Automatic Detection of Floating-Point Exceptions

Earl T. Barr
Thanh Vo, Vu Le, and Zhendong Su

Presented by Saheel
Numerical Code

• Ubiquitous
• Safety critical

Google’s Driverless Car
Numerical Code

- Ubiquitous
- Safety critical

Google’s Driverless Car
Numerical Code

- Ubiquitous
- Safety critical
- **Hard to get right**
Contribution

**Ariadne**: A practical symbolic analysis engine that finds witnesses for floating-point exceptions

- Realizes a sweet spot in balance of analysis cost and effectiveness
- Finds over 2,000 *real*, non-rounding, FP exceptions in the GNU Scientific Library
Key Insight

Exception-triggering inputs form dense clusters

• If one input triggers an exception, many do, and one will do so over both real and floating-point arithmetic.
Exception Triggering Inputs
Outline

• Floating-Point Primer
• Ariadne Frontend
• Ariadne Backend
• Evaluation
Floating-Point Format

\[ \pm b_0.b_1b_2\ldots b_{p-1} \times 2^E \]

-1^{\text{sign}} \times 1.\text{fraction} \times 2^{\text{exponent} - \text{bias}}
Floating-Point Number Line

-8
-DBL_MAX
-1.797 x 10^{308}
-DBL_MIN
-2.225 x 10^{-308}
0
DBL_MIN
2.225 x 10^{-308}
DBL_MAX
1.797 x 10^{308}
∞
Floating-Point Number Line

INEXACT is the price for using constant-sized CPU registers for arithmetic.

\[ \text{x is rounded} \]

\[ \text{INEXACT exception thrown} \]
Floating-Point Number Line

$\Omega - \lambda \quad 0 \quad \lambda - \Omega$

Overflow

Usable Range

Usable Range

Overflow

$x + y = \infty$

OVERFLOW exception thrown

$x \quad y \quad x+y$
Floating-Point Number Line

Overflow       Usable Range       Underflow       Usable Range       Overflow

-∞             -Ω                -λ                0                Ω

x - y = ?
UNDERFLOW exception
Floating-Point Number Line

Overflow | Usable Range | Underflow | Usable Range | Overflow

-∞ | -Ω | -λ | 0 | λ | Ω
### Floating-Point Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVALID</td>
<td>0 / 0, 0 × ∞, (\sqrt{-1})</td>
<td>NaN</td>
</tr>
<tr>
<td>DIVIDEBYZERO</td>
<td>Finite nonzero / 0</td>
<td>±∞</td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>Ω + 1</td>
<td>±∞</td>
</tr>
<tr>
<td>UNDERFLOW</td>
<td>(\lambda / 2)</td>
<td>Subnormal</td>
</tr>
<tr>
<td>INEXACT</td>
<td>10 / 3</td>
<td>Rounded value</td>
</tr>
</tbody>
</table>
Floating-Point Numbers

- Floating-point numbers are a “leaky abstraction”
- The semantics of floating-point are intricate and unintuitive
Lack of Associativity

\[ A + \alpha_1 + \alpha_2 + \ldots + \alpha_n \]

Assume \( A \ll \alpha_i \) and \( A = \sum_{i=1}^{n} \alpha_i \)

\[ (((((A + \alpha_1) + \alpha_2) + \ldots) + \alpha_n) = A \]

\[ A + (\ldots((\alpha_1 + \alpha_2) + (\alpha_3 + \alpha_4)) + \ldots + (\alpha_{n-1} + \alpha_n)\ldots) = 2A \]
Catastrophic Cancellation

\[ a = 1.xxx \ xxx1 \]
\[ b = 1.xxx \ xxx0 \]
\[ a - b = 0.000 \ 0001 \ uuu \ uuuu \]
\[ = 1.uuu \ uuuu \times 10^{-7} \]
where each \( u \) is a random, unassigned digit

*Adapted from slides by Gerald W. Recktenwald*
Avoiding Floating-Point Exceptions

How hard is it to prevent floating-point exceptions via “carefully written programs”?
Averaging Two Numbers

\[
\frac{x + y}{2}
\]
Averaging Two Numbers

double av1 ( double x, double y ) {
    return ( x + y ) / 2.0;
}

Averaging Two Numbers

double av1 ( double x, double y ) {
    return ( x + y ) / 2.0;
}

\[
\frac{x + y}{2} + \frac{\Omega + \Omega}{2} = \frac{2\Omega}{2} = \Omega
\]
Averaging Two Numbers

double av1 ( double x, double y ) {
    return ( x + y ) / 2.0;
}

\[
\frac{x + y}{2} + \frac{\Omega + \Omega}{2} = \frac{2\Omega}{2} = \Omega
\]

