Automatic Detection of Floating-Point Exceptions

By

THANH V. VO

B.S. (Vietnam National University, Hanoi) 2007

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA,

DAVIS

Approved:

Zhendong Su, Chair

Premkumar Devanbu

Kent Wilken

Committee in Charge

2011

Contents

Thanh V. Vo
December 2011
Computer Science

Automatic Detection of Floating-Point Exceptions

Abstract

It is well-known that floating-point exceptions can be disastrous and writing numerical programs that do not throw floating-point exceptions is very difficult. Thus, it is important to automatically detect such errors. In this thesis, we present Ariadne, a practical, symbolic execution system specifically designed and implemented for detecting floating-point exceptions. The key idea is to systematically transform a numerical program to explicitly check each exception triggering condition. This transformation effectively reduces the difficult problem of symbolic reasoning over floating-point numbers to that over the reals, which has extensive tool support. We also devise novel, practical techniques to solve nonlinear constraints. We have conducted an extensive evaluation of Ariadne over 424 scalar functions in the widely used GNU Scientific Library (GSL). Our results show that Ariadne is practical and identified a large number of real runtime exceptions in GSL. The GSL developers confirmed our preliminary findings and look forward to Ariadne's public release, which we plan to do in the near future.

Acknowledgments

First of all, I would like to thank my family, who are the major source of mental support through all the challenges in research and course work.

It is an honor for me to be a student of professor Zhendong Su. His briliant ideas and suggestions are always the most important factor that advance my work. He has constantly shown his patience to me and the project. I am very lucky to work with such a nice and knowledgable advisor.

I would like to thank professor Prem Devanbu for his instruction and feedback. Attending his classes and seminars, I have understood more about software engineering and have learned how to do good research in general.

I am grateful to professor Kent Wilken for his lectures in our compiler optimization class. Interesting discussions with him have enhanced my knowledge in compiler both in theory and practice, a fundamental foundation of this work.

This thesis would not have been possible without the help of Dr. Earl Barr who has worked closely with me on my daily research. Whenever I have had any conceptual or technical questions, he has been always there to help me out. Earl indeed is my informal advisor.

I am indebted to many of my colleagues in Software Systems Lab and friends at Davis. They are always very kind and supportive; their questions and comments have polished the work.

1. Introduction

On June 4 1996, the European Space Agency's Ariane 5 rocket veered off course and self-destructed because the assignment of a floating-point number to an integer caused an overflow [31]. The loss is estimated to have been US$370m. Scientific results increasingly rest on software that is usually numeric and may invalidate results when buggy [24]. Numerical software, which uses floating-point arithmetic, is prominent in critical control systems in devices in national defense, transportation, where numeric control software has been implicated in Toyota's infamous failures [30], and health care, where it is being used in haptic controllers for remote surgery. Clearly, we are increasingly reliant on numerical software.

Computers have finite storage and must approximate real numbers, whose radix expansion may be unbounded. Floating-point numbers are a finite precision encoding of real numbers. Floating-point operations are not closed: their result may have an absolute value greater than the largest floating-point number and *overflow*; it may be nonzero and smaller than the smallest nonzero floating-point number and *underflow*; or it may lie between two floating-point numbers, require rounding, and be *inexact*. *Dividing by zero* and the *invalid* application of an operation to operands outside its domain, like taking the square root of a negative number, also generate exceptions. The IEEE 754 standard defines these exceptions [18].

Writing numerical software that does not throw floating-point exceptions is difficult [13, 16]. For example, it is easy to imagine writing **if** `(x != y) { z = 1 / (x−y); }` [13] and then later contending with the program mysteriously failing due to a spurious divide by zero. As

another example, consider a straightforward implementation to compute the 2-norm of a vector [16, 17]:

```
sum = 0;
for (i = 0; i < N; i++)
  sum = sum + x[i] * x[i];
norm = sqrt(sum);
```

For many slightly large or small vector components, this code may overflow or underflow when evaluating `x[i]*x[i]` or adding the result to sum. In spite of exceptions during its computation, the norm itself may, nonetheless, be representable as an unexceptional floating-point number. Even when some exceptions are caught and handled, others may not be; these are *uncaught exceptions*. The divide by zero in the first example is an uncaught exception. At the same time, the code to catch and handle exceptions at a floating point operation may be unnecessary — a particular exception may be impossible given the domain of possible inputs to the operation. In this case, the exception-handling code may degrade the performance of the system. Thus, we would like to identify both uncaught and unnecessarily caught exceptions.

Symbolic analysis has successfully tested and validated software [4, 12, 28]. However, little work has considered symbolic analysis of floating-point code. One natural approach is to equip a satisfiability modulo theory (SMT) solver with a theory for floating-point. Much work has pursued formalizing IEEE floating-point standards in various proof systems such as Coq, HOL, and PVS. This has proved to be a challenging problem and ongoing efforts simplify and deviate from the IEEE floating-point standard [27].

We propose an alternate approach to symbolically executing numerical software. Our key insight is that, if we transform floating-point code to create explicit program points at which floating-point exceptions could occur, we can leverage existing SMT solvers that are equipped with the theory of real numbers, such as Microsoft's Z3 [5]. Before a floating point exception occurs, all floating-point values are reals. Thus, if we symbolically execute a numeric program without polluting any of its floating-point values with the special values

that, by default, represent floating-point exceptions, we can directly feed the resulting constraints into an SMT solver equipped with the theory of reals. To this end, we transform a numeric program to explicitly check the operands of each floating-point operation before executing that operation. If a check fails, the transformed program throws the floating-point exception that would have occurred if the guarded floating-point operation had executed, then terminates. In the transformed program, all floating-point values, from program entry to exit, are reals. To discover whether the original numeric code could throw a particular floating point exception, we symbolically execute the code. If we can reach any of the floating-point-exception-throwing program points we injected, we have found a floating-point exception and we can query the SMT solver to return an input that causes the original program to throw that exception.

Our approach, which we have christened Ariadne, combines testing and verification. When we find an exception, we produce a test case that triggers that exception that developers can add to their test suite and use to fix the bug. Ariadne is sound: when it traverses a path from start to exit without finding an exception, no instruction on that path can thrown a floating-point exception. When it can show that all paths to the exception-throwing program point are unsatisfiable, it verifies that the program does not throw floating-point exceptions.

To realize the Ariadne symbolic execution engine, we have adapted and enhanced the symbolic execution engine KLEE from Stanford [4]. We taught KLEE to handle floating-point and use Z3 as its SMT solver instead of STP [11], which does not support the theory of the reals. Interesting numeric code often contains multivariate, nonlinear constraints. We fed these constraints to iSAT, a state of the art nonlinear solver [8], but it only handled less than 1%, rejecting most of our constraints because they contain large constant values that exceed iSAT's bounds. We also directly gave them to Z3, which recently added support for multivariate, nonlinear constraints but it could only handle a small pecentage. These results motivate us to devise a novel method for handling nonlinear constraints involving rational functions (to support division), which we also incorporated into KLEE. Our contributions follow:

- A technique for transforming numeric code to make floating-point exception handling explicit;
- An LLVM and Klee-based implementation of our technique and its evaluation on the GNU Scientific library (GSL);
- A tool that automatically converts fixed precision numeric code into arbitrary precision numeric code to detect potentially avoidable overflows and underflows; and
- Based on classic results, a method for handling nonlinear constraints involving the rational functions over the reals.

We evaluated our tool on the GNU Scientific Library (GSL) version 1.14. We analyzed all functions in the GSL that take scalar inputs, this family of functions includes elementary and many differential equation functions. Across the 424 functions we analyzed, our tool discovered inputs that generated 2288 floating-point exceptions. 94% of these are underflows and overflows, while the remainders are divide-by-zero and invalid exceptions. We reported preliminary results to the GSL community and they confirmed that our warnings were valid and said they look forward to the public release of our tool.

To motivate and clarify our problem, chapter 2 presents and explains numerical code that contains floating-point exceptions. We open chapter 3 with terminology, next describe the two transformations on which our approach rests — the first preserves the invariant that the floating-point variables are special-free and thus constraints containing them are suitable for SMT solvers and the second symbolically handles a subset of external functions that are difficult to internalize — then present algorithms we use to solve multivariate, nonlinear constraints. Chapter 4 describes the realization of these transformations and algorithms. In chapter 5, we present the results of our analysis of the GSL special functions. We discuss closely related work in and chapter 6 we summarize in chapter 7.

