Effective Floating-Point Analysis via Weak-Distance Minimization

Zhoulai Fu
IT University of Copenhagen, Denmark
zhfu@itu.dk

Zhendong Su
ETH Zurich, Switzerland
zhendong.su@inf.ethz.ch

Abstract

This work studies the connection between the problem of analyzing floating-point code and that of function minimization. It formalizes this connection as a reduction theory, where the semantics of a floating-point program is measured as a generalized metric, called *weak distance*, which *faithfully* captures any given analysis objective. It is *theoretically guaranteed* that minimizing the weak distance (*e.g.*, via mathematical optimization) solves the underlying problem. This reduction theory provides a general framework for analyzing numerical code. Two important separate analyses from the literature, branch-coverage-based testing and quantifier-free floating-point satisfiability, are its instances.

To further demonstrate our reduction theory's generality and power, we develop three additional applications, including boundary value analysis, path reachability and overflow detection. Critically, these analyses do not rely on the modeling or abstraction of floating-point semantics; rather, they explore a program's input space guided by *runtime computation and minimization* of the weak distance. This design, combined with the aforementioned theoretical guarantee, enables the application of the reduction theory to real-world floating-point code. In our experiments, our boundary value analysis is able to find all reachable boundary conditions of the GNU `sin` function, which is complex with several hundred lines of code, and our floating-point overflow detection detects a range of overflows and inconsistencies in the widely-used numerical library GSL, including two latent bugs that developers have already confirmed.

CCS Concepts • **Theory of computation → Program analysis**.

Keywords Program Analysis, Mathematical Optimization, Theoretical Guarantee, Floating-point Code

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314632

ACM Reference Format:
Zhoulai Fu and Zhendong Su. 2019. Effective Floating-Point Analysis via Weak-Distance Minimization. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3314221.3314632

1 Introduction

Modern infrastructures, from aerospace and robotics to finance and physics, heavily rely on floating-point code. However, floating-point code is error-prone, and reasoning about its correctness has been a long-standing challenge. The main difficulty stems from the subtle, albeit well-known, semantic discrepancies between floating-point and real arithmetic. For example, the associativity rule in real arithmetic $a+(b+c) = (a+b)+c$ does not hold in floating-point arithmetic.[1] Consider the C code in Fig. 1(a) for a further motivating example. We assume the rounding mode is the default round-to-nearest as defined in the IEEE-754 standard. The code may appear correct upon first sight. However, if we set the input to 0.999 999 999 999 999 9, the branch "`if (x < 1)`" will be taken, but the subsequent "`assert (x + 1 < 2)`" will fail (`x + 1 = 2` in this case). Now, if we run the same code under a different rounding mode, say round-toward-zero, the assertion becomes valid.

```
void Prog(double x) {
  if (x < 1){
    x = x + 1;
    assert(x < 2);
  }}
```

(a)

```
void Prog(double x) {
  if (x < 1){
    x = x + tan(x);
    assert(x < 2);
  }}
```

(b)

Figure 1. Motivating examples: Do the assertions hold?

To reason about such counterintuitive floating-point behavior, one may believe that a formal semantic analysis is necessary. Indeed, state-of-the-art SMT solvers such as MathSAT [10] can determine that the floating-point constraint $x < 1 \wedge x + 1 \geq 2$ is satisfiable under the round-to-nearest mode and unsatisfiable under the round-toward-zero mode. However, the tight coupling between analysis and semantics

[1] One may verify that, on a typical x86-64 machine with the round-to-nearest mode, $0.1+(0.2+0.3) = 0.6$, but $(0.1+0.2)+0.3 = 0.600\,000\,000\,000\,000\,1$.

can quickly become problematic for real-world code involving floating-point arithmetic operations, nonlinear relations, or transcendental functions. Suppose we change "`x = x + 1`" to "`x = x + tan(x)`" as shown in Fig. 1(b). SMT-based methods will find it difficult to reason about `tan(x)` because the implementation of `tan(x)` is system-dependent, and its semantics is not standardized in IEEE-754 [1].

Recent work has introduced two highly effective floating-point analyses that do not need to directly reason about program logic [16, 17]. Both analyses drastically outperform traditional techniques. The first is for achieving high branch coverage in floating-point code [17]. It transforms the program under test into another program whose minimum points trigger uncovered conditional branches. The analysis achieves an average of 90%+ branch coverage within seconds on Sun's math library code. The second concerns floating-point constraint satisfiability [16]. A floating-point formula in conjunctive normal form (CNF) is transformed into a program whose minimum points correspond to the models of the formula. A Monte Carlo optimization backend is applied to find these models, if any. The solver produces consistent satisfiability results as both MathSAT and Z3 with average speedups of over 700X on SMT-competition benchmarks.

This work generalizes these ideas and develops a unified theory that applies to a broad category of floating-point analysis problems. The theory consists of a faithful reduction procedure from the problem domain of floating-point analysis to the problem domain of mathematical optimization. At the core of theory, floating-point program semantics is measured as a generalized metric called *weak distance*, and it is guaranteed that minimizing the weak distance, such as via mathematical optimization (MO), leads to a solution to the underlying floating-point analysis problem, and vice versa.

Our reduction offers a key practical benefit. As modern MO techniques work by executing an objective function, our approach does not need to analyze floating-point program semantics, an intractable and potentially cost-ineffective task. Instead, it directs input space exploration by executing another, potentially simpler floating-point program that models the weak distance to capture the desired analysis objective.

It is a common misconception that MO is useful only for continuous objective functions — continuity is preferred, but not necessary. Modern, advanced algorithms exist that can handle functions with discontinuity, high-dimensionality, or time-consuming computation. Our approach uses MO techniques as black-boxes and can directly benefit from the state-of-the-art. Our main contributions follow:

- We develop a reduction theory for floating-point analysis via mathematical optimization. To our knowledge, this is the first rigorous investigation of the general connection between the two problem categories. In particular, we have introduced the concept of weak distance that enables viewing the analysis problem regarding a floating-point program equivalently as the optimization problem of another floating-point program.
- We study three instances of the theory, including boundary value analysis, path reachability, and floating-point overflow detection. These problems are known to be important, challenging, and have been treated separately in the literature. The proposed reduction technique allows us to approach these distinct problems uniformly in the same theory and to effectively realize them under a common implementation architecture.
- We conduct a set of experiments to empirically validate our approach. Our boundary value analysis is able to trigger all reachable boundary conditions of the GNU `sin` function, which is complex with several hundred lines of code. Our floating-point overflow detection has detected a range of overflows, inconsistencies, and two latent, already confirmed bugs in the widely-used GSL.

To facilitate the exposition of our approach and its reproduction, we provide a variety of examples throughout the paper. We believe that these examples also help inform the reader both our approach's strengths and limitations.

The rest of the paper is organized as follows. Section 2 formulates the problem, and Section 3 develops our reduction theory. Section 4 illustrates the theory with three examples. Section 5 lays out the implementation architecture and discuss the limitations of our approach, while Section 6 presents our experiments and results. Section 7 surveys related work, and finally Section 8 concludes.

Notation. As usual, we denote the set of integers and real numbers by $\mathbb{Z}$ and $\mathbb{R}$ respectively. We write $\mathbb{F}$ for the set of floating-point numbers of the IEEE-754 `binary64` format. We will often write $\vec{x}$ to represent an N-dimensional floating-point vectors $(x_1, \ldots, x_N)$ in $\mathbb{F}^N$.

2 Floating-Point Analysis

The term *floating-point analysis* in this paper refers to a broad class of problems in program analysis, testing, and verification, where the goal is to determine if floating-point program is correct or has some desired properties. In this section, we first briefly review the semantic constructs of the problem, formulate, and then frame it as a problem of finding unsafe inputs. We show five instances of the defined problem, three of which will be studied in later sections (the other two were explored in the literature).

2.1 Problem Formulation

A large body of research on floating-point analysis can be characterized as semantic methods. Reasoning about a program `Prog` relies on its semantics, a mathematical object $[\![\texttt{Prog}]\!]$ that models the possible states of the program. The correctness of the program is usually formulated as $[\![\texttt{Prog}]\!] \cap \Omega = \emptyset$, where Ω refers to the set of unsafe program states.

We formulate floating-point analysis from a program input perspective, that is to determine whether an input exists such that executing the program on that input can go wrong.

Definition 2.1. Let Prog be a program with an input domain $\text{dom}(\texttt{Prog}) = \mathbb{F}^N$, where N is the number of its inputs. Let S be a subset of the input domain of Prog. A floating-point analysis problem, denoted by

$$\langle \texttt{Prog}; S \rangle, \tag{1}$$

is the process of seeking an element of S. We say that an algorithm solves the floating-point analysis problem if the following two conditions hold:

(a) The algorithm must find an $\vec{x} \in S$ if $S \neq \emptyset$.
(b) The algorithm must report "not found" if $S = \emptyset$.