OVERFLOW
Averaging Two Numbers

double av2 ( double x, double y ) {
    return ( x / 2.0 + y / 2.0 );
}

\[
\frac{x + y}{2} = \frac{\Omega}{2} + \frac{\Omega}{2} = \Omega
\]
Averaging Two Numbers

double av2 ( double x, double y ) {
    return ( x / 2.0 + y / 2.0 );
}

\[ \frac{x + y}{2} = \frac{\lambda}{2} + \frac{\lambda}{2} \]

UNDERFLOW
Averaging Two Numbers

double av1 ( double x, double y ) {
    return ( x + y ) / 2.0;
}

double av2 ( double x, double y ) {
    return ( x / 2.0 + y / 2.0 );
}

⇒ Overflows when x and y are large and have the same sign.

⇒ No overflow but obvious underflow.
Averaging Two Numbers

\[
\frac{x + y}{2} + \frac{x}{2} - \frac{x}{2} = \frac{x + x + y - x}{2} = \frac{2x + y - x}{2} = x + \frac{y - x}{2}
\]
Averaging Two Numbers

Consider \( x = \Omega, \ y = \Omega \)

\[
\Omega + \frac{\Omega - \Omega}{2} = \Omega + \frac{0}{2} = \Omega
\]
Averaging Two Numbers

What about $x = -\Omega$, $y = \Omega$?

\[-\Omega + \frac{\Omega - (-\Omega)}{2} = \Omega + \frac{2\Omega}{2}\]
Averaging Two Numbers

double av1 ( double x, double y ) {
    return ( x + y ) / 2.0;
}

double av2 ( double x, double y ) {
    return ( x / 2.0 + y / 2.0 );
}

double av3 ( double x, double y ) {
    return ( x + ( y - x ) / 2.0 );
}

double av4 ( double x, double y ) {
    return ( y + ( x - y ) / 2.0 );
}

⇒ Overflows when x and y are large and have the same sign.

⇒ No overflow but obvious underflow.

⇒ Overflows when x and y are large and differ in sign.

⇒ Overflows when x and y are large and differ in sign.
Sterbenz’ Average

double average( double x, double y ) {
    int samesign;
    if ( x >= 0 ) {
        if ( y >= 0 )
            samesign = 1;
        else
            samesign = 0;
    } else {
        if ( y >= 0 )
            samesign = 0;
        else
            samesign = 1;
    }

    if ( samesign ) {
        if ( y >= x )
            return av3( x, y );
        else
            return av4( x, y );
    } else
        return av1( x, y );

} //average

That’s a lot of work and it only prevents overflow.
double av1 ( double x, double y ) {
    return ( x + y ) / 2.0;
}

Ariadne

Overflow, line 2
x = \Omega, y = \Omega
double av3 ( double x, double y ) {
    return ( x + ( y - x ) / 2.0 );
}

Overflow, line 2
x = -Ω, y = Ω
Ariadne’s Design Principles

• Convert floating-point detection into a reachability problem
• Model floating-point arithmetic with real arithmetic
Ariadne Architecture

- Input Code (C, C++, Fortran)
- Reify FP Exceptions
- Input Code + Explicit FP Exceptions
- Symbolic Execution
- Constraints

Ariadne Frontend
Rewriting Addition

double av1 ( double x, double y ) {
    return ( x + y ) / 2.0;
}

Rewriting Addition

double av1 ( double x, double y ) {
    double z = x + y;
    return z / 2.0;
}

Rewriting Addition

double av1 ( double x, double y ) {
    double z = x + y;
    double abs1 = fabs( z );
    if ( DBL_MAX < abs1 ) {
        perror( "OVERFLOW!\n" );
        exit( 1 );
    }

    return z / 2.0;
}
Rewritten Program’s CFG

\[ z = x + y \]

