i;
7  }
8  if (y <= x)  z = y;
9  else  z = x;
10 if (y′ <= x′)  z′ = y′;
11 else  z′ = x′;
12
13 assert(−ϵ ≤z-z′ ≤ ϵ)
```

3.3 Robustness analysis of self-composed programs

Via self-composition, the robustness analysis problem can be reduced to solving a standard reachability (safety) problem. The recent success of automatic analysis and verification tools (such as SLAM [1], CBMC [18], and ASTRÉE [5]) aiming at checking reachability properties in programs makes this approach promising. In the following, we will present two popular reachability analysis approaches that fit for analyzing robustness, i.e., software model checking and symbolic execution.

Checking robustness by software model checking Software model checking [29] provides an automatic approach to check whether a program satisfies a property by exploring the state space of the program. For the robustness analysis problem, the property

to be checked is an assertion at the end of the self-composed programs stating that the output change is bounded by ϵ, i.e., $\texttt{assert}(-\epsilon \leq \text{y-y}' \leq \epsilon)$. A main advantage of using software model checking is that it will generate a counterexample when the robustness property does not hold. The counterexample shows an execution trace which violates the robustness property. A counterexample is very helpful for the users to identify the source of non-robustness.

Finding maximum output change (or input perturbation) by symbolic execution Symbolic execution [10, 31] is a technique to analyze a program by executing the program with symbolic rather than concrete values as program inputs. The process of symbolic execution essentially generates and explores a symbolic execution tree which represents all execution paths followed during the process. Each tree node represents a symbolic execution state, while each edge represents a program transition between the states. At any tree node, the symbolic execution state includes a program counter, a *path condition* (PC) that encodes the constraints on the symbolic inputs to reach that node, a *path function* (PF) that represents the current values of the program variables as function of symbolic inputs when the path condition holds true. The path condition is a boolean expression over the symbolic inputs. The path function describes the expected result of the program, under the given path condition. Due to conditional branches and loops in a program, the symbolic execution of a program will result in a set of paths, each of which is described by a pair $\langle PC, PF\rangle$ of the path condition PC and the associated path function PF.

We now show how to use symbolic execution to conduct a robustness analysis of program P with inputs $x_1, \ldots, x_n$ and output y. First, the analysis algorithm performs symbolic execution on the self-composed program $P; P'$. Assume that the algorithm collects, at the end of the self-composed program, a set $\mathcal{S}$ of pairs $\langle PC, PF\rangle$ of the path condition PC and the associated path function PF. Then for each $s \triangleq \langle PC, PF\rangle \in \mathcal{S}$, we compute the maximum output change $\overline{\epsilon}_\delta^s$:

$$\begin{array}{ll} \max & |PF_y - PF_{y'}| \\ s.t. & (|x_i - x_i'| \leq \delta \wedge \bigwedge\limits_{1\leq j\leq n, j\neq i} x_j = x_j') \wedge PC \end{array}$$

Here, PF_y and $PF_{y'}$ denote the symbolic expressions that the path function PF maps the variables y and y' to, respectively. Let $\overline{\epsilon}_\delta$ be the maximum element of $\{\overline{\epsilon}_\delta^s \mid s \in \mathcal{S}\}$, i.e., $\overline{\epsilon}_\delta = \max(\{\overline{\epsilon}_\delta^s \mid s \in \mathcal{S}\})$. If $\overline{\epsilon}_\delta \leq \epsilon$, then the original program P is (δ, ϵ)-*robust*.

Similarly, given the bound of output change ϵ, computing the maximum allowed input perturbation is reduced to solving a series of the following optimization problems for each $s \triangleq \langle PC, PF\rangle \in \mathcal{S}$:

$$\begin{array}{ll} \max & |x_i - x_i'| \\ s.t. & (\bigwedge\limits_{1\leq j\leq n, j\neq i} x_j = x_j') \wedge PC \wedge |PF_y - PF_{y'}| \leq \epsilon \end{array}$$

And $\overline{\delta}_\epsilon$ will be the maximum element of $\{\overline{\delta}_\epsilon^s \mid s \in \mathcal{S}\}$, i.e., $\overline{\delta}_\epsilon = \max(\{\overline{\delta}_\epsilon^s \mid s \in \mathcal{S}\})$.