2. Illustrative Examples

Floating-point arithmetic is unintuitive. Sterbenz [29] illustrates this fact by computing $\frac{x+y}{2}$, the average of two numbers. Even for a "simple" function like this, it requires considerable knowledge to implement it well. The four functions av1–av4 in Figure 2.1 are a few possible average formulas. They are all equivalent over the reals, but not over floating-point numbers. For instance, av1 overflows when x and y have the same sign and sufficiently large, and av2 underflows when x and y are sufficiently small. Sterbenz performs a very interesting analysis of this problem and defines average that uses av1, av3, and av4 according to the signs of the inputs x and y. The function average is designed to have no overflow. Our tool Ariadne explores all the paths in average and confirms that it indeed does not raise any overflows. To run Ariadne, we first issue "`llvm-gcc -a sterbenz.c`" to compile and apply the operand checking, explicit floating-point exception transformation to create `sterbenz.c`, then issue "`ariadne sterbenz.bc`" to produce `sterbenz.out`, which contains the analysis results. Ariadne discovers 6 underflows, reporting, for example, x = −3.337611e−308 and y = 2.225074e−308 as an input that triggers this exception at line 4.

In the Ariane 5 disaster, the cast of double precision (64-bit) floating point number to a 16-bit signed integer caused an integer overflow [31]. Written in C[1], a similar cast is "**unsigned short** x = (**unsigned int**)d;". The cast was intentional. The violation of its implicit precondition that the absolute value of d be less than INT_MAX was not. It seems reasonable to assume d was the output of some complex, perhaps cypher physical, computation, or the

[1] The Ariane software was written in Ada.

```
1  #include <stdio.h>
2  #include <float.h>
3  double av1(double x, double y) {
4     return (x+y)/2.0;
5  }
6  double av2(double x, double y) {
7     return (x/2.0 + y/2.0);
8  }
9  double av3(double x,double y) {
10    return (x + (y-x)/2.0);
11 }
12 double av4(double x, double y) {
13    return (y + (x-y)/2.0);
14 }
15 double average(double x, double y) {
16    int samesign;
17    if ( x >= 0 ) {
18       if (y >=0)
19          samesign = 1;
20       else
21          samesign = 0;
22    } else {
23       if (y >= 0)
24          samesign = 0;
25       else
26          samesign = 1;
27    }
28    if ( samesign ) {
29       if ( y >= x)
30          return av3(x,y);
31       else
32          return av4(x,y);
33    } else
34       return av1(x,y);
35 }
```

Figure 2.1: Sterbenz' average function.

fact that it could violate the cast's precondition would have been noticed. Ariadne detects integer overflows due to casts involving floating-point numbers. Had Ariadne been applied to Ariane control software, it would have injected **if** (abs(d) $>$ INT_MAX) throw IntOverflow, then symbolically executed that complex computation and reported an exception-triggering input that demonstrated that the precondition could be violated.

We close with a function that contains each of the four floating-point exceptions Ariadne

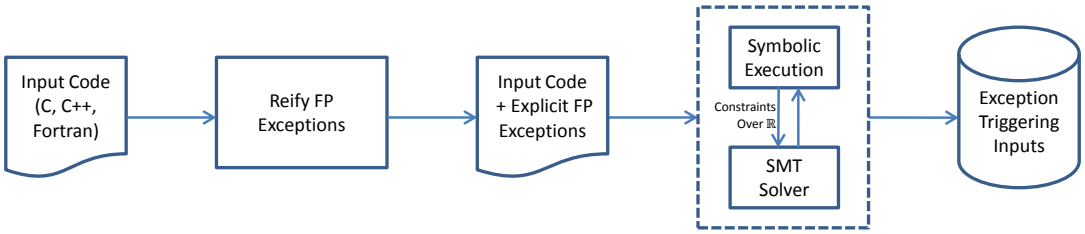

Figure 2.2: The architecture of Ariadne.

```
1  int gsl_sf_bessel_Knu_scaled_asympx_e(
2      const double nu, const double x, gsl_sf_result * result
3  ) { /* x >> nu*nu+1 */
4    double mu   = 4.0*nu*nu;
5    double mum1 = mu-1.0;
6    double mum9 = mu-9.0;
7    double pre  = sqrt(M_PI/(2.0*x));
8    double r    = nu/x;
9    result->val = pre * (1.0 + mum1/(8.0*x) +
       mum1*mum9/(128.0*x*x));
10   result->err = 2.0 * GSL_DBL_EPSILON * fabs(result->val) + pre
       * fabs(0.1*r*r*r);
11   return GSL_SUCCESS;
12 }
```

Figure 2.3: A function from GSL's special function collection.

detects, drawn from our test corpus, the GSL special functions. Figure 2.3 contains the GSL's implementation of gsl_sf_bessel_Knu_scaled_asympx_e . This function computes the scaled irregular modified Bessel function of fractional order. Run on this function, Ariadne reports the following: The division operation at line 7 throws an Invalid exception when $x < 0$ and a Divide-by-Zero when $x = 0$. When $nu = -5.789604e+76$ and $x = 2.467898e+03$, the evaluation of $mum1 * mum9$ at line 9 overflows. Finally, when $nu = -7.458341e-155$ and $x = 2.197413e+03$, the evaluation of $mu = 4.0 * nu * nu$ at line 4 underflows.

3. Approach

Figure 2.2 depicts the architecture of Ariadne, which has two main phases. Phase one transforms an input numeric program into a form that explicitly checks for floating-point exceptions before each floating-point operation and, if one occurs, throws it and terminates execution. The transformed program contains conditionals, such as $x > \text{DBL\_MAX}$, that cannot hold during concrete execution, where DBL_MAX denotes the maximum floating-point value. The transformed program, including its peculiar conditionals, are amenable to symbolic execution and any program point that symbolic execution reaches without throwing an exception has the property that none of its floating-point variables contain a special value, such as NaN (Not a Number) or ∞. Thus, the numeric constraints generated during symbolic execution involve those floating-point values that are a subset of $\mathbb{R}$ and suitable input to an SMT solver supporting the theory of the reals. During phase two, Ariadne symbolically executes the transformed program, and for each path that reach floating-point exceptions injected in phase one, it attempts to solve the corresponding path constraint. If a satisfying assignment is found, Ariadne reports the assignment as a concrete input that triggers the exception. Our symbolic execution is standard; the key challenge in its realization is how to effectively solve the collected numerical constraints, many of which are multivariate, nonlinear.

We first present pertinent background in numerical analysis and introduce notation (Section 3.1), then describe our transformation (Section 3.2) and how we solve the numerical constraints (Section 3.3).

Exception	Example	Default Result
Overflow	$\lvert x+y\rvert > \Omega$	$\pm\infty$
Underflow	$\lvert x/y\rvert < \lambda$	Subnormals
Divide-by-Zero	$x/0$	$\pm\infty$
Invalid	$0/0, 0\times\infty, \sqrt{-1}$	NaN
Inexact	$\text{nearest}(x \text{ op } y) \neq x \text{ op } y$	Rounded result

Table 3.1: Floating-point exceptions; $x, y \in \mathbb{F}$ [17, §2.3].

3.1 Background and Notation

Four integer parameters define a floating-point number system $\mathbb{F} \subset \mathbb{R}$: the base (radix) β, the precision t, the minimum exponent $e_{\min}$, and the maximum exponent $e_{\max}$. With these parameters and the mantissa $m \in \mathbb{Z}$ and the exponent $e \in \mathbb{Z}$,

$$\mathbb{F} = \{\ \pm m\beta^{e-t} \mid (0 \le m \le \beta^t - 1 \wedge e_{\min} \le e \le e_{\max}) \\ \vee (0 < m < \beta^{t-1} \wedge e = e_{\min})\ \}$$

The floating-point numbers for which $\beta^{t-1} \le m \le \beta^t \wedge e_{\min} \le e \le e_{\max}$ holds are *normal*; the numbers for which $0 \le m < \beta^{t-1} \wedge e = e_{\min}$ holds are *subnormal* [17, §2.1].

$\Omega = \beta^{e_{\max}}(1-\beta^{-t})$	Maximum
$-\Omega = -\beta^{e_{\max}}(1-\beta^{-t})$	Minimum
$\lambda = \beta^{e_{\min}-1}$	Smallest normalized
$æ = \beta^{e_{\min}-t}$	Smallest denormalized

Floating-point operations can generate exceptions shown in Table 3.1. Some applications of the operations on certain operands are undefined and cause the Divide-by-Zero and Invalid exceptions. Finite precision causes the rest. Inexact occurs frequently [13, 19] and is often an unavoidable consequence of finite precision. For example, when dividing 1.0 by 3.0, an

Inexact exception occurs because the ratio $1/3$ cannot be exactly represented as a floating-point number. For this reason, Ariadne focuses on discovering Overflow, Underflow, Invalid, and Divide-by-Zero exceptions.

