A Genetic Algorithm for Detecting Significant Floating-Point Inaccuracies

Daming Zou*†, Ran Wang*†, Yingfei Xiong*†, Lu Zhang*†, Zhendong Su‡, Hong Mei*†
*Key Laboratory of High Confidence Software Technologies (Peking University), MoE, China
†Institute of Software, School of EECS, Peking University, China
‡Department of Computer Science, University of California, Davis, USA
{zoudm, lilianwangran, xiongyf, zhanglucs, meih}@pku.edu.cn, su@ucdavis.edu

***Abstract*—It is well-known that using floating-point numbers may inevitably result in inaccurate results and sometimes even cause serious software failures. Safety-critical software often has strict requirements on the upper bound of inaccuracy, and a crucial task in testing is to check whether significant inaccuracies may be produced.**

The main existing approach to the floating-point inaccuracy problem is error analysis, which produces an upper bound of inaccuracies that may occur. However, a high upper bound does not guarantee the existence of inaccuracy defects, nor does it give developers any concrete test inputs for debugging.

In this paper, we propose the first *metaheuristic search-based approach to automatically generating test inputs* that aim to trigger significant inaccuracies in floating-point programs. Our approach is based on the following two insights: (1) with FPDebug, a recently proposed dynamic analysis approach, we can build a reliable fitness function to guide the search; (2) two main factors — the scales of exponents and the bit formations of significands — may have significant impact on the accuracy of the output, but in largely different ways. We have implemented and evaluated our approach over 154 real-world floating-point functions. The results show that our approach can detect significant inaccuracies in the subjects.

I. Introduction

Inaccuracy caused by floating-point numbers is a well-known problem in software development. In critical software systems, disastrous results may be caused by floating-point inaccuracy. An example well cited in the literature is the failure of a Patriot missile to intercept an incoming missile in the first Persian Gulf War, due to the accumulated floating-point errors during the continuous tracking and guidance. This failure caused the loss of 28 lives and around 100 injuries.

Floating-point numbers are usually less inaccurate because they use finite number of digits to represent a real number. For example, when we represent 0.1 as a (single precision) floating-point number, because 0.1 cannot be represented in finite digits in a binary fraction, the value is in fact 0.100000001490116119384765625. One consequence of the inaccuracy in representation is the rounding error, e.g., when adding a small number to a large number, significant digits in the small number may be rounded off due to the limited digits to represent the final result. Accumulated rounding errors during the computation may result in significant inaccuracy in the output.

Because of the importance of ensuring accuracy in floating-point calculation, several approaches [1]–[3] have been proposed to detect floating-point inaccuracies. Most approaches use static analysis based on interval arithmetic [4] or affine arithmetic [5], trying to determine the possible range of errors in the result of the computation. However, we observe several limitations in reporting the ranges of errors:

- Due to the limitations of static analysis, the computed range is often an over-approximation of the actual error, and the difference is often very large. Even in the latest approach [1], the computed upper bound is usually several orders of magnitude larger than the actual error, and is sometimes infinite. As a result, even if a large upper bound is reported, it is still unknown whether or not a problem in accuracy exists.
- To debug an inaccuracy problem, it would be more convenient to have an input that triggers the problem, so that developers can follow the execution to discover the root cause of the problem. However, as reported by Bao and Zhang [6], there may be only a small portion of inputs among all possible inputs causing significant inaccuracies in the output. Thus, it would be quite difficult for developers to obtain such an input manually.

To overcome these problems, this paper proposes a metaheuristic search-based approach that aims to generate a test input for a program to maximize the error of the output. To the best of our knowledge, our approach is the first metaheuristic test generation approach aiming to detect floating-point inaccuracies. Existing test generation approaches [7], [8] for floating-point programs focus on maximizing path coverage rather than output errors. Our approach is based on the following two insights. First, with FPDebug, an approach recently proposed by Benz et al. [9], we can obtain the likely error occurred in the output of a particular concrete execution. Thus, we can build a fitness function around this error to guide the search. Second, there are two main factors of the input

We sincerely acknowledge Dr. Hao Zhong at Shanghai Jiaotong University for his useful and helpful feedback on an early draft of this paper, and Xinrui He at Peking University for her help on setting up the experiments.

The authors from Peking University are supported by the National Basic Research Program of China under Grant No. 2014CB347701, and the National Natural Science Foundation of China under Grant No. 61202071, 61225007, 61421091, 61332010. Zhendong Su is partially supported by United States NSF Grants 1117603, 1319187, and 1349528.

Yingfei Xiong is the corresponding author.

TABLE I: IEEE 754 floating-point representation

	Sign	Exponent	Significand
Single Precision	1	8	23
Double Precision	1	11	52

floating-point numbers that may affect the error of the output: the scales of exponent and the bit formation of significand, but their relations to accuracy exhibit different characteristics. These factors could be exploited for designing efficient search algorithms.

In summary, we make the following major contributions:

- We perform an empirical analysis to uncover the relations between different factors and the accuracy of output. The results suggest that both the scales of the exponents and the bit formation of significands may substantially affect accuracy. While only exponents in a small interval lead to significant inaccuracies, a large portion of significands may lead to significant inaccuracies.
- We design a novel genetic algorithm, locality-sensitive genetic algorithm (LSGA), based on the results of the empirical analysis. Our basic idea is to evolve the exponent to hit the small interval, while randomly generate the significand as the large portion of bit formation is easy to hit. The fitness function is built upon the output of FPDebug [9].
- We evaluate our approach by (1) a sanity check on six classic examples [10] of stable and unstable algorithms, and (2) a series of experiments on 154 floating-point functions selected from the latest version of the GNU Scientific Library. Our experiments compare three search algorithms, including our own, a standard genetic algorithm, and a random search algorithm. Our results reveal that (1) our algorithm exhibits absolute superiority over the other two algorithms, and (2) our approach is able to find extremely large inaccuracies in widely-used real world scientific functions.

The rest of the paper is organized as follows. Section II introduces some background knowledge, while Section III further motivates our research. Section IV presents the empirical analysis and our algorithm design. Section V reports the two evaluations. Section VI discusses the main limitations and possible future research. Section VII discusses related work, and Section VIII concludes.

II. Background

A. Format of Floating-Point Numbers

According to IEEE 754 standard [11], a floating-point number contains three parts: sign, exponent, and significand. Table I depicts the numbers of bits of the three parts in either a single precision number or a double precision number.