4 Robustness analysis of floating-point programs

In this section, we consider the robustness analysis problem of floating-point programs. In Section 3.3, we propose to utilize software model checking and symbolic execution to perform robustness analysis of self-composed programs (in exact real arithmetic). However, most existing software model checkers and symbolic execution tools can not be directly applied to floating-point programs, since they rely on constraint solvers that often assume good algebraic properties such as associativity and distributivity over the reals which do not hold for floating-point arithmetic. To handle floating-point arithmetic, we have to resort to bit-precise modeling of floating-point arithmetic or abstracting floating-point arithmetic to real number arithmetic.

CBMC (C Bounded Model Checker) [18] is one of the few software model checkers that have considered floating-point arithmetic. CBMC employs a sound and complete decision procedure for floating-point arithmetic [7, 26]. It precisely encodes floating-point operations as functions on bit-vectors. Each floating-point operation is further modeled as a formula in propositional logic. The formula is then handled by a SAT-solver in the backend to check for satisfiability.

When we consider symbolic execution of floating-point programs, both the path condition and the path function will involve floating-point expressions. Hence,to compute the maximum output change (or maximum allowed input perturbation), we need optimization methods supporting floating-point constraints. However, as far as we know, even for linear programming, there is no available sound solver supporting floating-point constraints. To this end, in this paper, we abstract the optimization problem with floating-point constraints into an interval linear programming problem (i.e., linear programming problem with interval coefficients) over the reals. The main idea is to use the so-called *floating-point linearization* technique [35, 36] to abstract floating-point expressions into linear real number expressions with interval coefficients (in the form of $\Sigma_i[a_i, b_i]x_i$) .

4.1 Floating-point abstraction

In this subsection, we will explain how to abstract floating-point expressions into interval linear expressions over the reals.

First, let us consider the upper bound on rounding errors due to one floating-point operation. Let $R_{\mathbf{f},r}(x)$ denote the floating-point rounding function that maps a real number x to a floating-point number (or a runtime error due to, for example, overflows) with respect to the floating-point format $\mathbf{f}$ and the rounding mode r. The amount of the rounding error due to $R_{\mathbf{f},r}(x)$ depends on the category of x.

- If x is in the range of normalized numbers, then $|R_{\mathbf{f},r}(x) - x| \leq \varepsilon_{\text{rel}} \cdot |x|$ where $\varepsilon_{\text{rel}} = 2^{-\mathbf{p}}$ (wherein $\mathbf{p}$ is the number of bits of fraction in the significand of the floating-point format $\mathbf{f}$). In this case we consider the relative rounding error ε_{rel}.
- If x is in the range of denormalized number, then $|R_{\mathbf{f},r}(x) - x| \leq \varepsilon_{\text{abs}}$, where $\varepsilon_{abs} = mf_{\mathbf{f}}$ (wherein $mf_{\mathbf{f}}$ is the smallest non-zero positive denormalized floating-point number in the floating-point format $\mathbf{f}$, which is also the gap between two neighboring denormalized numbers). In this case, we consider the absolute rounding error ε_{abs}.

The rounding errors of these two cases can be unified as

$$|R_{\mathbf{f},r}(x) - x| \leq \max(\varepsilon_{\text{rel}} \cdot |x|, \varepsilon_{\text{abs}})$$

Since max is not a linear operation, we derive an over-approximation

$$|R_{\mathbf{f},r}(x) - x| \leq \varepsilon_{\text{rel}} \cdot |x| + \varepsilon_{\text{abs}}$$

Furthermore, when $b \geq 0$, $|y| \leq b$ is equivalent to $y = [-1, 1] \times b$. Hence,

$$R_{\mathbf{f},r}(x) - x = [-1, 1](\varepsilon_{\text{rel}} \cdot |x| + \varepsilon_{\text{abs}})$$

that is,

$$R_{\mathbf{f},r}(x) = [1 - \varepsilon_{\text{rel}}, 1 + \varepsilon_{\text{rel}}] \times x + [-\varepsilon_{\text{abs}}, \varepsilon_{\text{abs}}]$$

