NUMFUZZ: A Floating-Point Format Aware Fuzzer for Numerical Programs

Chenghu Ma[†‡] Liqian Chen[†✉] Xin Yi[†] Guangsheng Fan[†‡] Ji Wang[†‡]
[†]College of Computer, National University of Defense Technology, Changsha, China
[‡]HPCL, National University of Defense Technology, Changsha, China
{machenghu, lqchen, yixin09, guangshengfan, wj}@nudt.edu.cn

***Abstract*—It is difficult to write a numerical program that does not incur floating-point exceptions in practice. To detect floating-point exceptions, most existing methods use static analysis, which may induce false alarms (due to over-approximation), or suffer from scalability issues (since solving floating-point constraints is expensive). Fuzzing is a widely used technique to finding bugs, but existing fuzzing techniques have not yet considered the specific format of floating-point and are lack of guidance for detecting floating-point exceptions.**

In this paper, we propose a floating-point format aware coverage-based grey-box fuzzing to detect floating-point exceptions for numerical programs. More specifically, we propose a novel mutation strategy for floating-point format aiming at producing valid floating-point test inputs. Moreover, we present a new guidance aiming to search for test inputs that are closer to exposing exceptions. We implement our approach as a tool, named NUMFUZZ, based on AFL. We have conducted experiments to evaluate NUMFUZZ on GNU Scientific Library (GSL) and Sun's C math library respectively. The preliminary experimental results suggest that our approach has promising ability in detecting floating-point exceptions and achieving high floating-point branch coverage in real-world numerical programs.

***Index Terms*—Fuzzing, Floating-point exception, Dynamic analysis**

I. INTRODUCTION

Floating-point arithmetic is widely used in software that requires lots of scientific and engineering computing. In modern computers, floating-point numbers are an approximate representation of real numbers using a finite number of bits [1], [2]. Thus, the execution of a floating-point operation may incur exceptions. These floating-point exceptions may lead to disastrous consequences in safety-critical systems such as control systems for aerospace, finance, transportation and medicine, etc. Therefore, it is critical to detect floating-point exceptions in numerical software for ensuring the reliability of the software.

Based on the results of floating-point operations, the IEEE 754 standard [1] defines five types of exceptions, including *Overflow*, *Underflow*, *Inexact*, *Invalid* and *Divide-By-Zero*. An *Overflow* exception is triggered when the result of an operation exceeds the greatest normal floating-point number in magnitude in the current precision, while an *Underflow* exception is triggered when the result falls between zero and the smallest normal floating-point number. An *Inexact* exception is triggered when the result falls between two floating-point numbers, and thus needs rounding. An *Invalid* or *Divide-By-Zero* exception is triggered when the operand of a floating-point operation outsides its domain (e.g., $x < 0.0$ for *sqrt(x)*). To execute program continually when an exception is triggered, the computer may choose to mask the exception and use the special value (*subnormal number*, *infinity* or *NaN*) to represent the result.

In practice, it is difficult to write a numerical program that will not incur floating-point exceptions [2], [3]. Hence, there are several methods [4]–[6] have been proposed to detect floating-point exceptions. Most methods use static analysis based on symbolic execution [7], trying to find an input that triggers a floating-point exception at runtime. However, a real-world numerical program may have tens of thousands of floating-point operations, and rounding errors are pervasive in floating-point arithmetic. Meanwhile, solving floating-point constraints is very expensive using SMT solvers.

Grey-box fuzzing is currently the most popular technique to find vulnerabilities because it is easy to deploy and of good scalability and feasibility [8]–[10]. Usually, fuzzing techniques leverage coverage information as guidance for exploration of different program paths. However, directly using existing grey-box fuzzers for detecting floating-point exceptions does not work, because the floating-point exception does not give rise to a crash in numerical programs. On the other hand, it is quiet difficult to produce test inputs causing floating-point exception in numerical programs for existing coverage-based fuzzing techniques, because there may be a little part of inputs among all possible inputs causing floating-point exceptions in numerical programs. Existing coverage-based fuzzers may cover a statement that is potential to incur exception, but will not deliberately trigger the exception.

In this paper, we present a floating-point format aware fuzzing to automatically detect floating-point exceptions for numerical programs. More specifically, we propose a novel mutation strategy based on floating-point format aiming at enhancing grey-box fuzzing to detect floating-point exceptions by effectively generating valid floating-point test inputs. Moreover, we also introduce a new fitness guidance used within our approach, which explicitly prefers to search for test inputs that are closer to exposing exceptions. We have implemented our approach as a tool, named NUMFUZZ, based on AFL

[*]Liqian Chen is corresponding author.

TABLE I
FLOATING-POINT FORMAT OF SINGLE AND DOUBLE

	Sign	Exponent	Mantissa
32-bit Single	1	8	23
64-bit Double	1	11	52

[11]. NUMFUZZ first instruments neccessary checks for each floating-point operation, which explicitly cause the program abort when the exception is triggered in numerical program. Then, it employs the proposed floating-point format aware mutation strategy and the proposed fitness guidance to run the instrumented program to detect floating-point exceptions. We have evaluated NUMFUZZ on GNU Scientific Library (GSL) and Sun's math library respectively to demonstrate the effectiveness of NUMFUZZ. NUMFUZZ achieves 91.9% of branch coverage on average on Sun's math library, which outperforms AFL and CoverMe [12]. Besides, it has detected 264 floating-point exceptions on 154 interface functions of GSL, which dramatically outperforms AFL.

The rest of the paper is organized as follows. Section II introduces the necessary background of floating-point format and fuzzing technique, while Section III gives an overview of our approach via a motivating example. Section IV presents the technical details of our approach. Section V details the implementation and evaluation of our technique. Section VI discusses related work, and Section VII concludes the paper.