Let us denote the sign as s, the value of the exponent as e, and the value of the significand as f. If all bits of the exponent are 1, this floating number is one of the special values: ∞, $-\infty$ or NaN, where NaN indicates errors in computation such as division by zero. Otherwise, the value of a floating-point number is depicted by Formula 1:

$$(-1)^s \times f \times 2^e \tag{1}$$

Suppose that the significand is in the form of $b_0 b_1 \ldots b_{n-1}$, the value of the significand, f, is defined by Formula 2:

$$f = \begin{cases} \sum_{i=0}^{n-1} \frac{b_i}{2^{i+1}}, & \text{if all bits of } e \text{ are } 0 \\ 1 + \sum_{i=0}^{n-1} \frac{b_i}{2^{i+1}}, & \text{otherwise} \end{cases} \tag{2}$$

Suppose that the exponent is in the form of $b_0 b_1 \ldots b_{n-1}$, the value of the exponent, e, is defined by Formula 3:

$$e = \sum_{i=0}^{n-1} 2^i b_i - 2^{(n-1)} + 1 \tag{3}$$

B. Genetic Algorithm

A Genetic Algorithm (GA) [12] is a metaheuristic search technique that simulates the process of natural selection for solving optimization problems. In a GA, each candidate solution is called an individual, and has a set of properties that can be represented in a binary form. There is also a fitness function f to evaluate how close an individual is to an optimal solution.

The process of GA typically starts from a set of individuals, randomly selected or pre-defined, to form the first generation. The population size is problem-dependent. When to create the next generation, all the individuals in the current generation are put into a selection pool, in which each individual has a probability, which is dependent on the fitness of the individual, to be a parent for generating individuals in the next generation. To create an individual in the next generation, a pair of parent solutions are selected from the selection pool. Then, two types of operations (i.e., crossover and mutation) are used to create a child of the two parents. The generated children, optionally plus the individuals from the previous generation, form the next generation. The selection-creation loop is repeated until reaching a termination condition, such as finding a good enough solution, reaching the maximum number of generations, and reaching the time limit.

There are several key components in designing a genetic algorithm, such as the selection method (i.e., how to select individuals for reproduction), crossover operator (i.e., how to produce a child from two parents), mutation operator (i.e., how to mutate a child), and initial population (i.e., how the first generation is populated).

III. Motivating Example

Example. To motivate our research, let us consider function $F(x)$ defined in the following code snippet. $F(x)$ is a carefully constructed example to demonstrate the problem of floating-point inaccuracy, containing several major operations on floating-point numbers (addition, subtraction, division) and a common code pattern (adding up many numbers in a loop). $F(x)$ should always returns a constant number in real arithmetic.

```
float F(float x)
```

```
{
1:  int i,n=8192;
2:  float y,z;
3:  y=z=n;
4:  if (x<0) x=-x;
5:  for (i=0;i<n;i++)
6:    y=y+x;
7:  y=y/z;
8:  return (0.125f+x)/(y-0.875f);
}
```

As the loop on Lines 5 and 6 will be executed n times, the value of y before executing Line 7 should be equal to $n*(1+|x|)$. To make the analysis easy, we set n to $8192 = 2^{13}$. As a result, the value of y after executing Line 7 should be equal to $1 + |x|$. Therefore, $F(x)$ should always return 1.

However, due to the accumulation of rounding errors in the loop, the value of y after executing Line 7 cannot be exactly $1 + |x|$. This error will be further magnified on Line 8. As a result, $F(x)$ may produce significant inaccuracy for some input x. The most inaccurate output is 1.0039062.

Range of problematic inputs. The range of x for $F(x)$ to produce significant inaccuracies is not large. The reason is that, when the value of $|x|$ becomes larger (e.g., 10 times larger than 0.125), the value of $F(x)$ will be mainly decided by $|x|$, not the accumulated rounding error. In fact, all the cases of inaccuracy over 0.001 occur when x is between -0.4 and 0.4. Note that, when the value of x is very close to 0, $F(x)$ cannot produce very large inaccuracy. The reason is that a very small $|x|$ (e.g., smaller than 0.0001) would not produce a large enough rounding error when executing the loop only 8192 times.

Considering that the number of possible values of x is huge when testing $F(x)$, generating tests to trigger significant inaccuracies of $F(x)$ should be difficult. In fact, if we want to trigger an inaccuracy that is close to the largest inaccuracy 0.0039062, e.g., the absolute error is larger than 0.0039, the range of $|x|$ is much smaller (i.e., between 0.0004874 and 0.0004883). Note that existing coverage-based test generation cannot help here, since any value of x can achieve 100% branch coverage.

Factors affecting accuracy. This example demonstrates that there are two distinctive factors that may affect the error of the output, and the impact of one factor may be very different from the other. The first factor is the scale of the exponent in the input. Since the scale of a floating number is mainly determined by its exponent, only when the exponent falls into a small range can the input trigger a large error. The second factor affecting the accuracy of the output is the bit formation of the significand. Due to the "round to even" policy adopted by floating-point numbers, large inaccuracies appear only when the accumulated rounding errors do not cancel each other. This requires that the formation of bits in the significand exhibits certain patterns.

IV. Approach

A. Problem Definition

Given a program using floating-point numbers (denoted as P) with M input parameters (denoted as I_1, I_2,...,I_M), we deem the problem of detecting floating-point inaccuracy in P as a search problem. The search space of this search problem is the space represented by all the possible combinations of the values of the M input parameters. The aim of the search problem is to find a particular input (i.e., a combination of the values of the M input parameters) that maximize the error of the output.

To solve this search problem, we need a criterion to determine whether one input would lead to a more inaccurate output than another input. Search algorithms will use this criterion to guide the searching process. Unlike coverage, which is widely used as the criterion in search-based test generation, it is not straightforward to set up the criterion needed in our approach. Fortunately, Benz et al. [9] recently proposed a dynamic analysis technique which dynamically increase the precision of the floating-point numbers to calculate both the likely absolute error and the likely relative error of the output of a program for an arbitrary input.[1] Following the common practice [16], we use relative error to measure the inaccuracy of the output. Let us denote the real result as R and the actual output as A. The relative error is defined as $|R - A|/|R|$.

With this basic framework, different metaheuristic search techniques can be adopted for detecting significant inaccuracies. However, it is not easy to discover significant inaccuracies through searching. First, as our motivating example and an existing study [6] have shown, often only a very small portion among all possible floating-point numbers may lead to serious inaccuracies. When there is more than one input parameter for a floating-point function, the probability of hitting a large error becomes very low. Second, FPDebug has a slow down of several hundreds of times [9], such that we could test only a relatively small number of input values during the search process. As a result, it is critical to design an effective search algorithm that hits large inaccuracies quickly. To design such an algorithm, we perform a small empirical analysis to understand the relation between the accuracy of the output and different factors.

