Achieving High Coverage for Floating-Point Code via Unconstrained Programming

Zhoulai Fu Zhendong Su
University of California, Davis, USA
zhoulai.fu@gmail.com su@ucdavis.edu

Abstract

Achieving high code coverage is essential in testing, which gives us confidence in code quality. Testing floating-point code usually requires painstaking efforts in handling floating-point constraints, *e.g.*, in symbolic execution. This paper turns the challenge of testing floating-point code into the opportunity of applying unconstrained programming — the mathematical solution for calculating function minimum points over the entire search space. Our core insight is to derive a representing function from the floating-point program, any of whose minimum points is a test input guaranteed to exercise a new branch of the tested program. This guarantee allows us to achieve high coverage of the floating-point program by repeatedly minimizing the representing function.

We have realized this approach in a tool called CoverMe and conducted an extensive evaluation of it on Sun's C math library. Our evaluation results show that CoverMe achieves, on average, 90.8% branch coverage in 6.9 seconds, drastically outperforming our compared tools: (1) Random testing, (2) AFL, a highly optimized, robust fuzzer released by Google, and (3) Austin, a state-of-the-art coverage-based testing tool designed to support floating-point code.

CCS Concepts **•Software and its engineering → Dynamic analysis**

Keywords Unconstrained Programming, Representing Function, CoverMe

1. Introduction

Test coverage criteria attempt to quantify the quality of test data. Coverage-based testing [33] has become the state-of-the-practice in the software industry. The higher expectation for software quality and the shrinking development cycle have driven the research community to develop a spectrum of automated testing techniques for achieving high code coverage.

A significant challenge in coverage-based testing lies in the testing of numerical code, *e.g.*, programs with floating-point arithmetic, non-linear variable relations, or external function calls, such as logarithmic and trigonometric functions. Existing solutions include random testing [14, 23], symbolic execution [17, 24], and various search-based strategies [12, 25, 28, 31], which have found their way into many mature implementations [16, 39]. Random testing is easy to employ and fast, but ineffective in finding deep semantic issues and handling large input spaces; symbolic execution and its variants can perform systematic path exploration, but suffer from path explosion and are weak in dealing with complex program logic involving numerical constraints.

Our Work This paper considers the problem of coverage-based testing for floating-point code and focuses on the coverage of program branches. We turn the challenge of testing floating-point programs into the opportunity of applying unconstrained programming — the mathematical solution for calculating function minima over the entire search space [44].

Our approach has two unique features. First, it introduces the concept of *representing function*, which reduces the branch coverage based testing problem to the unconstrained programming problem. Second, the representing function is specially designed to achieve the following theoretical guarantee: Each minimum point of the representing function is an input to the tested floating-point program, and the input necessarily triggers a new branch unless all branches have been covered. This guarantee is critical not only for the soundness of our approach, but also for its efficiency — the unconstrained programming process is designed to cover only new branches; it does not waste efforts on covering already covered branches.

We have implemented our approach into a tool called CoverMe. CoverMe first derives the representing function from the program under test. Then, it uses an existing unconstrained programming algorithm to compute the minimum points. Note that the theoretical guarantee mentioned above allows us to apply any unconstrained programming algorithm

PLDI'17, June 18–23, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062383

```
 1 double tanh(double x){
 2   int jx, ix;
 3   jx = *(1+(int*)&x);  // High word of x
 4   ix = jx&0x7fffffff;
 5   if(ix>=0x7ff00000) {
 6     if (jx>=0) ...;
 7     else       ...;
 8   }
 9   if (ix < 0x40360000) {
10     if (ix<0x3c800000) ...;

12     if (ix>=0x3ff00000) ...;
13     else ...;
14   }
15   else ...;
16   return ...;
17 }
```

Figure 1: Benchmark program `s_tanh.c` taken from Fdlibm.

as a black box. Our implementation uses an off-the-shelf Monte Carlo Markov Chain (MCMC) [11] tool.