II. PRELIMINARY

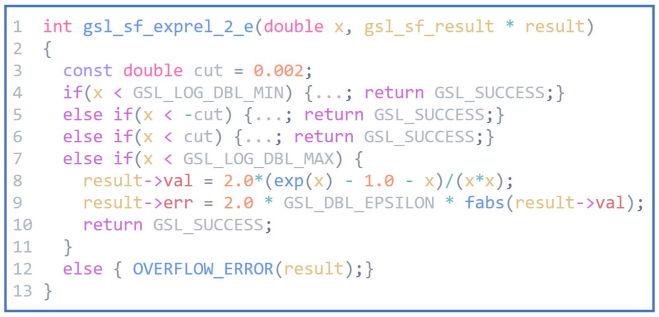
A. Floating-Point Format

Floating-point numbers are an approximate representation of real numbers using a finite number of bits. According to the IEEE 754 standard [1], a floating-point number consists of three parts: sign, exponent, mantissa (also called significand). When all the bits in the exponent of a floating-point representation are all 1, the value of the floating-point number is one of several special values: $+\infty, -\infty$ or NaN (*Not-a-Number*), where NaN denotes exceptions in computation, e.g., divide-by-zero. Otherwise, a floating-point number can be depicted by

$$(-1)^S \times M \times 2^E, \text{ where}$$

- $S \in \{0, 1\}$ is the value of the 1-bit sign, representing a floating-point number is positive (if $S = 0$) or negative (if $S = 1$).
- $E = e - bias$ indicates exponent, where e is the biased unsigned integer with p bits, and $bias = 2^{p-1} - 1$.
- $M = m_0.m_1m_2 \ldots m_n$ indicates mantissa, where the leading bit m_0 is a hidden bit that does not need to be stored, and $m_1m_2 \ldots m_n$ is a n-bit fraction.

Table I shows the numbers of bits of the sign, exponent, and mantissa in a 32-bit single as well as a 64-bit double precision floating-point number according to the IEEE 754 standard.



```
1  int gsl_sf_exprel_2_e(double x, gsl_sf_result * result)
2  {
3    const double cut = 0.002;
4    if(x < GSL_LOG_DBL_MIN) {...; return GSL_SUCCESS;}
5    else if(x < -cut) {...; return GSL_SUCCESS;}
6    else if(x < cut) {...; return GSL_SUCCESS;}
7    else if(x < GSL_LOG_DBL_MAX) {
8      result->val = 2.0*(exp(x) - 1.0 - x)/(x*x);
9      result->err = 2.0 * GSL_DBL_EPSILON * fabs(result->val);
10     return GSL_SUCCESS;
11   }
12   else { OVERFLOW_ERROR(result);}
13 }
```

Fig. 1. Sliced code of *gsl_sf_exprel_2* function.

B. Fuzzing Technique

Fuzzing is an effective software testing technique for finding vulnerabilities in software, which was originally proposed by Miller et al. [13] in 1990s. The primary idea of fuzzing is to create and feed the target program with a large number of test inputs by mutating existing test data. These test inputs are prospective to trigger software vulnerabilities.

A traditional fuzzer consists of four main components: *testcase generator* for generating test cases, *monitor* for monitoring program execution status, *testcase selector* for selecting interesting test cases and *bug detector* for finding potential bugs. The process of fuzzing test typically starts with generating plenty of test cases by *testcase generator* for target program which may be either binary or source code. These test cases will be added into a seed pool. Then the fuzzer takes a test case from the seed pool to feed to target program, and executes the target program. During the execution of target program, the fuzzer leverages the *monitor* to monitor the program state and utilizes the *testcase selector* to determine whether to retain the test case into the seed pool for further mutation according to the feedback information. At the mean time, the fuzzer makes use of the *bug detector* to find potential bugs when the target program crashes or has other abnormal behaviors. The fuzzing loop is repeated until reaching the predefined time limit or stopping by users.

III. OVERVIEW

In this section, we give an overview of our approach via a motivating example to illustrate how our approach detects floating-point exceptions. The example is the *gsl_sf_exprel_2* function[1] from GSL. The sliced source code of the *gsl_sf_exprel_2* function is shown in Fig. 1. The sliced code in Fig. 1 shows that the true branch of the `else if` statement at Line 7 contains some floating-point operations. Among these floating-point operations, there are 2 subtractions (Line 8), 4 multiplications (Lines 8-9), 1 division and 1 *exp()* function call (Line 8), aggregately 8 floating-poinnt operations directly or indirectly involving the input *x*. In the sliced code, `GSL_LOG_DBL_MIN` is about -7.083964e+02 and `GSL_LOG_DBL_MAX` is about 7.097827e+02.

[1]With a few changes for illustration

TABLE II
EXCEPTION-RELATED CHECKERS

Checker	Operator	Pseudocode
_Check_double()	$\odot^*$	/* the checker takes the result of the arithmetic operation as the argument a */ if(fabs(a) > DBL_MAX){ Report_error("Overflow"); abort(); } if(fabs(a) < 0 && fabs(a) < DBL_MIN){ Report_error("Underflow"); abort(); }
_Check_div()	division	/* the checker takes the numerator and denominator of the division operation as arguments a and b */ if(b == 0){ if(a == 0) Report_error("Invalid"); else Report_error("Div-By-Zero"); abort(); } if(fabs(a) > fabs(b)*DBL_MAX){ Report_error("Overflow"); abort(); } if(fabs(a) > 0 && fabs(a) < fabs(b)*DBL_MIN){ Report_error("Underflow"); abort(); }
_Check_sqrt()	sqrt()	/* the checker takes the argument of the function sqrt() as argument a */ if(a < 0){ Report_error("sqrt Invalid"); abort(); }
_Check_log()	log()	/* the checker takes the argument of the function log() as argument a */ if(a < 0){ Report_error("log Invalid"); abort(); } if(a == 0){ Report_error("log Div-By-Zero"); abort(); }
_Check_exp()	exp()	/* the checker takes the argument of the function exp() as argument a */ if(a > log(DBL_MAX)){ Report_error("exp Overflow"); abort(); } if(a < log(DBL_MIN)){ Report_error("exp Underflow"); abort(); }
_Check_pow()	pow()	/* the checker takes the argument of the function pow() as arguments a and b */ if(a == 0 && b <= 0){ Report_error("pow Invalid"); abort(); }

* The operator $\odot \in \{+, -, *\}$

Instrumentation. To detect exceptions over floating-point operations, we first instrument bug *checkers* for each operation which has potential to trigger floating-point exceptions. According to the difference of the operation types, the corresponding bug *checkers* are also different. The details of the bug *checkers* are shown in Table II. Besides, we instrument *ErrBits* function for each floating-point operation that has potential to trigger floating-point exceptions, where the *ErrBits* function is used to measure how close a test case comes to triggering a potential exception at the given exception location. The definition of the *ErrBits* function will be detailed in Section IV-A.