Throughout this section, we use $\mathbb{Q}$ to denote its arbitrary precision subset. Let $fp : \mathbb{Q} \to \mathbb{F}$ convert a rational into the nearest floating-point value, rounding down. For $next : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$, $next(x, y)$ returns the floating-point number next to x in the direction of y.

To model path constraints, we consider formulas ϕ in the mixed theory of reals and uninterpreted functions. For a formula ϕ, $fv(\phi) = \vec{x}$ denotes the free variables in ϕ. We use S to denote satisfiable or SAT; $\bar{S}$ to denote unsatisfiable or UNSAT; and U for UNKNOWN. $A^n_{\mathbb{Q}} = (V \times \mathbb{Q})^n \uplus \{\bar{S}, U\}$ is a disjoint union that is either an n length vector of simultaneous assignments over $\mathbb{Q}$ to the variables V, or it is UNSAT or UNKNOWN. $A^n_{\mathbb{F}} = (V \times \mathbb{F})^n \uplus \{\bar{S}, U\}$ is the subset of $A^n_{\mathbb{Q}}$ whose bindings are restricted to $\mathbb{F}$. For isSAT: $A^n_{\mathbb{Q}} \to \mathbb{B}$, isSAT($a$) returns true if $a \notin \{\bar{S}, U\}$.

3.2 Program Transformation

We first define Φ, the term rewriting system we use to inject explicit floating-point exception handling into numeric code. For simplicity of presentation, we assume that nested arithmetic expressions have been flattened. We assume that the program terminates after throwing a floating-point exception.

3.2.1 Basic Arithmetic Operations

The operator $\odot \in \{+, -, *\}$, and variables x and y bind to floating-point expressions. We have the following *local rewriting rules*:

```
1  double z = mu - 1.0;
2  double abs1 = fabs(z);
3  if (DBL_MAX < abs1) {
4      perror("Overflow!\n");
5      exit(1);
6  }
7  if (0 < abs1 && abs1 < DBL_MIN) {
8      perror("Underflow!\n");
9      exit(1);
10 }
11 double mum1 = z;
```

Figure 3.1: Subtraction Transformation (line 5 of example in Figure 2.3): Ariadne symbolically executes this code over arbitrary precision rationals, where the conditionals can be tested; `fabs` returns the absolute value of a double.

$$\Phi(x \odot y) = \begin{cases} \text{Overflow} & \text{if } |x \odot y| > \Omega \\ \text{Underflow} & \text{if } 0 < |x \odot y| < \lambda \\ x \odot y & \text{otherwise} \end{cases}$$

$$\Phi(x \,/\, y) = \begin{cases} \text{Invalid} & \text{if } y = 0 \wedge x = 0 \\ \text{Divide-by-Zero} & \text{if } y = 0 \wedge x \neq 0 \\ \text{Overflow} & \text{if } |x| > |y|\Omega \\ \text{Underflow} & \text{if } 0 < |x| < |y|\lambda \\ x \,/\, y & \text{otherwise} \end{cases}$$

We also have a global contextual rewriting rule $\Phi(C[e]) = C[\Phi(e)]$, where e denotes an expression of the form $x \odot y$ or $x \,/\, y$, and $C[\cdot]$ denotes a program context. Ignoring time-dependent behavior, it is evident that $[\![p]\!] = [\![\Phi(p)]\!]$ when the numeric program p terminates without throwing a floating-point exception.

Figure 3.1 shows a concrete example of how Φ, specialized to C, transforms a subtraction expression (line 5 of example in Figure 2.3). The result of the subtraction is checked and, if an exception could occur, execution terminates. The division transformation, depicted in Figure 3.2, is the most involved transformation. The original expression is taken from line 8 of the example in Figure 2.3. A floating-point division operation can throw four distinct

```
1  if (x == 0) { // denominator is zero
2     if (nu == 0) {
3        perror("Invalid!\n");
4     } else {
5        perror("DivZero!\n");
6     }
7     exit(1);
8  }
9  double abs1 = fabs(nu);
10 double abs2 = fabs(x);
11 if (abs1 > abs2 * DBL_MAX) {
12    perror("Overflow!\n");
13    exit(1);
14 }
15 if (0 < abs1 && abs1 < abs2 * DBL_MIN) {
16    perror("Underflow!\n");
17    exit(1);
18 }
19 double r = nu/x;
```

Figure 3.2: Division Transformation (line 8 of example in Figure 2.3).

exceptions, each of which the transformed code explicitly checks. For instance, given the inputs nu = 0.0 and x = 0.0, the original expression would have terminated normally and returned NaN, but the transformed code would detect and output an Invalid exception then terminate.

3.2.2 Elementary Mathematical Functions

Calls into an unanalyzed library usually terminate symbolic execution. Calls to elementary mathematical functions, like `sqrt`, `log`, `exp`, `cos`, `sin` and `pow` are frequent in our constraints. These functions have no polynomial representation. We next describe simple, yet effective transformations to deal with these functions.

We handle `sqrt` precisely and directly by adding y, a fresh independent symbolic variable, as shown in Figure 3.3. For the `sqrt` expression on line 7 of the example from Figure 2.3, we add a fresh symbolic variable y and the constraint $y \cdot y = \text{M\_PI}/(2.0x)$. The others we handle by introducing dependent symbolic variables that allow us to defer concretization and continue symbolic execution and exploration for floating-point exceptions,

$$
\begin{aligned}
\Phi(\texttt{sqrt}(x)) &= \begin{cases} \text{Invalid} & \text{if } x < 0 \\ y \cdot y = x & \text{otherwise} \end{cases} \\
\Phi(\texttt{exp}(x)) &= \begin{cases} \text{Overflow} & \text{if } x > \texttt{log}(\Omega) \\ \text{Underflow} & \text{if } x < \texttt{log}(\lambda) \\ d & \text{otherwise} \end{cases} \\
\Phi(\texttt{log}(x)) &= \begin{cases} \text{Invalid} & \text{if } x \leq 0 \\ d & \text{otherwise} \end{cases} \\
\Phi(\texttt{cos}(x)) &= d \quad \text{where } -1 \leq d \leq 1 \\
\Phi(\texttt{sin}(x)) &= d \quad \text{where } -1 \leq d \leq 1 \\
\Phi(\texttt{pow}(x,y)) &= \begin{cases} \text{Invalid} & \text{if } x = 0 \wedge y \leq 0 \\ d & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 3.3: Rewriting rules for elementary mathematical functions. With the exception of `sqrt`, these rewriting rules approximate the functions. We assume that each rule returns a fresh symbolic variable.

Dependent variable	Expression
d_0	$sin(\frac{x}{y})$
d_1	$log(x^2 + \frac{1}{x})$

Table 3.2: A dependent symbolic variable table.

rather than simply terminating upon encountering one of these elementary functions. To introduce a dependent symbolic variable, we rewrite each use of an elementary function as shown in Figure 3.3. We also record the dependent symbolic variable and the expression on which it depends in the dependent symbolic variable table, for use during concretization (Section 3.3.1): Table 3.2 illustrates what happens when the transformation encounters $\texttt{sin}(\frac{x}{y})$ and $\texttt{log}(x^2 + \frac{1}{x})$, which use the input symbolic variables x, y. It is possible for a dependent symbolic variable to depend on another dependent variable, as when `log(log(x))` is encountered. Our rewriting of *pow* could make additional checks for exceptions, such as $y > 0 \wedge y\texttt{log}(x) > \texttt{log}(\Omega)$, but as we mentioned earlier, we have found our current handling of `pow` and the other elementary functions very effective over the functions we have analyzed.

We use the following terminology in our later discussions. An *independent symbolic variable* is an input variable and is necessarily unbound in the dependent symbolic variable table. A *dependent symbolic variable* is a symbolic variable that the table binds to an

expression whose free variables contain at least one symbolic variable.

3.3 Solving Numeric Constraints

Designing SMT solvers that can effectively support multivariate, nonlinear constraints is an active area of research. Off-the-shelf SMT solvers focus on linear constraints and do not solve general nonlinear constraints because linear constraints are prevalent in integer code and solving nonlinear constraints is very expensive. However, real-world numeric applications contain many nonlinear constraints. In our experiment over the GSL special functions, 81% of queries are nonlinear, 68% are multivariate and 60% are both nonlinear and multivariate.

Algorithm 1 is the Ariadne solver for numerical constraints. During symbolic execution, it is called to solve path constraints. In addition to a path constraints ϕ, it takes the dependent symbolic variable table built during the transformation phase, the transformed numeric program itself and the location (program point) of the query.