In the definition above, we have framed the analyzed program to have only floating-point input parameters. That is useful (although not always necessary in practice) when we reduce the problem to optimization techniques. The set S is usually implicitly defined as inputs triggering unsafe states, $S \stackrel{\text{def}}{=} \{\vec{x} \in \text{dom}(\texttt{Prog}) \mid [\![\texttt{Prog}]\!](\vec{x}) \cap \Omega \neq \emptyset\}$, where $[\![\texttt{Prog}]\!](\vec{x})$ denotes the runtime states of executing $\texttt{Prog}(\vec{x})$. Most important are conditions (a) and (b), as they define the theoretical requirements for a solution to the problem.

2.2 Instances

Instance 1 (Boundary Value Analysis)**.** In software testing, inputs that explore *boundary conditions* are usually regarded to have a higher payoff than test inputs that do not. The problem of generating such inputs is called *boundary value analysis* [28].

Boundary conditions refer to equality constraints derived from arithmetic comparisons. For example, the program in Fig. 1 has two boundary conditions, $\texttt{x} = 1$ and $\texttt{x} = 2$ corresponding to the two branches.

Boundary value analysis of program Prog can be defined as a floating-point analysis problem $\langle \texttt{Prog}; S \rangle$ where S is the set of inputs triggering a boundary condition.

Instance 2 (Path Reachability)**.** Given a path π of program Prog, one needs to generate an input that triggers π. Such a problem has been studied as an individual topic [26], or more commonly, as a subproblem, *e.g.*, in dynamic symbolic execution [18, 22]. Path reachability can be specified as $\langle \texttt{Prog}; S \rangle$ with S being the set of inputs triggering τ.

Instance 3 (Overflow Detection)**.** Let Prog be the program under test. We are interested in finding a set of inputs that trigger overflows on most, if not all, of the floating-point operations in Prog. Floating-point overflow detection was suggested in the 90s as one of the major criteria for validating critical systems [38], but it is only until recently a systematic approach has been proposed [5].

To frame overflow detection as an instance of floating-point analysis, we introduce a parameter L for tracking the floating-point operations that have overflowed with previously generated inputs. Then the overflow detection problem can be formulated as a set of floating-point analysis problems parameterized by L, $\langle \texttt{Prog}; S_L \rangle$, where

$$S_L \stackrel{\text{def}}{=} \{\vec{x} \in \text{dom}(\texttt{Prog}) \mid \exists l \notin L, [\![\texttt{Prog}]\!](\vec{x}) \Downarrow_l \text{overflow}\}. \tag{2}$$

Above, the notation $[\![\texttt{Prog}]\!](\vec{x}) \Downarrow_l$ overflow reads as, the floating-point operation at l overflows if Prog runs with the input $\vec{x}$.

Instance 4 (Branch Coverage-based Testing)**.** This instance concerns finding inputs that cover as many branches of the tested program as possible. [17] formulates the problem by introducing a parameter B of the branches that have already been covered. Then, branch coverage-based testing can be formulated as $\langle \texttt{Prog}; S_B \rangle$ with S_B denoting the set of inputs that trigger a branch outside B.

Instance 5 (Quantifier-free Floating-Point Satisfiability)**.** Consider a constraint in the conjunctive normal form (CNF), $c \stackrel{\text{def}}{=} \bigwedge_{i \in I} \bigvee_{j \in J} c_{i,j}$, where each $c_{i,j}$ is a binary comparison between two floating-point expressions. Suppose we have the following program,

```
void Prog(double x_1, ..., double x_N) {if (c);}
```

where $\texttt{x}_i$, $1 \leq i \leq N$, are the free variables of the floating-point constraint c. Then, the problem of deciding whether c is satisfiable or not, and the problem of solving the path reachability problem of Prog in the sense of Def. 2.1(a-b) (where the path consists of the true branch), are equivalent.

Remark. An instance of floating-point analysis may be viewed as a sub-instance of another. For example, the overflow detection problem (Instance 3) may be addressed as branch coverage-based testing (Instance 4) with appropriate code instrumentation. That is to say, if there exists an ideal implementation that could achieve full branch coverage for any given program, the same implementation could find all possible floating-point overflows. In reality, such an ideal implementation rarely exists, and we can usually expect to construct more refined, dedicated solutions for sub-instances.

3 Technical Formulation

Problem reduction is a common term in computability theory. Simply speaking, a problem A reduces to a problem B if solving B allows for a solution to A. For example, the problem of finding the maximum of a scalar function f reduces to finding the minimum of $-f$.

In this section, we first present a reduction algorithm in a metric space, which provides with a natural tool for bringing floating-point analysis to bear on function minimization techniques. Then, we generalize the reduction procedure with the concept of weak distance, which forms the core of our theoretical development.

3.1 Reduction in a Metric Space

A *metric space*, denoted by (M, η), is composed of a set M and a distance function $\eta : M \times M \to \mathbb{R}$. The distance function satisfies $\forall m, n, o \in M$, $\eta(m, n) = 0$ iff $m = n$, $\eta(m, n) = \eta(n, m)$, and $\eta(m, n) \leq \eta(m, o) + \eta(o, n)$ (triangle inequality).

Let $\langle \texttt{Prog}; S \rangle$ be a floating-point analysis problem. Since each input is in $\mathbb{F}$, we can embed the input domain $\text{dom}(\texttt{Prog})$ in a real-valued, N-dimensional Euclidean space $\mathbb{R}^N$. Following a standard practice in mathematics, one can lift η to a "point-to-set distance" function D from a program input $x \in \text{dom}(\texttt{Prog})$ to the set S,

$$D(\vec{x}) \stackrel{\text{def}}{=} \min\{\eta(\vec{x}, \vec{x}') \mid \vec{x}' \in S\}. \tag{3}$$

By definition, $D(\vec{x})$ is always nonnegative. It becomes smaller when $\vec{x}$ gets closer to S and vanishes when $\vec{x}$ goes inside. Thus, we can expect to solve the floating-point analysis problem precisely by minimizing D: (1) If the minimum found is 0, then the minimum point (where the minimum is reached) must be an element of S; and (2) if the minimum is strictly positive, we can guarantee that S is empty. In other words, we can solve the floating-point analysis problem in the sense of Def. 2.1(a-b) with the algorithm below.

Algorithm 1. The input is a floating-point analysis problem $\langle \texttt{Prog}; S \rangle$, and the algorithm consists of the following steps.

(1) Construct the distance function D (Eq. 3).
(2) Minimize D. Let $\vec{x}^*$ be the minimum point.
(3) If $D(\vec{x}^*) = 0$ return $\vec{x}^*$. Otherwise, return "not found".

The issue of the algorithm above, however, lies in that implementing D involves $\eta(\vec{x}, \vec{x}')$ for all $\vec{x}' \in S$. Even though S is finite ($S \subseteq \text{dom}(\texttt{Prog}) = \mathbb{F}^N$), it is usually defined through the unknown part of the floating-point analysis problem. For example, in the case of boundary value analysis (Section 2), to implement D is to compute the distance between an arbitrary input and the inputs triggering a boundary value; the later is exactly the solution set we seek (so, it is unknown).

3.2 Weak-Distance Minimization

We introduce the concept of *weak distance* to address the aforementioned issue regarding Algorithm 1.

Definition 3.1 (Weak Distance)**.** Let $\langle \texttt{Prog}; S \rangle$ be a floating-point analysis problem. A program $\texttt{W}$ is said to be a *weak distance* of $\langle \texttt{Prog}; S \rangle$ if it is has the type $\text{dom}(\texttt{Prog}) \to \mathbb{F}$, and it satisfies the following properties:

(a) For all $\vec{x}$, $\texttt{W}(\vec{x}) \geq 0$.
(b) Each zero of the weak distance is a solution to the floating-point analysis problem, that is, $\texttt{W}(\vec{x}) = 0 \implies \vec{x} \in S$.
(c) The zeros of the weak distance include all the solutions to the floating-point analysis problem, that is, $\vec{x} \in S \implies \texttt{W}(\vec{x}) = 0$.

Unlike the distance function D (Eq. 3), weak distance is defined to be a computer program. It is "weaker", or in fact more general, than the distance because D also satisfies the properties in Def. 3.1(a-c). Thus, an implementable distance function is a weak distance. Another generic weak distance for any floating-point problem $\langle \texttt{Prog}; S \rangle$ can be the characteristic function,

$$\lambda \vec{x}. \begin{cases} 0 & \text{if } \vec{x} \in S \\ 1 & \text{otherwise} \end{cases} \tag{4}$$

under the condition that S is decidable (so that an algorithm can decide $\vec{x} \in S$).

Next we develop the theory of *weak-distance minimization*. The lemma below allows for a systematic reduction algorithm that solves the floating-point analysis problem.

Lemma 3.2. *Let $\texttt{W}$ be a weak distance of the floating-point analysis problem $\langle \texttt{Prog}; S \rangle$, $\texttt{W}^*$ be the minimum of $\texttt{W}$, and* $\text{argmin}\,\texttt{W}$ *be the set of its minimum points.*

(a) Deciding the emptiness of S is equivalent to checking the sign of $\texttt{W}^*$. That is, $S = \emptyset \Leftrightarrow \texttt{W}^* > 0$.
(b) Assume that $S \neq \emptyset$. We have $S = \text{argmin}\,\texttt{W}$.