For the motivating example shown in Fig. 1, we instrument the check function *_Check_double()* after the floating-point arithmetic $\odot^2$ operations (Lines 8-9), the check function *_Check_div()* after the floating-point division operation (Line 8) and the check function *_Check_exp()* after the *exp()* function call. The function body implementations of these check functions are shown in Table II. Moreover, we instrument the *ErrBits* function after each floating-point operation in the sliced code of the motivating example. After the instrumentation stage, we will get the instrumented program, and then step into the next stage, i.e., the Fuzzing loop stage.

[2]The operator $\odot \in \{+, -, *\}$

Fuzzing Loop. In this stage, our fuzzer NUMFUZZ takes the instrumented program and the initial seeds as inputs. Differently from the traditional grey-box fuzzers, we propose a novel mutation strategy for floating-point format, aiming to generate valid floating-point test inputs. We will detail our mutation strategy in Section IV-B. NUMFUZZ generates new test inputs utilizing the mutation strategy we proposed, and feeds them to the instrumented program. Then NUMFUZZ collects the branch coverage information and *ErrBits* information as guidance for whether retaining the test inputs as the interesting inputs for further mutation. If the test inputs are closer to the exceptions or lead to new branch coverage, they are saved as interesting test inputs for further mutation. NUMFUZZ repeats

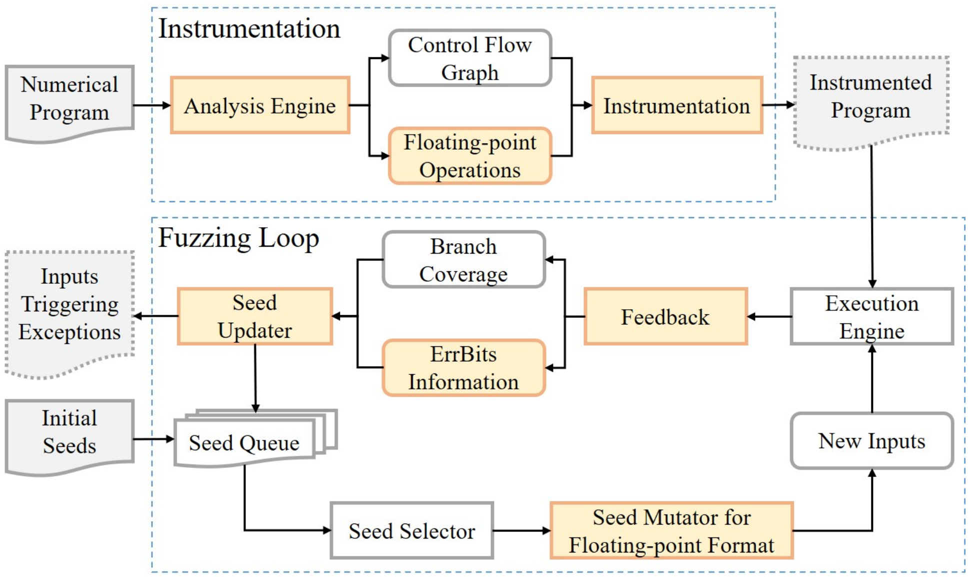


Fig. 2. The architecture of NUMFUZZ.

the fuzzing loop until reaching the predefined time limit or stopped by users.

For the sake of illustration of the example in Fig. 1, we suppose that the initial seed value of *x* is 0.5. At Line 7 in Fig. 1, the program executes the true branch of the last `else if` statement. Based on the mutation strategy we proposed for floating-point format, NUMFUZZ can easily generate a new valid test input i_1. Suppose i_1 is 2.56e+02. Then the input i_1 still hits the true branch of the last `else if` statement, i.e., none of new branches have been covered. At this time, i_1 would be discarded by the traditional coverage-based grey-box fuzzers, which misses the chance to generate a new input which is hopeful to expose exceptions. However, the *ErrBits* information guidance of NUMFUZZ considers that i_1 is closer to the *Overflow* exception (the result of the first multiplication operation in expression at Line 8 is closer to the *Overflow* exception), and thus retains it as an interesting input. When i_1 is further mutated, NUMFUZZ may generate a new valid input (e.g., i_2 = 5.120004e+02) that is closer to the *Overflow* exception. After several mutations, NUMFUZZ generates an exception-trigger input (i_e = 7.094447e+02) that triggers the *Overflow* exception in the first multiplication operation of the expression at Line 8. Note that traditional coverage-based grey-box fuzzers can hardly discover the *Overflow* exception, because they may generate inputs that cover those statements which have potential to trigger exception, but can hardly generate inputs triggering such exception. This is due to the fact that the mutated inputs do not incur new branch coverage and due to the lack of guidance for exception triggering.

IV. APPROACH

In this section, we present the details of our approach. Figure 2 depicts the architecture of NUMFUZZ, which contains two main stages: *instrumentation* and *fuzzing loop*. We detail each stage in the subsections.

A. Instrumentation

For the given target numerical program, we first conduct static analysis to obtain two kinds of information: *control flow graph* and *floating-point operations*. Then we use the information to help NUMFUZZ to decide where to instrument the numerical target program. Based on the *control flow graph* information of the numerical target, NUMFUZZ instruments the program to record branch coverage for guidance of program path explorations. In addition, based on the *floating-point operations* information, we instrument the program to explicitly check each floating-point operation for floating-point exceptions and collect *ErrBits* information for guiding the fuzzing process towards triggering more floating-point exceptions.

Control Flow Graph. Similarly to traditional coverage-based grey-box fuzzers (e.g., AFL), NUMFUZZ leverages the *control flow graph* information of the numerical program to collect the branch coverage information for guidance of program path explorations. It instruments every branch for collecting the branch coverage during runtime.

