Efficient Generation of Error-Inducing Floating-Point Inputs via Symbolic Execution

Hui Guo
Department of Computer Science
University of California, Davis, USA
higuo@ucdavis.edu

Cindy Rubio-González
Department of Computer Science
University of California, Davis, USA
crubio@ucdavis.edu

ABSTRACT

Floating point is widely used in software to emulate arithmetic over reals. Unfortunately, floating point leads to rounding errors that propagate and accumulate during execution. Generating inputs to *maximize* the numerical error is critical when evaluating the accuracy of floating-point code. In this paper, we formulate the problem of generating high error-inducing floating-point inputs as a code coverage maximization problem solved using symbolic execution. Specifically, we define inaccuracy checks to detect large precision loss and cancellation. We inject these checks at strategic program locations to construct specialized branches that, when covered by a given input, are likely to lead to large errors in the result. We apply symbolic execution to generate inputs that exercise these specialized branches, and describe optimizations that make our approach practical. We implement a tool named FPGen and present an evaluation on 21 numerical programs including matrix computation and statistics libraries. We show that FPGen exposes errors for 20 of these programs and triggers errors that are, on average, over 2 orders of magnitude larger than the state of the art.

CCS CONCEPTS

• **Mathematics of computing → Numerical analysis**; • **Software and its engineering → Software testing and debugging**; **Software verification and validation**.

KEYWORDS

floating-point, testing, catastrophic cancellation, roundoff errors, symbolic execution

ACM Reference Format:
Hui Guo and Cindy Rubio-González. 2020. Efficient Generation of Error-Inducing Floating-Point Inputs via Symbolic Execution. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380359

1 INTRODUCTION

Floating-point numbers are widely used as a standard to represent reals in modern computers. The limited precision in floating-point representation and computation, however, remains a known threat to the correctness, accuracy and stability of floating-point programs. Numerical bugs due to rounding errors, nonreproducibility and floating-point exceptions are common in floating-point programs [18]. In particular, the propagation and accumulation of rounding errors have resulted in catastrophic failures [1, 6, 33].

Floating-point errors. The floating-point error of a computation refers to the sum of the rounding errors accumulated in the result of the computation. This includes errors due to inaccurate initial data as well as errors generated during the computation due to floating-point finite precision. The amount of floating-point error included in the final result of a program specifies the accuracy of the code. While there are several tools (e.g., [10, 25, 27]) that, given a set of inputs, help detect and identify accuracy and stability problems of floating-point code, few testing tools have been developed to trigger and expose floating-point errors. Existing floating-point testing tools [9, 11, 19] mainly focus on triggering floating-point exceptions, or maximizing code coverage of floating-point programs, while finding inputs to test accuracy is left to developers. Generating inputs to *maximize* the numerical error is critical when evaluating the accuracy of floating-point code. Besides testing, maximizing numerical error is significantly important in identifying inaccurate code areas for automated floating-point program repair or optimization [21, 24, 29, 31, 32, 38].

Error-inducing floating-point inputs. Finding inputs that trigger large numerical errors is non-trivial. As observed in a previous study [10], only a very small portion of the input domain can cause large errors. It is challenging to identify such inputs because rounding errors are unintuitive and difficult to reason about. Floating-point optimization techniques [21, 29, 31] use random inputs that satisfy common distributions or code coverage criteria. The state-of-the-art error-inducing input generators [16, 37, 39] perform limited analysis over the floating-point errors generated during execution, and mainly rely on searching or sampling to identify error-inducing inputs. This leads to a lack of support for numerical programs with multi-dimensional input data due to the large input space. Both LSGA [39] and EAGT [37] are designed for floating-point programs with a small number of scalar inputs.[1]

In reality, a large number of numerical programs, e.g., matrix computation libraries that are widely used in scientific computing, machine learning libraries, and software in computer graphics and data analysis, take multi-dimensional input data such as arrays,

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380359

[1] LSGA does not provide an algorithm to generate multiple floating-point inputs, and is evaluated on programs with at most four scalar floating-point inputs (with the majority taking only one scalar floating-point input). EAGT relies on the approximation of condition numbers, and only focuses on programs with one scalar floating-point input.

vectors or matrices. S3FP[16] is the state-of-the-art tool for generating error-inducing multi-dimensional floating-point inputs. S3FP divides each input number interval and randomly permutes the subintervals to zoom into tighter input ranges in exploring the input space. However, its black-box nature results in a smaller chance to find inputs that trigger the highest errors.

In this paper, we propose a white-box algorithm to generate high error-inducing inputs for floating-point programs especially with multi-dimensional input data. Specifically, we check the floating-point errors generated during execution and use such error patterns to identify inputs that are likely to trigger high errors in the result. By adding error checks, we transform the problem of generating high error-inducing inputs into the code coverage maximization problem that can be solved by performing symbolic execution.

Symbolic execution. Symbolic execution (e.g., [13, 14, 20, 23]) enhances program testing by symbolizing inputs to achieve higher code coverage. In symbolic execution, instead of concrete values, the inputs are represented as symbols that indicate arbitrary numbers. Each program operation on concrete values is replaced by an operation on symbolic values, and accordingly the value of each variable is an expression in terms of the input symbols. When encountering a conditional statement, the execution is forked into two to each follow the *true* and *false* branch, and each child process adds the corresponding constraints into its program state to identify the executed path. Once the program terminates, or an error occurs, e.g., division by zero, the path constraints are solved by satisfiability modulo theory (SMT) constraint solvers to find concrete values for the symbolic inputs.

Recent works [19, 28] have incorporated floating-point arithmetic in symbolic execution. Like integer variables, floating-point variables are symbolized to be integrated into different path constraints and floating-point inputs are generated through SMT solvers with floating-point support (e.g., Z3 [7]). However, a floating-point input that covers one path of the program does not necessarily trigger large numerical errors on that path, which is shown in our evaluation. To find inputs that trigger high errors, a more specific algorithm is needed.

Key insight. Our key insight is that by injecting inaccuracy checks after floating-point arithmetic operations, we force symbolic execution to explore the probability of the occurrence of severe rounding errors or numerical cancellation at each injection site. Severe rounding can cause significant precision loss, which could affect the accuracy of the result, and a cancellation could be catastrophic due to loss of significance. In floating-point code, it is hard to predict which part of the program could possibly involve severe rounding or cancellation, and what the impact in the result is if either happens. Our technique enables symbolic execution to explore rounding and cancellation possibilities in different code areas. For each input generated, we measure the error it exposed, and select only those that trigger the largest error.

The main *challenge* stems from an inherent limitation of symbolic execution: path explosion [15]. Due to the exponential growth of the number of feasible paths, CPU and memory usage becomes high. More importantly, as more constraints are added into a path constraint, it takes longer for such constraints to be solved. The necessity for inaccuracy check injection aggravates the problem by introducing additional paths to be explored. To alleviate path explosion, we carefully manage the number of symbolic input variables by separating the input variables into two groups and concretizing the group with larger size. Second, we only instrument the floating-point operations of interest in the core loop of the algorithm, and use two sampling strategies to dynamically enable injection. Lastly, we formulate the inaccuracy checks using bitwise operations, which significantly reduce the number of branches at each injection point.

We implement our approach in a tool named FPGen and demonstrate that FPGen is effective at generating inputs that expose large floating-point errors. Our evaluation on 3 summation algorithms and 18 numerical programs from the Meschach library [5] and the GNU Scientific library [2] shows that FPGen is able to expose errors for nearly all programs, and the order of the magnitude of exposed relative errors is -6.35 on average, which indicates that the result has only around 6 accurate digits.

We compare FPGen's generated inputs against random input generation, the state-of-the-art error-inducing input generator S3FP [16], and KLEE-Float [28], a symbolic execution engine that provides floating-point support. The results show that random input generation and S3FP trigger errors in 13 out of 21 programs while FPGen triggers errors in all programs except for one. Furthermore, FPGen triggers larger errors than all other approaches for 15 out of 21 programs. The order of the magnitude of the exposed relative errors is -12.69 on average for random input generation and -8.46 on average for S3FP. On the other hand, KLEE-Float fails to trigger errors for all programs. Regarding to the magnitude of errors the tools can trigger, FPGen improves the state-of-the-art input generator S3FP by more than 2 orders of magnitude.