CoverMe has achieved high or full branch coverage for the tested floating-point programs. Fig. 1 lists the program `s_tanh.c` from our benchmark suite Fdlibm [6]. The program takes a `double` input. In Line 3, variable `jx` is assigned with the high word of `x` according to the comment given in the source code; the right-hand-side expression in the assignment takes the address of `x` (`&x`), cast it as a pointer-to-int (`int*`), add 1, and dereference the resulting pointer. In Line 4, variable `ix` is assigned with `jx` whose sign bit is masked off. Lines 5-15 are two nested conditional statements on `ix` and `jx`, which contain 16 branches in total according to Gcov [7]. Testing this type of programs is beyond the capabilities of traditional symbolic execution tools such as Klee [16]. CoverMe achieves full coverage within 0.7 seconds, dramatically outperforming our compared tools, including random testing, Google's AFL [1], and Austin [26] (a tool that combines symbolic execution and search-based heuristics). See details in Sect. 6.

Contributions This work introduces a promising automated testing solution for programs that are heavy on floating-point computation. Our approach designs the representing function whose minimum points are guaranteed to exercise new branches of the floating-point program. This guarantee allows us to apply any unconstrained programming solution as a black box, and to efficiently generate test inputs for covering program branches.

Our implementation, CoverMe, proves to be highly efficient and effective. It achieves 90.8% branch coverage on average, which is substantially higher than those obtained by random testing (38.0%), AFL (72.9%), and Austin (42.8%).

Paper Outline We structure the rest of the paper as follows. Sect. 2 presents background material on unconstrained programming. Sect. 3 gives an overview of our approach, and Sect. 4 presents the algorithm. Sect. 5 describes our implementation CoverMe, and Sect. 6 describes our evaluation. Sect. 7 surveys related work and Sect. 8 concludes the paper. For completeness, Sect. A-D of an extended version of this paper [5] provide additional details on our approach.

Notation We write $\mathbb{F}$ for floating-point numbers, $\mathbb{Z}$ for integers, $\mathbb{Z}_{>0}$ for strictly positive integers. we use the ternary operation $B\ ?\ a\ :\ a'$ to denote an evaluation to a if B holds, or a' otherwise. The lambda terms in the form of $\lambda x.f(x)$ may denote mathematical function f or its machine implementation according to the given context.

2. Background

This section presents the definition and algorithms of unconstrained programming that will be used in this paper. As mentioned in Sect. 1, we will treat the unconstrained programming algorithms as black boxes.

Unconstrained Programming We formalize unconstrained programming as the problem below [19]:

$$\begin{array}{ll} \text{Given} & f : \mathbb{R}^n \to \mathbb{R} \\ \text{Find} & x^* \in \mathbb{R}^n \text{ for which } f(x^*) \le f(x) \text{ for all } x \in \mathbb{R}^n \end{array}$$

where f is the objective function; x^*, if found, is called a minimum point; and $f(x^*)$ is the minimum. An example is

$$f(x_1, x_2) = (x_1 - 3)^2 + (x_2 - 5)^2, \tag{1}$$

which has the minimum point $x^* = (3, 5)$.

Unconstrained Programming Algorithms We consider two kinds of algorithms, known as local optimization and global optimization. Local optimization focuses on how functions are shaped near a given input and where a minimum can be found at local regions. It usually involves standard techniques such as Newton's or the steepest descent methods [34]. Fig. 2(a) shows a common local optimization method with the objective function $f(x)$ that equals 0 if $x \le 1$, or $(x-1)^2$ otherwise. The algorithm uses tangents of f to converge to a minimum point quickly. In general, local optimization is usually fast. If the objective function is smooth to some degree, the local optimization can deduce function behavior in the neighborhood of a particular point x by using information at x only (the tangent here).

Global optimization for unconstrained programming searches for minimum points over $\mathbb{R}^n$. Many global optimization algorithms have been developed. This work uses Monte Carlo Markov Chain (MCMC) [11]. MCMC is a sampling method that targets (usually unknown) probability distributions. A fundamental fact is that MCMC sampling follows the target distributions asymptotically, which is formalized by the lemma below. For simplicity, we present the lemma in the form of discrete-valued probabilities [11].