Floating-point Operations. We instrument bug *checkers* and *ErrBits* function for each floating-point operation. To give rise to a crash during running a program when the floating-point exception occurs, we instrument the corresponding bug *checker* for each floating-point operation (see Table II), which facilitates NUMFUZZ to check and find exception-triggering inputs. After the instrumentation, the instrumented program would throw and report an exception when the exception is triggered by the test input, then abort.

We use *ErrBits* function to measure the distance between the result O of a floating-point operation and the exception

Algorithm 1 Floating-point Exception Detection Fuzzing

```
Input: an instrumented program $P$, and a set of initial inputs $I_0$.
Output: test cases $BuggyS$ triggering floating-point exceptions.
1: $BuggyS \leftarrow \emptyset$
2: $CurSet\_ErrBits \leftarrow \emptyset$
3: $SeedQueue \leftarrow I_0$
4: while time not expire do
5:     $s \leftarrow$ SELECT($SeedQueue$)
6:     $s' \leftarrow$ FPMUTATE($s$)
7:     $(trace, CurSet\_ErrBits) \leftarrow$ EXECUTE($s'$)
8:     if TRIGGERCRASH($trace$) then
9:         $BuggyS \leftarrow BuggyS \cup s'$
10:    else
11:        if NEWPATH($trace$) $\vee$ ISCLOSER($CurSet\_ErrBits$) then
12:            $SeedQueue \leftarrow SeedQueue \cup s'$
13:        end if
14:    end if
15: end while
16: return $BuggyS$
```

value E (e.g., E is 0 for division). Following [14], we define the *ErrBits* function as follows:

$$ErrBits\,(O, E) = \log_2 |\{f \in \mathbb{F} \mid \min(O, E) \le f \le \max(O, E)\}|$$

The *ErrBits* function represents (the 2-base logarithm of) the number of floating-point values between the result of a floating-point operation O and the exception value E. Intuitively, if the result of the *ErrBits* function for a floating-point operation over the current program execution is lower than before, the test input is more potential to trigger exception, and thus should be retained as interesting input for further mutation. That is to say, these inputs are hopeful to expose floating-point exceptions in next cycle of mutation. By using the *ErrBits* information as guidance, NUMFUZZ can mutate the test inputs towards exposing the floating-point exceptions in a numerical program.

B. Fuzzing Loop

Algorithm 1 describes the main procedure of NUMFUZZ. First, the algorithm uses SELECT function to select an input s from seed pool *SeedQueue* (Line 5), and mutates it to generate a mutant s' through FPMUTATE function that is proposed to mutate floating-point number (Line 6). At Line 7, NUMFUZZ then executes input s' on the instrumented numerical program, and monitors its execution. If the input s' triggers floating-point exceptions, it is added to output $BuggyS$ (Line 9). Otherwise, NUMFUZZ analyzes its branch coverage and *ErrBits* information (Line 11). If it is closer to floating-point exception or has new branch coverage, it is retained as interesting seed and added into the *SeedQueue* for the further mutation (see Line 12). NUMFUZZ checks whether s' is closer to floating-point exception based on ISCLOSER function (see Section IV-B2) at Line 11. After Algorithm 1 terminates, the exception-triggering test inputs are obtained in $BuggyS$.

As traditional coverage-based grey-box fuzzers, NUMFUZZ leverages the branch coverage information for guidance of program path explorations. NUMFUZZ utilizes the NEWPATH function to consider whether the newly generated test input reaches a new branch or not during runtime. Note that the main difference of this process with tranditional coverage-based greybox fuzzers falls in that NUMFUZZ presents a new mutation strategy for floating-point number and adds *ErrBits* information guidance to retain interesting inputs, which is explained in detail in the following.

1) Mutation Strategy: Considering floating-point format, our strategy mutates sign, exponent, mantissa of a floating-point number respectively. Since the sign bit is 1-bit, we flip it with half probability, i.e., switch between 0 and 1. For exponent and mantissa, we design several mutators as follows.

Bitflip. For exponent, we flip one random bit at a random position or flip a randomly-chosen continuous chuck of n-bits, where n is 2, 3, 4 and 8. The flip operation of mantissa is the same as exponent but adds a new value for n, that is 16.

Arithmetic. For exponent, we add an integer to it, where the integer is from 1 to 32. The opposite mutate operation is subtracting an integer from it. The arithmetic operation of mantissa is the same as exponent but the added or subtracted integer is $2^m - 1$, where m is from 1 to 33.

Havoc. The mechanism of havoc is the same as traditional greybox fuzzers, while the main difference lies in its mutators. For floating-point number, we design several havoc mutators including `bitflip e`, `bitflip m`, `addition e`, `addition m`, `decrease e`, `decrease m`, and `random bit`.

- `bitflip e`: flipping one bit at a random position of the exponent.
- `bitflip m`: flipping one bit at a random position of the mantissa.
- `addition e`: adding an integer to the exponent, where the integer is randomly generated in the range 1 to 32.
- `addition m`: adding an integer to the mantissa, where the integer is $2^p - 1$, wherein p is randomly generated in the range 1 to 33.
- `decrease e`: subtracting an integer from the exponent, where the integer is randomly generated in the range 1 to 32.
- `decrease m`: subtracting an integer from the mantissa, where the integer is $2^p - 1$, wherein p is randomly generated in the range 1 to 33.
- `random bit`: setting a randomly-chosen continuous chuck of n-bits to a random value, where n is 4 for exponent, while n is 8, 16 and 32 for mantissa.

2) ErrBits Metric: First, let us introduce several notations. Let $Set\_ErrBits$ be a global map that retains the least *ErrBits* at each location that has potentail to trigger floating-point exception. Let $CurSet\_ErrBits$ be a map from all exception locations to their corresponding *ErrBits* as attained in the current run. The ISCLOSER() function takes $CurSet\_ErrBits$ from current program execution as argument, compares it with $Set\_ErrBits$ and checks whether the current test input is closer to the exception or not, and updates $Set\_ErrBits$ when needed. Similarly to the hit-count table of AFL [11], NUMFUZZ maintains an exclusive table in shared memory for recording and communicating the *ErrBits* information during runtime. During the program execution, if the test input is closer to the exception based on the *ErrBits* information than other test inputs saved so far, NUMFUZZ would retain it as interesting input for further mutation to generate new test inputs which are hopeful to expose the exception.