The contributions of this paper are as follows:

- We enable symbolic execution to find high error-inducing inputs by incorporating precision loss and cancellation checks under floating-point computations, and describe various optimizations to scale symbolic execution, including managing the number of symbolic variables and selecting injection sites (Section 3).
- We evaluate FPGen on a set of 21 numerical programs including matrix computation and statistics libraries, and show that FPGen outperforms the state of the art in the majority of the programs. Moreover, FPGen advances the state of the art by triggering errors that are more than 2 orders of magnitude larger (Section 4).

The rest of this paper is organized as follows. Section 2 illustrates testing of floating-point programs using 3 summation algorithms. Section 3 describes our inaccuracy checks, and optimizations that make symbolic execution effective at generating error-inducing inputs. Section 4 describes our experimental evaluation. Finally, Section 5 discusses related work and we conclude in Section 6.

2 FLOATING-POINT ACCURACY TESTING

In this section, we first illustrate the problem of exposing floating-point inaccuracy using three well-known floating-point summation algorithms: recursive summation, pairwise summation, and compensated summation. Second, we show how the state of the art in error-inducing input generation (along with other two baselines) fails to find inputs that trigger errors in these algorithms while we successfully craft such an input manually. Finally, we discuss our insight for generating inputs that maximize error.

```
1 double recursive_summation(double* A, int size){
2   for (int i = size-1 ; i > 0 ; i--)
3     A[i-1] += A[i];
4   return A[0];}
```

(a) Recursive Summation

```
1 double pairwise_summation(double a1, double a2,
2   double a3, double a4){
3   a1 += a2; a3 += a4;
4   a1 += a3;
5   return a1;}
```

(b) Pairwise Summation

```
1 double compensated_summation(double* A, int size){
2   double sum, a, e=0;
3   for (int i = size-1 ; i > 0 ; i--){
4     sum = A[i];
5     a = A[i-1] + e;
6     A[i-1] = sum + a;
7     e = (sum - A[i-1]) + a;}
8   return A[0];}
```

(c) Compensated Summation

Figure 1: Floating-point summation algorithms.

2.1 Floating-Point Summations

To achieve better accuracy when adding floating-point numbers, a variety of summation algorithms have been proposed. Figure 1 shows 3 summation algorithms that compute the sum over the elements of a DOUBLE array.[2] The recursive summation algorithm iteratively adds each element in the array in reverse order. It is simple and the most frequently used, but its accuracy depends on the order in which numbers are given. Pairwise summation adds array elements in pairs to avoid large rounding errors introduced when adding each of the elements to the partial sum. Lastly, the compensated summation algorithm uses a correction term, i.e., e (line 2), to diminish the rounding error incurred in the addition operation (line 6) of the last iteration. More details on the accuracy of these (and other) summation algorithms can be found in [22].

2.2 Error-Inducing Inputs for Summations

First, we investigate the effectiveness of three existing approaches for generating error-inducing inputs. Specifically, we use the C++ `rand` function as random number generator, S3FP [16], the state-of-the-art error-inducing input generator for programs with multi-dimensional input data, and KLEE-FLOAT [28], a symbolic execution engine that supports floating point. We implement an additional 128-bit quadruple precision version of each summation. For each input array generated, we run both original and high-precision programs to calculate the error in the result. As shown in Table 1a, for an input array of size 4 in range $[-100, 100]$, the random generator and S3FP searched 1000 input arrays but failed to find one that exposes error in the summations. KLEE-FLOAT generated an array of zeros that covers the *only* path in the programs while exposing no errors.

Second, we manually crafted an array A that triggers high numerical error on each summation algorithm. The values for each array element can be found in Table 1b. We refer to the manual approach simply as MANUAL. Table 1a shows that MANUAL exposes a relative error of 12 in the recursive and compensated summations, and a relative error of 1 in the pairwise summation.

[2]For simplicity and readability, we omit the bulky code of pairwise summation for an array of size N, and illustrate it using 4 DOUBLE variables.

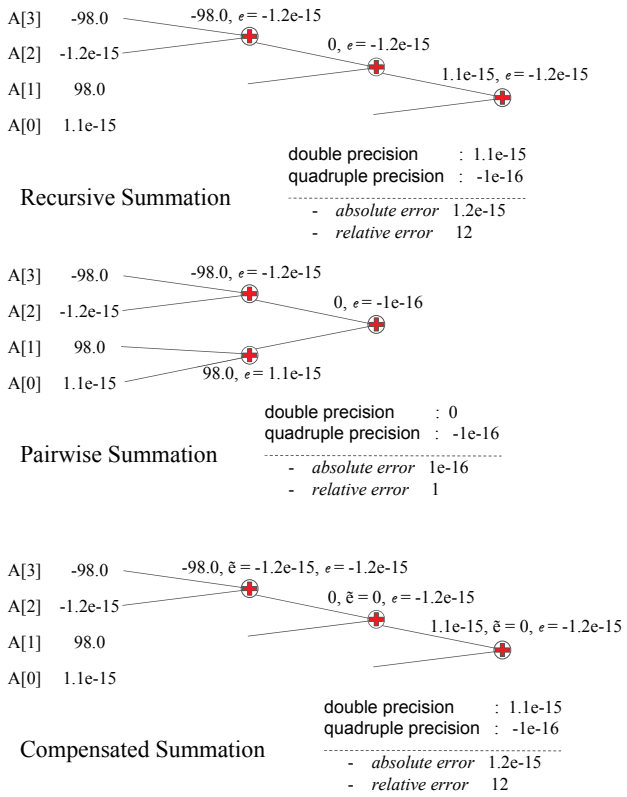


Figure 2: Summations on the manually crafted input array.

Figure 2 shows the computation process of each summation algorithm over the manually crafted input to shed light on the generation of error-inducing inputs. Program executions of recursive summation in DOUBLE and QUADRUPLE precision produce summation results 1.1e−15 and −1e−16, respectively. Using the result of QUADRUPLE execution as the ground truth, the *absolute* error of DOUBLE execution is 1.2e−15, and the *relative* error is 12. We further examine the result and the error incurred at each addition operation in DOUBLE precision. As shown in Figure 2, the first addition adds $A[3]$ to $A[2]$. Because $A[2]$ is less than the least significant digit (i.e., ULP) of $A[3]$, −1.4e−14, $A[2]$ is rounded off (shown as the rounding error, e). The second addition adds the result so far to $A[1]$, unfortunately both values cancel out. The local error remains −1.2e−15 since no new rounding errors occur. Lastly, $A[1]$, which contains the partial summation result 0, is added to $A[0] = 1.1e{-}15$. The final result, 1.1e−15, is stored in $A[0]$, and the error accumulated in the result is −1.2e−15. Because the magnitude of the error is close to the result of the summation, the relative error is high.

Similarly, the first two additions in pairwise summation generate two errors by rounding off the smaller operand, and the last addition cancels the two partial summation results. The cancellation causes the magnitude of the error to be comparable to the result, leading to a large relative error. Compensated summation maintains

Table 1: Accuracy testing of summation algorithms. The input is an array of size 4 in the range $[-100, 100]$.

(a) Testing results.

Approach	#Inputs	Maximum Relative Error		
		Recursive Sum.	Pairwise Sum.	Compensated Sum.
Random	1000	0	0	0
S3FP	1000	0	0	0
KLEE-Float	1	0	0	0
Manual	1	12	1	12

(b) Manual values for input array.

Array Element	FP Value
A[3]	-98.0
A[2]	-1.2e-15
A[1]	98.0
A[0]	1.1e-15

a correction term, i.e., $\tilde{e}$ in the third subgraph of Figure 2. It captures the rounding error generated by the current addition operation, and will be added when applying the next addition. For example, after the first addition, $\tilde{e}$ holds the rounding error $-1.2e{-}15$, introduced by adding $A[3]$ and $A[2]$. However, because the next term to add, $A[1] = 98.0$, has an ULP greater than $-1.2e{-}15$, the correction term is dropped in the second addition. Also, cancellation occurs in the second iteration. In the end, the compensated summation performs the same as recursive summation, and its result over array A is $1.1e{-}15$ with a relative error of 12.