Lemma 2.1. *Let x be a random variable, A be an enumerable set of the possible values of x, f be a target probability*

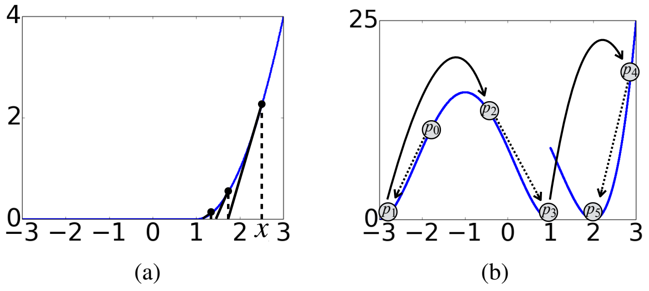


Figure 2: (a) Local optimization example with objective function $\lambda x.x \le 1\ ?\ 0 : (x-1)^2$. The local optimization algorithm uses tangents of the curve to quickly converge to a minimum point; (b) Global optimization example with objective function $\lambda x.x \le 1\ ?\ ((x+1)^2-4)^2 : (x^2-4)^2$. The MCMC method starts from p_0, converges to local minimum p_1, performs a Monte-Carlo move to p_2 and converges to p_3. Then it moves to p_4 and converges to p_5.

distribution for x, i.e., *the probability of x taking value $a \in A$ is $f(a)$. Then, an MCMC sampling sequence $x_1,\dots,x_n,\dots$ satisfies the property that Prob$(x_n = a) \to f(a)$.*

For example, consider the target distribution of coin tossing with 0.5 probability of getting a head. An MCMC sampling is a sequence of random variables $x_1,\dots, x_n,\dots$, such that the probability of x_n being head converges to 0.5.

Using MCMC to solve unconstrained programming problems provides multiple advantages in practice. First, Lem. 2.1 ensures that MCMC sampling can be configured to attain the minimum points with higher probability than the other points. Second, MCMC integrates well with local optimization. An example is the Basinhopping algorithm [29] used in Sect. 5. Third, MCMC techniques are robust; some variants can even handle high dimensional problems [38] or non-smooth objective functions [20]. Our approach uses unconstrained optimization as a black box. Fig. 2(b) provides a simple example. Steps $p_0 \to p_1$, $p_2 \to p_3$, and $p_4 \to p_5$ employ local optimization; Steps $p_1 \to p_2$ and $p_3 \to p_4$, known as Monte-Carlo moves [11], avoid the MCMC sampling being trapped in the local minimum points.

3. Overview

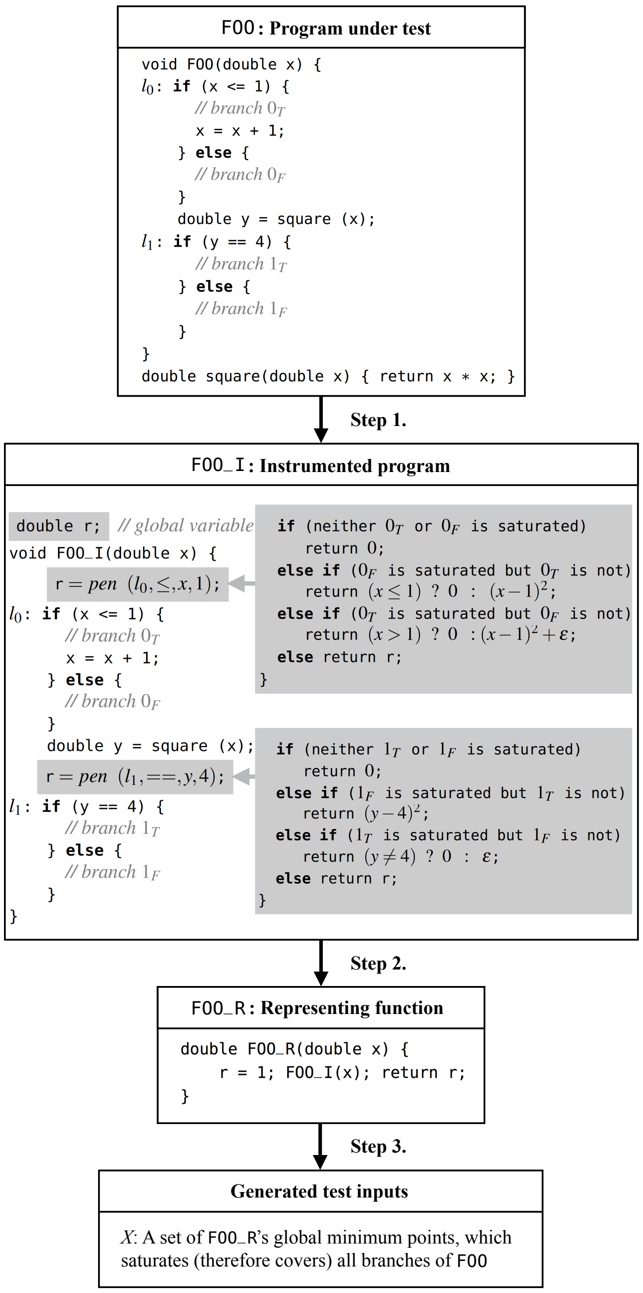
This section states the problem and illustrates our solution.