In general, we could abstract floating-point operations into interval linear expressions in real number semantics. E.g.,

$$x \oplus_{\mathbf{f},r} y$$

that is

$$R_{\mathbf{f},r}(x + y)$$

can be abstracted into

$$[1 - \varepsilon_{\text{rel}}, 1 + \varepsilon_{\text{rel}}] \times (x + y) + [-\varepsilon_{\text{abs}}, \varepsilon_{\text{abs}}]$$

that is

$$[1 - \varepsilon_{\text{rel}}, 1 + \varepsilon_{\text{rel}}] \times x + [1 - \varepsilon_{\text{rel}}, 1 + \varepsilon_{\text{rel}}] \times y + [-\varepsilon_{\text{abs}}, \varepsilon_{\text{abs}}]$$

The advantage of this kind of rounding mode insensitive floating-point abstractions is that the result is sound with respect to arbitrary rounding modes, since $R_{\mathbf{f},r}(x)$ always satisfies $R_{\mathbf{f},-\infty}(x) \leq R_{\mathbf{f},r}(x) \leq R_{\mathbf{f},+\infty}(x)$ while $|R_{\mathbf{f},r}(x) - x| \leq \varepsilon_{\text{rel}} \cdot |x| + \varepsilon_{\text{abs}}$ has already taken into account the extreme cases of $r = -\infty$ and $r = +\infty$. This is of practical importance, since we may not know the exact rounding mode for each floating-point operation. E.g., C99 provides the `fesetround()` function to set the current rounding mode. Of course, when we know the exact rounding mode for the floating-point operation, we could make the floating-point abstraction more precise. E.g., if the current rounding mode is toward nearest, then

$$R_{\mathbf{f},n}(x) = [1 - \varepsilon_{\text{rel}}/2, 1 + \varepsilon_{\text{rel}}/2] \times x + [-\varepsilon_{\text{abs}}/2, \varepsilon_{\text{abs}}/2]$$

In addition, if we know the range of x, we may also define more precise floating-point abstractions. E.g., if we know that x is in the range of denormalized numbers, then

$$R_{\mathbf{f},r}(x) = x + [-\varepsilon_{\text{abs}}, \varepsilon_{\text{abs}}]$$

For the sake of generality, in this paper, we use the following rounding mode insensitive floating-point abstraction:

$$R^{\sharp}_{\mathbf{f},?}(x) = [1 - \varepsilon_{\text{rel}}, 1 + \varepsilon_{\text{rel}}] \times x + [-\varepsilon_{\text{abs}}, \varepsilon_{\text{abs}}]$$

where we assume $|x| < Mf_{\mathbf{f}}$.

More clearly, we use the following abstraction for floating-point arithmetic:

$$
\begin{aligned}
R^{\sharp}_{\mathbf{f},?}(x \oplus_{\mathbf{f},?} y) &= [1-\varepsilon_{\text{rel}}, 1+\varepsilon_{\text{rel}}] \times x + [1-\varepsilon_{\text{rel}}, 1+\varepsilon_{\text{rel}}] \times y + [-\varepsilon_{\text{abs}}, \varepsilon_{\text{abs}}] \\
R^{\sharp}_{\mathbf{f},?}(x \ominus_{\mathbf{f},?} y) &= [1-\varepsilon_{\text{rel}}, 1+\varepsilon_{\text{rel}}] \times x + [-1-\varepsilon_{\text{rel}}, -1+\varepsilon_{\text{rel}}] \times y + [-\varepsilon_{\text{abs}}, \varepsilon_{\text{abs}}] \\
R^{\sharp}_{\mathbf{f},?}(x \otimes_{\mathbf{f},?} y) &= [1-\varepsilon_{\text{rel}}, 1+\varepsilon_{\text{rel}}] \times x \times y + [-\varepsilon_{\text{abs}}, \varepsilon_{\text{abs}}] \\
R^{\sharp}_{\mathbf{f},?}(x \oslash_{\mathbf{f},?} y) &= [1-\varepsilon_{\text{rel}}, 1+\varepsilon_{\text{rel}}] \times x/y + [-\varepsilon_{\text{abs}}, \varepsilon_{\text{abs}}]
\end{aligned}
$$

Specially, for a constant number c that appears in the source code, we use the following abstraction:

$$R^{\sharp}_{\mathbf{f},?}(c) = [R^{\sharp}_{\mathbf{f},-\infty}(c), R^{\sharp}_{\mathbf{f},+\infty}(c)]$$

4.2 Symbolic execution of abstracted floating-point programs

From Section 4.1, we see that floating-point expressions can be soundly abstracted into real number expressions with interval coefficients. Since the multiplication $x \times y$ and division x/y are not linear expressions when both x and y are not constant numbers, in order to obtain linear expressions with interval coefficients, we replace y with its interval range denoted as $[\underline{y}, \overline{y}]$. In symbolic execution, y is always an expression over the symbolic input values. We assume users provide the interval ranges for those symbolic input values. Then, all floating-point expressions can be abstracted as interval linear expressions. Therefore, the resulting path conditions of symbolic execution consist of interval linear constraints while the resulting path functions consist of interval linear expressions.