Input generation insight. We observe two general patterns in the summations over the manual input array A: (a) rounding that particularly occurs when adding two floating-point numbers whose exponents vary widely, e.g., $-98.0+(-1.2e{-}15)$, and (b) cancellation that affects large terms. The first pattern introduces rounding errors in intermediate results while the second pattern exposes the errors by canceling the accurate significant digits. We propose to inject *inaccuracy checks* at floating-point operations to detect rounding and cancellations. Our checks do not require maintaining a high-precision shadow execution to calculate intermediate errors, but check inaccuracies solely based on the operands and the result of the computation.

In response to where to inject inaccuracy checks, we observe in our example that the rounding pattern arises in the first addition while the cancellation pattern occurs on different addition operations in the 3 summation programs (2nd addition in recursive and compensated summation, and 3rd addition in pairwise summation). In reality, it is unattainable to predict the floating-point operations at which rounding and cancellation need to happen. This is the reason why we create a search space on the inaccuracies of computations and conduct the search using symbolic execution. In summary, the goal of our inaccuracy checks is to create a search space to allow symbolic execution to explore the inaccuracies of different code areas for high error-inducing inputs. This observation on the inaccuracy patterns can be generalized to other floating-point code. It is highly likely that an input that conforms to our inaccuracy checks will produce a high numerical error in the result.

3 TECHNICAL APPROACH

Our approach to generate floating-point inputs that expose large numerical errors consists of three main components, which are illustrated in Figure 3. We first apply a program transformation that injects checks for precision loss and cancellation into the program P. Second, we apply various optimizations to mitigate path explosion during symbolic execution, including reducing the number

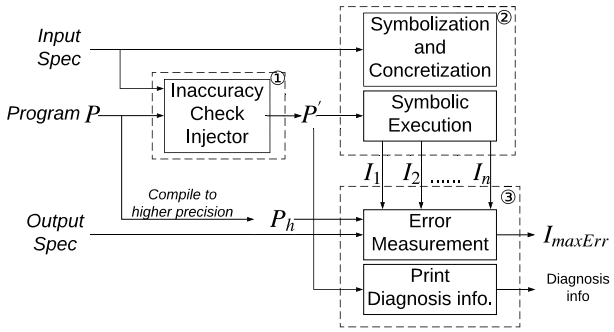


Figure 3: FPGen workflow.

of symbolic variables by concretizing some input variables using random values. Input specifications are required for the identification of input variables. Symbolic execution is then performed in the transformed program P'. Finally, we assess the quality of the generated inputs $I_1, I_2, ..., I_n$ by measuring their errors with respect to a higher-precision version of the program P_h. The input that exposes the largest error, I_{maxErr}, is then selected. We also diagnose the root cause of the numerical error to further help the programmer in identifying the program expressions that contribute the most to the numerical inaccuracy.

3.1 Inaccuracy Check Injector

The goal of the inaccuracy check injector is to transform a given program so that the result of each injected floating-point arithmetic operation is explicitly checked for precision loss and cancellation errors. In this section, we assume a three-address code representation in which each arithmetic operation has at most two operands. We first define each of the two checks separately, and then we describe how we combine the two when injecting them into the program.

3.1.1 Check for Precision Loss. Rounding errors are inherent to floating point, and occur when an operation results in a value that cannot be exactly represented in floating point. This leads to a loss of precision in the computed result. In this paper, we focus on precision loss that results in most bits of a data value being discarded. For example, a number smaller than $1.2e{-}38$ is rounded to 0 when represented in single precision[3]. The rounding error incurred is equal to the data value itself, and all bits of precision are effectively discarded.

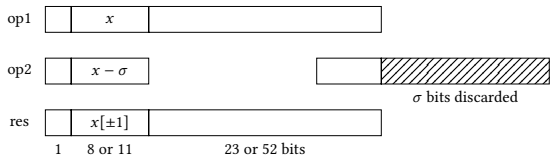
[3] Assuming no subnormal numbers.

In most cases this type of precision loss[4] is unintentional and can be the symptom of a numerical bug. For example, consider the summation of 1, 1e−8, and −1. If we apply the summation in the order of 1 + 1e−8 + (−1) in single precision, the sum is 0 because 1 + 1e−8 equals to 1 due to precision loss that causes all bits of precision of 1e−8 to be lost. However, if we change the order to 1 + (−1) + 1e−8, we are able to attain the exact result of the summation, 1e−8, without generating numerical errors. From the view of programmers, the intention is to add the three given numbers. However, the first order leads to a result that only adds two of the numbers due to precision loss. This violates programmers' intention and therefore is a hidden numerical bug. Precision loss checks are designed to expose such errors.

We inject explicit precision loss checks after floating-point addition and subtraction operations.[5] For each given floating-point operation, we compare the exponents of the two operands. The intuition behind is that the addition of two floating-point values of similar magnitude will result in a more accurate result than the addition of two values whose magnitude differ significantly. We define the precision loss check as follows:

$$|exp(op1) - exp(op2)| \geq \sigma \tag{1}$$

where $exp(op1)$ and $exp(op2)$ represent the exponents of the two source operands, respectively, and σ is an integer constant that defines the lower bound of the exponent difference. In other words, σ represents the number of bits of precision that are discarded. For example, if σ is greater than the number of significand digits, i.e., 23 in single precision and 52 in double precision, all bits of precision in the corresponding operand will be discarded.



The figure above visualizes the computation on *op1* and *op2*, which leads to σ bits of *op2* being discarded. Each operand and the result of the computation are described in the floating-point format that contains three components: sign (1 bit), exponent (8 bits in FLOAT, 11 bits in DOUBLE) and fraction (23 bits in FLOAT, 52 bits in DOUBLE). As shown, the exponent of *op1* is x, σ greater than the exponent of *op2*. When performing an addition or subtraction on *op1* and *op2*, the fraction of *op2* is shifted to the right by σ bits. The first few bits ($23 - \sigma$ bits in FLOAT, $52 - \sigma$ bits in DOUBLE) of *op2* are used to compute the fraction of the result *res* while the last σ bits are discarded. Finally, *res* is normalized and its exponent can be adjusted by 1.

[4]For simplicity, the rest of this paper refers to *precision loss* when most bits of precision are discarded.

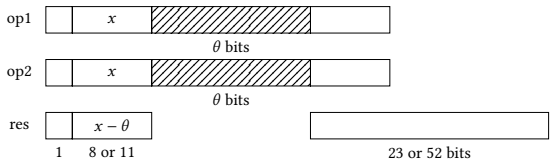
[5]Among the floating-point arithmetic operations, +, −, ×, ÷, $\sqrt{}$, %, addition and subtraction are the common operations that are likely to discard most bits of precision in one of its operand data values.

3.1.2 Check for Cancellation Errors. A cancellation occurs when two floating-point numbers with opposite sign and nearly equal magnitude are added. The most significant bits are canceled with the least (often inaccurate) significant bits taking precedence. Consider the decimal numbers 1.9874 and −1.9856. Rounding these numbers to three decimal digits results in 1.987 and −1.986, each with a rounding error of 4e−4. If we add these numbers, the first three digits cancel each other and the result of the addition is 0.001, which is comparable in magnitude to the rounding errors. The relative error of the result is $|0.001-0.0018|/0.0018 = 4.4e{-}1$, which can be unacceptable. Such a cancellation error can have serious repercussions. The affected value could change the control flow of the program if used in a conditional expression, or be amplified through the rest of the computation, thus potentially introducing a large numerical error in the final result.