Notation Let `FOO` be the program under test with N conditional statements, labeled by $l_0,\dots,l_{N-1}$. Each l_i has a `true` branch i_T and a `false` branch i_F. We write dom(`FOO`) to denote the input domain of program `FOO`.

3.1 Problem Statement

Definition 3.1. The problem of *branch coverage-based testing* aims to find a set of inputs $X \subseteq$ dom(`FOO`) that *covers* all branches of `FOO`. Here, we say a branch is "covered" by X if it is passed through by executing `FOO` with an input of X.

We scope the problem with three assumptions. They will be partially relaxed in our implementation (Sect. 5):



FOO: Program under test

```
void FOO(double x) {
l0: if (x <= 1) {
       // branch 0T
       x = x + 1;
    } else {
       // branch 0F
    }
    double y = square (x);
l1: if (y == 4) {
       // branch 1T
    } else {
       // branch 1F
    }
}
double square(double x) { return x * x; }
```

Step 1.

FOO_I: Instrumented program

```
double r; // global variable
void FOO_I(double x) {
    r = pen (l0,≤,x,1);
l0: if (x <= 1) {
       // branch 0T
       x = x + 1;
    } else {
       // branch 0F
    }
    double y = square (x);
    r = pen (l1,==,y,4);
l1: if (y == 4) {
       // branch 1T
    } else {
       // branch 1F
    }
}
```

```
if (neither 0T or 0F is saturated)
   return 0;
else if (0F is saturated but 0T is not)
   return (x ≤ 1) ? 0 : (x−1)^2;
else if (0T is saturated but 0F is not)
   return (x > 1) ? 0 : (x−1)^2 + ε;
else return r;
}
```

```
if (neither 1T or 1F is saturated)
   return 0;
else if (1F is saturated but 1T is not)
   return (y−4)^2;
else if (1T is saturated but 1F is not)
   return (y ≠ 4) ? 0 : ε;
else return r;
}
```

Step 2.

FOO_R: Representing function

```
double FOO_R(double x) {
    r = 1; FOO_I(x); return r;
}
```

Step 3.

Generated test inputs

X: A set of FOO_R's global minimum points, which saturates (therefore covers) all branches of FOO

Figure 3: An illustration of our approach. The goal is to find inputs that saturate (therefore cover) all branches of `FOO`, *i.e.*, $\{0_T, 0_F, 1_T, 1_F\}$.

(a) The inputs of `FOO` are floating-point numbers;

(b) Each Boolean condition in `FOO` is an arithmetic comparison between two floating-point variables or constants; and

(c) Each branch of `FOO` is feasible, *i.e.*, it is covered by an input of program `FOO`.

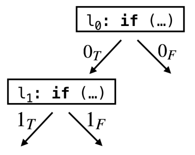
The concept below is crucial. It allows us to rewrite the branch coverage-based testing problem as defined in Def. 3.1 to an equivalent, but easier-to-solve one.

Definition 3.2. Let X be a set of inputs generated during the testing process. We say that a branch is *saturated* by X if the branch itself and all its descendant branches, if any, are covered by X. Here, a branch b' is called a descendant branch of b if there exists a control flow from b to b'. We write

$$\text{Saturate}(X) \tag{2}$$

for the set of branches saturated by X.