Finally, the problems of computing the maximum output change and the maximum allowed input perturbation are reduced to solving a series of interval linear programming problems. For example, computing the maximum output change requires the solutions of the following interval linear programming problems:

$$
\begin{aligned}
\max \quad & \Sigma_i [\underline{a}_i, \overline{a}_i] \times x_i + \Sigma_i [\underline{a}'_i, \overline{a}'_i] \times x'_i + b \\
s.t. \quad & (|x_i - x'_i| \le \delta \wedge \bigwedge_{1 \le j \le n, j \ne i} x_j = x'_j) \\
& \wedge \bigwedge_k \Sigma_i [\underline{A}_{ki}, \overline{A}_{ki}] \times x_i + \Sigma_i [\underline{A}'_{ki}, \overline{A}'_{ki}] \times x'_i \le c_k
\end{aligned}
$$

where $\Sigma_i [\underline{a}_i, \overline{a}_i] \times x_i + \Sigma_i [\underline{a}'_i, \overline{a}'_i] \times x'_i + b$ denotes the abstracted output change $PF_y - PF_{y'}$ (or $PF_{y'} - PF_y$) while $\bigwedge_k \Sigma_i [\underline{A}_{ki}, \overline{A}_{ki}] \times x_i + \Sigma_i [\underline{A}'_{ki}, \overline{A}'_{ki}] \times x'_i \le c_k$ denotes the abstracted PC.

Example 5. Consider the self-composed program `piecewise_linear` in Example 3. Suppose we would like to compute the maximum output change, given the input perturbation $\delta = 0.1$ over x. The self-composed program includes four paths overall. Let's consider for example the path that takes the `else` branch in both the unprimed program

P and the primed program P'. Since x, y are of `float` type, $\varepsilon_{\text{rel}} = 2^{-23}$ and $\varepsilon_{\text{abs}} = 2^{-149}$ for the 32-bit single precision floating-point format. We will have

$$\begin{aligned}
PC:&\quad x \geq R^{\sharp}_{\mathbf{f},-\infty}(1.001) \wedge x' \geq R^{\sharp}_{\mathbf{f},-\infty}(1.001)\\
PF_y:&\quad [1-2^{-23}, 1+2^{-23}] \times [R^{\sharp}_{\mathbf{f},-\infty}(1001.0), R^{\sharp}_{\mathbf{f},+\infty}(1001.0)] \times x + [-2^{-149}, 2^{-149}]\\
PF_{y'}:&\quad [1-2^{-23}, 1+2^{-23}] \times [R^{\sharp}_{\mathbf{f},-\infty}(1001.0), R^{\sharp}_{\mathbf{f},+\infty}(1001.0)] \times x' + [-2^{-149}, 2^{-149}]
\end{aligned}$$

Thus, we get the following interval linear programming problem:

$$\begin{aligned}
\max\quad & [1-2^{-23}, 1+2^{-23}] \times [R^{\sharp}_{\mathbf{f},-\infty}(1001.0), R^{\sharp}_{\mathbf{f},+\infty}(1001.0)] \times x\\
& +[-1-2^{-23}, -1+2^{-23}] \times [R^{\sharp}_{\mathbf{f},-\infty}(1001.0), R^{\sharp}_{\mathbf{f},+\infty}(1001.0)] \times x' + [-2^{-148}, 2^{-148}]\\
s.t.\quad & x - x' \leq 0.1 \wedge -x + x' \leq 0.1\\
& \wedge -x \leq -R^{\sharp}_{\mathbf{f},-\infty}(1.001) \wedge -x' \leq -R^{\sharp}_{\mathbf{f},-\infty}(1.001)
\end{aligned}$$

Solving the above interval linear programming problem by the method described in Section 2.2 will give us 100.10023889571252. After we deal with all other paths in the same way, we will find that 100.10023889571252 is the maximum output change with respect to the given input perturbation 0.1. Hence, the program `piecewise_linear` in floating-point arithmetic is at least (0.1, 100.10023889571252)-*robust*.