Proof. Proof of (a): Suppose $S \neq \emptyset$. Let $\vec{x}_0$ be an element of S. We have $\texttt{W}^* \geq 0$ by Def. 3.1(a). In addition, we have $\texttt{W}^* \leq \texttt{W}(\vec{x}_0)$ since $\texttt{W}^*$ is the minimum. Then we have $\texttt{W}^* \leq 0$ because $\texttt{W}(x_0) = 0$ due to Def. 3.1(c). Thus $\texttt{W}^* = 0$. Conversely, $\texttt{W}^* = 0$ implies that there exists an $\vec{x}^* \in S$ such that $\texttt{W}(x^*) = 0$. By Def. 3.1(b), $\vec{x}^* \in S$. Thus $S \neq \emptyset$.

Proof of (b): Let $0_{\texttt{W}}$ denote the set of the zeros of $\texttt{W}$. Below, we show $0_{\texttt{W}} \subseteq \text{argmin}\,\texttt{W} \subseteq S \subseteq 0_{\texttt{W}}$ under the condition $S \neq \emptyset$. We have $0_{\texttt{W}} \subseteq \text{argmin}\,\texttt{W}$, as a zero of $\texttt{W}$ is necessarily a minimum point; we have $\text{argmin}\,\texttt{W} \subseteq S$ because $S \neq \emptyset$ implies $\texttt{W}^* = 0$ by Lem. 3.2(a). For an arbitrary $\vec{x}^* \in \text{argmin}\,\texttt{W}$, $\texttt{W}(x^*) = \texttt{W}^* = 0$ holds. Therefore $\vec{x}^* \in X$ by Def. 3.1(b); we have $S \subseteq 0_{\texttt{W}}$ from Def. 3.1(c). □

Algorithm 2 (Weak-Distance Minimization)**.** The input is a floating-point analysis problem $\langle \texttt{Prog}; S \rangle$.

(1) Construct a program $\texttt{W} : \text{dom}(\texttt{Prog}) \to \mathbb{F}$ that satisfies Def. 3.1(a-c).
(2) Minimize $\texttt{W}$. Let $\vec{x}^*$ be the minimum point.
(3) Return $\vec{x}^*$ if $\texttt{W}(\vec{x}^*) = 0$, or otherwise, return "not found".

The theorem below follows directly from Lem. 3.2.

Theorem 3.3 (Theoretical Guarantee)**.** *Let $\langle \texttt{Prog}; S \rangle$ be a floating-point analysis problem. The weak-distance minimization algorithm solves the problem in the sense of Def. 2.1(a-b).*

(a) It returns an $\vec{x}^* \in S$ if $S \neq \emptyset$, and
(b) It returns "not found" if $S = \emptyset$.

4 Weak-Distance Minimization Illustrated

We have applied weak-distance minimization to a number of floating-point analysis problems that are common and important in the industry and research. Applications to boundary

value analysis, path reachability, and FP overflow detection are presented in this section. (The problems have been described in Section 2.) In each context, we demonstrate how to construct a weak distance, and we illustrate how minimizing the weak distance with an MO backend solves the underlying floating-point analysis problem.

We first present the algorithms on boundary value analysis and path reachability with a small, artificial example, to give readers the intuition. Then, we use a real-world scientific program to initiate discussions about overflow detection, of which we also describe the algorithm with details.

Throughout this section, we need to argue whether a constructed program indeed meets the requirements Def. 3.1(a-c) for being a weak distance. To simplify the presentation, we will use real arithmetic to argue their computational values (although a weak distance is a floating-point program), and we defer discussions about FP inaccuracies to Section 5.

We start by reviewing notations and basic concepts in MO.

4.1 Mathematical Optimization (MO)

An MO problem [27] is to compute $\min\{f(x) \mid x \in X\}$, where f is the *objective function*, and X is the *search space*. MO algorithms can be divided into two categories. Local optimization focuses on where a local minimum can be found near a given input, and global optimization determines the function minimum over the entire search space.

Notation. Let f be a function over a metric space (M, η). We call $x^* \in M$ a *local minimum point* if there exists a neighborhood of x^*, namely $\{x \mid \eta(x, x^*) < \delta\}$ for some $\delta > 0$, so that all x in the neighborhood satisfy $f(x) \geq f(x^*)$. The value of $f(x^*)$ is called the *local minimum* of the function f. If $f(x^*) \leq f(x)$ for all $x \in M$, we call $f(x^*)$ the *global minimum* of the function f, and x^* a *global minimum point*. Below, we say minimum (resp. minimum point) for global minimum (resp. global minimum point).

In this work, we see MO as an off-the-shelf black-box technique that produces a sampling sequence from a combination of local and global optimization. The local optimization generates a sequence of samplings $\vec{x}_{\vec{s}}^{\,0}, \ldots, \vec{x}_{\vec{s}}^{\,n}, \ldots$ that converges to a local minimum point $\vec{x}_{\vec{s}}^{\,*}$ near a starting point $\vec{s}$. Such a local MO is then applied over a set of starting points SP. It is expected that at least one of $\{\vec{x}_{\vec{s}}^{\,*} \mid \vec{s} \in SP\}$ reaches a global minimum point, although no general guarantee can be claimed.

4.2 Reducing Boundary Value Analysis to MO

Consider boundary value analysis for the program in Fig. 2. The boundary values are the inputs that trigger either `x = 1.0` at the first branch or `y = 4.0` at the second branch. Readers can check that -3.0, 1.0, and 2.0 are three boundary values.

To automatically detect these inputs with weak-distance minimization, we start by instrumenting `Prog` with a global variable `w`. The variable `w` is multiplied with `abs(x - 1.0)` before the first branch and with `abs(y - 4.0)` before the second branch. Fig. 3(a) highlights the instrumented parts in `Prog_w`. Then, a driver program `W` captures the value of `w` by invoking `Prog_w`. Note that the variable `w` is initialized to 1 in `Prog_w`. By design, the following properties hold: (1) $W(x) \geq 0$ for any $x \in \mathbb{F}$; (2) $W(x) = 0$ iff $x = 1.0$ before the first branch or $y = 4.0$ before the second branch. In other words, the nonnegative function `W` has encoded the boundary values into its zeros. Fig. 3(b) shows the graph of `W`. The graph attains 0 at -3.0, 1.0, and 2.0. Thus, the boundary value analysis problem reduces to the minimization problem of `W`. The program `W` is our constructed weak distance.

```
void Prog(double x) {
  if (x <= 1.0) x++;
  double y = x * x;
  if (y <= 4.0) x--;
}
```

Figure 2. A simple floating-point program.

The minimization problem can then be solved by off-the-shelf MO techniques. Fig. 3(c) shows the sampling we have collected with the Scipy's Basinhopping backend [37]. Observe that all three expected boundary values, depicted as the horizontal lines in the figure, are reached by the samples.

4.3 Reducing Path Reachability to MO

In this example, we intend to trigger a path passing through the two branches of the program in Fig. 2.

We instrument `Prog` with a global variable `w` and inject

```
w = w + (a <= b)? 0 : a - b
```

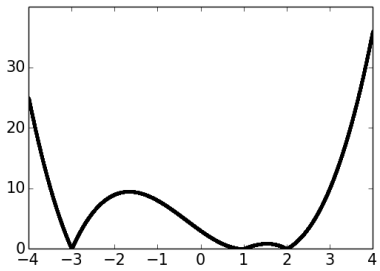
before each branch of the form `a <= b`. Fig. 4(a) shows the instrumented program `Prog_w`. The program `W` initializes `w` to `0` and retrieves its value after executing `Prog_w(x)`. The following property holds: a program input `x` minimizes `W` iff `x` triggers both branches. Thus, the path reachability problem reduces to the problem of finding the minimum of the `W`. Fig. 4(b) shows the graph of the weak distance. It attains 0 for all $x \in [-3, 1]$, which correspond to the set of inputs that can trigger the path. Fig. 4(c) shows the MO sampling with `W` as the objective function. The solution space is highlighted. Observe that the highlighted region is reached by a large number of samples, with noticeably more samples reaching inside than outside. These results confirm the effectiveness of using MO for this problem.