To summarize, we adopt two mechanisms of guidance for guiding NUMFUZZ to retain the interesting inputs for further mutation to discover the floating-point exceptions. Test inputs that reach new branch are saved as interesting inputs, which is inherited from the traditional coverage-based fuzzing. The other guidance mechanism is the *ErrBits* information we proposed to use. More specifically, if the test inputs is closer to the exception than other previously generated test inputs, it would be retained as the interesting input for further mutation.

V. EVALUATION

In this section, we present the implementation and evaluation of our approach. We have built a prototype tool named NUMFUZZ based on AFL [11]. NUMFUZZ consists of instrumentation and fuzzing loop components. The instrumentation components is implemented based on the LLVM, while the execution engine of NUMFUZZ is built on top of AFL.

A. Experimental Setup

1) Benchmarks: To measure the performance of NUMFUZZ for achieving high floating-point code coverage and detecting floating-point exceptions, our benchmarks include two sets.

a) Floating-point exception benchmarks: We conduct our experiments to evaluate the detection-performance of floating-point exceptions of NUMFUZZ on subjects chosen from the widely used GNU Scientific Library[3] (GSL), version 2.7.1. The special function package of GSL has about 487 functions but only has 178 interface functions. 154 of these 178 interface functions take and return floating-point numbers, and thus we choose these functions as our experimental subjects. It is not necessary to detect floating-point exceptions for internal functions, because these functions will be called by interface functions and normally are not visible for end users.

b) Floating-point code coverage benchmarks: To measure the performance of NUMFUZZ for achieving high floating-point code coverage, we conduct a set of experiments on subjects chosen from the Freely Distributable Math Library[4] (Fdlibm), version 5.3. Note that we conduct our experiments on subjects following existing work [12] for direct comparisons.

[3]https://www.gnu.org/software/gsl/

[4]http://www.netlib.org/fdlibm/

2) Comparison: We have compared NUMFUZZ with AFL [11] and CoverMe [12] respectively on different benchmarks.

a) Floating-point branch coverage comparison: We compare NUMFUZZ against the start-of-the-art tool named CoverMe and in achieving high coverage for floating-point code. We also compare our tool with AFL. Because CoverMe does not support functions from GSL[5], we did not compare the floating-point branch coverage with it on GSL benchmarks.

b) Floating-point exception detection comparison: We compare our tool with the traditional coverage-based greybox fuzzer, AFL, which integrates a variety of guidance strategies and leverages genetic algorithms to efficiently detect vulnerabilities. For each tested function, we run each fuzzer for 10 minutes, perform each experiment for 3 times, and evaluate their statistical performance. There are also other approaches with aiming to detect floating-point exceptions, including [4], [5] and [6]. However, these tools are not publicly available so we cannot compare NUMFUZZ with them.

We have performed all of our experiments on a machine running the 64-bit Ubuntu 20.04 LTS with a 3.30GHz Intel (R) Core (TM) i9-10940X CPU and 32GB RAM.

B. Evaluation Results

1) Floating-point Branch Coverage Performance: The results of our experiments to evaluate the ability of NUMFUZZ in achieving high floating-point branch coverage are shown in Table III. The column "File" and "Function" indicates the programs and the corresponding functions respectively, which is sorted by their names. The column "#Branches" gives the numbers of branches of the corresponding functions.

The column "Time" gives the spent time in seconds by NUMFUZZ, CoverMe and AFL respectively. We set the spent time of AFL as ten times of the NUMFUZZ time, following existing work [12]. To measure the branch coverage, we employ AFL-cov [15] for NUMFUZZ and AFL, while we utilize the Gnu coverage tool Gcov [16] for Coverme, following existing work [12]. The column "Branch Coverage" shows the branch coverage results of NUMFUZZ, CoverMe and AFL respectively. NUMFUZZ achieves 100% coverage for 17 out of all tested functions, while CoverMe achieves 100% coverage for 10 out of all tested functions and AFL achieves 100% coverage for 2 out of all tested functions. For all tested functions, AFL achieves an average of 82.4% branch coverage, while NUMFUZZ achieves an average of 91.9% branch coverage, as shown in the last row of the Table III. The average improvement is 9.5% between NUMFUZZ and AFL (see the last row). Note that, NUMFUZZ can handle mixed types of input, while CoverMe can not, e.g., the last 8 functions shown in Table III. Except the last 8 functions, NUMFUZZ achieves an average of 91.9% branch coverage, which provides 3.3% coverage improvement on average, compared CoverMe. For total tested functions, the average improvement is 18.1%

[5]All of interface functions from GSL is wrapped by macro processing and some of them invoke other external functions. However, CoverMe only handles the entry function, i.e., it does not handle other external functions invoked by the entry function.

TABLE III
NumFuzz versus CoverMe and AFL in Achieving Branch Coverage for Floating-point Programs