5 Implementation and experimental results

We have implemented a robustness analysis tool RAFP, based on the symbolic execution and floating-point abstraction techniques presented in Section. 4. Given an input perturbation δ over one input variable of the program, RAFP can compute the maximum output change. Furthermore, if the user also provides a candidate output change ϵ and would like to check whether the program is (δ, ϵ)-*robust*, RAFP will check this property during the process of computing maximum output change and will stop once one path violating the property is found. Also, given an output change ϵ, RAFP can compute the maximum allowed input perturbation for floating-point programs. RAFP is built on top of Symbolic PathFinder (SPF) [38] which is a symbolic execution engine for Java programs. We use SPF to extract the path conditions together with the associated path functions. For linear programming, RAFP makes use of the Java Binding for GLPK (GNU Linear programming kit) called GLPK-Java [41].

To conduct experiments on checking robustness properties of floating-point programs via software model checking, we choose CBMC (C Bounded Model Checker) [18] which implements bounded model checking for ANSI-C programs using SAT/SMT solvers. CBMC utilizes a bit-precise modeling for floating-point operations and employs a sound and complete decision procedure for floating-point arithmetic. CBMC provides an option `--floatbv` to use IEEE floating point arithmetic and options for choosing rounding modes. However, CBMC does not support to use different rounding modes for the floating-point operations in the same program. In other words, all floating-point operations in a program are of the same rounding mode during the analysis. We use the default rounding mode `--round-to-nearest` during our experiments. Moreover, CBMC provides `__CPROVER_assume()` and `__CPROVER_assert()`

statements, which are needed for robustness analysis of self-composed programs. Both statements take Boolean conditions. The `__CPROVER_assume()` statement restricts that the program traces should satisfy the assumed condition. For the `__CPROVER_assert()` statement, CBMC will check whether the asserted condition holds true for all runs of the program.

Table 1. Experimental results for benchmark examples

program	δ_{in}	RAFP		CBMC			
		$\epsilon_{\max}$	$t(ms)$	ϵ_{unr}	$t(ms)$	ϵ_r	$t(ms)$
piecewise_linear	0	2.3889571220452006e-4	29	⋆	⋆	⋆	⋆
	0.01	10.010238895712442	33	?	>1h	?	>1h
	0.1	100.10023889571252	34	?	>1h	?	>1h
Max1	0	1.4012987984203268e-45	44	⋆	⋆	⋆	⋆
	0.01	0.010000000000000007	55	0.0099	215	0.01	16066
	0.1	0.10000000000000006	54	0.099	227	0.1	16262
MorePaths	0	1.0000000027939686	59	⋆	⋆	⋆	⋆
	0.01	1.0000000027939686	79	0.99	279	1.0	379
	0.1	1.0000000027939686	86	0.99	314	1.0	471
Orientation	0	2.0	55	⋆	⋆	⋆	⋆
	0.1e-8*$\mathcal{E}$	2.0	68	0	1840	1.0	6845
	0.1e-3*$\mathcal{E}$	2.0	73	0	1338	1.0	13327
	0.1e-2*$\mathcal{E}$	2.0	80	1.0	14165	2.0	413
Filtered_Orientation	0	1.0	54	⋆	⋆	⋆	⋆
	0.1e-8*$\mathcal{E}$	1.0	70	0	1044	1.0	29641
	0.1e-2*$\mathcal{E}$	1.0	73	0	898	1.0	30261
	0.1*$\mathcal{E}$	2.0	75	0	719	1.0	26463
	$\mathcal{E}$	2.0	68	1.0	13206	2.0	654

We have conducted experiments on a selection of benchmark examples using both RAFP and CBMC. Table 1 shows the comparison of performance and the resulting output changes. The column"δ_{in}" shows the considered input perturbation over one input variable of the program. The column "$\epsilon_{\max}$" shows the resulting maximum output change computed by RAFP with respect to the given input perturbation. The column "ϵ_{unr}" gives the largest possible output change that we have tried with CBMC such that the program is not $(\delta_{in}, \epsilon_{unr})$-*robust* with respect to the given input perturbation. The column "ϵ_r" gives the smallest output change that we have tried with CBMC such that the program is $(\delta_{in}, \epsilon_r)$-*robust* with respect to the given input perturbation [1]. Since CBMC uses the same rounding mode for all floating-point operations in the same program during the analysis, the output change is always 0 when the given input perturbation is 0. Hence, for those rows that specify input perturbation as 0, we do not need to run

[1] Note that CBMC can be used only to check whether a program is (δ, ϵ)-*robust* and can not be used to compute the amount of output change with respect to the given input perturbation. During our experiments, we try CBMC with different candidate values of ϵ to find ϵ_{unr} and ϵ_r.