To solve ϕ, Algorithm 1 first concretizes ϕ using T, the table of dependent symbolic variables built during the transformation phase, to convert ϕ into a univariate formula by binding variables to values in $\mathbb{F}$, if necessary (Section 3.3.1). Next, to convert a nonlinear univariate formula into a linear formula, again as necessary, Algorithm 1 applies a linearization technique that finds the roots of each nonlinear polynomial, then rewrites the constraints into an equivalent disjunction of interval checks (Section 3.3.2). While ϕ is univariate and nonlinear at this point, it is a formula in the theory of the reals, so the solver may return a solution in $\mathbb{Q} \setminus \mathbb{F}$. The findFloat algorithm (Section 3.3.3) finds a floating-point solution for a univariate, linear formula, if one exists.

Finally, a satisfying assignment in $\mathbb{F}^n$ may reach and trigger an exception when symbolically executed over arbitrary precision $\mathbb{Q}$ but *not* when concretely executed over $\mathbb{F}$. Consider "if$(x+1 > \Omega)$ Overflow". The binding $x = \Omega$ is a valid solution in the theory of reals; it also passes the findFloat test since this test only enforces $x \in \mathbb{F}$. Under concrete

Algorithm 1 ariadneSolve: $\Phi \times \mathbb{T} \times P \times \mathbb{N} \rightarrow A_{\mathbb{F}}^{n}$ solves numeric path constraints. Note that $n = |fv(\phi)|$, for $\phi \in \Phi$.

inputs $\phi \in \Phi$, a numeric constraint from p.
$T \in \mathbb{T}$, a table of dependent symbolic variables.
$p \in P$, the numeric program under analysis.
$l \in \mathbb{N}$, the program point of query.

1: **for** 1..MAXCONCRETIZATIONS **do**
2: $\phi', \vec{a}$:= concretize(ϕ, T) // *NOP when ϕ is univariate.*
3: **continue if not** isSAT$(\vec{a})$
4: $\vec{a} := \vec{a} +$ findFloats(linearize(ϕ'))
5: **continue if not** isSAT$(\vec{a})$
6: **if** $p(\vec{a}) \neq l$ **then**
7: **return** $\bar{S}$ **if** $|fv(\phi)| = 1$
8: $(x, f) =$ choose$(\vec{a})$
9: $\phi := \phi \wedge x \neq f$
10: **continue**
11: **break if** isSAT$(\vec{a}) \vee |fv(\phi)| = 1$
12: **return** $\vec{a}$

execution with standard floating-point semantics, however, the difference in magnitude of the operands means that the result of the addition operation must be truncated to fit into the finite precision of floating-point which "absorbs" the 1 into x's value of Ω. In effect, this manifests as an Inexact exception in the concrete execution. To check for and avoid reporting solutions such as this one accounts for the concrete execution check at line 6. Here, the satisfying assignment $\vec{a}$ is used as input to determine whether the specified program point can be reached. If this check fails, a variable binding from $\vec{a}$ is chosen nondeterministically and converted into an inequality so that either the next iteration tries a different solution or ϕ is no longer satisfiable. The algorithm ends when the maximum number of concretization attempts is exhausted, or $\vec{a}$ is satisfiable and passes the concrete execution test or ϕ is univariate and completes a single iteration, as it is not concretized.

3.3.1 Concretizing Multivariate Constraints

Algorithm 2 uses concretization to convert a formula that contains multivariate polynomial constraints into a formula containing only univariate polynomial constraints. Its inputs

Algorithm 2 concretize : $\Phi \times \mathbb{T} \rightarrow \Phi \times A_{\mathbb{F}}^{n-1}$. This algorithm first concretizes the independent symbolic variables that define each dependent variable, concretely evaluates the resulting expression to compute the dependent variable, before concretizing any remaining independent symbolic variables.

inputs $fv(\phi)$, the set of free variables in ϕ.
T, a dependent symbolic variable table.

1: $\vec{a} := \langle\rangle$
2: $C = \{v_i \mid \langle \cdots, (s_i, v_i), \cdots \rangle = \vec{a}\}$ // *Variables in $\vec{a}$ as a set.*
3: $\forall (d, y) \in \text{sort}(T)$ **do**
4: $\quad \forall z \in fv(y) - C$ **do** // *Skip already concretized variables.*
5: $\quad\quad a' := \text{bindVariable}(\phi, z)$
6: $\quad\quad$ **return** ϕ, a' **if not** isSAT(a')
7: $\quad\quad \phi := \phi[f/x]$ for $(x, f) = a'$
8: $\quad\quad \vec{a} := \vec{a} + a'$
9: $\quad f := \text{eval}(y[\vec{z}/\vec{a}(\vec{z})])$
10: $\quad \vec{a} := \vec{a} + (d, f)$
11: $\quad \phi := \phi[f/d]$
12: **while** $|fv(\phi)| > 1$ **do**
13: $\quad a' := \text{findBinding}(\phi, \text{choose}(fv(\phi))$
14: $\quad$ **return** ϕ, a' **if not** isSAT(a')
15: $\quad \vec{a} := \vec{a} + a'$
16: $\quad \phi := \phi[f/s]$ for $(s, f) = a'$
17: **return** $\phi, \vec{a}$

are $fv(\phi)$, the set of free variables, which are symbolic in our context, of the formula ϕ and T, the dependent symbolic variable table built during the transformation. Independent I and dependent D symbolic variables (defined in Section 3.2.2) partition these free variables.

To clarify the presentation in this section, we introduce the helper function checkbinding: $\Phi \times V \times \mathbb{F} \rightarrow \mathbb{B}$:

$\forall c_i \in \phi$ **do**
$\quad$ **return false if** evalexpr($c_i[f/s]$) = **false**
return true

The function evalexpr walks the expression tree that results from the substitution and evaluates constant expressions. It returns false if the recursive evaluation of constant expressions reduces the expression tree to false. Its complexity is $O(n)$, where n is the number of nodes in the expression tree.

In lines 4–11, we handle dependent symbolic variables. Because a dependent symbolic variable may depend on another dependent symbolic variable, we sort T (line 3) by the count of distinct dependent variables in its expression field, ascending from zero. Then, we concretize those independent symbolic variables that appear in the expression of a dependent symbolic variable and have not already been concretized (*i.e.* not in C), using bindVariable: $\Phi \times V \rightarrow A^n_{\mathbb{F}}$, which finds a concrete binding for a given variable:

for 1..MAXBINDINGS **do**
  $f :=$ random(**dom**(s))
  return (s, f) **if** checkbinding(ϕ, s, f)
return U // *UNKNOWN is returned.*

At line 9, we first substitute each symbolic variable in y, where we use $\vec{a}(\vec{z})$ to denote the vector of concrete bindings in $\vec{a}$ for the variables in $\vec{z}$, then concretely evaluate the resulting concretized expression to concretize the dependent symbolic variable. For example, consider $T[d_1] = log(x^2 + \frac{1}{x})$ in Table 3.2. When x is concretized to 1, $d_1 = log(2) = 1$.

When more than one independent symbolic variable remain after handling dependent symbolic variables, we concretize them until one remains in the final while loop at lines 12–16[1]. The function *findBinding*: $\Phi \rightarrow A^n_{\mathbb{F}}$ abstracts different algorithms for finding a binding of a symbolic variable to a concrete value in its domain. For instance, findBinding could select the symbolic variable with the highest degree in ϕ or uniformly at random. Alternately, findBinding could select the variable that appears most often. Algorithm 3 is the implementation we used. In it, the choosen function nondeterministically selects an element from a set, but could be defined to implement an arbitrary selection policy.

Algorithm 2 returns a rewritten ϕ that has at most one symbolic variable. Algorithm 2 calls evalexpr for each independent symbolic variable over expression trees whose maximum number of nodes is n. For each dependent variable, we must evaluate its corresponding

[1]The handling of dependent symbolic variables may concretize all independent symbolic variables and convert ϕ into a ground formula, *i.e.*, a formula with no free variables. When this happens, linearize echoes the ground formula and findFloat returns true or false via an empty assignment binding.

Algorithm 3 findBinding : $\Phi \times V \to A^n_{\mathbb{F}}$. This algorithm finds a concrete binding for some variable in the multivariate formula ϕ.

input ϕ, a multivariate formula.
s, a symbolic variable.
Tested $:= \emptyset$
Candidates $:= \{y \mid y \in fv(\phi) \wedge \text{typeof}(y) = \text{typeof}(s)\}$
repeat
$f := \text{random}(\textbf{dom}(s))$
return (s, f) **if** checkbinding(ϕ, s, f)
Tested $:=$ Tested $\cup \{s\}$
$s :=$ choose(Candidates $\setminus$ Tested)
until Candidates $\setminus$ Tested $= \emptyset$
return U // *UNKNOWN is returned.*

expression, say of complexity α. Thus, the complexity of concretize is $O(|I|n + |D|\alpha)$.