B. Empirical Analysis

In Section III we have seen that there are two main factors that may affect the accuracy of the output, each corresponding to one of the three main components of a floating-point numbers. These two factors can be exploited to design an effective search algorithm. To further understand the relation between the two factors and the accuracy of the output, we performed a small empirical analysis.

Analysis setup. In our analysis, we randomly chose four functions from the GNU Scientific Library (GSL). More description of GSL can be found in Section V. The four functions are bessel_K0, Ci, erf, and legendre_Q1. Function bessel_K0 computes the cylindrical Besssel function, Ci computes the Cosine integral, erf computes the Gauss error function, and legendre_Q1 computes the Legendre function. We deliberately

[1] These errors are likely errors because increasing the precision does not guarantee more accurate result in all cases [13]–[15].

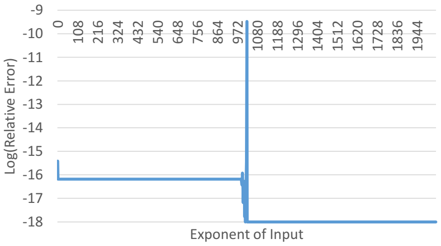

Fig. 1: erf at significand 0x34873b27b23c6

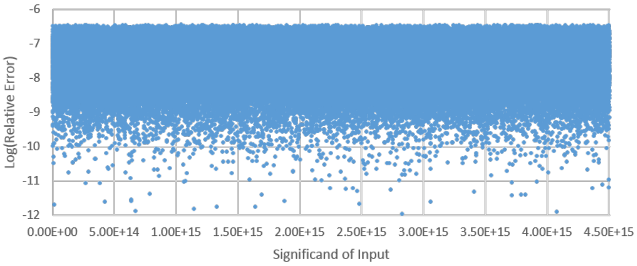

Fig. 2: erf at exponent 1023

chose functions with one floating-point input parameter to simplify our analysis.

We invoked these programs in FPDebug with different inputs, and monitored how the relative errors of output change based on the change of input. To better understand the two factors individually, we fixed one factor and changed the other. First we fixed significand and changed exponent. For each program, we randomly generated three significands. For each significand, we combined it with every possible exponent allowed by the double precision format to form an input to a subject program, and then we executed the program in FPDebug to get the error for each input. Second we fixed exponent and changed significand. We randomly generated two exponents for each program. However, because of the slow down from FPDebug, it is not possible to test the whole space of significand in double precision. As a result, we generated 100,000 random significands and tested each pair of significand and exponent. In all tests, we set the sign bit to 0.

Results. Fig. 1 shows different relative errors at different exponents for function erf when the significand is fixed at 0x34873b27b23c6. Fig. 2 shows different relative errors at different significands for erf when the exponent is fixed at 1023. Note that in the X axes of both figures, we interpret both the exponents and significands as unsigned integers, and report their integer values. We shall use the two figures as examples to illustrate our results, and the experiments on all functions exhibit very similar patterns.

First, we observe that both the exponents and the significands have a great impact on the relative errors. In all functions, by changing the significands we could achieve a difference of 5∼9 orders of magnitude, and by changing the exponents we could achieve a difference of 4∼18 orders of magnitude.

Second, we observe that the exponents that invoke large relative errors stay in a small interval of the axis, while the significands that invoke large errors distribute evenly on the axis. This result confirms our analysis about the two factors in Section III. How exponents affect accuracy is decided by their scales, while how significands affect accuracy is decided by their bit formations.

Third, as we can see from the figures, the portion of exponents that invoke large relative errors is usually very small. In the worst case, there is no more than 0.1% of the exponents invoking errors that are within two orders of magnitude difference of the largest error. On the other hand, the portion of significands that invoke large relative errors is usually large. Even in the worst case, we still have more than 28% of the significands invoking errors that are within two orders of magnitude difference of the largest error. Note that this result is consistent with the existing study [6]: Although a large portion of significands may invoke large errors, the probability of a random input invoking a large error is still very small, as both the significand and the exponent need to invoke large errors.

Fourth, there usually exist exponents that are near the interval of large errors and lead to errors higher than the average. As shown in Fig. 1, there is small burst of error near the exponent of 1000, before the large burst near 1023. Though the errors from the small burst are still much smaller than the largest error, they are higher than most other errors.

Fifth, we observe that the small interval of exponents leading to large errors is likely to be near the value 1023, which is just the median of all double-precision exponents. This indicates that the exponents around the median may have higher probability to invoke large relative errors.

The third observation suggests that the key to design an effective search algorithm is to hit the small interval of the exponents that lead to large errors, which is much more difficult than hitting a significand that lead to a large error. The fourth and fifth observations indicate possible strategies towards this problem. The next section explains how we design our algorithm based on these observations.

C. Our Genetic Algorithm

As revealed by the fourth observation, exponents near the small interval of large relative errors may also invoke a relatively high error. Though it is difficult to hit the small interval of large errors, it is much easier to hit an exponent near it, and gradually evolve the exponent to hit the small interval. Based on the fifth observation, exponents invoking large errors often appear near the median of all possible exponents, so searching around the median might be more effective than searching in other places.

Based on these ideas, we design a genetic algorithm, named *locality-sensitive genetic algorithm* (LSGA). A high-level outline of our algorithm is shown in Algorithm 1. Let us first consider programs with one floating-point input parameter. Our genetic algorithm first randomly generates a set of exponents (line 1). This generation process tries to make the generated exponents evenly distributed in the space of all exponents,

Algorithm 1 Outline of LSGA

```
1: population ← generateInitialPopulation()
2: for i ← N do
3:     input ← select(population)
4:     input ← mutate(input)
5:     population.add(input)
6: return maxError(population)
```

but favors exponents around the median. These exponents are combined with randomly generated significands and signs to form the initial population. During every iteration, we pick an individual with a high relative error (line 3), mutate it (line 4) and put it back to the population (line 5) without removing the original one from the population. The mutation process adds a random number to the exponent, regenerates its significand randomly, and flips its sign bit with a probability. This process repeats until a predefined number of iterations have been reached, and the result with the highest relative error is returned (line 6). Programs with multiple floating-point input parameters are processed in a similar way, but each time we deal with a set of floating-point numbers rather than one number.

The algorithm always randomly generates the significand because, as revealed by our empirical analysis, a randomly generated significand already has a high probability to invoke a large error. To avoid further complication for our algorithm, we use random search for the significands. We regenerate the significands at every mutation to increase the diversity of the population. Our algorithm also drops the cross-over operation in the standard genetic algorithm, because we do not find sensible operation to combine two exponents based on their scales. As a matter of fact, this coincides with the design of many existing genetic algorithms, which favor mutation over cross-over [17].