The control-flow graph on the right illustrates Def. 3.2. Suppose that an input set X covers $\{0_T, 0_F, 1_F\}$. Then $\text{Saturate}(X) = \{0_F, 1_F\}$. Branch 1_T is not saturated because it is not covered; branch 0_T is not saturated neither because its descendant branch 1_T is not covered.



Our approach reformulates the branch coverage-based testing problem with the lemma below.

Lemma 3.3. *Let `FOO` be the program under test. Assume Def. 3.1(a-c) hold. Then, a set of inputs $X \subseteq \text{dom}(\texttt{FOO})$ saturates all `FOO`'s branches iff X covers all `FOO`'s branches.*

By consequence, the goal of branch coverage-based testing defined in Def. 3.1 can be equivalently stated: To find a set of inputs $X \subseteq \text{dom}(\texttt{FOO})$ that saturates all `FOO`'s branches.

3.2 Illustration of Our Approach

Fig. 3 illustrates our approach. The program under test has two conditional statements l_0 and l_1. Our objective is to find an input set that saturates all branches, namely, 0_T, 0_F, 1_T, and 1_F. Our approach proceeds in three steps:

Step 1 We inject the global variable `r` in `FOO`, and, immediately before each control point l_i, we inject the assignment

$$\texttt{r} = pen \tag{3}$$

where *pen* invokes a code segment with parameters associated with l_i. The idea of *pen* is to capture the *distance of the program input from saturating a branch that has not yet been saturated*. Observe that in Fig. 3, both injected *pen* return different values depending on whether the branches at l_i are saturated or not. `FOO_I` denotes the instrumented program.

Step 2 This step constructs the representing function that we have mentioned in Sect. 1. The representing function is the driver program `FOO_R` shown in Fig. 3. It initializes `r` to 1, invokes `FOO_I` and then returns `r` as the output of `FOO_R`. That is to say, $\texttt{FOO\_R}(x)$ for a given input x calculates the value of `r` at the end of executing $\texttt{FOO\_I}(x)$.

The key in Steps 1 and 2 is to design *pen* so that `FOO_R` meets the two conditions below:

C1. $\texttt{FOO\_R}(x) \geq 0$ for all x, and

C2. $\texttt{FOO\_R}(x) = 0$ iff x saturates a new branch. In other words, a branch that has not been saturated by the generated input set X becomes saturated with $X \cup \{x\}$, *i.e.*, $\text{Saturate}(X) \neq \text{Saturate}(X \cup \{x\})$.

Conditions C1 and C2 are essential because they allow us to transform a branch coverage-based testing problem to an unconstrained programming problem. Ideally, we can then saturate all branches of `FOO` by repeatedly minimizing `FOO_R` as shown in the step below.

Step 3 We calculate the minimum points of `FOO_R` via unconstrained programming algorithms described in Sect. 2. Typically, we start with an input set $X = \emptyset$ and $\text{Saturate}(X) = \emptyset$. We minimize `FOO_R` and obtain a minimum point x^* which necessarily saturates a new branch by condition C2. Then we have $X = \{x^*\}$ and we minimize `FOO_R` again which gives another input x^{**} and $\{x^*, x^{**}\}$ saturates a branch that is not saturated by $\{x^*\}$. We continue this process until all branches are saturated. Note that when Step 3 terminates, $\texttt{FOO\_R}(x)$ must be strictly positive for any input x, due to C1 and C2.

Tab. 1 illustrates a scenario of how our approach saturates all branches of program `FOO` given in Fig. 3. Each "**#n**" below corresponds to a line in the table. We write pen_0 and pen_1 to distinguish the two *pen* injected at l_0 and l_1 respectively. (**#1**) Initially, no branch has been saturated. Both pen_0 and pen_1 set $\texttt{r} = 0$, and `FOO_R` returns 0 for any input. Suppose $x^* = 0.7$ is found as the minimum point. (**#2**) The branch 1_F is now saturated and 1_T is not. Thus, pen_1 sets $\texttt{r} = (y-4)^2$. Minimizing `FOO_R` gives $x^* = -3.0$, 1.0, or 2.0. We have illustrated how these minimum points can be computed in unconstrained programming in Fig. 2(b). Suppose $x^* = 1.0$ is found. (**#3**) Both 1_T and 1_F, as well as 0_T, are saturated by the generated inputs $\{0.7, 1.0\}$. Thus, pen_1 returns the previous `r` and `FOO_R` amounts to pen_0, which returns 0 if $x > 1$ or $(x-1)^2 + \varepsilon$ otherwise, where ε is a small predefined constant (Sect. 4). Suppose $x^* = 1.1$ is found as the minimum point. (**#4**) All branches have been saturated. In this case, both pen_0 and pen_1 return `r`. `FOO_R` returns 1 for all x since `FOO_R` initializes `r` as 1. Suppose the minimum found is $x^* = -5.2$. It necessarily satisfies $\texttt{FOO\_R}(x^*) > 0$, which confirms that all branches have been saturated (due to C1 and C2).