4.4 Reducing Overflow Detection to MO

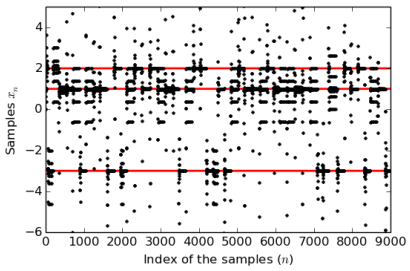
The GSL Bessel function in Fig. 5 contains 23 elementary FP operations involving $+$, $-$, $*$ or $/$. Suppose that the function has been compiled into a modern IR (intermediate representation) so that each FP operation corresponds to exactly one instruction in the IR. For example, the first statement in the

```
double w;
void Prog_W (double x) {
  w = w * abs(x - 1.0);
  if (x <= 1.0) x++;
  double y = x * x;
  w = w * abs(y - 4.0);
  if (y <= 4.0) x--;
}
double W (double x) {
  w = 1; Prog_W(x); return w;
}
```

(a)



(b)

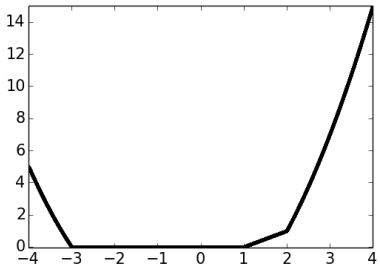


(c)

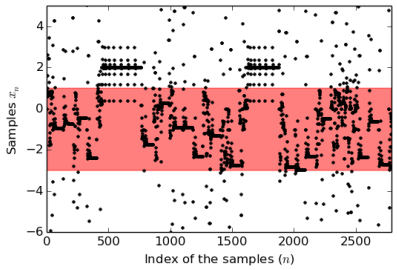
Figure 3. Applying weak-distance minimization to boundary value analysis. Goal: To find an input that triggers `x = 1` at the first branch or `y = 4` at the second branch. (a) Weak distance. (b) Graph of the weak distance `W`(x). (c) MO samples. In (c), the x-axis is the index of the samples n, and the y-axis is the value of the sampled input x_n. Horizontal lines are three known boundary values -3.0, 1.0, and 2.0.

```
double w;
void Prog_w(double x)) {
w = w + (x <= 1.0 ? 0 : x - 1);
  if (x <= 1.0) x++;
  double y = x * x;
w = w + (y <= 4.0 ? 0 : y - 4);
  if (y <= 4.0) x--;
}
double W(double x) {
  w = 0; Prog_w(x); return w;
}
```

(a)



(b)



(c)

Figure 4. Applying weak-distance minimization to path reachability. Goal: To find an input triggering both branches (a) Weak distance; (b) Graph of the weak distance `W`(x); (c) MO samples. In (c), the x-axis is the index of the samples n, and the y-axis is the value of the sampled input x_n. A known solution space is $[-3, 1]$.

```
int gsl_sf_bessel_Knu_scaled_asympx_e(const double nu,
const double x, gsl_sf_result* result) {
  double mu = 4.0 * nu * nu;
  double mum1 = mu - 1.0;
  double mum9 = mu - 9.0;
  double pre = sqrt(M_PI / (2.0 * x));
  double r = nu / x;
  result->val = pre * (1.0 + mum1 / (8.0 * x) +
                  mum1 * mum9 / (128.0 * x * x));
  result->err = 2.0 * GSL_DBL_EPSILON *
    fabs(result->val) + pre * fabs(0.1 * r * r * r);
  return GSL_SUCCESS;
}
```

Figure 5. A Bessel function from GSL `bessel.c`

source, `mu = 4.0 * nu * nu`, compiles to two LLVM IR instructions of the form

l_1: `t = double fmul 4.0 nu`
l_2: `mu1 = double fmul t nu`

Below, we will use the term "instruction" to refer to the IR code and "statement" to the source code. Thus, the Bessel function has a set of 23 instructions. We write $\bar{L}$ to denote this set. Each instruction $l \in \bar{L}$ is of the form "`a = ...`". We call `a` the assignee of l.

Our ultimate goal in this application is to detect all the instructions in the Bessel function that can overflow, and for each of the operations, a program input triggering the overflow. Following the explanation in Section 2, it suffices that we consider the parameterized floating-point analysis problem $\langle \texttt{Prog}; S_L \rangle$, where $L \subseteq \bar{L}$ tracks FP instructions that have already overflowed with previously generated inputs.

As before, we start by constructing a weak distance. We first illustrate this weak distance in a simple case, that is when $L = \bar{L} \setminus \{l_1, l_2\}$. In other words, we intent to trigger

overflow on all the instructions given that all except two, l_1 and l_2, have already overflowed. We inject

```
if (l_i is not in L) w = |a_i| < MAX ? MAX - |a_i| : 0
```

after l_i, where a_i refers to the assignee of the assignment at statement l_i (t and mu1 for $i = 1, 2$ respectively).

By design, $w \geq 0$; and $w = 0$ iff an overflow at l_2 occurs (the injected code after l_2 overwrites w's previous value). [2] Thus, if we have a weak distance function that maps inputs of the Bessel function to the value of w after executing the instrumented program, minimizing the weak distance should allow us to find an input that triggers an overflow on l_2. Then, L is to be updated to $L \cup \{l_2\}$, making the injected code after l_2 behave like a no-op. If we continue minimizing the weak distance, we can trigger overflows on l_1 as well. When overflow has been triggered for both instructions, all the inserted code acts like a no-op, and the weak distance returns the initial value of w, implying that no more overflow can be found.

The algorithm below applies in the general case. The algorithm follows what we have described above except for some implementation details that we will explain shortly.

Algorithm 3 (Overflow Detection with Weak-Distance Reduction)**.** Let Prog be the program under analysis containing $nFP_{\texttt{Prog}}$ floating-point instructions. We use the set L to track the set of instructions in Prog that have overflowed with previously generated inputs, and we use the set X to track the set of the generated inputs. The MO backend is Basinhopping. It takes a floating-point program and a starting point as inputs and returns a minimum point.

(1) Instrument Prog with a global variable w of type double.
(2) Inject the following code after each floating-point operation l in Prog, where the assignee of l is denoted by a. Let Prog_w denote the instrumented program.

```
if (l is not in L) {
  w = (|a| < MAX)? MAX - |a|: 0;
  if (w == 0) return;
}
```

(3) Let W be the following program

```
double W(double x_1, ..., double x_N) {
  w = 1; Prog_w(x_1, ..., x_N); return w;
}
```

(4) Pick a random starting point $\vec{s} \in \mathbb{F}^N$.
(5) Let $\vec{x}^* = \texttt{Basinhopping}(\texttt{W}, \vec{s})$
(6) If $\texttt{W}(\vec{x}^*) = 0$, then set $X = X \cup \{\vec{x}^*\}$.
(7) Pick *target* to be the last instruction executed in the previous round of the Basinhopping cycle in step (5), and let $L = L \cup \{target\}$.
(8) If $|L| \leq nFP_{\texttt{Prog}}$, go to step (4).
(9) Return X.

[2] More precisely, w = 0 implies mu ≥ MAX, not mu > MAX. But we dismiss the situation of |mu| attaining MAX exactly, which is extremely unlikely to happen in the floats.

Algorithm 3(1-3) constructs the weak distance. This part from the injected code `if (w==0) return;` is to ensure that the instrumented program terminates whenever it reaches 0. The rest of the algorithm from (4) is to minimize the weak distance in multiple iterations. Each iteration launches the optimization backend from a random starting point. From each starting point the backend Basinhopping computes the minimum point $\vec{x}^*$. If the minimum is 0, $\vec{x}^*$ is to be added to X. The Basinhopping procedure is a Markov Chain Monte Carlo (MCMC) sampling over the space of the local minimum points [23]. Its algorithmic detail can be found in Section 4 of [17] but is irrelevant for this presentation as we use the backend as a black-box.

The use of *target* in step (7) of the algorithm is heuristic. The idea is that the algorithm targets one instruction in each minimization round. Following the instrumentation in step (2), the assignment of the variable w of the last instruction that has not yet been triggered with overflow overwrites the precedent assignments of w. Thus, we populate L with *target* to ensure termination of the algorithm. If a minimum 0 is found, *target* is to be put in L, meaning that overflow on the targeted instruction has been tracked by L. If a nonzero minimum is found, then either the instruction *target* overflows (such as the case `y = x * 0`), or the optimization backend fails to find the true minimum 0. In either case, *target* is added to L so that L not only includes overflowed instructions, but also those that cannot be triggered with overflow or do not overflow. In this way, we can guarantee termination: minimizations rounds.

Experimental results on the Bessel function will be presented in Section 6. Briefly, we have found 21 overflows out of 23 FP operations. In particular, the one that triggers the overflow on l_1 is nu = 1.8E308; the one that triggers overflow on l_2 is nu = 3.2E157.

Remark. It is worth nothing that the weak distance constructed in this section depends on a global variable L. This is where optimization of weak distance differs from MO used in the literature, where the objective function is usually pure, with no side effects. Def. 3.1 only specifies the minimum set of rules for constructing a weak distance, which allows for significant flexibility when it comes to implementing such a weak distance as is shown in this application. Another difference between traditional MO and MO of weak distance lies in their termination conditions. Traditional MO usually does not know when the optimization process should terminate, whereas in the case of weak distance, if a minimum 0 is reached, MO should stop as no smaller minimum can be found due to Def. 3.1(1).