Benchmark			Time(s)[*]			Branch Coverage(%)			Improvement(%)	
File	Function	#Branches	AFL	CoverMe	NumFuzz	AFL	CoverMe	NumFuzz	NumFuzz vs. AFL	NumFuzz vs. CoverMe
e_acos.c	ieee754_acos(double)	12	23	5.3	2.3	91.7	91.7	100.0	8.3	8.3
e_acosh.c	ieee754_acosh(double)	10	4	0.7	0.4	80.0	100.0	100.0	20.0	0.0
e_asin.c	ieee754_asin(double)	14	38	5.2	3.8	85.7	92.9	92.9	7.2	0.0
e_atan2.c	ieee754_atan2(double,double)	44	267	13.4	26.7	90.9	59.1	91.4	0.5	32.3
e_atanh.c	ieee754_atanh(double)	12	3	5.4	0.3	91.7	91.7	91.7	0.0	0.0
e_cosh.c	ieee754_cosh(double)	16	3	5.4	0.3	81.3	93.8	93.8	12.5	0.0
e_exp.c	ieee754_exp(double)	24	4	5.6	0.4	58.3	95.8	95.8	37.5	0.0
e_fmod.c	ieee754_fmod(double,double)	60	269	12.8	26.9	70.0	56.7	80	10.0	23.3
e_hypot.c	ieee754_hypot(double,double)	22	110	9.4	11.0	81.8	81.8	95.5	13.7	13.7
e_j0.c	ieee754_j0(double)	18	12	6.0	1.2	83.3	94.4	94.4	11.1	0.0
	ieee754_y0(double)	16	4	0.5	0.4	81.2	100.0	100.0	18.8	0.0
e_j1.c	ieee754_j1(double)	16	19	6.6	1.9	87.5	93.8	93.8	6.3	0.0
	ieee754_y1(double)	16	15	1.4	1.5	93.8	100.0	100.0	6.2	0.0
e_log.c	ieee754_log(double)	22	24	2.1	2.4	86.4	90.9	95.5	9.1	4.6
e_log10.c	ieee754_log10(double)	8	3	3.3	0.3	87.5	87.5	87.5	0.0	0.0
e_pow.c	ieee754_pow(double,double)	114	1634	14.6	163.4	85.9	78.9	89.5	3.6	10.6
e_rem_pio2.c	ieee754_rem_pio2(double,double*)	30	5	8.2	0.5	83.3	90.0	100.0	16.7	10.0
e_remainder.c	ieee754_remainder(double,double)	22	7	8.2	0.7	81.8	86.4	90.9	9.1	4.5
e_scalb.c	ieee754_scalb(double,double)	14	7	9.7	0.7	57.1	92.9	100.0	42.9	7.1
e_sinh.c	ieee754_sinh(double)	20	3	5.5	0.3	80.0	90.0	90.0	10.0	0.0
e_sqrt.c	ieee754_sqrt(double)	46	184	11.1	18.4	95.7	78.3	84.8	-10.9	6.5
k_cos.c	kernel_cos(double,double)	8	1	8.6	0.1	87.5	87.5	87.5	0.0	0.0
s_asinh.c	asinh(double)	12	4	5.5	0.4	91.7	91.7	91.7	0.0	0.0
s_atan.c	atan(double)	26	9	5.7	0.9	88.5	92.3	92.3	3.8	0.0
s_cbrt.c	cbrt(double)	6	3	0.3	0.3	83.3	83.3	83.3	0.0	0.0
s_ceil.c	ceil_bis(double)	30	10	5.7	1.0	90.0	83.3	90.0	0.0	6.7
s_cos.c	cos(double)	8	3	0.2	0.3	87.5	100.0	100.0	12.5	0.0
s_erf.c	erf(double)	20	3	5.5	0.3	90.0	100.0	100.0	10.0	0.0
	erfc(double)	24	11	0.2	1.1	100.0	100.0	100.0	0.0	0.0
s_expm1.c	expm1_bis(double)	42	31	0.7	3.1	95.2	97.6	97.6	2.4	0.0
s_floor.c	floor_bis(double)	30	9	5.6	0.9	73.3	76.7	90.0	16.7	13.3
s_ilogb.c	ilogb_bis(double)	12	10	6.2	1.0	66.7	41.7	41.7	-25.0	0.0
s_log1p.c	log1p_bis(double)	36	34	6.9	3.4	91.7	88.9	91.7	0.0	2.8
s_logb.c	logb_bis(double)	6	1	0.2	0.1	50.0	83.3	50.0	0.0	-33.3
s_modf.c	modf(double,double*)	10	7	2.2	0.7	100.0	100.0	100.0	0.0	0.0
s_nextafter.c	nextafter_bis(double,double)	44	221	13.9	22.1	81.8	79.6	90.9	9.1	11.3
s_rint.c	rint(double)	20	224	1.7	22.4	95.0	90.0	100.0	5.0	10.0
s_sin.c	sin(double)	8	1	0.2	0.1	87.5	100.0	100.0	12.5	0.0
s_tan.c	tan_bis(double)	4	2	0.2	0.2	75.0	100.0	100.0	25.0	0.0
s_tanh.c	tanh_bis(double)	12	5	0.4	0.5	75.0	100.0	100.0	25.0	0.0
e_jn.c	ieee754_jn(int,double)	42	520	–	52.0	83.3	–	92.9	9.6	92.9
	ieee754_yn(int,double)	26	129	–	12.9	73.1	–	100.0	26.9	100.0
e_lgamma_r.c	ieee754_lgamma_r(double, int*)	47	213	–	21.3	93.6	–	100.0	6.4	100.0
s_frexp.c	frexp(double, int*)	6	2	–	0.2	66.7	–	83.3	16.6	83.3
s_ldexp.c	ldexp(double, int)	8	7	–	0.7	50.0	–	100.0	50.0	100.0
s_scalbn.c	scalbn(double, int)	16	45	–	4.5	81.2	–	81.2	0.0	81.2
k_sin.c	kernel_sin(double, double, int)	6	2	–	0.2	66.7	–	83.3	16.6	83.3
k_tan.c	kernel_tan(double, double, int)	16	13	–	1.3	93.8	–	93.8	0.0	93.8
Average		**23**	**87**	**5.3**	**8.7**	**82.4**	**88.6**	**91.9**	**9.5**	**18.1**

[*] We first run our tool NumFuzz and record the time that the tool achieves the highest coverage in ten minutes. Then we set the spent time of AFL as ten times of that of NumFuzz, since AFL is not specifically designed for numerical programs and thus needs more time to obtain good coverage.