We can see that there are three main components of the algorithm: generate an initial population, select, and mutate. In the following we explain each in detail.

Initial Population. We generate n exponents as the initial population based on a uniform distribution, but the exponents in the interval $[median - t, median + t]$ has a probability five times as high as other exponents. Number $median$ is the median value of all possible exponents, being 1023 for double-precision and 127 for single-precision. Number t is equal to $2^{\lceil k/2 \rceil}$ in our current implementation, where k is the number of digits in the exponent part.

To implement this distribution, we separate the space of exponents into n intervals, where the lengths of the intervals within $[median - t, median + t]$ are five times as small as those outside $[median - t, median + t]$. For programs with a single floating-point input parameter, we randomly generate an exponent within each interval, and get n inputs. For programs with multiple floating-point input parameters, we randomly pick an interval and generate an exponent within the interval for each parameter. We repeat the generation n times for n sets of parameters.

For each generated exponent, we randomly generate the sign bit and the significand to form a floating-point number. Then we run the program in FPDebug with each set of input parameters to get the relative error of its output.

Selection Method. In the selection step, we would like to favor the inputs that lead to a larger error. A standard selection method for this purpose is roulette wheel selection [18], where the probability of selecting individual i is $f_i / \sum_{j=1}^{n} f_j$, where f_i is the fitness (in our case, relative error) of individual i and n is the total number of individuals. However, this method is not suitable for our case because the probability of hitting a large error is small. If we have not hit a large error, it is likely that in our population there is only a few number of individuals whose errors are slightly larger than the others. Roulette wheel selection cannot select those slightly better individuals because their number is too small. On the other hand, if we really encounter an individual who has several orders of magnitude larger error than other individuals, it is unlikely we will choose any other individual.

To overcome this problem, we choose the group-based rank selection, which is more suitable to our case based on existing studies [18]. We first group the population by their relative errors. Every two individuals in a group have a difference of no more than two orders of magnitude in relative error. Then we select a group by their rank. We first select the group with highest relative errors with a probability of p. If the first group is not selected, we select the group with the second highest relative error with probability of p, and etc. After we have selected a group, we pick a random individual from the group. Based on our experience with a few small programs, we set $p = 0.6$.

Mutation Operation. As mentioned before, given a floating-point number, our mutation operation adds a value v (can be positive or negative) to its exponent, flip its sign bit with a probability q, and randomly regenerate its significand. After mutation, we run the program with the mutated input in FPDebug to get the relative error on the output.

The value v is decided by a normal distribution $\mathcal{N}(0, \sigma^2)$, where σ^2 is the length of the interval in the initial population which this exponent falls in. This setting ensures that the change to the exponent is always small, and is even smaller around the median because the initial population is already condensed around the median.

To ensure the sign bit is not frequently changed, we set q as 0.1 in our current implementation.

Number of Iterations. Since the maximum population size in our algorithm is confined by the number of times that FPDebug can be executed within a time frame, it is important to properly balance between the number of initial population (denoted as n) and the number of mutations (denoted as m). We currently set $n = m$ in our implementation. This setting is different from common genetic algorithms where $n \ll m$. However, since our main task is to hit the small interval of exponents that lead to large errors, it is important to have a large initial population

TABLE II: Sanity Check Results

	Newton	Inv	Root	Poly	Exp	Cos
LSGA	2.8E-16	3.2E-16	8.1E+76	1.7E-11	1.0E+00	9.2E-01
BGRT	1.7E-16	2.5E-16	1.3E-14	4.7E-14	2.1E-15	1.2E-16

to cover the whole space.

V. Evaluation

A. Sanity Check

We first evaluated our technique on a set of collected test subjects from related work [10]. These subjects consist of six classic examples of stable and unstable implementations, as shown in Table II, where the first two programs are stable and the rest are unstable. Stable implementations are likely to produce accurate results than unstable ones.

Our technique is able to confirm those stable implementations and find relevant inputs that lead to large errors for the unstable ones. For each of the two stable programs in the set, the maximal relative error detected by our approach is smaller than 1×10^{-15}. For each of the four unstable programs in the set, the maximal relative error detected by our approach is larger than 1×10^{-11}. This result provides a preliminary evidence on the feasibility and usefulness of our approach.

Interestingly, in parallel to our work, Chiang et al. [19] also explore search algorithms for inputs that cause large errors, and in their experiments a guided random search algorithm (BGRT) works best. We also compared our technique with BGRT on the six subjects. As we can see from Table II, on none of the subjects BGRT outperforms LSGA, and BGRT also cannot distinguish stable and unstable subjects. One possible explanation is that guided random research cannot exploit the benefit of the fourth observation. Consequently, though BGRT found relatively large errors, it could not hit the largest one.

B. Experiment Overview and Research Questions

To further understand the performance of our approach on real world programs, we performed an experimental study using 154 functions from the GNU Scientific Library. In our study, we experimentally compared our approaches with two standard search techniques, a random search (RAND) and a standard genetic algorithm (STD), serving as the control techniques.

To compare the three techniques, a limit should be set to terminate the search. Based on our testing, executing programs in FPDebug occupies more than 99.5% of the execution time in all three search algorithms. For convenience, we set a limit on the number of times that FPDebug can be invoked by each algorithm. Then we compared the effectiveness of the three techniques on each function under this limit.

In general, our experimental study aims to investigate the following two research questions. The first research question (**RQ1**) is concerned with which of the three technique tends to find larger inaccuracies for each experimented function. The second research question (**RQ2**) concerns whether our approach is able to detect potential accuracy problems in practice.

TABLE III: Subjects with Different Parameters

Total	1-Para	2-Para	3-Para	4-Para
154	104	37	8	5

C. Experimental Setup

We conducted our experimental study on a virtual machine running the Ubuntu-10.04.4, hosted on a PC with a 2.3GHz Intel Pentium i5-2410M CPU and 6GB memory.

1) Subjects: Our subjects are also chosen from the latest version (i.e., version 1.16) of the GNU Scientific Library (GSL)[2] as subjects. GSL is an open-source numerical library for C and C++ programmers. The library provides a wide range of mathematical routines such as random number generators, special functions, and least-squares fitting. GSL has been used in previous studies [10], [20] on analyzing programs with floating-point numbers. GSL has in total 210 functions involving intensive floating-point computations. From these functions, whose total size is 48K lines of code, we chose all the functions with all inputs being floating-point numbers and the output being also a floating-point number. There are also three functions where FPDebug is reported to not work properly [9], so we removed the three functions. As a result, we used 154 functions in our experimental study. Each of the 154 functions has up to 4 parameters. Table III depicts the numbers of functions having 1, 2, 3, and 4 parameters, respectively. All the parameters and the return values of the 154 functions are of double precision.