5 Implementation Architecture

This section contributes to a system for implementing weak-distance minimization involving three layers as illustrated in Figure 6. Each layer, the Client, Analysis Designer, or

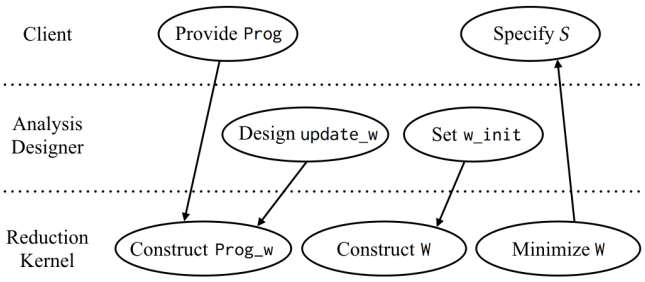


Figure 6. Implementation architecture for weak-distance minimization.

Reduction Kernel, is responsible for a different facet of the implementation so that developers with varying areas of expertise can evolve different portions of the system separately. Limitations that could affect applicability will also be discussed in each layer.

5.1 The Client Layer

The *Client* defines the floating-point analysis problem by providing the program under analysis `Prog` and specifying a set of program inputs $S \subseteq \text{dom}(\texttt{Prog})$. If `Prog` invokes other functions, the Client also needs to provide the invoked functions as well. For example, consider the boundary value analysis problem for the following (artificial) program

```
void Prog(double x){if (g(x) <= h(x)){...}}
```

If the analysis is also concerned with boundary values within `g` and `h`, the Client must provide instrument-able versions of `g` and `h`.

Following Definition 2.1, the Client is expected to provide a *valid* specification, in the sense that $\text{dom}(\texttt{Prog}) = \mathbb{F}^N$ for some $N \in \mathbb{Z}$ and $S \subseteq \text{dom}(\texttt{Prog})$. If $\text{dom}(\texttt{Prog}) \neq \mathbb{F}^N$, the Client needs to specify a valid floating-point analysis problem $\langle \texttt{Prog}_v; S_v \rangle$ such that $\text{dom}(\texttt{Prog}_v) = \mathbb{F}^N$, and that a solution found in S_v can be mapped to an element in S. For example, if the objective is to find boundary values of a function with interface `Prog(int)` instead of `Prog(double)`, the Client can specify the boundary value analysis problem with the function `Prog_v(double x) {Prog(d2i(x);}` where `d2i` converts a `double` to an `int`, through a kind of truncation for example. In this way, every boundary value x^* found should be mapped to `d2i`(x^*) as a solution to the original problem. As another example, if the objective is to analyze a function of the interface `Prog(double*)`, the actual analyzed program can be `Prog_v (double x){double* p; *p = x; Prog(p);}`. As a third example, the Bessel function in Figure 5 accepts two `double` values and a pointer value. The latter is for passing the computation results. The function inputs can be easily adapted to $\mathbb{F}^2$ if a global variable is used to hold the results.

Such tricks, however, have to be done manually, and not all floating-point programs may fit the requirement that $\text{dom}(\texttt{Prog}) = \mathbb{F}^N$. That is where we would like to point out a limitation of our approach.

Limitation 1. If the program under analysis has an input parameter other than `double`, it is possible that the Client's problem does not fit Definition 2.1, or manual efforts may be needed from the Client side to reduce the case to a valid floating-point analysis problem.

5.2 The Analysis Designer Layer

The *Analysis Designer* aims to construct a weak distance for $\langle \texttt{Prog}; S \rangle$. The process usually depends on two parameters. One is `w_init`, the initial value of the instrumented variable `w`. The other is a piece of code `update_w` that is to be instrumented into the program under analysis and updates `w`. The choices of `w_init` and `update_w` depend on the category of floating-point analysis problems, following Definition 3.1(a-c) as guidelines.

In theory, the constructed weak distance should satisfy conditions (a-c) of Definition 2.1. In practice, however, it occurs that the Analysis Designer builds a function that satisfies the conditions in real arithmetic but not in floating-point arithmetic. Consider the path reachability problem with the following program:

```
void Prog(double x){if (x == 0) ...; return;}
```

The expected input is 0. But if we inject `w = w + x * x` before the branch and retrieve the value of `w` through a `W` program as before, then $\texttt{W}(x) = 0$ does not necessarily imply $x = 0$, *e.g.*, $\texttt{W}(1\text{E-}200) = 0$ as well, but 1E-200 does not trigger the path `if (x == 0)`. This example shows a limitation that the Analysis Designer faces.

Limitation 2. Constructing a weak distance that strictly complies with Definition 2.1 can be tricky due to floating-point inaccuracy. The Analysis Designer may reason about floating-point programs with real arithmetic. Formally, if Algorithm 2(1) fails to produce a weak distance for a given floating-point analysis problem $\langle \texttt{Prog}; S \rangle$, then the algorithm returns an $\vec{x}^*$ but $\vec{x}^* \notin S$, causing unsoundness.

Remark. One way to avoid unsoundness is check whether $\vec{x}^* \in S$ in Algorithm 2(3), assuming that S is decidable. For example, in the path reachability case above, one can run the program to see if the input 1E-200 indeed passes through the branch `if (x == 0)`.

Note that while inaccuracy is inherent in floating-point code, some can be avoided in constructing weak distance. For example, the use of the absolute value instead of the square operator, as shown in the example above and in Figure 3, helps avoid overflow/underflow, which allows us to mitigate Limitation 2. Also, one can implement `W` with higher-precision arithmetic, or the integer-valued ULP distance [31].

```
double r;
void Prog_w(double x){
w = w * (x == 1)? 0: 1;
l0: if (x <= 1) x++;
double y = x * x;
w = w * (y == 4)? 0: 1;
if (y <= 4) x--;
}
```

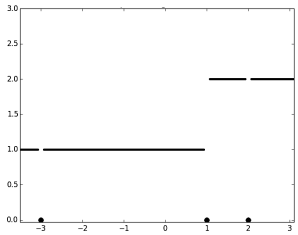


Figure 7. Boundary value analysis with a characteristic function as the weak distance.

5.3 The Reduction Kernel Layer

This layer takes as inputs the program `Prog` provided by the Client, the initial value `w_init` and the stub `update_w` set by the Analysis Designer, and operates the following three steps: (1) Inject `w` and `update_w` into `Prog` to construct `Prog_w`, (2) initialize `w` to `w_init`, invoke `Prog_w` and return `w` in `W`, and (3) Minimize `W` with mathematical optimization.

Steps (1-2) are simple program instrumentation, which can be achieved with the Clang/LLVM infrastructure for the C family of languages. Step (3) introduces an external MO backend. In theory, if the minimum is zero, the minimum point can be returned as a solution expected by the Client; if the minimum is strictly positive, then $S = \emptyset$. In practice, however, it is possible that the MO backend produces a sub-optimal result. In that case, if the exact minimum is strictly larger than zero, Algo. 2 concludes that $S = \emptyset$, which is correct. But if the exact minimum is 0, the reduction algorithm also concludes $S = \emptyset$, which is incorrect, a situation that we call *incompleteness*.

As an illustration, consider the boundary value analysis problem (Figure 3) again. Suppose we implement a characteristic function as shown in Figure 7 (a), which is a weak distance as explained in Section 3. However, existing optimization tools should have difficulty to find any of -3.0, 1.0 or 2.0 as the function is flat almost everywhere (Figure 7 (b)). In this case, weak distance does not help guide the search process, and the optimization of this weak distance degenerates into pure random testing.

Limitation 3. MO problems are intractable in general. There is no guarantee that the optimization backend produces an actual minimum. As a result, incompleteness may occur: the reduction algorithm may return "not found" whereas $S \neq \emptyset$.

6 Experiments

We have presented weak-distance minimization from both a theoretical and an applicative perspective. Our experiment attempts to assess the gap between them, namely, how much we can answer a floating-point analysis problem if its reduced MO problem can only be partially solved in general?

Table 1. Different MO backends applied on two weak distances.

	Boundary Value Analysis		Path R.	
	W^*	x^*	W^*	x^*
Basinhopping	0	1.0, 2.0, -3.0 0.999 999 999 999 999 9	0	$[-3, 1]$
Differential E.	4.43E-18	NA	0	$[-3, 1]$
Powell	0	1.0, 2.0	0	$[-3, 1]$

We have designed three sets of experiments that aim to answer this question. All the experiments were performed on a laptop with a 2.6 GHz Intel Core i7 running Ubuntu 14.04 with 4G RAM.

6.1 Checking Different MO Backends

Our first experiment is a sanity check. Since MO is seen as a black-box technique in our approach, we should be able to apply different MO backends and observe differences regarding completeness (Section 5). To this end, we have used three MO backends on the example program in Fig. 2. The first backend is Basinhopping [23], an MCMC (Markov Chain Monte Carlo) algorithm that samples over local minimum points. The second is Differential Evolution [35], an evolutionary programming algorithm utilizing a parallel direct search evolution strategy. The last one is Powell [30], a local search that does not need to calculate function derivatives. All three are taken from the `SciPy` package [3]. We applied the backends on two weak distances used in our boundary value analysis (Fig. 3) and path reachability examples (Fig. 4). For each weak distance `W`, we recorded all the minimal found W^* and their corresponding minimum points x^*. Tab. 1 presents our experimental data.