CBMC and thus we mark the table entry with $\star$ in this case. Our tool RAFP utilizes rounding mode insensitive floating-point abstraction and thus in principle it holds that $\epsilon_{\max} \geq \epsilon_r \geq \epsilon_{unr}$, which is confirmed by the experimental results.

The program `piecewise_linear` corresponds to the program shown in Example 2. `Max1`, `MorePaths` come from JPF Continuity [8]. `Max1` is a floating-point program that implements $\max(x, y)$, and thus it is $(\delta_{in}, \delta_{in})$-*robust*. `MorePaths` is a floating-point program that involves both a step function and a max function, and thus it is $(\delta_{in}, 1.0)$-*robust* for all $\delta_{in} \leq 1.0$. `Orientation` (which corresponds to the program shown in Fig. 1) together with `Filtered_Orientation` are extracted from the computational geometry algorithms library CGAL [21] and address robust geometric computation. `Filtered_Orientation` is an improved version of `Orientation` via static filter technique. The approximate result of computing the sign of a determinant is compared with a given positive filter bound $\mathcal{E}$ (rather than compared with zero). When the approximate result is in the interval $[-\mathcal{E}, \mathcal{E}]$, `Filtered_Orientation` gives 0. During our experiments, we set $\mathcal{E}$=1.5e-5 (and for the sake of comparison, we express the input perturbation in terms of $\mathcal{E}$ also for `Orientation` although here $\mathcal{E}$ does not appear). The outputs of `Orientation` and `Filtered_Orientation` are always -1 (negative), 0 (zero), or 1 (positive). Hence, in Table 1, the resulting output changes for these two programs are always 0, 1.0, or 2.0. From Table 1, we could find that `Filtered_Orientation` is more robust than `Orientation`. E.g., given the input perturbation δ=0.1e-2*$\mathcal{E}$, CBMC finds that for ϵ=1.0, `Filtered_Orientation` is robust while `Orientation` is not. Similarly, given the input perturbation δ=0.1e-2*$\mathcal{E}$, RAFP gives $\epsilon_{\max} = 1.0$ for `Filtered_Orientation` but gives $\epsilon_{\max} = 2.0$ for `Orientation`.

The column "t(ms)" presents the analysis times in milliseconds when the analyzers run on a 2.5GHz PC with 4GB of RAM running Windows 7. [2] From Table 1, we could see that RAFP outperforms CBMC in time efficiency. Especially for `piecewise_linear`, CBMC could not even finish the analysis process in 1 hour. The low efficiency of CBMC is because that CBMC uses a sound and complete decision procedure for floating-point arithmetic. Especially, the multiplication and division floating-point operations may generate formulae that are expensive to decide and quite hard for SAT solvers to solve [33]. Hence, checking robustness properties of floating-point programs via CBMC may have limitations in scalability due to the current expensive decision procedures for floating-point logic. During our experiments, the approach via symbolic execution of abstracted floating-point programs is much more efficient. In principle, symbolic execution may suffer from the path explosion problem. However, the recent success of symbolic execution tools such as KLEE [9] on analyzing large-scale programs [10], makes this approach promising.

6 Related work

Robustness analysis of programs. Robustness is a standard correctness property for control systems [39]. Robustness analysis of programs has received increasing attention in the recent years. Majumdar and Saha [34] took a first step toward analyzing the

[2] RAFP runs further on a Java Virtual Machine (JVM) while CBMC runs further on a virtual machine VMWare running Fedora 12.