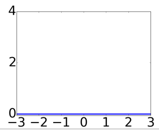
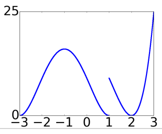
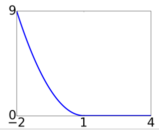
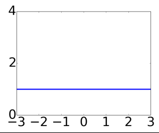
4. Algorithm

We provide details corresponding to the three steps in Sect. 3.2. The algorithm is summarized in Algo. 1.

Algorithm for Step 1 The outcome of this step is the instrumented program `FOO_I`. As explained in Sect. 3.2, the essence is to inject the variable `r` and the assignment $\texttt{r} = pen$ before each conditional statement (Algo. 1, Lines 1-4).

To define *pen*, we first introduce a set of helper functions that are sometimes known as *branch distance*. There are many

Table 1: A scenario of how our approach saturates all branches of FOO by repeatedly minimizing FOO_R. Column "Saturate": Branches that have been saturated. Column "FOO_R": The representing function and its plot. Column "x^*": The point where FOO_R attains the minimum. Column "X": Generated test inputs.

#	Saturate	FOO_R		x^*	X
1	$\emptyset$	$\lambda x.0$		0.7	$\{0.7\}$
2	$\{1_F\}$	$\lambda x.\begin{cases}((x+1)^2-4)^2 & x \le 1 \\ (x^2-4)^2 & \text{else}\end{cases}$		1.0	$\{0.7, 1.0\}$
3	$\{0_T, 1_T, 1_F\}$	$\lambda x.\begin{cases}0 & x > 1 \\ (x-1)^2+\varepsilon & \text{else}\end{cases}$		1.1	$\{0.7, 1.0, 1.1\}$
4	$\{0_T, 1_T, 0_F, 1_F\}$	$\lambda x.1$		-5.2	$\{0.7, 1.0, 1.1, -5.2\}$

different forms of branch distance in the literature [25, 31]. We define ours with respect to an arithmetic condition $a\ op\ b$.

Definition 4.1. Let $a, b \in \mathbb{R}$, $op \in \{==, \le, <, \ne, \ge, >\}$, $\varepsilon \in \mathbb{R}_{>0}$. We define *branch distance* $d_\varepsilon(op, a, b)$ as follows:

$$d_\varepsilon(==, a, b) \overset{\text{def}}{=} (a-b)^2 \tag{4}$$

$$d_\varepsilon(\le, a, b) \overset{\text{def}}{=} (a \le b)\ ?\ 0 : (a-b)^2 \tag{5}$$

$$d_\varepsilon(<, a, b) \overset{\text{def}}{=} (a < b)\ ?\ 0 : (a-b)^2 + \varepsilon \tag{6}$$

$$d_\varepsilon(\ne, a, b) \overset{\text{def}}{=} (a \ne b)\ ?\ 0 : \varepsilon \tag{7}$$

and $d_\varepsilon(\ge, a, b) \overset{\text{def}}{=} d_\varepsilon(\le, b, a)$, $d_\varepsilon(>, a, b) \overset{\text{def}}{=} d_\varepsilon(<, b, a)$. We use ε to denote a small positive floating-point close to machine epsilon. The idea is to treat a strict floating-point inequality $x > y$ as a non-strict inequality $x \ge y + \varepsilon$, *etc.* We will drop the explicit reference to ε when using the branch distance.