Cancellation checks have been used in prior work [10, 25] to detect program instability on the fly. Specifically, the cancellation check is defined as follows:

$$\max\{exp(op1), exp(op2)\} - exp(res) \geq \theta \tag{2}$$

where $exp(x)$ represents the exponent of floating-point number x, $op1$ and $op2$ are the two operands, and res is result of an addition or subtraction operation. The value θ denotes the lower bound of the number of significant bits that are canceled. For example, two numbers are canceled out to 0 if θ is 23 bits in single precision. As visualized below, operand *op1* and operand *op2* have the same exponent x, and the exponent of the computation result *res* is reduced to $x - \theta$. The first θ bits of *op1* and *op2* are discarded due to cancellation. The few least significant bits of *op1* and *op2* ($23 - \sigma$ bits in FLOAT, $52 - \sigma$ bits in DOUBLE), which are inaccurate due to rounding errors, are used to compute the most significant bit of *res*.



3.1.3 Check Injection. First, we construct an inaccuracy detector that checks for precision loss and cancellations in floating-point computations. To facilitate symbolic execution, we divide the program execution under the computation into three branches. As shown in Figure 4, one branch is guarded by the precision loss condition formalized in Equation (1) to explore inputs that lose precision in the execution; one branch is secured with the cancellation condition described in Equation (2) to select inputs that can cause catastrophic cancellation; and the third branch satisfies neither of the two conditions, and can be referred as the accurate branch. Precision loss and cancellation conditions are contradictory as cancellation requires the two source operands to be nearly equal and precision loss happens only when the two source operands are in significant different order of magnitude. Therefore, no cancellation occurs in the precision loss branch.

Inaccuracy thresholds. We select thresholds σ and θ for precision loss and cancellation, respectively. Threshold σ represents the number of bits of precision that are discarded in precision loss from

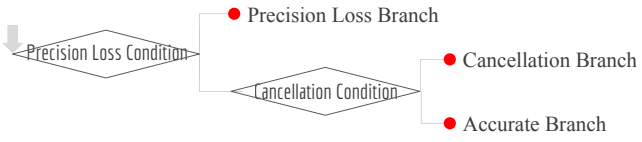


Figure 4: Inaccuracy check branches.

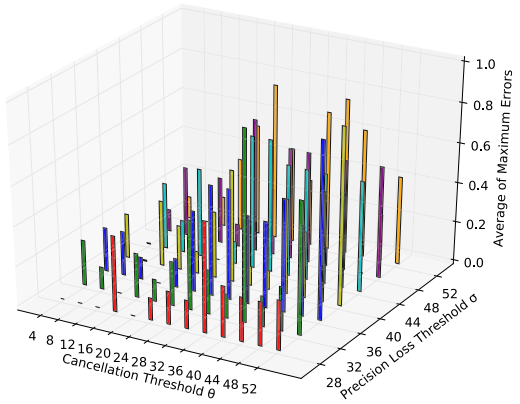


Figure 5: Average of maximum errors on 3 summation programs with different inaccuracy thresholds.

Equation (1), and θ represents the number of bits in the significand that cancel from Equation (2). The larger σ is, the more bits of precision are discarded in the operand value with the smaller magnitude of the two. Similarly, the larger θ is, the more bits of the significand are canceled in the two operand values. Both will cause significant inaccuracies. In double precision, σ and θ are positive integers no greater than 52 (the number of bits in the significand). We select the values of σ and θ based on an empirical evaluation on the three summation programs presented in Section 2.

We start the search with the parameter setting $\{\sigma = 28, \theta = 4\}$ and investigate all multiples of 4 for σ and θ.[6] For each parameter setting, we compute the average of the top 3 errors triggered in each program and combine the results of the three programs by calculating the mean value. Figure 5 shows the mean value of the errors in the three summation programs while the threshold parameters σ and θ vary. The coordinate of a bar indicates (θ, σ), and the height denotes the mean value of the maximum errors. The bars use distinct colors for different values of parameter σ. The parameter setting that ranks first is $\{\sigma = 32, \theta = 40\}$. We use this setting in our experimental evaluation, which yields fruitful results.

Check condition binarization. The two check conditions in Figure 4 involve computations such as the absolute value of an integer in Equation (1) and the maximum of two integers in Equation (2). These operations generate additional branches in the binary code. The following two statements show these branches using the conditional operator (? :) in C.

[6]Our initial value is $\sigma = 28$ because we consider that discarding at least 28 bits could affect precision sufficiently.

```c
#define exp_bt(pa) (long)((*(unsigned long*)(pa)
  >>52)&0x7ff)
```

(a) exp

```c
#define abs_mask(a) ((a)>>(sizeof(long)*8-1))
#define abs_bt(a) (((a)+abs_mask(a))^abs_mask(a))
```

(b) abs

```c
#define max_bt(a, b) ((a)^(((a)^(b))&-((a)<(b))))
```

(c) max

Figure 6: Bitwise utility functions.

```
abs(a) = (a > 0)? a : -a;
max(a, b) = (a > b)? a : b;
```

The expansion of branches increases the number of paths exponentially and makes it more difficult for the symbolic execution engine to find inaccuracy patterns. To alleviate this problem, we designed three highly optimized bitwise utility functions that do not include any branches. The utility functions are *exp_bt* to obtain the exponent of a floating-point number with specified precision as a long integer, *abs_bt* to obtain the absolute value of a long integer, and lastly, *max_bt* that returns the largest of two long integers. Figure 6 presents the detailed implementation of the three functions in double precision using C macros.

Selecting injection sites. In this paper, we mainly focus on maximizing numerical error for floating-point code that uses multi-dimensional input data. On the selection of injection sites for inaccuracy checks, we are particularly interested in loops that iterate on the input data and update the variable that holds the result. Assuming three-address code, we inject inaccuracy checks under the addition and subtraction operations. Unfortunately, it is not practical to check for inaccuracies in each iteration. Therefore, we provide two sampling strategies, uniform and logarithmic, to dynamically select loop iterations for inaccuracy checks. Both sampling strategies supply the parameter *start* and *step* for customization. Parameter *start* allows the user to start the counter from any iteration, and *step* specifies the step to the next sampled iteration. In logarithmic sampling, *step* indicates the initial step, which is multiplied by 10 each time the counter increases by one order of magnitude.

Remark. Based on our observation of inaccuracy patterns (described in Section 2), we inject inaccuracy branches to enable symbolic execution to find error-inducing inputs. Note that previous work [10, 25] has used Equation (2) to *detect* cancellation when running a program on a *given* set of inputs. In contrast, our focus in this paper is to *generate* inputs that maximize error. To the best of our knowledge, we are the first to formulate precision loss in which most bits of one operand are discarded, and combine it with cancellations for inaccuracy checking. Our technique is the first to enable a widely-used technique such as symbolic execution to find floating-point inputs that maximize error. Symbolic execution itself, however, is unable to generate such inputs as shown in Section 4.

3.2 Symbolic Execution with Concretization

After injecting precision inaccuracy checks, we proceed to symbolically execute the program under test. In this paper, we use one of the most popular and mature symbolic execution tools, KLEE [13],

Data: Input variables : *inVars*, Input specifications : *inSpecs*, Program : *P*, Program with injected inaccuracy checks : *P'*, Time budget : *tBudget*, Timeout parameters : τ_0, τ_1
Result: Maximum error, the error-inducing input

```
1  tStart = time();
2  /* Random search for an error-inducing input.          */
3  randomErrMax = 0; inBase = NULL;
4  rStart = time();
5  while time() - rStart < τ0 do
6      generate a random input in the input domain : in;
7      err = compute-error(in, P);
8      if err > eMax then
9          randomErrMax = err; inBase = in;
10     end
11 end
12 /* Separate the input variables into operands.         */
13 op1Vars, op2Vars = partition-variables(inVars, inSpecs);
14 /* Initilization.                                      */
15 errMax = randomErrMax; errInput = inBase;
16 concVars = op1Vars; symbVars = op2Vars;
17 while time() - tStart < tBudget do
18     /* Concretize convVars using base values.         */
19     concretize the variables in convVars using inBase;
20     symbolize the variables in symbVars;
21     STAT, sInputs = symbolic-execution(convVars, symbVars, P', τ1);
22     if STAT = timeout then
23         sNum = length(symbVars);
24         s1Vars, s2Vars = random-divide(symbVars, sNum/2, sNum/2);
25         convVars = convVars + s1Vars;
26         symbVars = s2Vars;
27     else
28         for sInput in sInputs do
29             err = compute-error(join-input(sInput, inBase), P);
30             if err > errMax then
31                 errMax = err; errInput ← join-input(sInput, inBase);
32             end
33         end
34         sNum = length(symbVars); lop2=length(op2Vars);
35         s1Vars, s2Vars = random-divide(op2Vars, lop2-sNum, sNum);
36         convVars = op1Vars + s1Vars;
37         symbVars = s2Vars;
38     end
39 end
40 return errMax, errInput
```

Algorithm 1: Symbolic execution with concretization.

as our symbolic execution engine. KLEE models the environment to explore all legal values while ensuring the accuracy of the program state, maintains memory efficiently to allow exploring as many as hundreds of thousands of paths simultaneously, and provides a set of heuristic search strategies that users can select from. More importantly, it has an extension, KLEE-Float [3], which provides support for floating-point arithmetic and thus enables symbolic execution of floating-point programs.