2) Control Techniques: As mentioned earlier, we use a random search and a standard genetic algorithm as control techniques. The random search randomly generates every bit of the input parameters at each iteration, and returns the maximum relative error in all iterations. The standard genetic algorithm is designed by configuring the classic framework for genetic algorithm [21] using standard operations [21], [22]. More concretely, the algorithm starts from N randomly generated inputs as initial generation. For each generation, the algorithms repeat $n/2$ reproduction iterations to produce n children for the next generation, with each iteration producing two children. In every reproduction iteration, the algorithm picks two individuals from the current generation based on roulette wheel selection. The weight of each individual is the logarithm of the relative error. As recommended by the framework [21], computing logarithms is suitable for largely different fitnesses. Then each pair of parameters at the same position in the two individuals are crossed over with a probability C. The crossover is performed by treating the numbers as bit vectors, and exchange the first i bits of the two numbers with a randomly generated i. Finally, every bit of the two individuals are flipped with probability M. After a new generation is produced, the process starts with the new generation. Finally, the individual with the largest error in all generations is returned. Following the recommendation in existing papers [21], [22], we set $N = 20, C = 0.96, M = 0.1$.

[2] http://www.gnu.org/software/gsl/

3) Experimental Procedure: To answer the first research question, for each of the subject function, we used each of the three search techniques to calculate the maximum relative error that the technique can find. The limit of times for invoking FPDebug is 200, which are approximately equal to 60 seconds. In other words, random search will generate 200 inputs, the standard genetic algorithm will have 10 generations with 20 individuals each, and our algorithm will have 100 initial individuals and 100 mutations. There are mainly two reasons for us to use a relatively short iteration limit. Of course, more experiments are needed to further study the practical iteration limit for each of the techniques.

First, all the subject functions are library functions, which are building blocks for real world software. As a result, the execution time of one test input for a real world program can be hundreds or even thousands of times of that for a subject function in our experimental study. Thus, a small iteration limit for our subjects may be equivalent to a long time execution for real world programs. In other words, only techniques that can achieve satisfactory results on our subjects in a short iteration limit would have more practical value for real world programs.

Second, as the number of subject functions in our experimental study is large, a relatively small iteration limit would help us control our experimental procedure. Note that each technique may need to be executed many times against each subject function due to calibration.

The second research question is difficult to answer because it is difficult to decide whether a large error is a problem or not. Large errors may be fundamentally inevitable in many computations and are not considered problems. Interestingly, many GSL functions report an estimated absolute error for each execution. Using this information, we conservatively deem a large relative error $\geq 0.1\%$ as a potential problem when the actual absolute error is 10 times larger than the estimated one, as the large error is probably unexpected by the developer and may cause serious consequences.

For the generated test inputs that trigger an error larger than 0.1%, we invoke the associated functions with these inputs in FPDebug, and compare the estimated absolute error with the actual error reported by FPDebug. When the actual error is 10 times larger than the estimated one, we consider it a potential problem.

D. Threats to Validity

The main threat to the internal validity lies in the possible faults in our implementation. To reduce this threat, we reviewed all our source code before conducting the experiments. Note that, as Benz et al. [9] have made their tool publicly downloadable, we implemented the three techniques by directly invoking their tool, helping us further reduce this threat.

The main threat to external validity is concerned with the representativeness of our subjects. To reduce this threat, we used a large number of widely used functions (which have also been used in previous studies [10], [20]) as subjects in our study. Conducting more experiments using more real world programs as subjects would help us further reduce these threats.

TABLE IV: Maximum Inaccuracies Detected

Total	RAND	STD	LSGA	Tied
154	11 (7%)	24 (16%)	105 (68%)	14 (9%)

TABLE V: Sign Test on Inaccuracy Detection

	n_+	n_-	N	p
LSGA vs. RAND	127	12	139	<4.14e-22
LSGA vs. STD	110	30	140	<2.46e-11
STD vs. RAND	93	40	133	<6.52e-06

The main threat to construct validity is the limit of times that FPDebug can be invoked. To reduce this threat, we used a short limit to make the experimented techniques applicable for real world programs whose execution time may be much larger than that of our subjects. Note that, a technique that can detect inaccuracy in a short time would become more effective (or at least as effective) when used under a long time limit.

E. Experimental Results

In this subsection, we present the experimental results for the two research questions. All our experimental data are online.[3]

1) RQ1: Effectiveness of Inaccuracy Detection: Given a subject (denoted as s) and a technique (denoted as t), if the maximum relative error returned by the technique is larger than both the two maximum relative errors returned by the other two techniques, we deem that t is the best technique for s. Therefore, for each of three techniques, we count the number of subjects for which the technique is the best. For some subjects, no single technique is superior to both the other two, we deem that no technique is the best and denoted this situation as a tie.

The result of this comparison is depicted in Table IV. As we can see from the table, LSGA finds the maximum errors in the majority of the subjects, while random search finds the maximum errors in the least number of subjects. This results indicates that LSGA are more effective than the other two algorithms, while standard genetic algorithm is better than random search.

To further confirm whether the differences between the three techniques are statistically significant, we perform the sign test on each pair of techniques. The result of our sign test on inaccuracy detection is depicted in Table V. From the table we can see that the differences between each pair of techniques are significant, as p is much smaller than 0.05, the usual threshold for significant difference. In particular, LSGA has a very small p when compared with the other two techniques, which indicate that LSGA has absolute superiority over the other two techniques.

The sign test only considers which technique performs better for each subject, but does not consider whether the differences of relative errors found by different techniques are significant. If the relative errors found by two techniques are very close, both techniques are usable in practice. We deem that the difference between the two relative errors, e_1 and e_2 is significant, if

[3] http://sei.pku.edu.cn/%7exiongyf04/papers/ICSE15.html

TABLE VI: Significantly Larger Inaccuracies Detected

	Left	Right
LSGA vs. RAND	55 (36%)	3 (2%)
LSGA vs. STD	44 (29%)	5 (3%)
STD vs. RAND	25 (16%)	9 (6%)

e_1/e_2 or e_2/e_1 is larger than 10. Then we calculate, for each pair of techniques, how many significantly larger errors one techniques found over the other.