For boundary value analysis: Basinhopping detected three expected boundary values, -3.0, 1.0, 2.0, and one that we were unaware of, 0.999 999 999 999 999 9. It is the same number we have discussed in Section 1. We can quickly check that this number indeed triggers the boundary condition `y = 4.0` at the second branch of the program. Differential Evolution, on the other hand, did not find any boundary value. It achieved the minimum $W^* = 1\text{E-}18$ and concluded "not found." [3] At last, Powell found two boundary values, 1.0 and 2.0, but missed -3.0. For path reachability, the expected solution space is $[-3, 1]$. Each tested backend reached the minimum 0, and they all found a large number of x^* between - 3.0 and 1.0. Thus, the results show that using different MO backends is possible, although their results may vary.

6.2 Boundary Value Analysis on GNU `sin`

Our second experiment is a case study with the `sin` function implemented in the GNU C Library (Glibc) 2.19 [2].

[3] This could be due to our misuse of the implementation or its configuration, *e.g.*, by setting an overly large tolerance.

```
1  double sin (double x) {
2    int k, m;
3    ...
4    k = 0x7fffffff & m;
5    if (k < 0x3e500000) // |x| < 1.490120E-08 { ... }
6    else if (k < 0x3feb6000) // |x| < 8.554690E-01 { ... }
7    else if (k < 0x400368fd) // |x| <2.426260E+00 { ... }
8    else if (k < 0x419921fb) // |x| < 1.054140E+08 { ... }
9    else if (k < 0x7ff00000) // |x| < 2^1024 { ... }
10   else ...
11 }
```

Figure 8. An implementation of the `sin` function from Glibc 2.19.

[4] To compute the correctly rounded values for all inputs, the implementation uses Chebyshev, Taylor polynomials, or lookup tables to approximate $\sin(x)$ for different input ranges [4]. Details of the implementation are not necessary for the discussion that follows and are omitted here.

Fig. 8 displays a simplified version of the `sin` function, which has five branches from Lines 5 to 9 corresponding to the aforementioned input ranges. Each of the branches involves the absolute value function, for which we count two boundary conditions, *e.g.*, $x = 2^{-26}$ and $x = -2^{-26}$ for Line 5. In total, this `sin` function contains 10 boundary conditions. Among them, the two associated with the last branch, $x = 2^{1024}$ and $x = -2^{1024}$, are unreachable as 2^{1024} is strictly larger than the largest `double` floating-point number (except `+inf`). In this experiment, we test the capability of weak-distance minimization in triggering these boundary conditions.

To carry out the experiment, we first instrumented a weak distance for the `sin` function manually. We introduced a global variable `w` in `sin`, injected `w = w * abs(k - c)` before each branch of the form `if (k < c)`, and retrieved the value of `w` through the program `W`, namely weak distance, as illustrated in Fig. 3 . We used Basinhopping to minimize `W` from a set of random starting points. The MO process produced a total of 6 365 201 samples for triggering all the reachable boundary conditions, which took 66.3 seconds. Fig. 9 illustrates the sampling process in terms of the numbers of triggered boundary conditions.

To analyze the experimental data, we collected all the samples in a python variable `Raw`. From `Raw`, we filtered the samples that attained the minimum 0 on the weak distance, namely, $\mathtt{BV} \stackrel{\text{def}}{=} \{x \in \mathtt{Raw} \mid \mathtt{W}(x) = 0\}$. The samples in `BV` are the boundary values that our analysis reported. There were 945 314 samples in `BV`, that is 14.9% of the sample size of `Raw`.

We analyzed the data from two aspects:

(i) Soundness, namely, whether these reported boundary values, *i.e.*, the samples in `BV`, trigger boundary conditions as expected. For this purpose, we manually instrumented the `sin` function with `if (k == c) hits++` before each branch `if (k < c)`. We then ran this instrumented program with all the samples in `BV`. We observed that `hits`, initialized 0, became 945 314 in the end, which is exactly the sample size of `BV`. This confirms that each reported boundary value triggers exactly one boundary condition.[5]

(ii) Completeness, in the sense that all reachable boundary conditions should have been triggered. Fig. 9 explained above already showed the completeness. Here, we grouped the samples of `BV` by its triggered boundary conditions. Tab. 2 shows the boundary values suggested by the developers (row **ref**), the minimum and maximum of our found boundary values (rows **min** and **max**), and the number of times the boundary condition has been reached (row **hits**). The results confirm that each of the 8 reachable boundary conditions is reached. It also shows that our reported boundary values are very close to those suggested by developers.

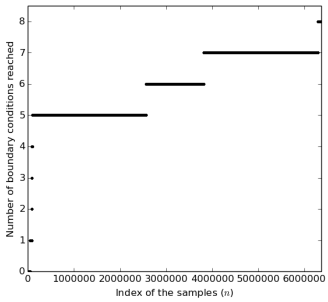


Figure 9. Boundary value analysis on GNU `sin`: The number of the triggered boundary conditions (y-axis) in terms of the sampling (x-axis).

Table 2. Case study with Glibc `sin`: Boundary value analysis.

	Br. Line 5 `k<0x3e500000`	Br. Line 6 `k<0x3feb6000`	Br. Line 7 `K<0x400368fd`	Br. Line 8 `k<0x419921fb`
+				
ref	1.490 120E-8	8.554 690E-1	2.426 260	1.054 140E8
min	1.490 117E-8	8.554 688E-1	2.426 264	1.054 143E8
max	1.490 117E-8	8.554 692E-1	2.426 266	1.054 144E8
hits	1	374485	44342	55
-				
ref	-1.490 120E-8	-8.554 690E-1	-2.426 260	-1.054 140E8
min	-1.490 118E-8	-8.554 692E-1	-2.426 266	-1.054 144E8
max	-1.490 116E-8	-8.554 688E-1	-2.426 264	-1.054 143E8
hits	89563	422036	14777	55

[4] Glibc's implementations of `sin` are system-dependent. The implementation that we use is for x86-64 Linux. It can be found in the routine `_sin (double x)` of `sysdeps/ieee754/dbl-64/s_sin.c`.

[5] Readers may be surprised to see that such a large number of boundary values can be associated with only 8 boundary conditions. Consider, for example, a program `Prog(double x){if (1 + x == 1) ...;}`. Clearly, 0 is a boundary value, but so are 1E-17 and many other small floating-point numbers, as $1 + x = 1$ in floating-point arithmetic if x is close to 0.

Table 3. Result summary: Floating-point overflow detection.

Benchmark			no. issues			
File	Function names	|Op|	|O|	|I|	|B|	T (sec)
bessel	`bessel_Knu_scaled.`	23	21	4	0	6.0
hyperg	`gsl_sf_hyperg_2F0_e`	8	4	2	0	5.9
airy	`gsl_sf_airy_Ai_e`	26	2	2	2	10.4

The column |Op| refers to the number of floating-point operations. |O|, |I|, |B| refer to the numbers of detected overflow, inconsistencies, and bugs, respectively.

6.3 Finding Bugs in GSL

Our third experiment applies weak-distance minimization to detecting floating-point overflows. In science and applied mathematics, there is an extensive use of the GNU Scientific Library (GSL), which has been ported to many programming languages, such as Java, Python, Ocaml and Octave. Thus, GSL provides a practical context for this experiment. We choose three *special functions*[6] from the GNU Scientific Library (GSL) as benchmarks: the Airy, Bessel, and Hypergeometric functions. Our criteria for selecting these benchmarks are as follows: they need to be be sufficiently sophisticated to challenge existing solvers, and yet not overly complicated to prevent us from analyzing the detected issues manually (using tools like `gdb`).

We implemented Algo. 3 in a prototype called FPOD and evaluated its ability to detect floating-point overflow. The front-end constructs the weak distance with an LLVM pass, which works on code from the C family such as GSL; the backend uses Basinhopping as before. We carried out the experiment in the following steps. First we used FPOD to generate inputs triggering overflows in the benchmarks. Then we replayed with the inputs for an inconsistency check. The inconsistency refers to a situation when a GLS computation manifests exceptional conditions but its returned `status` shows `GSL_SUCCESS`. Finally, we manually analyzed the root cause of each inconsistency with `gdb` to see if the inconsistency is a serious issue. We reported two such issues to GSL developers, both of which have been confirmed. Tab. 3 gives a quantitative summary of our findings.

6.3.1 Floating-point Overflows

An overflow is detected whenever the backend of FPOD returns 0 as the minimum. We recorded each overflow as a pair $(\mathtt{op}, x^*)$, where `op` is the floating-point operator, and x^* the input data triggering the overflow. If a strictly positive minimum is produced, we relaunch `Basihopping` with other starting points in case that failing to find a minimum 0 is due to incompleteness (Section 5).

[6]Special functions are functions with established names and notations due to their importance in mathematical analysis, physics and other applications.

Table 4. Floating-point overflow detected in `Bessel`.