3.3.2 Linearizing Univariate Nonlinear Constraints

Given a univariate, nonlinear constraint, we transform it into a disjunction of linear constraints, suitable for an off-the-shelf SMT solver. Algorithm 4 linearizes a formula in the theory of the reals into an equivalent formula by replacing each comparison involving a nonlinear polynomial with an equivalent collection of disjuncts. For each conjunct in ϕ, it loops over each disjunct rewriting those disjuncts that make nonlinear polynomial comparisons. The complexity of this step is quadratic since finding roots of univariate polynomials can be done in quadratic time. The correctness of this rewriting rests on Lemma 3.3.1, which we present next.

Lemma 3.3.1. *For a rational function $f(x) = \frac{P(x)}{Q(x)}$ where $x \in \mathbb{R}$, there exists a polynomial $R(x)$ from whose distinct real roots, $x_1, x_2, \cdots, x_m$, where $x_i < x_j$ when $i < j$, the mapping of intervals*

$$b = \{(1, (x_m, \infty)), (2, (x_{m-1}, x_m)), \cdots, (m, (-\infty, x_1))\}$$

can be formed such that

Algorithm 4 linearize: $\Phi \rightarrow \Phi$. This algorithm rewrites a nonlinear polynomial constraint into an equivalent disjunction of linear interval predicates.

input $\phi = \bigwedge c_i$, a univariate numeric path constraint.
  $\forall c_i \in \phi$ **do** // *All conjuncts in* ϕ.
    $c_i' := c_i$
    $\forall d_i \in c_i$ **do** // *All disjuncts in* c_i.
      if $d_i = R(x) \prec 0 \wedge \text{degree}(R(x)) > 1$ **then**
        $x_1, x_2, \cdots, x_m, \cdots, x_n = \text{polysolver}(R(x))$
        The roots $x_1, x_2, \cdots, x_m$ are the distinct real roots of $R(x)$, ordered such that $\forall i, j \in [1..m], i < j \Rightarrow x_i < x_j$.
        $b = \{(1, (\infty, x_m)), (2, (x_m, x_{m-1})), \cdots, (m, (x_1, -\infty))\}$
        $\text{eq} := \bigvee_{i=1}^{m} x = x_i$
        $\text{lt} := \bigvee_{i=1}^{m} x \in b(i) \wedge i = 2k, k \in \mathbb{N}$
        $\text{gt} := \bigvee_{i=1}^{m} x \in b(i) \wedge i = 2k+1, k \in \mathbb{N}$
        $\psi :=$ **match** $\prec$ **with**
            $= \rightarrow \text{eq}$
          $| \text{¡} \rightarrow \text{lt}$
          $| \text{¿} \rightarrow \text{gt}$
          $| \leq \rightarrow \text{eq} \vee \text{lt}$
          $| \geq \rightarrow \text{eq} \vee \text{gt}$
        $c_i' := c_i'[\psi / d_i]$
    $\phi := \phi[c_i'/c_i]$ **if** $c_i \neq c_i'$
  return ϕ

$$f(x) = 0 \equiv \bigvee_{i=1}^{m} x = x_i$$
$$f(x) < 0 \equiv \bigvee_{i=1}^{m} x \in b(i) \wedge i = 2k, k \in \mathbb{N}$$
$$f(x) > 0 \equiv \bigvee_{i=1}^{m} x \in b(i) \wedge i = 2k+1, k \in \mathbb{N}$$

Proof. The rational function $f(x) = \frac{P(x)}{Q(x)}$, by definition. For $\prec \in \{<, >, =\}$, $\frac{P(x)}{Q(x)} \prec 0 \equiv P(x)Q(x) \prec 0 \wedge Q(x) \neq 0$ (multiply by $Q(x)^2$) so our problem reduces to that of solving the predicate $R(x) \prec 0$ which either has the form $P(x)Q(x)$ or $Q(x)$. We normalize $R(x)$ so that its leading coefficient is positive. Assume $R(x)$ has degree n and m real roots $x_1 \leq x_2 \leq \cdots \leq x_m$. From the Fundamental Theorem of Algebra, we have $R(x) = (x -$

Algorithm 5 findFloat: $\Phi \to A^n_{\mathbb{F}}$ takes a path constraint and returns a satisfying assignment over $\mathbb{F}$ if one exists.

input $\phi \in \Phi$, a path constraint.

1: **while true do**
2: $\quad \vec{a} :=$ SMT.solve(ϕ)
3: $\quad$ **return** $\vec{a}$ **if not** isSAT($\vec{a}$)
4: $\quad (x, v) = \vec{a}$
5: $\quad \vec{a} :=$ SMT.solve($\phi[fp(v)/x]$)
6: $\quad$ **return** $\vec{a}$ **if** isSAT($\vec{a}$)
7: $\quad \vec{a} :=$ SMT.solve($\phi[\text{next}(fp(v), \Omega)/x]$)
8: $\quad$ **return** $\vec{a}$ **if** isSAT($\vec{a}$)
9: $\quad \phi := \phi \wedge (x < fp(v) \vee x > \text{next}(fp(v), \Omega))$

$x_1)(x-x_2)\cdots(x-x_m)(x-x_{m+1})\cdots(x-x_n)$. If a polynomial has the complex root $a+bi$, that root's conjugate is also a root, so the product of the non-real complex roots of $R(x)$ is positive. Thus, the distinct, real roots of $R(x)$ determine the truth value of $R(x) \prec 0$. If $m = 0$, then $R(x) > 0$ and $R(x) \leq 0$ is false. Otherwise, the real roots determine the signs of their factors when $R(x) \neq 0$: The roots greater than x form negative factors while the roots less than x form positive factors. Thus, $R(x) < 0$ when the number of roots greater than x is odd and $R(x) > 0$ otherwise. Let b be a map that orders the intervals into which x can fall as follows $\{(1,(x_m,\infty)),(2,(x_{m-1},x_m)),\cdots,(m,(-\infty,x_1))\}$. Then, checking $R(x) > 0$ is equivalent to $x \in b(1) \vee x \in b(3) \vee x \in b(5) \vee \cdots$ and checking $R(x) < 0$ is $x \in b(2) \vee x \in b(4) \vee x \in b(6) \vee \cdots$ while checking $R(x) = 0$ is $(x = x_m) \vee \cdots \vee (x = x_1)$. □

3.3.3 Finding a Floating-point Solution

Given a univariate, linear constraint ϕ, we need to find a solution over $\mathbb{F}$. Algorithm 5 loops over the intervals that ϕ defines. It tries to solve ϕ at line 2, then substitutes the nearest float less or equal to the solution into ϕ and tries to solve the resulting formula at line 5. If that fails, it then substitutes the nearest float greater than or equal the solution into ϕ and tries to solve the resulting formula at line 7. If this fails, it rules out the interval under consideration at line 9 and considers the next interval. The complexity of this step is linear, and we prove the correctness of Algorithm 5 next.

Lemma 3.3.2. *Algorithm 5 finds a floating-point solution to a univariate linear constraint, if one exists.*

Proof. The solution for a univariate system of linear inequalities is a set of intervals. A linear inequality has the form $ax+b>0$. Either $a=0$ and b determines the truth value of the inequality, or the solution of the clause falls into one of two intervals defined by the root of clause. Thus, the path constraint ϕ is a set of intervals $\bigcup[a_i,b_i]$[2]. For $x\in[a_i,b_i]$, let x_f be the largest floating-point number that less than or equal to x. If either $x_f\in[a_i,b_i]$ or $\text{next}(x_f,\Omega)\in[a_i,b_i]$ then a floating-point value exists that satisfies the path constraints. Otherwise, we have $x_f\notin[a_i,b_i]\wedge\text{next}(x_f,\Omega)\notin[a_i,b_i]\wedge[a_i,b_i]\cap[x_f,next(x_f,\Omega)]\neq\emptyset$ which implies $x_f<a_i<b_i<\text{next}(x_f,\Omega)$. In this case, there is no floating-point number in the interval $[a_i,b_i)]$. So we add the constraint $x<x_f\vee x>\text{next}(x_f,\Omega)$ to rule out $[a_i,b_i]$ and consider a different interval. If the algorithm exhausts all the intervals without finding a satisfying floating-point value, no floating-point value for x satisfies the path constraint. □

[2]We consider the closed intervals; open intervals can be handled similarly.