The result is shown in Table VI. The "Left" column shows how many significantly larger inaccuracies the left technique found than the right. Similarly, the "Right" column shows how many significantly larger inaccuracies the right technique found than the left. As we can see from the table, LSGA found significantly larger errors than the other two techniques in a large number of subjects, while the other two techniques found significantly larger errors than LSGA only in rare cases.

Although random search performed the worst among the three techniques, the above result also suggests that random search still found relatively large errors on some subjects. To further understand why this happened, we analyze a sample function where random search returns a large error. This function is `gsl_sf_bessel_j1`, which solves spherical Bessel function:

$$j_1(x) = \frac{sin(x)}{x^2} - \frac{cos(x)}{x}$$

For this function, random search reaches the maximum relative error of 1.089913e+02, a fairly large error. The implementation code of this function is listed below:

```
int gsl_sf_bessel_j1_e(const double x,
                       gsl_sf_result * result)
{
1:   double ax = fabs(x);
2:
3:   /* CHECK_POINTER(result) */
4:
5:   if(x == 0.0) {
6:     result->val = 0.0;
7:     result->err = 0.0;
8:     return GSL_SUCCESS;
9:   }
10:  else if(ax < 3.1*GSL_DBL_MIN) {
11:    UNDERFLOW_ERROR(result);
12:  }
13:  else if(ax < 0.25) {
14:    const double y = x*x;
15:    const double c1 = -1.0/10.0;
16:    const double c2 =  1.0/280.0;
17:    const double c3 = -1.0/15120.0;
18:    const double c4 =  1.0/1330560.0;
19:    const double c5 = -1.0/172972800.0;
20:    const double sum = 1.0 + y * (c1 + y * (c2 + y *
         (c3 + y*(c4 + y*c5))));
21:    result->val = x/3.0 * sum;
22:    result->err = 2.0 * GSL_DBL_EPSILON *
         fabs(result->val);
23:    return GSL_SUCCESS;
24:  }
25:  else {
26:    gsl_sf_result cos_result;
27:    gsl_sf_result sin_result;
28:    const int stat_cos = gsl_sf_cos_e(x, &cos_result);
29:    const int stat_sin = gsl_sf_sin_e(x, &sin_result);
30:    const double cos_x = cos_result.val;
31:    const double sin_x = sin_result.val;
32:    result->val  = (sin_x/x - cos_x)/x;
33:    result->err  = (fabs(sin_result.err/x) +
         fabs(cos_result.err))/fabs(x);
34:    result->err += 2.0 * GSL_DBL_EPSILON *
         (fabs(sin_x/(x*x)) + fabs(cos_x/x));
35:    result->err += 2.0 * GSL_DBL_EPSILON *
         fabs(result->val);
36:    return GSL_ERROR_SELECT_2(stat_cos, stat_sin);
37:  }
}
```

In the above code, `GSL_DBL_MIN` is about 2.22507e-308.

This function divides the input space into four segments, and returns (1) a constant number (line 6), (2) an underflow error (line 11), (3) the value calculated by series expansion (lines 14-21), and (4) the value calculated by standard library functions (lines 28-32).

Since the program has no loop, the only possibility of producing such a large error is to subtract two similar numbers, known as cancellation [16]. Cancellation can happen at addition and subtraction, which exist on line 20 and line 32. In the case of line 20, because the path condition (line 13) is $|x| < 0.25$, y and each c_i produced between lines 14 and 19 will be largely different on scale, and we will not subtract two similar values. On the other hand, large error may be triggered on line 32, e.g., when x is a large number and $cos(x)$ happens to be near zero.

As a result, the probability of producing a large error for a random input is decided by the probability of executing line 32 and the probability of producing a large error when line 32 is executed. We obtain the former by analyzing the path condition and the latter by sampling. The path condition of line 32 is $|x| \geq 0.25$, and 50.14% of all double-precision floating-point numbers fall in this range. We then randomly created 100 test inputs satisfying $|x| \geq 0.25$, and 15 of them generated a relative error larger than 1.0. Putting the two probabilities together, 7.521% of random inputs will trigger an extremely large error. It is very easy for random search to locate an input within this range. Since there were 200 test inputs created in our experiment, there is a probability of 99.99995% to trigger a large error.

The analysis of this sample function explains why random search can find large errors in some cases: there exist subjects for which very significant inaccuracies can be easily triggered by chance. Nevertheless, our results also suggest most programs do not belong to this category, and metaheuristic search-based approaches would be useful in locating large errors in these programs.

2) RQ2: Ability to Identify Potential Problems: In the previous experiment, our algorithm generated test inputs for 59 functions where the relative error is larger than 0.1%. We invoked these functions with the generated test inputs, and functions returned estimated absolute errors together with the result. By comparing the actual and estimated absolute errors, we found 18 functions that have potential problems of inaccuracy. This result again outperforms the other two algorithms significantly, where the STD found seven potential problems and RAND found five.

The detailed result about the 18 functions is shown in Table VII. Each line is a potential problem our approach identifies. The first column lists the function names, the second

TABLE VII: Functions with Potential Bugs

Name	Relative Error	Estimated Absolute Error	Reported Absolute Error
airy_Ai_deriv	1.54E+06	1.04E-06	1.35E+00
airy_Ai_deriv_scaled	1.54E+06	1.04E-06	1.35E+00
clausen	5.52E-02	6.37E-17	2.31E-02
eta	9.58E+13	1.27E+37	2.71E+50
exprel_2	2.85E+00	4.44E-16	7.41E-01
gamma	1.07E-02	6.94E-14	1.05E-01
synchrotron_1	5.35E-03	4.47E-14	3.07E-04
synchrotron_2	3.67E-03	6.39E-14	1.86E-04
zeta	9.58E+13	3.41E+18	1.19E+32
zetam1	1.42E-02	1.51E+19	7.42E+30
bessel_Knu	6.08E-03	3.33E+22	9.05E+34
bessel_Knu_scaled	6.08E-03	2.66E+22	9.05E+34
beta	9.21E-03	4.91E-13	2.04E-01
ellint_E	8.92E-03	1.58E-16	3.14E-03
ellint_F	8.79E-03	1.86E-16	3.64E-03
gamma_inc_Q	1.36E+13	8.88E-16	1.25E-12
hyperg_0F1	5.80E+06	2.08E+37	7.33E+49
hyperg_2F0	4.35E-03	5.20E+02	3.19E+12

column lists the relative errors reported by FPDebug, the third column lists the estimated absolute errors reported by the subject functions, and the last column lists the absolute errors reported by FPDebug.