robustness of programs in control systems. They also utilized symbolic execution and optimization techniques to compute the maximum difference in program outputs with respect to the given input perturbation. However, they assumed exact real arithmetic in the program. Continuity as one aspect of robustness for software was firstly considered in [27]. Recently, Chaudhuri et al. presented logic-based mostly-automated methods to determine whether a program is continuous [12] or Lipschitz continuous [13, 14], and more recently to determine whether a decision-making program is consistent under uncertainty [11]. Quite recently, Shahrokni and Feldt [42] conducted a systematic review of software robustness. However, most existing work on robustness analysis does not handle floating-point arithmetic in the program. Bushnell [8] presented a symbolic execution based approach to identify continuties and discontinuties associated with path condition boundaries for floating-point software, but it did not consider the true floating-point semantics. Besides, Gazeau et al. [22] presented a non-local method for proving the robustness of floating-point programs, but which needs much manual work. Recently, Goubault and Putot [25] proposed an abstract interpretation based robustness analysis method for finite precision implementations.

Safety analysis of floating point programs. Monniaux [37] described common pitfalls in analyzing and verifying floating-point programs. Abstract interpretation [19] based techniques have shown quite successful on analysis of floating-point programs. In [23], Goubault analyzed the origin of the loss of precision in floating-point programs based on abstract interpretation. Following this direction, a static analyzer FLUCTUAT [24] was developed. The abstract interpretation based static analyzer ASTRÉE [5] checks for floating-point run-time errors based on the computed set of reachable values for floating-point variables. As in ASTRÉE, we rely on the floating-point abstraction technique of [35] to soundly abstract floating-point expressions into ones over the field of reals. Chen et al. [16, 17] utilized interval linear constraints to design numerical abstract domains and to construct sound floating-point implementations [15]. Ivančić et al. [28] used bounded model checking based on SMT solvers to detect numerical instabilities in floating-point programs, based on a mixed integer-real model for floating-point variables and operations. Brain et al. [6] recently improved the bit-precise decision procedure for the theory of floating-point arithmetic based on a strict lifting of the conflict-driven clause learning algorithm in modern SAT solvers to abstract domains. Barr et al. [2] presented a method to automatically detect the floating-point exception through symbolic execution.

Self-composition. The idea of self-composition is firstly used in the field of secure information flow [4, 20], to characterize non-interference. Terauchi and Aiken [44] proposed the type-directed transformation approach to make self-composition work in practice with off-the-shelf automatic safety analysis tools. Recently, Barthe et al. [3] proposed a general notion of product program that is beneficial to relational verification, which could be considered as the generalization of self-composition. Kovács et al. [32] presented a general method to analyze 2-hypersafety properties by applying abstract interpretation on the self-compositions of the control flow graphs of programs.

7 Conclusion

We have proposed a self-composition based approach for robustness analysis of programs, which enables making use of off-the-shelf automatic reachability analysis tools to analyze robustness properties of programs. Then, we have shown how to use software model checking and symbolic execution techniques on self-composed programs to analyze program robustness properties. In particular, we have considered the robustness analysis problem of floating-point programs. To deal with floating-point arithmetic during symbolic execution, we have utilized a rounding mode insensitive floating-point abstraction to abstract floating-point expressions into interval linear expressions in exact real arithmetic. On this basis, the maximum output change (when given the input perturbation) or maximum allowed input perturbation (when given the input perturbation) are computed based on symbolic execution and interval linear programming for abstracted floating-point programs. Experimental results of our prototype implementation are encouraging.

It remains for future work to exploit the intrinsic symmetry of self-composed programs to reduce the number of considered paths during robustness analysis. We also plan to improve the prototype implementation and to conduct more experiments on larger realistic floating-point programs.

Acknowledgements. This work is supported by the 973 Program under Grant No. 2014CB340703, the 863 Program under Grant No. 2011AA010106, the NSFC under Grant Nos. 61202120, 61120106006, 91318301, and the SRFDP under Grant No. 20124307120034.

Conflict of Interests. The author(s) declare(s) that there is no conflict of interests regarding the publication of this article.

References

1. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3. ACM Press, 2002.
2. E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *POPL*, pages 549–560. ACM, 2013.
3. G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *FM*, pages 200–214. Springer, 2011.
4. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSFW*, pages 100–114. IEEE, 2004.
5. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207. ACM Press, 2003.
6. M. Brain, V. D'Silva, A. Griggio, L. Haller, and D. Kroening. Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in System Design*, 2013. (online first).