Symbolic execution allows program inputs to be represented as symbols. The program is then interpreted using these symbolic values rather than concrete inputs. In the execution of a conditional statement, KLEE forks the current process into two, and each child process updates its program state by adding the branch constraints over the input symbols into its path constraints. Path constraints are solved when the program terminates and input symbols are concretized to specific values that exercise the given path. One of the main challenges faced by symbolic execution is path explosion. The number of feasible paths grows exponentially with the size of the program. Unfortunately, injecting inaccuracy checks exacerbates path explosion. To alleviate this problem, it is important to manage the number of symbolic variables. We concretize input variables prior to symbolic execution and find that it significantly reduces the number of paths to explore, making symbolic execution for our transformed programs practical.

First, we refer to input variables as scalars. The array and matrix input variables are broken down into multiple scalar input variables for concretization and symbolization, discussed in the rest of this section. Concretizing input variables to manage the number of symbolic variables is critical for the application of symbolic execution. First, symbolizing all input variables is redundant since symbolizing only one of the two operands in an operation suffices. Take the operation $x + y$ as an example. To trigger a cancellation in the operation, it is sufficient to symbolize either variable x or y, and the value of the other can be arbitrary. The concretization of redundant symbolic variables is effective in speeding up the constraint solver behind symbolic execution. Second, besides the redundant input variables, we randomly select input variables for concretization in order to perform symbolic execution.

Algorithm 1 describes the procedure of symbolic execution with concretization. Given program P, program P' with inaccuracy check injections, input variables, and input specifications that describe the input variables and how they are related in the computation, our algorithm returns the maximum error triggered in program P and the corresponding error-inducing input within a time budget. Specifically, we first conduct a random search over all input variables (line 2-11) and the input that triggers the highest error is kept as *base values* of the input variables for future concretization. We then partition the input variables into two groups based on the input specifications so that two operand variables are separated into different groups (line 13). Take matrix multiplication (MM) as an example, which performs multiplication over two matrix input variables. Each of the matrices is an operand of the multiplication, and in our operand partition, the two matrix entries are separated as $op1Vars$ and $op2Vars$.

Lastly, we perform symbolic execution with concretization to maximize the numerical error in program P (line 14-39). We first initialize the maximum error and the corresponding input using the result of random search (line 15), and then update them every time a higher error is triggered (line 30-32). The input variables are divided into concrete variables (shown as *concVars*) and symbolic variables (*symbVars*). Concrete variables use the corresponding concrete values from the base input (*inBase*) (line 19), and symbolic variables are declared as symbols (line 20).

The symbolic execution engine is invoked on the injected program P' with an execution time threshold τ_1 (line 21). If symbolic execution does not terminate within the time threshold τ_1, we reduce the number of symbolic variables by half (line 23-26) and

repeat.[7] The initial number of symbolic variables is the number of the operand variables with smaller size (line 16). If the symbolic execution engine terminates, we examine each input it generates by computing the numerical error each input triggers and update the largest error to the maximum error *errMax* and error-inducing input *errInput* (line 28-33). Finally, we shuffle the concrete and symbolic variables in the operand variable *op2Vars* (line 34-37) and repeat the above procedure until time is up.

3.3 Error Measurement

As discussed in the previous section, the inputs generated by random search and symbolic execution are evaluated by computing the numerical error they trigger (Algorithm 1, lines 7 and 29). To measure the error, we transform the program into higher precision (e.g., 128-bit precision). We compare the result produced by the original program against the one from the high-precision program. Moreover, the error is represented by the relative error of the two program results, i.e., $|r - r_0| / \max\{FLT\_MIN, |r_0|\}$ where r is the result produced by the original program, r_0 is the result of the transformed program in high precision, and *FLT_MIN* indicates the minimum representable positive floating-point number in FLOAT precision. In addition, we print the diagnosis information that contains the log of precision losses and cancellations and the corresponding code area an inaccuracy event occurs. This can help the programmer in identifying the program expressions that contribute the most to the numerical inaccuracy.

4 EXPERIMENTAL EVALUATION

We implemented our algorithm in a tool named FPGen.[8] FPGen includes a floating-point computation analyzer for C programs implemented using LibTooling [4]. The analyzer yields a list of code sites, i.e., statements that contain floating-point addition/subtraction operations located within loops, to select as inaccuracy injection sites. FPGen then performs symbolic execution with concretization on the transformed program to maximize the error in the result. We use KLEE-Float as the symbolic execution engine, which is built to run on LLVM [26] bitcode files.

In the evaluation of FPGen, all experiments were run on a workstation Intel(R) Xeon(R) Gold 6238 CPU (8 cores, 2.10GHz), 32GB RAM, and the operating system is Ubuntu 14.04.5 LTS.

The goal of this evaluation is to answer the following questions:

RQ1 How effective is FPGen at finding error-inducing inputs?

RQ2 How does FPGen compare to random input generation, the state-of-the-art tool S3FP, and KLEE-Float?

Benchmarks. We evaluate FPGen on the 3 summation algorithms described in Section 2, 9 matrix computation routines from the Meschach library [5], and 9 *unique* statistics routines from the GNU Scientific library (GSL) [2]. Meschach provides a series of basic computation routines on matrices and vectors in C. The routine `sum` adds the elements of a vector. As their names indicate, `1-norm` and `2-norm` compute the 1-norm and 2-norm of a vector, and routines `dot` and `convolution` calculate the dot product and convolution product of two vectors, respectively. `MV` multiplies a matrix by a vector, and `MM` multiplies two matrices. `LU` and `QR` factor a matrix to different forms. GSL provides a wide range of mathematical routines written in C and C++, and has been used for evaluation in prior work [37–39]. Specifically, we use the GSL statistics routines that take array data as input.[9] These routines compute the mean, variance, standard deviation and more advanced statistical terms such as absolute deviations, skewness and kurtosis for weighted samples. The functions mainly take two input arrays, one as the samples and one being the associated weights.

Experimental Setup. Table 2 presents the benchmarks and their input characteristics including kind of input (the number in the parentheses indicates the size of each input kind), size of symbolized input and size of concretized input in both the initial and final configurations of FPGen. As described in Algorithm 1, the initial partition of symbolized and concretized inputs is based on the input operands. For the summation programs, `MM`, `LU`, and `QR`, half of the elements of an array/matrix operand are symbolized and the rest of the input data is concretized. For the remaining programs, all elements of an array/vector operand are symbolized and all elements of the other operand are concretized. Moreover, the final configuration on size of symbolized and concretized inputs indicates the partition in which the best relative error is observed.[10]

With regard to the time threshold parameters τ_0 and τ_1 described in Algorithm 1, we use $\tau_0 = 0, \tau_1 = 30min$ for summation programs, $\tau_0 = 10min, \tau_1 = 55min$ for Meschach programs and $\tau_0 = 20min, \tau_1 = 33min$ for GSL programs. Lastly, all benchmarks use DOUBLE precision,[11] and we inject inaccuracy checks into the last addition/subtraction operation that updates the accumulator in the core loop of each program. The generated inputs are FLOAT numbers in $[-100, 100]$ to facilitate comparison with S3FP, which operates on FLOAT numbers and requires an input range.