4. Implementation

In this section, we present the implementation of our floating-point exception detection tool, Ariadne. Ariadne's implementation mirrors its operation: its components fall into either transformation or analysis. The transformation rests on LLVM 2.7 [21] and, for its arbitrary precision transformation, it uses GMP [9]. Its analysis phases extend version 2.7 of the KLEE symbolic execution engine [4] and uses the polynomial solver from GSL 1.14 to rewrite constraints [10]. We are indebted to the community for having made these tools available. To give back, we will publish our tool at `anon.utopia.com/tool`.

4.1 Transformations

Our transformations are implemented as an LLVM analysis and transform pass and operate on LLVM IR. For this reason, our transformations are language agnostic and, in particular, can handle C, C++ and Fortran, the three languages in which most numerical software is written. The principle transformations are Ariadne, loop bound, and arbitrary precision.

Ariadne This transformation rewrites every floating-point operation, injecting operand guards before each one, as described in Section 3.2. It also handles elementary functions and constructs the dependent symbolic variable table (Section 3.2.2. For an input module, it iterates over all functions, over all of basic blocks in a function, and finally over all instructions in a basic block. It performs its writing at the instruction-level, matching the Call instruction and arithmetic (FAdd, FSub, FMul, FDiv) and conversion (FPToSI, FPToUI) floating-point operations. When handling the Call instruction, the transformation detects

and handles elementary functions (chapter 3).

Loop Bound Symbolic execution maintains symbolic state for each path; path explosion can exhaust resources and prevent symbolic execution from reaching interesting program points. Loops exacerbate this problem. The loop bound transformation takes a bound parameter that rewrites every loop in its input to obey.

Arbitrary Precision In Section 5.2, we propose classifier that separates likely to be avoidable over- and under-flows from those that are likely to be unavoidable. A certain example of an unavoidable exception is $\frac{\lambda}{2}$. The idea is to transform a program that throws a floating-point exception into an equivalent program with its floats replaced with arbitrary precision numbers, then run it to check whether if it 1) completes and 2) that the result can be converted into floating-point without over- or under-flowing. The arbitrary precision transformation (AP), converts each usage of a float type to mpq_t, the GMP library's arbitrary precision rational type. AP converts each arithmetic operation to the corresponding GMP function. Handling structures was a challenge we had to overcome. AP recursively traverses each structure, transforming the type of float fields to mpq_t. Function calls presented another challenge: AP must distinguish between internal and external functions. AP changes the prototype of an internal function while marshaling and unmarshaling the parameters to external functions.

4.2 Analysis

KLEE does not support floating point numbers, because its underlying SMT solver is STP, which uses bit-vector theory, but does not support the theory of the reals. We modified KLEE 1) to use the Z3 SMT solver from Microsoft, which supports the theory of real numbers and uninterpreted functions and 2) to support floating-point symbolic variables, update its internal expressions when encountering floating-point operations, and output those expressions, with the symbolic variables labeled with type real , as input to Z3. To implement

linearization (Section 3.3.2) and find the roots of polynomials, we extended KLEE to use the GSL polynomial solver package and to internally represent every expression as a rational function, *i.e.* a fraction of two polynomials.

LLVM's `select` instruction does not have a direct representation as an Z3 abstract syntax tree (AST). The syntax of the select instruction is `select (cond, expr1, expr2)`. If `cond` is true, the value of the `select` expression is `expr1`; otherwise it is `expr2`. To construct a Z3 AST for the `select` instruction, we create a new symbolic variable and add a constraint to capture its semantics:

```
(cond = true && expr = expr1)
|| (cond = false && expr = expr2)
```

Limitations Like other symbolic execution engines, Ariadne does not handle dynamically allocated objects and its analysis is expensive [1]. Currently, we restrict symbolic variables to floating-point parameter and assign random values to integer parameters. We do not handle parameters whose type is pointer or **struct**. We intend to support these features in the future.

5. Evaluation

We first motivate our creation of a new multivariate, nonlinear solver, then present exceptions it found and discuss its performance as a tester and verifier. Ariadne finds many overflows and underflows (Xflows), many of which may be an unavoidable consequence of finite precision. We close with the presentation of an Xflow classifier that seeks to separate avoidable from unavoidable Xflows and the $\mathbb{F} \to \mathbb{Q}$ type transformation on which it rests.

We ran our experiments on a machine running Ubuntu 10.04.2 LTS (kernel 2.6.32-30-server) with 128GB RAM and Intel Xeon X7542 6-Core 2.66GHz 18MB Cache, for a total of 18 cores. The Ariadne engine is described in chapter 4. We choose to analyze the special functions of the GNU scientific library (GSL) version gsl-1.14, because the first phase of Ariadne focuses on scalar functions and most of GSL's special functions take and return scalars. The GSL is a mature, well-maintained, well-tested, widely deployed scientific library [10]. It is for these reasons that we selected it for analysis: finding nontrivial exceptions in it is both challenging and important. In spite of these challenges, Ariadne found nontrivial exceptions within it.

We transform each special function in the GSL library to make potential floating-point exceptions explicit. Before each floating-point operation, we test its operands to check for exceptions. The bodies of these checks throw the relevant exception and contain a program point that can only be reached if the guarded operation can throw the specific floating-point exception, as described in Section 3.2.

5.1 Analysis Performance and Results

External functions and loops cause difficulties for static analyzes, like the symbolic analysis underlying RKLEE. We handle a subset of external functions as described in Section 3.2.2. For loops, a classic workaround tactic is to impose a timeout. Another is to bound the loops. We did both. We imposed two timeouts — per-query and per-run. To bound loops, we wrote a transformation that injects a bound into each loop, as described in Section 4.1. Although we could have restricted the use of this loop bound transformation to functions whose analysis failed, we re-ran the analysis on all functions at each loop bound, seeking the Pareto optimal balance of loop bound and time-to-completion. The infinity loop bound means that we did not apply our loop bound transformation.

Our goal is to build a precise and practical tool to detect floating-point exceptions. To this end, we hybridized the Ariadne solver with the Z3: we handed each query, even multivariate, nonlinear ones, to Z3 first, and only queried the Ariadne solver on those queries to which Z3 reported UNKNOWN. We performed the analysis described here with maximum concretizations set to 10 as we found that this setting effectively balances performance and precision, and is a testament to the effectiveness of our findBindings heuristic (Algorithm 3). This hybrid solver (RKLEE) found a total of 2288 input-to-location pairs that triggered exceptions, distributed as shown in Figure 5.1.

Multiple paths or multiple inputs along a single path might reach an exception-triggering program point. If that exception is a bug, its fix might require a single condition, implying that all the bug-triggering inputs are in an equivalence class, or logic bounded by the number of distinct inputs that trigger it. Thus, we report exceptions as $[X,Y]$: X counts unique exception locations found and is a lower-bound on the number of potential bugs and Y counts the unique input to location pairs and is an upper-bound.

Figure 5.2 shows the distribution of functions in terms of the potential exceptions they contain. As expected, this distribution is skewed: Given the maturity and popularity of the GSL, it is not surprising that most of its functions contain no latent floating-point

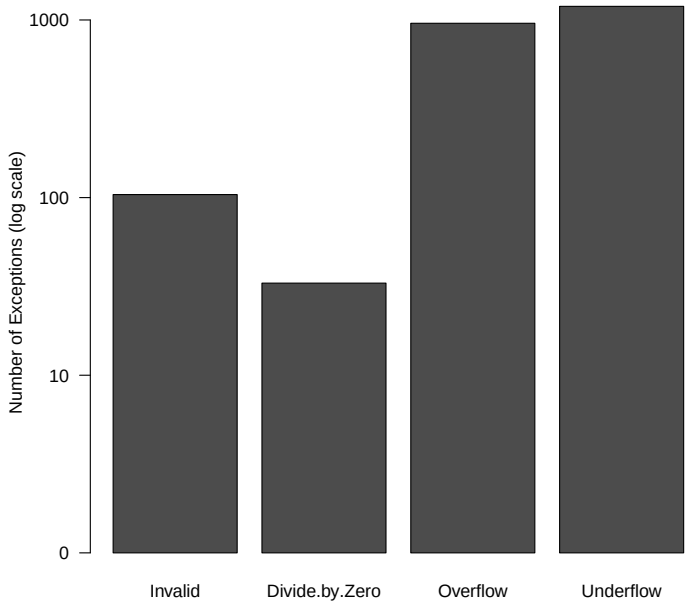

Figure 5.1: Ariadne found 2288 input-to-location pairs that trigger floating-point exceptions in the GNU Scientific library, distributed as shown.