Floating-point operations that have overflowed	nu*, x*
double mu = 4.0 * nu*nu	1.8E308, -1.5E2
double mu = 4.0*nu * nu	3.9E157, 2.5E2
double mum1 = mu - 1.0	2.8E157, 3.3E2
double mum9 = mu - 9.0	3.4E157, -4.2E1
double pre = sqrt(M_PI/(2.0 * x))	4.5E1, 1.3E308
double pre = sqrt(M_PI / (2.0*x))	missed
double r = nu / x	1.4E308, -7.6E-1
val=pre*(1.0 + mum1/(8.0 * x) + mum1*mum9/(128.0*x*x))	2.8E2, 8.6E307
val=pre*(1.0 + mum1 / (8.0*x) + mum1*mum9/(128.0*x*x))	3.2E157, 5.3E1
val=pre*(1.0 + mum1/(8.0*x) + mum1*mum9/(128.0*x*x))	4.3E157, 4.1E1
val=pre*(1.0 + mum1/(8.0*x) + mum1 * mum9/(128.0*x*x))	8.4E77, -2.5E2
val=pre*(1.0 + mum1/(8.0*x) + mum1*mum9/(128.0 * x*x))	2.2E1, 3.6E307
val=pre*(1.0 + mum1/(8.0*x) + mum1*mum9/(128.0*x * x))	3.4E2, 2.4E307
val=pre*(1.0 + mum1/(8.0*x) + mum1*mum9 / (128.0*x*x))	9.4E77, -1.9E2
val=pre*(1.0 + mum1/(8.0*x) + mum1*mum9/(128.0*x*x))	9.9E77, 9.5E1
val=pre * (1.0 + mum1/(8.0*x) + mum1*mum9/(128.0*x*x))	1.1E78, 2.3E2
err=2.0 * EPSILON*fabs(val) + pre*fabs(0.1*r*r*r)	missed
err=2.0*EPSILON * fabs(val) + pre*fabs(0.1*r*r*r)	1.2E78, 3.1E2
err=2.0*EPSILON*fabs(val) + pre * fabs(0.1*r*r*r)	1.5E308, -6.0E-1
err=2.0*EPSILON*fabs(val) + pre*fabs(0.1 * r*r*r)	5.2E159, -1.9E2
err=2.0*EPSILON*fabs(val) + pre*fabs(0.1*r * r*r)	4.6E104, -8.7
err=2.0*EPSILON*fabs(val) + pre*fabs(0.1*r*r * r)	2.8E104, 1.7E1
err=2.0*EPSILON*fabs(val) + pre*fabs(0.1*r*r*r)	1.0E78, 1.7E2

In the first column, "val" is short for result->val, "err" is short for result->err, and "EPSILON" is short for GSL_DBL_EPSILON.

Our tool FPOD found overflows in each of the three benchmarks. In particular, we find that floating-point overflows present in almost every floating-point operation of the `bessel` benchmark. Tab. 4 shows the floating-point instructions that overflow and their corresponding input data. The only two misses occur for the "*" operator in `double pre = sqrt (M_PI/(2.0*x))` and the first "*" of `result-> err = 2.0 * GSL_DBL_EPSILON`. The latter miss is expected, as it involves multiplication of two constants.

A natural question that would arise is how many of these overflows are real issues. The analysis below attempts to answer this question.

6.3.2 Inconsistencies and Bugs

The selected GSL special functions follow the POSIX error-handling convention, in which an error code is to be returned

Table 5. Inconsistency detected in three GSL special functions and their root cause.

	$\vec{x}^*$	Problematic locations	status	val	err	Root causes
bessel						
	1.8E308, -1.5E2	`double mu = 4.0 * nu*nu`	0	-nan	-nan	Large input nu
	3.2E157, 5.3E1	`double mu = 4.0*nu * nu`	0	inf	inf	Large input nu
	8.4E77 -2.5E2	`double pre = sqrt(M_PI/(2.0*x))`	0	-nan	-nan	negative in sqrt
	9.9E77, 9.5E1	`result->val = pre * (... + mum1 * mum9/(128.0*x*x))`	0	inf	inf	Large input nu
hyperg						
	-1.4E2, -1.2E2, -1.0E2	`result->val = pre * U.val`	0	inf	inf	Large operands of *
	-6.2E2, -3.7E2, -1.5E2	`double pre = pow(-1.0/x, a)`	0	inf	inf	Large exponent of pow
airy						
	-1.842 761 152	`int stat_mp = airy_mod_phase(..., & theta)`	0	0.3	inf	division by zero
	-1.142E34	`int stat_cos = gsl_sf_cos_err_e(..., & cos_result)`	0	-inf	inf	Inaccurate cosine

indicating whether the computation succeeds. These functions are typically invoked as follows:

```
gsl_sf_result result;
int status = gsl_sf_bessel_e(x, &result);
if (status == GSL_SUCCESS){ //use result }
```

Above, `gsl_sf_result` is defined as

```
typedef struct { double val; double err;}
```

where `val` stores the computational result and `err` is for an error estimate. The returned `status`, according to the documentation, "indicates error conditions such as overflow, underflow or loss of precision. If there are no errors, the error-handling functions return `GSL_SUCCESS`." Thus, we refer to "inconsistency" a situation where `status` equals `GSL_SUCCESS` (which is macro for 0 under our environmental setting) and `result.val` or `result.err` equals to `inf`, `-inf`, `nan` or `-nan`. Tab. 5 summarizes our analysis results on a total of 8 inconsistencies. Calculating with the inputs $\vec{x}^*$ from the second column of the table, a developer should not see any erroneous `status`. If the developer starts to use `result`, however, the unexpected results due to the overflow will be discovered.

We investigated the root cause for these inconsistencies by analyzing their execution traces with the inputs $\vec{x}^*$. Five out of the eight inconsistencies are due to large function inputs or operands. For example, when `nu` = 1.8E308, the first "*" in `double mu = 4.0 * nu * nu` overflows. Another one is due to negative operand of `sqrt` in `bessel`. We believe that these issues are benign. So we focus on the two remaining inconsistencies. Both of them are from the `airy` function. One inconsistency is due to a division by zero, and another one is due to inaccurate `cos` used in GSL. We reported the two to GSL developers who later confirmed that both of our findings were indeed bugs.

Bug 1 The Airy function triggers a division-by-zero with the input $x_1 = -1.842761151977744$. The division-by-zero exception disappears if one slightly disturbs the input, say, to -1.84276115198. In the `airy_mod_phase` function that `gsl_sf_airy_Ai_e` invokes, the variable `result_m` is divided whereas it has vanished following a nontrivial computation (with a loop in function `cheb_eval_mode_e`, Lines 26-30).

Bug 2 The Airy function gives wrong results with the input x_2 = -1.14E34. GSL's calculation result is `-inf`. It is mathematically wrong because Airy functions are damped oscillatory for negative inputs. Using Mathematica, the same airy function returns -1.36E-9. Using gdb, we observe that `gsl_sf_airy_Ai_e` (-1.1E34) invokes `gsl_sf_cos_err_e (theta.val, theta.err, &cos_result)` with `theta.val` = -8.11E50 and `theta.err` = 7.50E35. After this invocation, `cos_result.val` becomes `-inf`, clearly beyond its expected [-1,1] bound.

Remark. Among the five instances introduced in Section 2: The branch-coverage based testing implementation CoverMe is experimentally compared with the fuzzing tool AFL, and bAUSTIN (an FP testing implementation relying on search-based strategies and symbolic execution)) [17]; the satisfiability solving tool XSat is compared with MathSat, Z3, and the Coral solver [16]; to our knowledge, no boundary value analysis or path reachability tools for FP code have been proposed; as for FP overflow detection, an indirect comparison can be made between our implementation of Algorithm 3 FPOD and Ariadne [5]. The latter also analyzed the `Airy` function, but it did not find bugs in it, whereas we have found two; Ariadne reported a single overflow for the `Bessel` function, whereas we have found 21.

7 Related Work

Constructing an axiomatic system [15, 19] is of central importance for ensuring program correctness. In the context of floating-point (FP) analysis, a major milestone toward the axiomatic construction is the IEEE-754 standardization [1] and its formalization [8]. The standard grew out of a period of time where floating-point performance, reliability, and portability were of the utmost importance, which, combined with

the inherent limitations of machine representation, arguably contributes to the semantic complexity of today's floating-point arithmetic [20, 32]. This complexity now challenges how to prove the correctness of floating-point programs, or $[\![\texttt{Prog}]\!] \cap \Omega = \emptyset$, in which Ω specifies the erroneous states.

Two kinds of approximation have been extensively studied in FP analysis. One is abstract interpretation [11], which systematically constructs an abstract semantics $[\![\texttt{Prog}]\!]^\sharp \supseteq [\![\texttt{Prog}]\!]$ and proves $[\![\texttt{Prog}]\!]^\sharp \cap \Omega = \emptyset$. Such kinds of approximation allows for detection of a large class of numerical bugs[5, 13], or for proving their absence [6, 24], and has become the basis of more sophisticated analyses [7, 9, 31], and program transformations [12, 24].