Baselines. We compare FPGen to (1) a random input generator we implemented in C++, (2) S3FP, the state-of-the-art floating-point error-inducing input generator for programs with multi-dimensional floating-point input, and (3) KLEE-Float, the floating-point symbolic execution engine used by FPGen. We use the default parameter settings for S3FP: C_{init} is randomized while parameters k and N_{part} are set to the value 1.

Error Measurement. The ground truth for our benchmarks is obtained by running higher-precision implementations of the programs on the generated inputs. Specifically, we implemented summations that use 128-bit precision, and perform LONG DOUBLE precision (80-bit extended precision) for Meschach and GSL routines. Meschach and GSL support compilation in DOUBLE and LONG

[7] In the experiments discussed in Section 4, we use "–max-time=τ_1" to halt the execution of the symbolic execution engine when time is up according to the threshold τ_1, and check whether it has ever reached any error injections and thus triggered errors with incomplete execution. If it reached error injections and triggered errors within time threshold τ_1, it is considered as an effective termination, otherwise considered as non-termination (i.e., requiring more time to explore the error paths).

[8] The source of FPGen is available on GitHub: https://github.com/ucd-plse/FPGen

[9] There are a total of 15 floating-point statistics routines in GSL, however, from the standpoint of symbolic execution, 6 of them (`wvariance-m`, `wsd-m`, `wtss-m`, `wabsdev-m`, `wskew-m`, and `wkurtosis-m`) are a replicate of 6 other routines (`wvariance`, `wsd`, `wtss`, `wabsdev`, `wskew`, and `wkurtosis`). We only report results for 9 distinct routines, but the results for all 15 GSL statistics routines are available for full reference.

[10] The final input size does not indicate the size of symbolized and concretized input in the last partition of the search. For some programs, further partitions yield smaller errors.

[11] GSL routines use LONG DOUBLE for the accumulators, and we manually modified them to be in consistent precision with the samples, i.e., DOUBLE in the experiments. We believe the change will not cause any overflow exceptions since all samples and their associated weights are in a specific input range.

Table 2: Input characteristics of benchmarks.

Benchmark(s)	Input Kind	Initial Input Size		Final Input Size	
		Symbolized	Concretized	Symbolized	Concretized
Summations	array(32)	16	16	16	16
`sum`, `2-norm`	vector(4)	4	0	4	0
`1-norm`	vector(4)	4	0	2	2
`dot`, `convolution`	2 vectors(4)	4	4	2	6
`MV`	vector(4), matrix(4×4)	4	16	2	18
`MM`	2 matrices(4×4)	8	24	8	24
`LU`	matrix(4×4)	8	8	8	8
`QR`	matrix(4×4)	8	8	4	12
`wmean, wvariance-w, wsd-w, wtss[-m]`	2 arrays(4)	4	4	4	4
`wabsdev[-m], wskew[-m], wkurtosis[-m]`	2 arrays(4)	4	4	4	4
`wvariance[-m], wsd[-m]`	2 arrays(4)	4	4	2	6

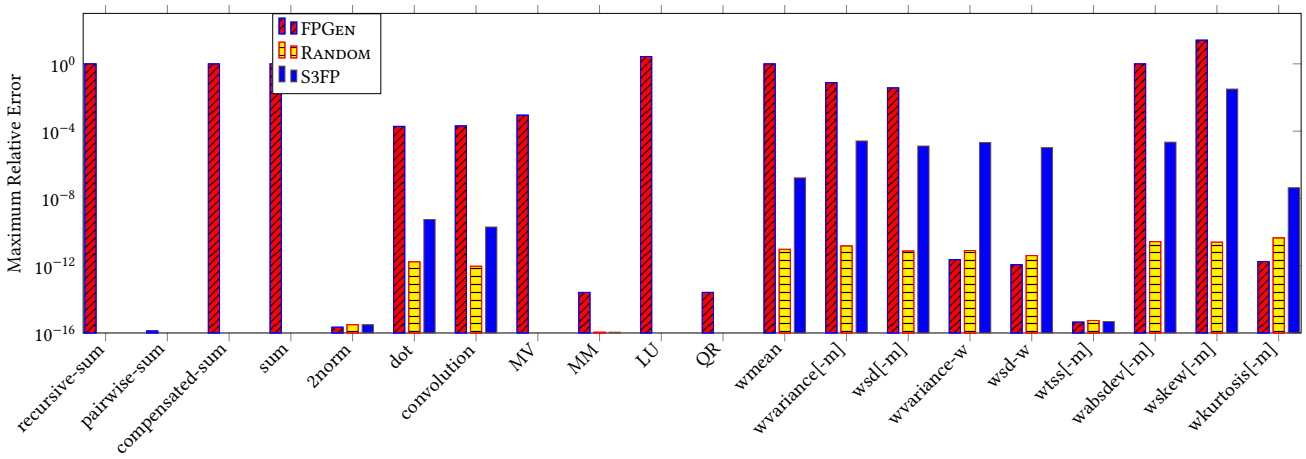


Figure 7: Comparison of maximum errors triggered by the error-inducing input generators.

double precision.[12] Note that all benchmarks are transformed to higher precision at the source code level, and we did not observe precision-specific operations that can potentially cause errors in the transformation [36]. For all our benchmarks, we calculate the relative error of the result produced by the original program with respect to the ground truth. Note that five of our benchmarks produce vectors or matrices as final result. In these cases, we report the maximum relative error observed across all elements.[13]

Experimental Results. We evaluate FPGen on the given 21 benchmarks, and compare it to (1) random input generation (referred to as Random), (2) the state-of-the-art input generator S3FP, and (3) KLEE-Float. For all experiments we consider a time budget of 2 hours. The results are shown in Table 3. Column "Rel. Error" indicates the maximum relative error triggered by generated inputs (the largest error triggered among the four tools is shown in bold), "# Inputs" denotes the total number of generated inputs , and "hh:mm:ss" describes the execution time. As shown, KLEE-Float is not able to trigger numerical errors on its own as it simply searches for inputs that cover program paths. Among the three error-inducing input generators, FPGen generates error-inducing inputs for 20 out of 21 benchmarks while the inputs generated by Random and S3FP trigger an error in 13 out of 21 programs. As shown in the first four rows in Table 3, Random and S3FP explored over five hundred thousand input arrays/vectors but failed to find one that exposes an error for the 3 summation programs and 5 Meschach routines. FPGen, however, triggered errors in these programs, except for `1-norm`, after exploring significantly fewer inputs within the time budget. Furthermore, the numerical errors triggered by FPGen are up to 1.0, which are comparable in order of magnitude to the errors triggered by the hand crafted input from Section 2.

Figure 7 visualizes the maximum relative error each error-inducing input generator triggered for all benchmarks except `1-norm` for which none of the generators triggered an error. The Y axis that indicates the maximum relative error triggered in each benchmark is proportional to the logarithm of the errors. As shown, FPGen

[12] The support of long double precision in Meschach is incomplete, and we manually adjusted few header files.

[13] S3FP only aims on triggering high error for one single output number. For vector/-matrix output, we adopts the same methodology described in the paper [16] which reports the relative error for the output element whose computation requires the highest number of floating-point operations. If all output elements involve the same number of operations, it reports the relative error of the first output element.

Table 3: Accuracy testing results for numerical library routines.