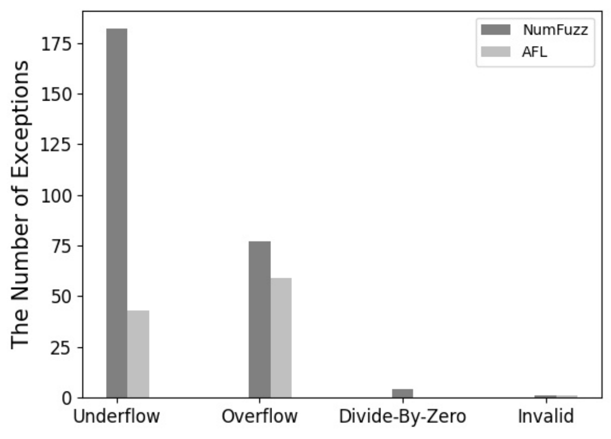


Fig. 3. The comparison of the number of floating-point exceptions detected by NumFuzz and AFL

TABLE IV
FLOATING-POINT EXCEPTIONS NUMFUZZ AND AFL FOUND

Function (gsl_sf_)	Inputs	Line	Exception	NUMFUZZ	AFL
airy_Ai_e	-1.8427611519777436e+00	274	DivByZero	✓	✗
exprel_2_e	7.0944466083035695e+02	406	Overflow	✓	✗
erf_Z_e	2.681561585988519e+154	380	Overflow	✓	✗
exp_e	-6.7741493348497897e+02	117	Underflow	✓	✗
fermi_dirac_mhalf_e	-6.8299992414604640e+02	1444	Underflow	✓	✗
legendre_H3d_0_e	4.30391878095824e-283 8.54409844947519e-60	272	DivByZero	✓	✗
laguerre_3_e	-1.00e+00 1.79769313286231e+307	225	Invalid	✓	✓

between NUMFUZZ and CoverMe (see the last row). There are some functions that NUMFUZZ can achieve higher coverage than CoverMe but the time overhead is bigger than CoverMe. For example, for the tested function (`e_pow.c`), NUMFUZZ spent 163.4 seconds in achieving 89.5% coverage. We have manually checked the test inputs for this function, and observed that NUMFUZZ can achieve 80.7% coverage at 14.6 seconds, which is competitive with CoverMe.

Note that, although NUMFUZZ is not specifically designed for achieving high branch coverage, we can see from Table III that our NUMFUZZ is competitive with CoverMe and AFL in achieving high floating-point branch coverage in numerical programs.

2) Floating-point Exception Detection Performance: The comparison of the number of floating-point exceptions detected by our tool NUMFUZZ and that by AFL, is shown in Figure 3. In total, NUMFUZZ detected 264 exceptions, including 182 *Underflow* exceptions, 77 *Overflow* exceptions, 4 *Divide-By-Zero* exceptions and 1 *Invalid* exception, while AFL found 103 exceptions, including 59 *Underflow* exceptions, 43 *Overflow* exceptions, 1 *Invalid* exception and none of *Divide-By-Zero* exception. From the distribution in Figure 3, we see that NUMFUZZ detects more *Underflow* and *Overflow* exceptions than *Divide-By-Zero* or *Invalid* exceptions. The number of *Divide-By-Zero* and *Invalid* exception is few, which is reasonable in theory because NUMFUZZ only detects interface functions of GSL's special function package and GSL is quite mature. Note that, on the other hand, the *Overflow* or *Underflow* exceptions may mask the *Divide-By-Zero* and *Invalid* exceptions. For example, an *Underflow* occurs in the calculation of the denominator and terminates the execution of the path before the division is computed.

The experimental results suggest the promising ability of NUMFUZZ in detecting floating-point exceptions in numerical programs, compared with AFL. In particular, it has outstanding ability in detecting *Underflow* and *Divide-By-Zero* exceptions. NUMFUZZ finds 4 *Divide-By-Zero* exceptions while AFL finds none of *Divide-By-Zero* exception. The *Divide-By-Zero* exceptions cause serious consequences. We particularly detail some floating-point exceptions detected by NUMFUZZ, as shown in Table IV. For example, NUMFUZZ detected a *Divide-By-Zero* exception in function *gsl_sf_airy_Ai_e* from `gsl/specfunc/airy.c`, while AFL fails to detect. With the input x = -1.8427611519777436e+00, the program will trigger a *Divide-By-Zero* exception in the operation `result_m.err/result_m.val` of the function *airy_mod_phase* invoked by function *gsl_sf_airy_Ai_e* at Line 274 of this file.

On the other hand, it is worthy noting that there are exception-triggering inputs which violate the preconditions in the comments internal to the source code. For example, in the *erfc8_sum* function from `gsl/specfunc/erfc.c`, an *Underflow* exception was triggered at Line 74 for the statement `den = x*den + Q[`i`]` in a loop when the input x = 1.25542034707733e+58. However, the input x is specified as $8 < x < 100$ in the comments internal to the source code and thus the *Underflow* exception is a false positive. Among all the 264 exceptions detected by NUMFUZZ, there are 6 false positives because of this reason. In the future, we plan to automatically collect the information in the comments internal to the source code for avoiding these false positives.

In summary, from the results of our all experiments, we can conclude that our approach has promising ability in detecting floating-point exceptions and achieving high floating-point branch coverage in real-world numerical programs.

VI. RELATED WORK

A. Analysis of Floating-point Programs

Abstract interpretation [17] is one of the techniques widely used in analyzing floating-point program. Miné [18] employs abstract interpretation-based static analysis using relational abstract domains to detect floating-point runtime exceptions. `Astreé` [19] leverages the relational abstract domains in [18], attempting to demonstrate the absence of runtime errors, over floating-point numbers. However, abstract interpretation has limitation that it does not sensitively analyze all program paths and may bring false positives.

Symbolic execution is another commonly used technique for analysis of floating-point programs. The symbolic execution tool `KLEE-FP` [20] is designed to support symbolic reasoning about the equivalence between floating-point values for cross-checking of floating-point and SIMD code, which is based on KLEE [21]. T. Barr et al. [5] first employ symbolic execution technique for automatic detection of floating-point runtime exceptions. They first consider floating-point arithmetic as real arithmetic, use SMT solver over real numbers to search for inputs that trigger exceptions, and then convert or search around those real number inputs to floating-point numbers and finally test them on the original floating-point program. To improve the efficiency of symbolic execution, Wu et al. [4] combine symbolic execution with value-range analysis to detect floating-point exceptions. They leverage value-range analysis to accelerate constraint solving, and thus can find more floating-point exceptions and eliminate false positives, comparing with the traditional symbolic execution techniques.