Another kind of approximations is to show that $[\![\texttt{Prog}]\!]^\flat \cap \Omega \neq \emptyset$, where $[\![\texttt{Prog}]\!]^\flat$ refers to an under-approximation of $[\![\texttt{Prog}]\!]$ that is usually computed from generating a concrete program input or trace via random testing or SMT solving [18]. Many analyses in this category adopt *symbolic execution* [21]. They repeatedly select a target path and gather the conjunction of logic conditions along the path, called path condition. They then use SMT solvers to calculate a model of of the path constraint. Symbolic execution and its variants have seen much progress since the breakthrough in SAT/SMT [29, 33], but they still have difficulties in handling scientific code that is heavy on numerical computation.

The idea of using Mathematical Optimization (MO) [27] for FP analysis has been explored. In the seminal work of Miller *et al.* [26], optimization methods have been used in generating test data, which have been taken up in the 1990s by Koral *et al.* [14, 22]. These methods have found their ways into many mature implementations [34, 36], and in particular, search-based testing [25]. The latter uses fitness functions to encode path conditions and maximizes/minimizes the functions to search for test inputs. The fitness function is similar to weak distance in the sense that both encode program properties into numerical functions. However, minimizing the fitness function assists in the problem solving but does not offer the stand-alone algorithm [25], whereas minimizing the weak distance is guaranteed to solve the original problem.

Such theoretical guarantees and their usage have been shown in two instances of this work, XSat [16] and CoverMe [17]. XSat transforms an FP constraint π into a nonnegative FP program $\texttt{R}_\pi$ that attains 0 if and only if the program input of $\texttt{R}_\pi$ is a model of π. The problem of deciding $\texttt{R}_\pi$ is then solved equivalently as minimizing $\texttt{R}_\pi$. The equivalence can be seen as a result of Theorem 3.3. In the context of branch-coverage based testing, CoverMe introduces a program FOO_R such that any input found by minimizing FOO_R triggers a branch that has not yet been covered in the tested program. This guarantee is proved rigorously in [17] but can now be obtained "for free" from Theorem 3.3.

This work shows that generalizing FOO_R or R to the concept of weak-distance allows us to connect the two problem domains, namely, MO and FP analysis. This generalization qualifies MO reductions for other FP analyses and helps design alternative weak distances. It also implies that advances in addressing the limitations in an instance of weak-distance minimization can also benefit other instances. As an example, XSat [16] employs ULP, an integer-based FP metric, to mitigate unsoundness caused by inaccuracy of FP operations (Limitation 2). Thus, the weak-distance minimization framework allows one to consider using ULP to mitigate unsoundness in all the other instances listed in Section 2.

8 Conclusion

The theory and applications of weak-distance minimization have been grounded on the concepts of weak distance and mathematical optimization. Theorem 3.3 stipulates that the problem of analyzing a program Prog reduces to an equivalent problem of minimizing an associated weak distance W. What this reduction theory provides is an efficient and general approach applicable to a variety of real-world floating-point analysis problems. Two instances of this approach, branch coverage-based testing and quantifier-free FP satisfiability solving, have been implemented in the literature. Three other instances, boundary value analysis, path reachability and overflow detection, have been developed and investigated in this work. Our experiments and case studies have provided compelling evidence and promising results in supporting the theoretical claims, and suggests that our approach is effective in analyzing and detecting issues in scientific code. Further investigations can seek to overcome or mitigate the discussed limitations.

Acknowledgments

We would like to thank our shepherd, Rahul Sharma, the anonymous PLDI reviewers, Jean-Jacque Lévy, and Leo Liberti for constructive and valuable feedback on earlier drafts of this paper. A special thanks goes to Dr. Fang Cao for inspiring discussions and detailed comments on this work throughout its whole duration.

This research was supported in part by the EU's H2020 program under grant agreement number 732287 and the United States National Science Foundation (NSF) grant 1618158.

References

[1] 1985. IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985* (1985).
[2] 2019. The GNU C Library. https://www.gnu.org/software/libc/.
[3] 2019. SciPy: open-source software for mathematics, science, and engineering. https://www.scipy.org/.
[4] Milton Abramowitz and Irene A Stegun. 1965. *Handbook of mathematical functions: with formulas, graphs, and mathematical tables*. Vol. 55. Courier Corporation.
[5] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic detection of floating-point exceptions.. In *POPL'13*. 549–560.

[6] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. 2003. A Static Analyzer for Large Safety-Critical Software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*. ACM Press, San Diego, California, USA, 196–207.

[7] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. 2009. Combining Coq and Gappa for Certifying Floating-Point Programs. In *Proceedings of the 16th Symposium, 8th International Conference. Held As Part of CICM '09 on Intelligent Computer Mathematics (Calculemus '09/MKM '09)*. 59–74.

[8] Martin Brain, Cesare Tinelli, Philipp Ruemmer, and Thomas Wahl. 2015. An Automatable Formal Semantics for IEEE-754 Floating-Point Arithmetic. In *Proceedings of the 2015 IEEE 22Nd Symposium on Computer Arithmetic (ARITH '15)*. 160–167.

[9] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous Floating-point Mixed-precision Tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. 300–315.

[10] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver. In *Proceedings of TACAS (LNCS)*, Nir Piterman and Scott Smolka (Eds.), Vol. 7795. Springer.

[11] Patrick Cousot and Radhia Cousot. 1977. 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 (POPL'77)*. 238–252.

[12] Eva Darulova and Viktor Kuncak. 2017. Towards a Compiler for Reals. *ACM Trans. Program. Lang. Syst.* 39, 2, Article 8 (March 2017), 28 pages.

[13] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. 2009. Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software. In *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems (FMICS '09)*. 53–69.

[14] Roger Ferguson and Bogdan Korel. 1996. The Chaining Approach for Software Test Data Generation. *ACM Trans. Softw. Eng. Methodol.* 5, 1 (Jan. 1996), 63–86.

[15] Robert W Floyd. 1967. Assigning meanings to programs. *Mathematical aspects of computer science* 19, 19-32 (1967), 1.

[16] Zhoulai Fu and Zhendong Su. 2016. XSat: A Fast Floating-Point Satisfiability Solver. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV'16)*. Toronto, Ontario, Canada.

[17] Zhoulai Fu and Zhendong Su. 2017. Achieving high coverage for floating-point code via unconstrained programming. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 306–319.

[18] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA*. 213–223.

[19] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580.

[20] William Kahan. 1996. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE* 754, 94720-1776 (1996), 11.

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

[22] B. Korel. 1990. Automated Software Test Data Generation. *IEEE Trans. Softw. Eng.* 16, 8 (Aug. 1990), 870–879.

[23] Zhenqin Li and Harold A Scheraga. 1987. Monte Carlo-minimization approach to the multiple-minima problem in protein folding. *Proceedings of the National Academy of Sciences of the United States of America* 84, 19 (1987), 6611.

[24] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. 22–32.

[25] Phil McMinn. 2004. Search-based Software Test Data Generation: A Survey. *Softw. Test. Verif. Reliab.* 14, 2 (June 2004), 105–156.

[26] W. Miller and D. L. Spooner. 1976. Automatic Generation of Floating-Point Test Data. *IEEE Trans. Softw. Eng.* 2, 3 (May 1976), 223–226.

[27] M. Minoux. 1986. *Mathematical programming: Theory and algorithms*. Wiley, New York.

[28] Glenford J. Myers. 2004. The art of software testing. I–XV, 1–234.

[29] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2006. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM* 53, 6 (2006), 937–977.

[30] Michael JD Powell. 1964. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The computer journal* 7, 2 (1964), 155–162.

[31] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic Optimization of Floating-point Programs with Tunable Precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. New York, NY, USA, 53–64.

[32] Charles Severance. 1998. IEEE 754: An interview with William Kahan. *Computer* 31, 3 (1998), 114–115.

[33] J. P. Marques Silva and Karem A. Sakallah. 1996. GRASP: A New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design (ICCAD '96)*. Washington, DC, USA, 220–227.

[34] Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S Păsăreanu. 2011. CORAL: Solving complex constraints for symbolic pathfinder. In *NASA Formal Methods Symposium*. 359–374.

[35] Rainer Storn. 1999. System design by constraint adaptation and differential evolution. *IEEE Transactions on Evolutionary Computation* 3, 1 (1999), 22–34.

[36] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex: White Box Test Generation for .NET. In *Proceedings of the 2Nd International Conference on Tests and Proofs (TAP'08)*. Berlin, Heidelberg, 134–153.

[37] David J. Wales and Jonathan P. K. Doye. 1998. Global Optimization by Basin-Hopping and the Lowest Energy Structures of Lennard-Jones Clusters Containing up to 110 Atoms. *The Journal of Physical Chemistry A* 101, 28 (March 1998), 5111–5116.

[38] B. A. Wichmann. 1992. A Note on the Use of Floating Point in Critical Systems. *Comput. J.* 35, 1 (April 1992), 41–44.