	Recursive Summation (32)			Pairwise Summation (32)			Compensated Summation (32)		
	Rel. Error	#Inputs	hh:mm:ss	Rel. Error	#Inputs	hh:mm:ss	Rel. Error	#Inputs	hh:mm:ss
Random	0.0000e+00	583704	02:00:00	0.0000e+00	579979	02:00:00	0.0000e+00	533939	02:00:00
S3FP	0.0000e+00	577227	02:00:00	0.0000e+00	551594	02:00:00	0.0000e+00	550118	02:00:00
KLEE-Float	0.0000e+00	1	≤00:00:01	0.0000e+00	1	≤00:00:01	0.0000e+00	1	≤00:00:01
FPGen	**1.0000e+00**	9472	02:00:00	**1.3174e-16**	1532	02:00:00	**1.0000e+00**	547	02:00:00
	`sum`			`1-norm`			`2-norm`		
	Rel. Error	#Inputs	hh:mm:ss	Rel. Error	#Inputs	hh:mm:ss	Rel. Error	#Inputs	hh:mm:ss
Random	0.0000e+00	542842	02:00:00	0.0000e+00	550180	02:00:00	**3.1216e-16**	544735	02:00:00
S3FP	0.0000e+00	550353	02:00:00	0.0000e+00	549360	02:00:00	**3.1170e-16**	542879	02:00:00
KLEE-Float	0.0000e+00	1	≤00:00:01	0.0000e+00	1	≤00:00:01	0.0000e+00	1	≤00:00:01
FPGen	**1.0000e+00**	43055	02:00:00	0.0000e+00	41690	02:00:00	**2.2117e-16**	41039	02:00:00
	`dot`			`convolution`			`MV`		
	Rel. Error	#Inputs	hh:mm:ss	Rel. Error	#Inputs	hh:mm:ss	Rel. Error	#Inputs	hh:mm:ss
Random	1.7010e-12	587409	02:00:00	9.2803e-13	561780	02:00:00	0.0000e+00	562187	02:00:00
S3FP	5.5831e-10	541171	02:00:00	1.9864e-10	529503	02:00:00	0.0000e+00	559708	02:00:00
KLEE-Float	0.0000e+00	1	≤00:00:01	0.0000e+00	1	≤00:00:01	0.0000e+00	1	≤00:00:01
FPGen	**1.9190e-04**	43649	02:00:00	**2.0446e-04**	42099	02:00:00	**8.9366e-04**	41180	02:00:00
	`MM`			`LU`			`QR`		
	Rel. Error	#Inputs	hh:mm:ss	Rel. Error	#Inputs	hh:mm:ss	Rel. Error	#Inputs	hh:mm:ss
Random	1.1102e-16	587108	02:00:00	0.0000e+00	544796	02:00:00	0.0000e+00	551384	02:00:00
S3FP	1.1102e-16	530526	02:00:00	0.0000e+00	543181	02:00:00	0.0000e+00	502424	02:00:00
KLEE-Float	0.0000e+00	1	≤00:00:01	0.0000e+00	9	00:01:20	0.0000e+00	24	02:24:51
FPGen	**2.5783e-14**	43965	02:00:00	**2.7327e+00**	40831	02:00:00	**2.5912e-14**	40944	02:00:00
	`wmean`			`wvariance (wvariance-m)`			`wsd (wsd-m)`		
	Rel. Error	#Inputs	hh:mm:ss	Rel. Error	#Inputs	hh:mm:ss	Rel. Error	#Inputs	hh:mm:ss
Random	9.4290e-12	526315	02:00:00	1.5039e-11	528128	02:00:00	7.5193e-12	529821	02:00:00
S3FP	1.6620e-07	526118	02:00:00	2.5955e-05	528292	02:00:00	1.2977e-05	535576	02:00:00
KLEE-Float	0.0000e+00	1	≤00:00:01	0.0000e+00	16	00:00:25	0.0000e+00	16	00:00:25
FPGen	**1.0000e+00**	89844	02:00:00	**7.6280e-02**	89221	02:00:00	**3.7439e-02**	88883	02:00:00
	`wvariance-w`			`wsd-w`			`wtss (wtss-m)`		
	Rel. Error	#Inputs	hh:mm:ss	Rel. Error	#Inputs	hh:mm:ss	Rel. Error	#Inputs	hh:mm:ss
Random	7.9593e-12	529220	02:00:00	3.9797e-12	531602	02:00:00	**5.5294e-16**	526324	02:00:00
S3FP	**2.0918e-05**	531397	02:00:00	**1.0459e-05**	528545	02:00:00	**4.7739e-16**	526869	02:00:00
KLEE-Float	0.0000e+00	16	00:00:25	0.0000e+00	16	00:00:25	0.0000e+00	16	00:00:25
FPGen	2.2858e-12	90107	02:00:00	1.1429e-12	89057	02:00:00	**4.4513e-16**	89318	02:00:00
	`wabsdev (wabsdev-m)`			`wskew (wskew-m)`			`wkurtosis (wkurtosis-m)`		
	Rel. Error	#Inputs	hh:mm:ss	Rel. Error	#Inputs	hh:mm:ss	Rel. Error	#Inputs	hh:mm:ss
Random	2.6840e-11	535959	02:00:00	2.5025e-11	497012	02:00:00	4.5107e-11	499180	02:00:00
S3FP	2.2077e-05	535286	02:00:00	3.1646e-02	507847	02:00:00	**4.3139e-08**	473608	02:00:00
KLEE-Float	0.0000e+00	16	00:00:25	0.0000e+00	1	≤00:00:01	0.0000e+00	16	00:00:24
FPGen	**1.0000e+00**	44041	02:00:00	**2.5675e+01**	89715	02:00:00	1.7733e-12	89794	02:00:00

outperforms Random and S3FP for 15 out of 20 benchmarks. In 2 of the other benchmarks, the three approaches are comparable to each other, and the order of magnitude of the errors are −16. For the remaining 3 benchmarks (from GSL), FPGen failed to reach an error path within the time budget and S3FP triggered the largest error among the input generators through black-box search. It requires future improvements on symbolic execution to assist FPGen reach more error paths, and thus trigger larger errors for these programs.

In summary, using precision loss and cancellation checks is effective in finding high error-inducing inputs especially for numerical programs with multi-dimensional input data. From the evaluation on the summation benchmarks, the matrix computation library Meschach, and the GSL statistics functions, FPGen significantly outperforms the state of the art.

RQ1: FPGen proves to be effective at finding error-inducing inputs by triggering errors in 20 out of 21 benchmarks and the errors are -6.35 on average in the order of magnitude.
RQ2: FPGen significantly outperforms the state of the art by triggering errors in 33% more programs while errors are more than 2 orders of magnitude larger on average.

Specifically, FPGen generated error-inducing inputs for 20 benchmark programs while the state-of-the-art generators trigger an error for 13 out of 21 programs. Moreover, regarding the maximum errors triggered by the generated inputs, FPGen (-6.35 on average in 20 programs) improves S3FP (-8.46 on average in 13 programs) by over two orders of magnitude. The order of magnitude of the maximum errors triggered by Random in 13 programs is -12.69 on average.

Discussion. The error-inducing inputs generated by FPGen can be used in many contexts including floating-point precision tuning and compiler testing for floating-point optimizations. Dynamic precision tuning (e.g., [31]) lowers precision while satisfying an accuracy constraint. Such approaches tune the programs with respect to a given test set. Augmenting such test sets with inputs generated by FPGen could lead to more robust precision optimizations. Moreover, compilers transform code for optimizations but the transformation is risky for floating-point code because floating-point arithmetic does not satisfy associative and distributive laws. To enhance the compiler optimizations for floating-point code, inputs that maximize the numerical error are required for testing.

With regard to the limitation of our tool, first, FPGen requires a specification for inaccuracy check injection (we used core loops in this paper). It remains future work to identify other code areas to inject inaccuracy checks. Second, we mainly rely on optimizations such as concretization to manage the number of symbolic variables to alleviate the scalability problem symbolic execution faces. In the future, it would be interesting to complement our work using techniques to speedup symbolic execution [8, 35].

5 RELATED WORK

Floating-Point Test Data Generation. S3FP [16], the state-of-the-art error-inducing input generator for numerical programs with multi-dimensional input data, is black-box. S3FP iteratively divides the search range of each input variable into two and permutes them randomly to generate a tighter search space. The tool evaluates each subspace by sampling inputs and selecting one for further exploration. The black-box nature of S3FP indicates that it is not as effective as FPGen when the input space becomes large. Other error-inducing input generators, i.e., LSGA [39], EAGT [37] and AutoRNP [37] target numerical program with few scalar inputs. LSGA uses a genetic algorithm to evolve the exponent of the inputs, however, it does not provide an algorithm in evolving multi-dimensional floating-point inputs, and the tool is not publicly available. EAGT and AutoRNP compute the approximation of the condition number in selecting inputs, and focus on programs with one scalar input.