exceptions. Ariadne reports 84 exceptions in gsl_sf_bessel_Inu_scaled_asympx_e defined in `gsl/specfunc/bessel.c`. For each operation, the Ariadne transformation introduces two paths to the operation each time it injects the call fabs to compute the absolute value of an operand. Thus, when the number of operations is O, it can add 2^O paths in the worst case. Further, we count distinct exception-triggering input to location pairs across all paths in Figure 5.2. For a single exception location, Ariadne can report multiple inputs that lead to the exception. Most of our constraints are nonlinear. For example, line 299 of this file is "**double** mu = 4.0∗nu∗nu;". Ariadne finds two different values for nu, *viz.* $-7.458341e-155$ and its absolute value $7.458341e-155$, that satisfy the conditional guarding Underflow. The computation of absolute value that Ariadne injects also creates new paths to an injected exception-throwing program point. Consider line 302 "**double** pre = 1.0/ sqrt (2.0∗M_PI∗x);". Note that the expression passed to sqrt actually contains only one multiplication because the compiler replaces the constant multiplication 2.0∗M_PI with a single constant. To check for Overflow in this sqrt, Ariadne injects the new conditional (written here in pseudocode)

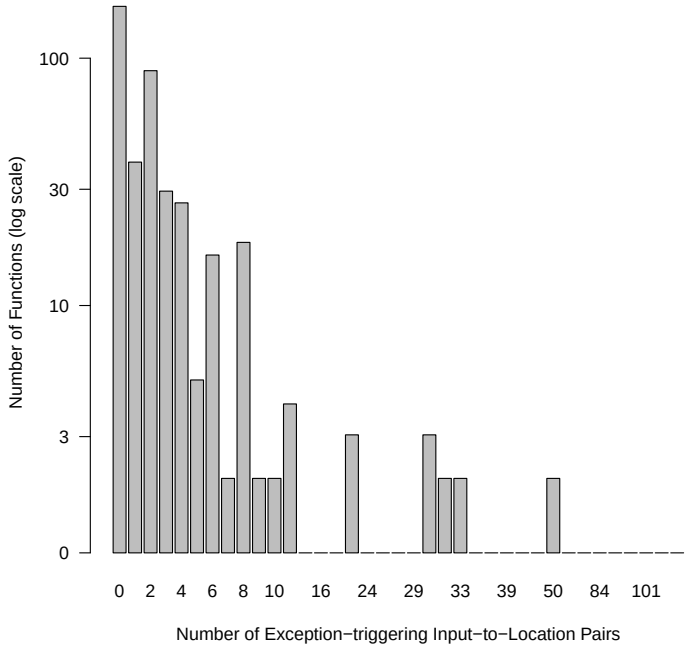

Figure 5.2: Bar plot of the number of functions (y-axis) with the specified number of input-to-location pairs that trigger floating-point exceptions (x-axis).

Function	Line	Inputs
gsl_sf_conicalP_1	989	2.000000e+01, 1.000000e+00
gsl_sf_bessel_Jnu_asympx_e	234	0.000000e+00, 0.000000e+00
gsl_sf_exprel_n_CF_e	89	-1.000000e+00, 1.340781e+154
gsl_sf_bessel_Knu_scaled_asympx_e	317	0.000000e+00, 0.000000e+00

Table 5.1: A selection of 4 of the 33 Divide-by-Zero exceptions Ariadne found.

```
1  y = (2.0*M_PI*x) if x > 0
2  y = -(2.0*M_PI*x) otherwise
3  if (y > DBL_MAX) Overflow
4  if (y < DBL_MIN) Underflow
5  double pre  = 1.0/sqrt(y);
```

In this example, Ariadne finds $x = 2.861117e+307$ that traverses 1, 3 and $x = -2.861117e+307$ that traverses 1, 2, 3; both inputs satisfy the conditional and trigger Overflow.

Perhaps the most serious exceptions are Divide by Zero and Invalid exceptions. Table 5.1 and Table 5.2 show the functions in which we found this type of exceptions, the line number, and exception triggering inputs for a selection of these exceptions. For example, Ariadne

Function	Line	Inputs
gsl_sf_bessel_Inu_scaled_asymp_unif_e	361	-2.225074e-308, 0.000000e+00
gsl_sf_bessel_Knu_scaled_asympx_e	317	-7.458341e-155, -6.445361e+02
gsl_sf_bessel_Jnu_asympx_e	234	-5.000000e-01, 0.000000e+00
gsl_sf_bessel_Inu_scaled_asympx_e	302	-1.500000e+00, -4.744681e+03

Table 5.2: A selection of 4 of the 104 Invalid exceptions Ariadne found.

Loop Bound	No Timeout	Functions Discharged	Exception-free	$\frac{\lvert S\rvert}{\lvert Q\rvert}$	$\frac{\lvert \bar{S}\rvert}{\lvert Q\rvert}$	$\frac{\lvert U\rvert}{\lvert Q\rvert}$	Ariadne Ratio	Exceptions Unique	Exceptions Shared	Total Time (hours)
infinity	319	88	38	0.55	0.22	0.23	0.67	[32,231]	[595,784]	94
64	321	91	38	0.57	0.30	0.12	0.80	[0,181]	[592,793]	104
32	323	91	38	0.57	0.30	0.12	0.79	[5,187]	[603,796]	101
16	318	91	38	0.57	0.30	0.13	0.80	[1,165]	[583,786]	100
8	321	91	38	0.57	0.30	0.13	0.79	[1,180]	[594,795]	101
4	321	91	38	0.57	0.30	0.13	0.79	[0,177]	[580,786]	103
2	320	91	38	0.57	0.30	0.12	0.80	[3,183]	[593,786]	108
1	319	91	38	0.57	0.30	0.13	0.79	[1,176]	[587,788]	102

Table 5.3: The results of RKLEE's analysis of the 424 GSL special functions at the specified loop bounds. Here, Q is the set of all queries issued during analysis and S, $\bar{S}$ and U partition Q into its SAT, UNSAT and UNKNOWN queries.

found both an Invalid and a Divide-by-Zero exception in gsl_sf_bessel_Knu_scaled_asympx_e from `gsl/specfunc/bessel.c`. Line 317 is "**double** pre = sqrt(M_PI/(2.0∗x));". When $x < 0$, sqrt throws Invalid; when $x = 0$, it throws Divide-by-Zero. The precondition of this function is $x >> nu*nu+1$; no explicit precondition on nu exists and the function is public. Ariadne independently discovers an implication of this precondition, which is a testament to its utility. Rather than rely on this precondition, defensive programming suggests adding the conditional **if**$(x > nu*nu+1)$ to check for domain error.

Table 5.3 shows the results from our analysis of the scalar GSL special functions. The data shows that, over the GSL special functions, the loop bound has little impact on the results. We believe that the reason for this is that the analyzed functions do not make heavy use of looping constructs.

RKLEE attempts to explore all paths within the time bound it is given. When RKLEE does not time out on a particular function, that function is added to the second, "No Timeout", column of the table. RKLEE *discharges* a function when it determines that every path is either satisfiable or unsatisfiable. The count of discharged functions is recorded in the third "Discharged" column. RKLEE found no exceptions in some of the discharged functions: these functions are "Exception-free". The three function columns have the property that the "No

Timeout" functions contain the "Discharged" functions which contain the "Exception-free" functions. The difference between "No Timeout" and "Discharged" is path abandonment when a query result is UNKNOWN; the difference between "Discharged" and "Exception-free" is the discovery of exceptions. RKLEE performs both testing, when it discovers an exception-triggering input, and verification, its determination that a function is exception-free is sound, subject to the loop bound setting used. We note that, while most of the exception-free functions at the infinity bound are loop-free, four are not.

The loop bound transformation monotonically reduces the number of paths in a program, so each of the function columns should be monotonically increasing. This pattern does not hold "No Timeout". There are two reasons for this variation. First, Z3 probabilistically hangs on some of the nonlinear, multivariate constraints RKLEE feeds it, in spite of its per-query timeout. Second, KLEE flips coins during path scheduling to ensure better coverage. In short, some of the analysis runs at certain loop bounds were simply unlucky.

The three Q columns report on RKLEE's overall effectiveness. Recall that in RKLEE, Ariadne only handles Z3's unknown queries. The Ariadne ratio column reports Ariadne effectiveness at solving Z3's unknowns in our context. Let $S_A, \bar{S}_A$ and U_A be the SAT, UNSAT and UNKNOWN queries that Ariadne handles. The Ariadne ratio then is $\frac{|S_A \cup \bar{S}_A|}{|S_A \cup \bar{S}_A \cup U_A|}$. In short, Ariadne found a substantial fraction of the exceptions and solved 67–80% of the queries that Z3 was unable to handle.