7. A. Brillout, D. Kroening, and T. Wahl. Mixed abstractions for floating-point arithmetic. In *FMCAD*, pages 69–76. IEEE, 2009.
8. D. Bushnell. Continuity analysis for floating point software. In *the 4th workshop on Numerical Software Verification (NSV'11)*, 2011.

9. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX Association, 2008.
10. C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.
11. S. Chaudhuri, A. Farzan, and Z. Kincaid. Consistency analysis of decision-making programs. In *POPL*, pages 555–568. Springer, 2014.
12. S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *POPL*, pages 57–70. ACM, 2010.
13. S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity and robustness of programs. *Commun. ACM*, 55(8):107–115, 2012.
14. S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. NavidPour. Proving programs robust. In *FSE*, pages 102–112. ACM, 2011.
15. L. Chen, A. Miné, and P. Cousot. A sound floating-point polyhedra abstract domain. In *APLAS*, volume 5356 of *LNCS*, pages 3–18. Springer Verlag, 2008.
16. L. Chen, A. Miné, J. Wang, and P. Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. In *SAS*, volume 5673 of *LNCS*, pages 309–325. Springer Verlag, 2009.
17. L. Chen, A. Miné, J. Wang, and P. Cousot. An abstract domain to discover interval linear equalities. In *VMCAI*, volume 5944 of *LNCS*, pages 112–128. Springer, 2010.
18. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.
19. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.
20. Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *SPC*, volume 3450 of *LNCS*, pages 193–209. Springer, 2005. Preliminary version in the informal proceedings of WITS'03.
21. E. Fogel and M. Teillaud. The computational geometry algorithms library cgal. *ACM Communications in Computer Algebra*, 47(3):85–87, 2013.
22. I. Gazeau, D. Miller, and C. Palamidessi. A non-local method for robustness analysis of floating point programs. In *QAPL*, volume 85 of *EPTCS*, pages 63–76, 2012.
23. E. Goubault. Static analyses of the precision of floating-point operations. In *SAS*, volume 2126 of *LNCS*, pages 234–259. Springer, 2001.
24. E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations: A simple abstract interpreter. In *ESOP*, volume 2305 of *LNCS*, pages 209–212. Springer, 2002.
25. E. Goubault and S. Putot. Robustness analysis of finite precision implementations. In *APLAS*, volume 8301 of *LNCS*, pages 50–57. Springer, 2013.
26. L. Haller, A. Griggio, M. Brain, and D. Kroening. Deciding floating-point logic with systematic abstraction. In *FMCAD*, pages 131–140. IEEE, 2012.
27. D. Hamlet. Continuity in software systems. In *ISSTA*, pages 196–200. ACM, 2002.
28. F. Ivancic, M. K. Ganai, S. Sankaranarayanan, and A. Gupta. Numerical stability analysis of floating-point computations using software model checking. In *MEMOCODE*, pages 49–58. IEEE, 2010.
29. R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.
30. L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C.-K. Yap. Classroom examples of robustness problems in geometric computations. In *ESA*, volume 3221 of *LNCS*, pages 702–713. Springer, 2004.
31. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

32. M. Kovács, H. Seidl, and B. Finkbeiner. Relational abstract interpretation for the verification of 2-hypersafety properties. In *CCS*, pages 211–222. ACM, 2013.
33. D. Kroening. The CPROVER User Manual. Available online at http://www.cprover.org/cbmc/doc/manual.pdf.
34. R. Majumdar and I. Saha. Symbolic robustness analysis. In *RTSS*, pages 355–363. IEEE, 2009.
35. A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004.
36. A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, December 2004.
37. D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 2008.
38. C. S. Pasareanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013.
39. S. Pettersson and B. Lennartson. Stability and robustness for hybrid systems. In *Proceedings of 35th IEEE Conference on Decision and Control*, pages 1202–1207, 1996.
40. J. Rohn. Solvability of systems of interval linear equations and inequalities. In *Linear Optimization Problems with Inexact Data*, pages 35–77. Springer, 2006.
41. H. Schuchardt. GLPK for Java. http://glpk-java.sourceforge.net/ (2014).
42. A. Shahrokni and R. Feldt. A systematic review of software robustness. *Information & Software Technology*, pages 1–17, 2013.
43. IEEE Computer Society. Ieee standard for binary floating point arithmetic. Technical report, ANSI/IEEE Std 745-1985, 1985.
44. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.