The intention of $d(op, a, b)$ is to quantify how far a and b are from attaining $a\ op\ b$. For example, $d(==, a, b)$ is strictly positive when $a \ne b$, becomes smaller when a and b go closer, and vanishes when $a == b$. The following property holds:

$$d(op, a, b) \ge 0 \quad \text{and} \quad d(op, a, b) = 0 \Leftrightarrow a\ op\ b. \tag{8}$$

As an analogue, we set *pen* to quantify how far an input is from saturating a new branch. We define *pen* following Algo. 1, Lines 14-23.

Definition 4.2. For branch coverage-based testing, the function *pen* has four parameters, namely, the label of the conditional statement l_i, op, and a and b from the arithmetic condition $a\ op\ b$.

(a) If neither of the two branches at l_i is saturated, we let *pen* return 0 because any input saturates a new branch (Lines 16-17).

(b) If one branch at l_i is saturated but the other is not, we set r to be the distance to the unsaturated branch (Lines 18-21).

(c) If both branches at l_i have already been saturated, *pen* returns the previous value of variable r (Lines 22-23).

For example, *pen* at l_0 and l_1 are invoked as $pen(l_i, \le, x, 1)$ and $pen(l_1, ==, y, 4)$ respectively in Fig. 3.

Algorithm for Step 2 This step constructs the representing function FOO_R (Algo. 1, Line 5). Its input domain is the same as that of FOO_I and FOO, and its output domain is double. FOO_R initializes r to 1. The initialization is to guarantee that FOO_R returns a non-negative value whenever all branches are saturated (Sect. 3.2, Step 2). FOO_R then invokes FOO_I(x) and returns the value of r at the end of executing FOO_I(x).

As mentioned in Sect. 3.2, it is important to ensure that FOO_R meets conditions C1 and C2. The condition C1 holds true since FOO_R returns the value of the instrumented r, which is never assigned a negative quantity. The theorem below states FOO_R also satisfies C2.

Theorem 4.3. *Let FOO_R be the program constructed in Algo. 1, and S the branches that have been saturated. Then, for any input $x \in \text{dom}(FOO)$, FOO_R$(x) = 0 \Leftrightarrow x$ saturates a branch that does not belong to S.*

Proof. We first prove the $\Rightarrow$ direction. Take an arbitrary x such that FOO_R$(x) = 0$. Let $\tau = [l_0, \dots l_n]$ be the path in FOO passed through by executing FOO(x). We know, from Lines 2-4 of the algorithm, that each l_i is preceded by an invocation of *pen* in FOO_R. We write pen_i for the one injected before l_i and divide $\{pen_i \mid i \in [1, n]\}$ into three groups. For the given input x, we let *P1*, *P2* and *P3* denote the groups of pen_i that are defined in Def. 4.2(a), (b) and (c), respectively. Then, we can always have a prefix path of $\tau = [l_0, \dots l_m]$, with $0 \le m \le n$ such that each pen_i for $i \in [m+1, n]$ belongs to *P3*, and each pen_i for $i \in [0, m]$ belongs to either *P1* or *P2*. Here, we can guarantee the existence of such an m because, otherwise, all pen_i belong in *P3*, and FOO_R becomes $\lambda x.1$. The latter contradicts the assumption that FOO_R$(x) = 0$. Because each pen_i for $i > m$ does nothing but performs r = r, we know that FOO_R(x) equals to the exact value of r that pen_m assigns. Now consider two disjunctive cases on pen_m. If pen_m is in *P1*, we immediately conclude that x saturates a new branch. Otherwise, if pen_m is in *P2*, we obtains the same from Eq. (8). Thus, we have established the $\Rightarrow$ direction of the theorem.

To prove the $\Leftarrow$ direction, we use the same notation as above, and let x be the input that saturates a new branch, and $[l_0, \dots, l_n]$ be the exercised path. Assume that l_m where $0 \le m \le n$ corresponds to the newly saturated branch. We know from the algorithm that (1) pen_m updates r to 0, and (2)