We can see that in most cases the actual errors are many orders of magnitudes larger than the estimated ones, indicating potentially serious problems in practice. By searching the GSL repository, we found that none of problems in the 18 functions has been reported as bugs, while there exist bug reports reporting relative errors in a few orders of magnitude. We have submitted bug reports for the identified functions, but have not received any feedback yet (possibly due to the fact that there is only one programmer actively maintaining GSL now).

VI. Limitations and Future Work

First, our approach relies on FPDebug to detect the accuracy of the output, and FPDebug detects the accuracy through promoting the precision of floating-point numbers. For example, if the original program uses single-precision numbers, FPDebug will use double-precision numbers to perform the same computation, and compare the results to get the relative error. In theory, this approach may not always produce the correct result, because precision-specific treatments may be used in programs. For example, the original program may predict some large error for the precision it used and add to the result a pre-defined value to compensate the error, or the original program may use bit operators to accelerate the computation, which usually works only for a certain precision. Raising precision on these programs may not lead to more accurate results. As a result, the inaccuracies reported by our approach are only indications of potential accuracy problems and are not guaranteed to be bugs. However, this probably is not a problem in practice. First, programmers are able to identify the false positives easily, as they know whether any precision-specific treatment is used in their programs. Second, precision-specific treatments are not very commonly used in practice. As a matter of fact, precision adjustment has been used in different approaches [6], [23]–[25], and no problem is reported as far as we know.

Second, as our approach is based on testing, we cannot guarantee the inaccuracy detected by our approach to be always the maximum inaccuracy the program under test can produce. Similarly, when there are several inputs in the input domain of the program under test to trigger significant inaccuracy, our approach may find only one of them. Note that, when there are several independent inaccuracy-related faults, it is very likely that these faults lead to multiple unrelated inaccuracy-inducing inputs. In fact, as we do not know the maximum inaccuracy of the program under test, we also do not know how close the inaccuracy produced by our approach is to the maximum inaccuracy. In future work, we need to establish some benchmarks of maximum inaccuracy through exhaustive search, and use the maximum inaccuracy to evaluate our approach. Furthermore, we should also investigate the injection of inaccuracy faults to create subject programs with controllable inaccuracy.

Third, although FPDebug provided by Benz et al. [9] needs to instrument the binary code of the program under test, our approach itself is in principle a black-box approach. That is to say, our approach does not analyze the code of the program under test, but completely relies on the results produced by FPDebug. This strategy should be suitable for programs with simple control structures like the function discussed in Section III. But there are also programs with both complex control structures and intensive floating-point computations. In future work, we plan to investigate approaches that can also utilize information (e.g., coverage information) obtained from code analysis. One possible benefit of using coverage information is that, as collecting coverage information is much cheaper than executing Benz et al.'s tool, using coverage information to avoid always executing Benz et al.'s tool could make our approach more efficient. Another possible benefit of using coverage information is that coverage information may guide our approach to explore more paths to find more than one independent inaccuracy-related faults.

Fourth, our approach is currently applicable to only floating-point parameters. If the function has other type of parameters, the user has to specify pre-defined values for them. In future work, we plan to investigate approaches that can also deal with other types of parameters. One possible way is to consider suitable representations of other types of parameters in our genetic algorithm. Thus, our genetic algorithm can be naturally extended to these types. Another possible way is to use different heuristics (e.g., a heuristic based on coverage) for other types to consider that they may play different roles from floating-point numbers in programs with intensive floating-point calculation.

Finally, it would be interesting to compare our approach with static analysis approaches that give an upper bound of errors. However, it is hard to be done at the current stage because we do not know the maximum possible errors of our subjects. Furthermore, to the best of our knowledge, there exists no nontrivial benchmark with precious bounds of errors. When a static analysis produces a large upper bound while our approach finds only a small error, we cannot decide which approach performs better. In future work, benchmarks of floating-point error can be developed so that different approaches can be compared.

VII. Related Work

Static analysis. Many approaches have been proposed to statically analyze the possible upper bound of errors. These approaches are typically built upon interval arithmetic [4] or affine arithmetic [5]. Interval arithmetic presents each number as a pair of a lower bound and a upper bound of values, and replaces basic arithmetic operations as operations between intervals. For example, adding two intervals resulting in another interval presenting the maximally possible range of the result: $[a, b]+[c, d] = [a+c, b+d]$. Affine arithmetic enhances interval arithmetic by distinguishing errors coming from different sources. In affine arithmetic, each number is represented as an affine: $v+x_1\epsilon_1+x_2\epsilon_2+\ldots$ where v is the precise value, x_i is the error coming from a source and ϵ_i is a symbol representing the source. By differentiating errors by their sources, affine arithmetic can produce better result in operations such as $n-n$ where the operands contain errors from the same sources.

Typical static analysis approaches replace the numbers and operations in the target program in their interval/affine form, and use standard program analysis techniques such as symbolic execution and data-flow analysis to obtain the possible errors on the output. For example, Putot et al. [2] present a static analysis that relies on abstract interpretation by interval form. Goubault and Martel [3] also propose an approach based on affine arithmetic to analyze nontrivial numerical computations. However, these approaches usually consider the possible errors for the whole input space, and cannot identify which input could produce a large error. Furthermore, due to the nature of static analysis, a large interval on the output does not guarantee the existence of a large error.

Static verification. Another branch of approaches try to verify the precision of floating-point programs. Given a property about the floating-point accuracy, these approaches try to verify, automatically or interactively, whether the property holds in a program. Boldo et al. [26], [27] build support for floating-point C programs in Coq, allowing one to easily build proofs for floating-point C program. Ayad and Marché [28] propose the use of multiple provers to try to automatically generate proofs of floating-point properties. Darulova and Kuncak [1] propose a type system to guarantee the precision of floating-point programs. However, the verification-based approaches suffer from the same problem as static analysis: failing to prove a properties does not necessarily implies the existence of a large error, and no input could be provided for further debugging.

Precision tuning. FPDebug [9], the key component enabling our approach, dynamically analyzes the program by performing all floating-point computation side by side in higher precision. The difference between the standard result and the result in high-precision is the error of the result. Bao and Zhang [6] proposes to reduce the cost of detection by not computing the precise error but marking and tracking potentially inaccurate values. Based on similar ideas of tuning precision, Lam et al. [24], Rubio-González [25], and Schkufza et al. [23] propose approaches that automatically reduce the precision of floating-point computation to enhance performance with an acceptable loss of precision. These approaches serve as evidences that changing precision is a feasible technique for various purposes.