DBL_MAX < fabs(z)

result = z

STOP

OVERFLOW

UNDERFLOW
Rewriting Addition

double av1 ( double x, double y ) {
    double z = x + y;
    double abs1 = fabs( z );
    if (DBL_MAX < abs1 ) {
        perror( "OVERFLOW!\n" );
        exit( 1 );
    }
    if (0 < abs1 && abs1 < DBL_MIN) {
        perror( "UNDERFLOW!\n" );
        exit( 1 );
    }
    return z / 2.0;
}
Rewritten Program’s CFG

\[ z = x + y \]

\[ 0 < \text{fabs}(z) \land \text{fabs}(z) < \text{DBL}_{\text{MIN}} \]

\[ \text{result} = z \]

\[ \text{DBL}_{\text{MAX}} < \text{fabs}(z) \]

STOP

OVERFLOW

UNDERFLOW
Reifying Floating-Point Exceptions

Exceptions are now explicit! Detecting exceptions mapped to reaching them.

\[ z = x + y \]

\[ 0 < \text{fabs}(z) \land \text{fabs}(z) < \text{DBL}_\text{MIN} \]

\[ \text{DBL}_\text{MAX} < \text{fabs}(z) \]

\[ \text{result} = z \]

OVERFLOW

UNDERFLOW
Elementary Functions

- log, sqrt, cos, sin, pow, ...
- “External functions”
  - Some handled in hardware
double y = log(x); ...
if ( x > 1 ) {
  ...
} ...
if ( y > 0 ) {
  ...
}

if ( x <= 0 ) {
  perror( "INVALID!\n" );
  exit( 1 );
}
double y = d; ...

Generating Path Constraints

Input Code (C, C++, Fortran) → Reify FP Exceptions → Input Code + Explicit FP Exceptions → Symbolic Execution → Constraints over $\mathbb{R}$

Ariadne Frontend
int d = read();

if (d == 17)
    abort();
else
    d = d + 7;

printf("continue");
Symbolic Execution -- using KLEE

```c
int d = read();
d = d + 7;
if (d == 17) abort();
printf("continue");
```

- `d = 17`  
  - PC: `s = 10`

- `d = s`  
  - PC: `∅`

- `d = s + 7`
  - PC: `∅`

- `d = s + 7`  
  - PC: `s ≠ 10`
Ariadne Path Constraints (PCs)

Multivariate 73% 62% Nonlinear 81%
Satisfiability Modulo Theories

• Boolean satisfiability (SAT)
  – \((x \lor y) \land z\)

• Satisfiability Modulo Theories (SMT)
  – SMT generalizes Boolean SAT to functions and predicates
  – Theories include real numbers, arrays, bit vectors, and uninterpreted functions
    • \(5x + 7y - z \leq 17\)
    • \(\text{select(store(store(a,1,-1),2,-2),1)}\)
SMT solvers

• Z3 by Microsoft
  – SMT solver with nlsat (non-linear arithmetic) support

• Z3-Ariadne
  – custom multi-variate non-linear constraint solver
Z3-Ariadne pipeline

Path constraints over $\mathbb{R}$

Multivariate Nonlinear

Concretize

Univariate Nonlinear

Solve polynomial

Roots, $R(x)$

Build formula

Disjunction of intervals

SMT Solver
Concretizing Multi-variante PCs

Authors found that multiple guesses don’t improve perf. by much
Nonlinearity remains...

Path constraints over $\mathbb{R}$

$x^3 + 2x^2y + 7x + 12 \geq 0$

Concretize

$x^3 + 10x^2 + 7x + 12 \geq 0$

Solve polynomial

Polynomial solver from GSL

Roots, $R(x)$

Build formula

Disjunction of intervals

SMT Solver
Nonlinear Univariate Constraints

• Normalize nonlinear univariate constraints
  – $R(x)$ \( \not\leq 0 \)

• $R(x) = (x - r_1)(x - r_2)...(x - r_n)$
  – Real roots determine the sign of $R(x)$
Real Roots Determine Sign

\[ R(x) \]

- \( x - r_1 \Rightarrow \text{positive} \)
- \( x - r_2 \Rightarrow \text{positive} \)
- \( x - r_3 \Rightarrow \text{negative} \)
- \( R(x) \Rightarrow \text{negative} \)
Nonlinear Univariate Constraints

• In our example, \( R(x) = x^3 + 10x^2 + 7x + 12 \)
  – and we want \( R(x) \) to be \( \geq 0 \)
Disjunction for $R(x) < 0$

$x < r_1 \lor r_2 < x \land x < r_3 \Rightarrow R(x) < 0$
Disjunction for $R(x) > 0$

$r_1 < x \land x < r_2 \lor r_3 < x \Rightarrow R(x) > 0$
Disjunction for $R(x) = 0$
Feed intervals to SMT solver