The "Unique" exception field shows exceptions not found by any other RKLEE analysis at a different loop bound. Let E_b be the exceptions found at loop bound b. Let E_b^u be the unique exceptions at b and E_b^s be the shared exceptions found at some other loop bound. $E_b^u = E_b - \bigcup_x E_x, \forall x \in B - \{b\}$. So at 32, the analysis discovered 5 unique exception-triggering locations and 187 unique pairs of input to a location that triggers an exception.

Our analysis is embarrassingly parallelizable, so the test harness forms a partition of functions for each core (in our case 18), then analyzes each function, one by one, against each loop bound. It records the analysis time per function per loop bound. The sum of all the analysis times for all the functions at each loop bound is reported in the total time

column. Again, we see that, with our corpus, the loop bound does not have much impact on analysis time.

5.2 Classifying Overflows and Underflows

Some overflows and underflows (Xflows) are an inevitable consequence of finite precision. The addition $\Omega + \Omega$ is a case in point. Some Xflows, however, may be avoided by modifying (*e.g.* changing the order of evaluation) or replacing the algorithm used to compute the solution. Because a developer's time is precious, we introduce the *avoidable Xflow classifier* that seeks to distinguish potentially avoidable Xflows from those that are an unavoidable consequence of finite precision.

Let $A : \mathbb{F}^n \to \mathbb{F}$ be an algorithm implemented using floating-point and $\mathscr{A}$ be that same algorithm implemented using an arbitrary precision $\mathbb{Q}$. When $A(\vec{i})$ overflows or underflows, we deem that exception *potentially avoidable* if $\mathscr{A}(\vec{i}) \in [-\Omega, \Omega] \wedge |\mathscr{A}(\vec{i})| \geq \lambda$.

This classifier rests on the intuition that, if the result of an arbitrary computation over arbitrary precision is a floating-point number, then it may be possible to find an algorithm that can compute that same result over floating-point without generating intermediate values that trigger floating-point exceptions. Unfortunately, this classifier is imperfect. First, Xflows the classifier deems unavoidable may, in fact, be avoided or mitigated via operand guards (input sanitization). For instance, x + y **if** x $<$ DBL_MAX/2 && y $<$ DBL_MAX/2 is safe when $x, y \leq \frac{\Omega}{2}$. Second, the fact that the result of a computation is a float-point number may depend on intermediate result that is not. For instance, consider a function that takes $f \in \mathbb{F}$ as input, computes $x := f\Omega$ and returns 2 if $x > 2\Omega$ and 1 otherwise. The result of this function is always an integer, but the fact, over $\mathbb{Q}$, it returns 2 when $f \geq 2$ greater than or equal to 2 does not imply that a corresponding floating-point algorithm exists that can produce 2.

To realize our Xflow classifier, we wrote a transformer that takes a numeric program and replaces its **float** and **double** types with an arbitrary precision rational type, for whose

	Overflows	Underflows
Interesting	98	250
Total	316	272
Ratio	0.31	0.92

Table 5.4: Potentially avoidable overflows and underflows.

implementation we use gmpq_t from GMP [9]. In addition to rewriting types, our arbitrary precision transformer also rewrites floating-point operations into rational operations. For instance, z = x + y becomes gmpq_add(z,x,y).

Xflows dominated the exceptions we found, comprising 94% of all exceptions. We defined and realized our classifier with the aim of filtering these Xflows. Nonetheless The number of potentially avoidable Xflows remains high in Table 5.4. Note that GSL is a worst case for us. In a numerical library such as the GSL, these ratios are high because the range of many mathematic functions is small but their implementation relies on floating-point operations whose operands have extremal magnitude.

6. Related Work

This section surveys closely related work. First, Ariadne is related to the large body of work on symbolic execution [20], such as recent representative work on KLEE [4] and DART [12]. Our work directly builds on KLEE and uses the standard symbolic exploration strategy. To the best of our knowledge, it is the first symbolic execution technique applied to the detection of floating-point runtime exceptions. We have proposed a novel programming transformation concept to reduce floating-point analysis to standard reasoning on real arithmetic and developed practical techniques to solve nonlinear constraints.

Our work is also related to static analysis techniques to analyze numerical programs. The main difference of our work to these techniques is our focus on bug detection rather than proving the absence of errors—every exception we detect is a real exception, but we may miss errors. We briefly discuss a few representative efforts in this area. Goubault [14] develops an abstract interpretation-based static analysis [2] to analyze errors introduced by the approximation of floating-point arithmetic. The work was later refined by Goubault and Putot [15] with a more precise abstract domain. Martel [22] presents a general concrete semantics for floating-point operations to explain the propagation of roundoff errors in a computation. In later work [23], Martel applies this concrete semantics and designed a static analysis for checking the stability of loops [17]. Miné [25] proposes an abstract interpretation-based approach to detect floating point errors. Underflows and round-off errors are considered tolerable and thus not considered real run-time errors. Monniaux [26] summarizes the typical errors when using program analysis techniques to detect bugs in or verify correctness of numerical programs. Astree [3] is a system for statically analyzing

C programs and attempts to prove the absence of overflows and other runtime errors, both over the integers and floating-point numbers. There are some other approaches to model the floating point computation for numerical programs. Fang *et al.* [6, 7] propose the use of affine arithmetic to model floating-point errors.

7. Conclusion and Future Work

We have presented our design and implementation of Ariadne, a symbolic execution engine for detecting floating-point runtime exceptions. We have also reported our extensive evaluation of Ariadne over a few hundred GSL scalar functions. Our results show that Ariadne is practical, primarily enabled by our novel combination of program rewriting to expose floating-point exceptional conditions and techniques for nonlinear constraint solving. Our immediate future work is to publicly release the tool to benefit numerical software developers. We would also like to investigate approaches to support non-scalar functions, such as those functions that accept or return vectors or matrices.

Bibliography

[1] P. Collingbourne, C. Cadar, and P. H. Kelly. Symbolic crosschecking of floating-point and SIMD code. In *EuroSys*, pages 315–328, 2011.

[2] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 234–252, 1977.

[3] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *ESOP*, pages 21–30, 2005.

[4] D. E. Daniel Dunbar, Cristian Cadar. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[5] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[6] C. F. Fang, T. Chen, and R. A. Rutenbar. Floating-point error analysis based on affine arithmetic. In *Proc. IEEE Int. Conf. on Acoust., Speech, and Signal Processing*, pages 561–564, 2003.

[7] C. F. Fang, R. A. Rutenbar, M. Püschel, and T. Chen. Toward efficient static analysis of finite-precision effects in DSP applications via affine arithmetic modeling. In *DAC*, DAC '03, pages 496–501, 2003.

[8] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007.

[9] FSF. GMP — the GNU multiple precision arithmetic library. `/http://gmplib.org/`.

[10] FSF. GSL — GNU scientific library. `http://www.gnu.org/s/gsl/`.

[11] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, pages 519–531, 2007.

[12] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*,

pages 213–223, 2005.

[13] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[14] E. Goubault. Static analyses of the precision of floating-point operations. In *SAS*, pages 234–259, 2001.

[15] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *SAS*, pages 18–34, 2006.

[16] J. Hauser. Handling floating-point exceptions in numeric programs. *ACM TOPLAS*, 18(2):139–174, Mar. 1996.

[17] N. J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics, 2nd edition, 2002.

[18] IEEE Computer Society. IEEE standard for floating-point arithmetic, August 2008.

[19] W. Kahan. A demonstration of presubstitution for ∞/∞ (Grail), 2005.

[20] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19:385–398, 1976.

[21] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[22] M. Martel. Propagation of roundoff errors in finite precision computations: a semantics approach. In *ESOP*, pages 194–208, 2002.

[23] M. Martel. Static analysis of the numerical stability of loops. In *SAS*, pages 133–150, 2002.

[24] Z. Merali. Computational science: ...error ...why scientific programming does not compute. *Nature*, 467:775–777, October 13 2010.

[25] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP'04*, pages 3–17, 2004.

[26] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30:12:1–12:41, May 2008.

[27] P. Rümmer and T. Wahl. An SMT-LIB theory of binary floating-point arithmetic. In *International Workshop on Satisfiability Modulo Theories (SMT) at FLoC*, 2010.

[28] K. Sen. CUTE: a concolic unit testing engine for C. In *FSE*, pages 263–272, 2005.

[29] P. H. Sterbenz. *Floating-Point Computation*. Prentice-Hall, 1974. ISBN: 0-13-322495-3.

[30] L. A. Times. Toyota's acceleration issue. http://articles.latimes.com/2009/

dec/05/opinion/la-ed-toyota5-2009dec05.

[31] Wikipedia. Ariane 5 flight 501. http://en.wikipedia.org/wiki/Ariane_5_Flight_501.