There are also many dynamic methods for analysis of floating-point programs. [22] and [23] employ genetic algorithm to automatically detect significant floating-point inaccuracies in numerical programs. AutoRNP [24] automatically detects and repairs high floating-point inaccuracy errors in numerical libraries. Similarly, ATOMU [25] utilizes condition numbers for atomic numerical operations to effectively detect floating-point inaccuracy errors in numerical programs. However, all the above works focus on floating-point inaccuracy errors, while our approach aims to detect floating-point exceptions. Recently, Fu et al. [6] present a dynamic analysis method for detection of floating-point overflow exception, which transforms the problem of detecting floating-point overflow exception into a search problem via weak-distance minimization. Compared to their work, we leverage fuzzing techniques and can detect general floating-point exceptions (e.g., *Underflow*). Moreover, we handle numerical programs with both floating-point and integer inputs.

B. Coverage-based Grey-box Fuzzing

Coverage-based grey-box fuzzing [8]–[10], [26]–[28] is a popular technique to find vulnerabilities and bugs, which has raised the wide-attention from both academic and industry. Coverage-based grey-box fuzzers generally leverages the coverage information as guidance for exploration of different program paths, e.g., the start-of-the-art coverage-based grey-box fuzzer, AFL [11].

Recently, to improve the effectiveness of fuzzing, grammar-based fuzzing [29] has received much attention to generate tests that meet the input syntax. NAUTILUS [30] combines the use of coverage feedback with the use of grammars as guidance to find deep bugs by generating valid test inputs. u4SQLi [31] proposes a set of mutation operations for SQLs to produce syntactically correct test inputs to touch deeper SQL statements. ProFuzzer [32], Zest [33] and Superion [34] automatically collect the syntactic or semantic knowledge in input fields, which aims to generate valid test inputs in syntactic and semantic to explore deeper program code. However, all of these works mentioned above have not considered mutation strategy of floating-point format for generating valid floating-point test inputs.

Different from these works, we propose a novel mutation strategy for floating-point format aiming to generate valid floating-point test inputs. Moreover, we explore a new guidance based on *ErrBits* to guide our fuzzer for whether retaining the test input as interesting test inputs for further mutation.

VII. Conclusion and Future Work

In this paper, we propose a floating-point format aware coverage-based grey-box fuzzing to detect floating-point exceptions for numerical programs. More specifically, we propose a novel mutation strategy for floating-point format, aiming at producing valid floating-point test inputs. Besides, we also present a new guidance, which explicitly prefers to search for test inputs that are closer to exposing exceptions. We have implemented our approach in a tool, named NUMFUZZ, based on AFL. We have conducted preliminary experiments to evaluate our tool in detecting floating-point exceptions and achieving high branch coverage respectively. The preliminary experimental results suggest that our tool achieve 91.9% of branch coverage on average on Sun's math library, which outperforms AFL and CoverMe. Moreover, NUMFUZZ has detected 264 floating-point exceptions on GSL, which is clearly better than AFL.

For future work, we will conduct experiments over more real-word numerical programs to detect runtime exceptions. Another direction of work is to extend our approach to support functions with vectors or matrices parameters.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (Nos. 61872445, 62032024).

References

[1] I. C. Society, "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.
[2] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM computing surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.
[3] J. R. Hauser, "Handling floating-point exceptions in numeric programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 18, no. 2, pp. 139–174, 1996.
[4] X. Wu, L. Li, and J. Zhang, "Symbolic execution with value-range analysis for floating-point exception detection," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2017, pp. 1–10.
[5] E. T. Barr, T. Vo, V. Le, and Z. Su, "Automatic detection of floating-point exceptions," *ACM Sigplan Notices*, vol. 48, no. 1, pp. 549–560, 2013.
[6] Z. Fu and Z. Su, "Effective floating-point analysis via weak-distance minimization," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 439–452.
[7] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.
[8] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.
[9] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.
[10] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.
[11] M. Zalewski. (2017) American fuzzy lop 2.52b. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[12] Z. Fu and Z. Su, "Achieving high coverage for floating-point code via unconstrained programming," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 306–319, 2017.

[13] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[14] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 1–11, 2015.

[15] "afl-cov: Code coverage analysis tool for afl." https://github.com/mrash/afl-cov.

[16] "Gcov: Gnu compiler collection tool." https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[17] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 238–252.

[18] A. Miné, "Relational abstract domains for the detection of floating-point run-time errors," in *European Symposium on Programming*. Springer, 2004, pp. 3–17.

[19] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The astrée analyzer," in *European Symposium on Programming*. Springer, 2005, pp. 21–30.

[20] P. Collingbourne, C. Cadar, and P. H. Kelly, "Symbolic crosschecking of floating-point and SIMD code," in *Proceedings of the sixth conference on Computer systems*, 2011, pp. 315–328.

[21] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[22] X. Yi, L. Chen, X. Mao, and T. Ji, "Efficient global search for inputs triggering high floating-point inaccuracies," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2017, pp. 11–20.

[23] D. Zou, R. Wang, Y. Xiong, L. Zhang, Z. Su, and H. Mei, "A genetic algorithm for detecting significant floating-point inaccuracies," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 529–539.

[24] X. Yi, L. Chen, X. Mao, and T. Ji, "Efficient automated repair of high floating-point errors in numerical libraries," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[25] D. Zou, M. Zeng, Y. Xiong, Z. Fu, L. Zhang, and Z. Su, "Detecting floating-point errors via atomic conditions," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–27, 2019.

[26] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 697–710.

[27] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[28] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *NDSS*, vol. 19, 2019, pp. 1–15.

[29] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, 2008, pp. 206–215.

[30] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars." in *NDSS*, 2019.

[31] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for SQL injection vulnerabilities: an input mutation approach," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 259–269.

[32] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," in *2019 IEEE symposium on security and privacy (SP)*. IEEE, 2019, pp. 769–786.

[33] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 329–340.

[34] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 724–735.