External errors. Our approach focuses on internal errors, which is about how much inaccuracy may be introduced during the computation of the program. External errors are errors from the sources outside the scope of the program. Such external errors in the input may be magnified during the computation of the program and result in significant inaccurate output. Different approaches have been proposed to analyze how robust a program is under an inaccurate input. Recent work include static verification of robustness by Chaudhuri et al. [29], dynamic analysis by Tang et al. [10], and dynamic sampling by Bao et al. [30]. However, these approaches cannot be used for internal errors as they are concerned with the execution of the subject program but not the precise output.

Test generation for floating-point programs. Test input generation is an important research topic and has been approached in different angles [22], [31], [32]. A typical approach [33] is to use symbolic execution to explore different paths and generate test inputs by solving path constraints. However, constraints with floating-point operations are usually difficult to solve. Miller and Spooner [7] first propose the use of search-based techniques instead of symbolic execution to generate test input data. Recently, Bagnara et al. [8] propose to use several search heuristics to enhance constraint-solving in concolic testing for floating-point programs. However, their goal of test generation is to increase path coverage, but not to detect floating-point inaccuracies. As far as we are aware, our approach is the first test generation approach to the detection of floating-point inaccuracies. Besides test input, test oracle generation is also an important problem in test automation. Recently, Zhang et al. [34] propose to infer metamorphic relations for regression testing. However, their approach only works for regression testing but not for initial testing.

VIII. Conclusion

In this paper we have shown that, with the recent advance in the dynamic analysis of floating-point errors, it has become possible for search-based test generation aiming at maximizing the likely errors. By exploiting the statistical characteristics of large floating-point inaccuracies, we have designed a specialized genetic algorithm in order to efficiently search for large inaccuracies in numerical programs. Our experimental results demonstrate that our approach is able to find many large inaccuracies in a widely-used library, indicating the proneness of numerical programs to large inaccuracies in practice.

References

[1] E. Darulova and V. Kuncak, "Trustworthy numerical computation in scala," in *Proc. OOPSLA*, 2011, pp. 325–344.

[2] S. Putot, E. Goubault, and M. Martel, "Static analysis-based validation of floating-point computations," in *Numerical software with result verification*, 2004, pp. 306–313.

[3] E. Goubault and S. Putot, "Static analysis of numerical algorithms," in *Proc. SAS*, 2006, pp. 18–34.

[4] T. Hickey, Q. Ju, and M. H. Van Emden, "Interval arithmetic: From principles to implementation," *Journal of the ACM*, vol. 48, no. 5, pp. 1038–1068, 2001.

[5] J. Stolfi and L. De Figueiredo, "An introduction to affine arithmetic," *TEMA Tend. Mat. Apl. Comput.*, vol. 4, no. 3, pp. 297–312, 2003.

[6] T. Bao and X. Zhang, "On-the-fly detection of instability problems in floating-point program execution," in *Proc. OOPSLA*, 2013, pp. 817–832.

[7] W. Miller and D. L. Spooner, "Automatic generation of floating-point test data," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 223–226, 1976.

[8] R. Bagnara, M. Carlier, R. Gori, and A. Gotlieb, "Symbolic path-oriented test data generation for floating-point programs," in *Proc. ICST*, 2013, pp. 1–10.

[9] F. Benz, A. Hildebrandt, and S. Hack, "A dynamic program analysis to find floating-point accuracy problems," in *Proc. PLDI*, 2012, pp. 453–462.

[10] E. Tang, E. T. Barr, X. Li, and Z. Su, "Perturbing numerical calculations for statistical analysis of floating-point program (in)stability," in *Proc. ISSTA*, 2010, pp. 131–142.

[11] W. Kahan, "IEEE standard 754 for binary floating-point arithmetic," *Lecture Notes on the Status of IEEE*, vol. 754, no. 94720-1776, p. 11, 1996.

[12] E. Falkenauer, *Genetic algorithms and grouping problems*. John Wiley & Sons, Inc., 1998.

[13] N. J. Higham, *Accuracy and stability of numerical algorithms*. Siam, 2002.

[14] B. D. McCullough and H. D. Vinod, "The numerical reliability of econometric software," *Journal of Economic Literature*, pp. 633–665, 1999.

[15] S. Zhao, "A calculator with controlled error, example section (in Chinese)," 2015, [Accessed 12-February-2015]. [Online]. Available: http://www.zhaoshizhong.org/download.htm

[16] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar. 1991.

[17] D. B. Fogel, *Evolutionary computation: toward a new philosophy of machine intelligence*. John Wiley & Sons, 2006, vol. 1.

[18] L. D. Whitley *et al.*, "The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best." in *ICGA*, vol. 89, 1989, pp. 116–123.

[19] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev, "Efficient search for inputs causing high floating-point errors," in *PPoPP*. ACM, 2014, pp. 43–52.

[20] E. T. Barr, T. Vo, V. Le, and Z. Su, "Automatic detection of floating-point exceptions," in *Proc. POPL*, 2013, pp. 549–560.

[21] J. J. Grefenstette, "Optimization of control parameters for genetic algorithms," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 16, no. 1, pp. 122–128, 1986.

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

[23] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic optimization of floating-point programs with tunable precision," in *PLDI*, 2014, pp. 53–64.

[24] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, "Automatically adapting programs for mixed-precision floating-point computation," in *ICS*, 2013, pp. 369–378.

[25] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *SC*, 2013, pp. 27:1–27:12.

[26] S. Boldo and J.-C. Filliâtre, "Formal verification of floating-point programs," in *Proc. ARITH*, 2007, pp. 187–194.

[27] S. Boldo and G. Melquiond, "Flocq: A unified library for proving floating-point algorithms in coq," in *Proc. ARITH*, 2011, pp. 243–252.

[28] A. Ayad and C. Marché, "Multi-prover verification of floating-point programs," in *Proc. IJCAR*, 2010, pp. 127–141.

[29] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour, "Proving programs robust," in *ESEC/FSE '11*, 2011, pp. 102–112.

[30] T. Bao, Y. Zheng, and X. Zhang, "White box sampling in uncertain data processing enabled by program analysis," in *OOPSLA '12*, 2012, pp. 897–914.

[31] T. Xie, L. Zhang, X. Xiao, Y. Xiong, and D. Hao, "Cooperative software testing and analysis: Advances and challenges," *Journal of Computer Science and Technology*, vol. 29, no. 4, pp. 713–723, 2014.

[32] D. Hao, L. Zhang, M. Liu, H. Li, and J. Sun, "Test-data generation guided by static defect detection," *Journal of Computer Science and Technology*, vol. 24, no. 2, pp. 284–293, 2009.

[33] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[34] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei, "Search-based inference of polynomial metamorphic relations," in *ASE*, 2014, pp. 701–712.