FPSE [9] and CoverMe [19] generate floating-point test inputs that maximize *code coverage*. FPSE [9] adopts a number of search heuristics to solve path conditions containing floating-point computations. CoverMe [19] translates the problem of covering a new branch in the floating-point code into a mathematical problem that can be solved by applying unconstrained programming. Such efforts are complementary to our testing approach, and can be adopted to enhance our symbolic-execution based approach.

Floating-point input generation tools have been developed to detect other specific problems. Chiang et al. [17] detect path divergence between a floating-point program and its high precision execution. Barr et al. [11] use symbolic execution to detect floating-point exceptions such as overflows and underflows. They also perform a transformation on the numerical program, and symbolically execute the transformed program to identify inputs that trigger an exception. The transformation, however, focuses on injecting exception checks before a floating-point operation, which is different from ours. Moreover, the transformed program is symbolically executed using real arithmetic while we use floating-point arithmetic.

Floating-Point Dynamic Analysis. Benz et al. [12] perform every floating-point computation side by side in higher precision and track the propagation of errors to detect accuracy problems. Lam et al. [25] conduct binary instrumentation on floating-point additions and subtractions to detect cancellations. They analyze the exponents of the operands and the result of the instrumented operations to determine the severity of a cancellation and report stack information for severe cancellations. Besides the runtime detection of mathematical cancellations, Bao and Zhang [10] propose to track the propagation of the cancellation error, which can be suppressed or inflated in the subsequent execution. Both cancellation detection techniques [10, 25] use the cancellation check equation described in our paper. However, their main purpose is to detect cancellation issues for existing inputs and cannot generate error-inducing inputs. Furthermore, it is important to combine precision loss with cancellation in the generation of error-inducing inputs. To the best of our knowledge, we are the first to present such an approach.

RAIVE [27] performs floating-point computation with a vector of values to capture rounding errors and report output variations. Similarly, Tang et al. [34] perturb the underlying numerical values and expressions to uncover instability problems in numerical code. Such dynamic analyses detect accuracy problems on given input data. Moreover, a large number of dynamic techniques (e.g., [21, 24, 29–32]) optimize floating-point code using a given input set. All of these techniques could benefit from the inputs FPGen generates.

6 CONCLUSION

We presented an approach to effectively generate floating-point inputs that trigger large errors. First, we formulated two inaccuracy checks for large precision loss and cancellation. The injection of inaccuracy checks after floating-point computation enables symbolic execution to explore specialized branches that cause numerical inaccuracy, which can lead to large errors in the final result. Second, we proposed optimizations to alleviate path explosion. In particular, this was achieved by strategically reducing the number of symbolic variables via concretization. We implemented our algorithm in a tool named FPGen, and presented an evaluation on 21 numerical programs including matrix computation and statistics libraries. Our results show that FPGen is able to expose errors for 20 of the evaluated programs while the state-of-the-art error-inducing input generator S3FP only triggers errors for 13 out of 21 programs. Moreover, FPGen triggered an error as large as 10^{-6} on average while the maximum error S3FP triggered is about 10^{-8} on average.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under award CCF-1750983, and by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under award DE-SC0020286.

REFERENCES

[1] Accessed: 2020-01-01. The Explosion of the Ariane 5. https://www.ima.umn.edu/~arnold/disasters/ariane.html.

[2] Accessed: 2020-01-01. GSL- GNU Scientific Library. https://www.gnu.org/software/gsl/.

[3] Accessed: 2020-01-01. KLEE with floating point support. https://github.com/srg-imperial/klee-float.

[4] Accessed: 2020-01-01. LibTooling. https://clang.llvm.org/docs/LibTooling.html.

[5] Accessed: 2020-01-01. Meschach Library. https://www.netlib.org/c/meschach/readme.

[6] Accessed: 2020-01-01. Toyota: Software to blame for Prius brake problems. http://www.cnn.com/2010/WORLD/asiapcf/02/04/japan.prius.complaints/index.html.

[7] Accessed: 2020-01-01. Z3. https://github.com/Z3Prover/z3.

[8] Eman Alatawi, Tim Miller, and Harald Sondergaard. 2018. Symbolic Execution with Invariant Inlay: Evaluating the Potential. In *2018 25th Australasian Software Engineering Conference, ASWEC 2018*.

[9] Roberto Bagnara, Matthieu Carlier, Roberta Gori, and Arnaud Gotlieb. 2013. Symbolic Path-Oriented Test Data Generation for Floating-Point Programs. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013*.

[10] Tao Bao and Xiangyu Zhang. 2013. On-the-fly detection of instability problems in floating-point program execution. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013*.

[11] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic Detection of Floating-point Exceptions. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013*.

[12] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A Dynamic Program Analysis to Find Floating-point Accuracy Problems. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012*.

[13] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI 2008*.

[14] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2008. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.* 12, 2, Article 10 (Dec. 2008), 38 pages.

[15] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90.

[16] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient search for inputs causing high floating-point errors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2014*.

[17] Wei-Fan Chiang, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2015. Practical floating-point divergence detection. In *International Workshop on Languages and Compilers for Parallel Computing*.

[18] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*.

[19] Zhoulai Fu and Zhendong Su. 2017. Achieving high coverage for floating-point code via unconstrained programming. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*.

[20] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. *SIGPLAN Not.* 40, 6 (June 2005), 213–223.

[21] Hui Guo and Cindy Rubio-González. 2018. Exploiting Community Structure for Floating-point Precision Tuning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018*.

[22] Nicholas J Higham. 1993. The accuracy of floating point summation. *SIAM Journal on Scientific Computing* 14, 4 (1993), 783–799.

[23] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394.

[24] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. LeGendre. 2013. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th international ACM conference on International conference on supercomputing, ICS 2013*.

[25] Michael O. Lam, Jeffrey K. Hollingsworth, and G. W. Stewart. 2013. Dynamic floating-point cancellation detection. *Parallel Comput.* (2013).

[26] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO 2004*.

[27] Wen-Chuan Lee, Tao Bao, Yunhui Zheng, Xiangyu Zhang, Keval Vora, and Rajiv Gupta. 2015. RAIVE: Runtime Assessment of Floating-point Instability by Vectorization. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015*.

[28] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F Donaldson, Rafael Zahl, and Klaus Wehrle. 2017. Floating-point symbolic execution: A case study in N-version programming. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*.

[29] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015*.

[30] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H. Bailey, and David Hough. 2016. Floating-point precision tuning using blame analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*.

[31] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: tuning assistant for floating-point precision. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2013*.

[32] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014*.

[33] Robert Skeel. 1992. Roundoff error and the Patriot missile. *SIAM News* (1992).

[34] Enyi Tang, Earl Barr, Xuandong Li, and Zhendong Su. 2010. Perturbing Numerical Calculations for Statistical Analysis of Floating-point Program (in)Stability. In *Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA 2010*.

[35] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped Symbolic Execution. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018*.

[36] Ran Wang, Daming Zou, Xinrui He, Yingfei Xiong, Lu Zhang, and Gang Huang. 2016. Detecting and Fixing Precision-specific Operations for Measuring Floating-point Errors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016*.

[37] Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2017. Efficient global search for inputs triggering high floating-point inaccuracies. In *2017 24th Asia-Pacific Software Engineering Conference, APSEC 2017*.

[38] Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2019. Efficient automated repair of high floating-point errors in numerical libraries. *Proceedings of the ACM on Programming Languages, POPL* (2019).

[39] Daming Zou, Ran Wang, Yingfei Xiong, Lu Zhang, Zhendong Su, and Hong Mei. 2015. A Genetic Algorithm for Detecting Significant Floating-Point Inaccuracies. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015*.