Path constraints over $\mathbb{R}$

$\begin{align*}
x^3 + 2x^2y + 7x + 12 & \geq 0 \\
x^3 + 10x^2 + 7x + 12 & \geq 0
\end{align*}$

Concretize

Solve polynomial

Roots, $R(x)$

Build formula

Disjunction of intervals

SMT Solver
Ariadne Architecture (revisited)

From the solver

$x \in \mathbb{R}$

$y = 5$

“guessed”, during concretization
Result Handling

• The SMT solver returns a solution over \( \mathbb{R} \)
  – NOT floating-point number

\[ x = \frac{10}{3} \]
findFloat

\[ x \in [l, h] \lor x \in [l', h'] \lor \ldots \]
findFloat

\[
x \notin [l, h] \lor x \in [l', h'] \lor \ldots
\]

Initially, \( x = a \in [l, h] \)
Eliminate “useless” intervals

\[
a_f \quad l \quad a_f \geq l \quad ? \quad a \quad \text{next}(a_f) \quad h \quad \text{next}(a_f) \quad \leq h \quad ?
\]
At this point we have inputs that are
- normal floating-point numbers
- guaranteed to trigger an exception
- when symbolically executed over arbitrary precision rationals
Problems with Real Arithmetic

\[ r = 10^{54} + 42 - 10^{54} \]

if( \( r == 42 \) )
    printf("Path under \( \mathbb{R} \).\n");
else
    printf("Path under FPA.\n");
Escape from Real Arithmetic

Testing the trigger
- Ariadne concretely executes a candidate exception-trigger input over floating-point to confirm it triggers the specified exception.
Complete Ariadne Architecture

Input Code (C, C++, Fortran) → Ariadne Frontend → Constraints over $\mathbb{R}$ → Solver → Results over $\mathbb{R}$

Find Floats

Run program on candidate input

Exception Triggering Inputs
Overflow, line 2
\( x = -\Omega, \ y = \Omega \)

Underflow, line 11
\( x = 0, \ y = 100 \)

DivideByZero, line 100
\( x = -3, \ y = 12 \)

…
Ariadne is sound

But not complete, i.e. it cannot guarantee that a program is exception-free, because it:

• approximates FP semantics with constraints over the reals, which notably ignore rounding (i.e. Inexact);
• concretizes multivariate constraints into univariate ones;
• rewrites loop conditions to control the number of iterations (Symbolic Execution path explosion prob.)
Evaluation

• GNU Scientific Library (GSL)
  – Widely used and well-tested library

• 467 special functions with scalar input/output
2091 Exception-Triggering Inputs

Number of Exceptions (log scale)

Invalid: 125
Divide.by.Zero: 44
Overflow: 805
Underflow: 1117
A Divide By Zero

gsl/specfunc/legendre_con.c

867: gsl_sf_conicalP_1_e(const double lambda,
           const double x, gsl_sf_result *result)

When \( \lambda = 20 \) and \( x = 1 \), after 113 lines and 8 control points, we reach

980:  const double sqrt_xm1 = sqrt(x - 1.0);
981:  const double sqrt_xp1 = sqrt(x + 1.0);
982:  const double sh = sqrt_xm1 * sqrt_xp1;
...
989:  int stat_V =
       conicalP_1_V( th, x/sh, lambda, 1.0, &V0, &V1);
Unavoidable Xflows

\[ \Omega + \Omega = 2\Omega \]

\[ \frac{\lambda}{2} = \text{subnormal} \]
Interesting Xflows

\[ x_1 \oplus x_2 \oplus x_3 \oplus \ldots \oplus x_n = ? \]
Over floating-point

\[ x_1 \oplus x_2 \oplus x_3 \oplus \ldots \oplus x_n = z \quad \land \quad |z| \in [\lambda, \Omega] \]
Over the reals

\[ \oplus \in \{+,-,*,/\} \text{ is an arithmetic operation} \]
# Interesting Xflows

<table>
<thead>
<tr>
<th></th>
<th>Overflows</th>
<th>Underflows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avoidable?</td>
<td>98</td>
<td>180</td>
</tr>
<tr>
<td>Total</td>
<td>155</td>
<td>217</td>
</tr>
<tr>
<td>Ratio</td>
<td>0.63</td>
<td>0.83</td>
</tr>
</tbody>
</table>
Questions?