Algorithm 1: Branch coverage-based testing

```
Input:  FOO       Program under test
        $n\_start$   Number of starting points
        LM        Local optimization used in MCMC
        $n\_iter$    Number of iterations for MCMC
Output: $X$   Generated input set
/* Step 1                                                  */
Inject global variable r in FOO
for conditional statement $l_i$ in FOO do
    Let the Boolean condition at $l_i$ be $a$ $op$ $b$ where
    $op \in \{\leq, <, =, >, \geq, \neq\}$
    Insert assignment $r = pen(l_i, op, a, b)$ before $l_i$
/* Step 2                                                  */
Let FOO_I be the newly instrumented program, and FOO_R be:
  double FOO_R(double x) {r = 1; FOO_I(x); return r;}
/* Step 3                                                  */
Let Saturate $= \emptyset$
Let $X = \emptyset$
for $k = 1$ to $n\_start$ do
    Randomly take a starting point $x$
    Let $x^* =$ MCMC(FOO_R, $x$)
    if FOO_R($x^*$) $= 0$ then $X = X \cup \{x^*\}$
    Update Saturate
return $X$
Function $pen(l_i, op, a, b)$
    Let $i_T$ and $i_F$ be the true and the false branches at $l_i$
    if $i_T \notin$ Saturate and $i_F \notin$ Saturate then
        return 0
    else if $i_T \notin$ Saturate and $i_F \in$ Saturate then
        return $d(op, a, b)$ /* $d$: Branch distance              */
    else if $i_T \in$ Saturate and $i_F \notin$ Saturate then
        return $d(\overline{op}, a, b)$ /* $\overline{op}$: the opposite of $op$     */
    else /* $i_T \in$ Saturate and $i_F \in$ Saturate              */
        return r
Function MCMC($f$, $x$)
    $x_L =$ LM($f$, $x$)
    /* Local minimization                                  */
    for $k = 1$ to $n\_iter$ do
        Let $\delta$ be a random perturbation generation from a
        predefined distribution
        Let $\widetilde{x_L} =$ LM($f$, $x_L + \delta$)
        if $f(\widetilde{x_L}) < f(x_L)$ then $accept = true$
        else
            Let $m$ be a random number generated from the
            uniform distribution on $[0, 1]$
            Let $accept$ be the Boolean $m < \exp(f(x_L) - f(\widetilde{x_L}))$
        if $accept$ then $x_L = \widetilde{x_L}$
    return $x_L$
```

each pen_i such that $i > m$ maintains the value of r because their descendant branches have been saturated. We have thus proven the $\Leftarrow$ direction of the theorem. ☐

Algorithm for Step 3 The main loop (Algo. 1, Lines 8-12) relies on an existing MCMC engine. It takes an objective function and a starting point and outputs x^* that it regards as a minimum point. Each iteration of the loop launches MCMC from a randomly selected starting point (Line 9). From each starting point, MCMC computes the minimum point x^* (Line 10). If FOO_R(x^*) $= 0$, x^* is to be added to X (Line 11). Thm. 4.3 ensures that x^* saturates a new branch if FOO_R(x^*) $= 0$. Therefore, in theory, we only need to set $n\_start = 2 * N$ where N denotes the number of conditional statements, so to saturate all $2 * N$ branches. In practice, however, we set $n\_start > 2 * N$ because MCMC cannot guarantee that its output is a true global minimum point.

The MCMC procedure (Algo. 1, Lines 24-34) is also known as the Basinhopping algorithm [29]. It is an MCMC sampling over the space of the local minimum points [30]. The random starting point x is first updated to a local minimum point x_L (Line 25). Each iteration (Lines 26-33) is composed of the two phases that are classic in the Metropolis-Hastings algorithm family of MCMC. In the first phase (Lines 27-28), the algorithm *proposes* a sample $\widetilde{x_L}$ from the current sample x. The sample $\widetilde{x_L}$ is obtained with a perturbation δ followed by a local minimization, *i.e.*, $\widetilde{x_L} = \text{LM}(f, x_L + \delta)$ (Line 28), where LM denotes a local minimization in Basinhopping, and f is the objective function. The second phase (Lines 29-33) decides whether the proposed $\widetilde{x_L}$ should be accepted as the next sampling point. If $f(\widetilde{x_L}) < f(x_L)$, the proposed $\widetilde{x_L}$ will be sampled; otherwise, $\widetilde{x_L}$ may still be sampled, but only with the probability of $\exp((f(x_L) - f(\widetilde{x_L}))/T)$, in which T (called the annealing temperature) is set to 1 in Algo. 1 for simplicity.

5. Implementation

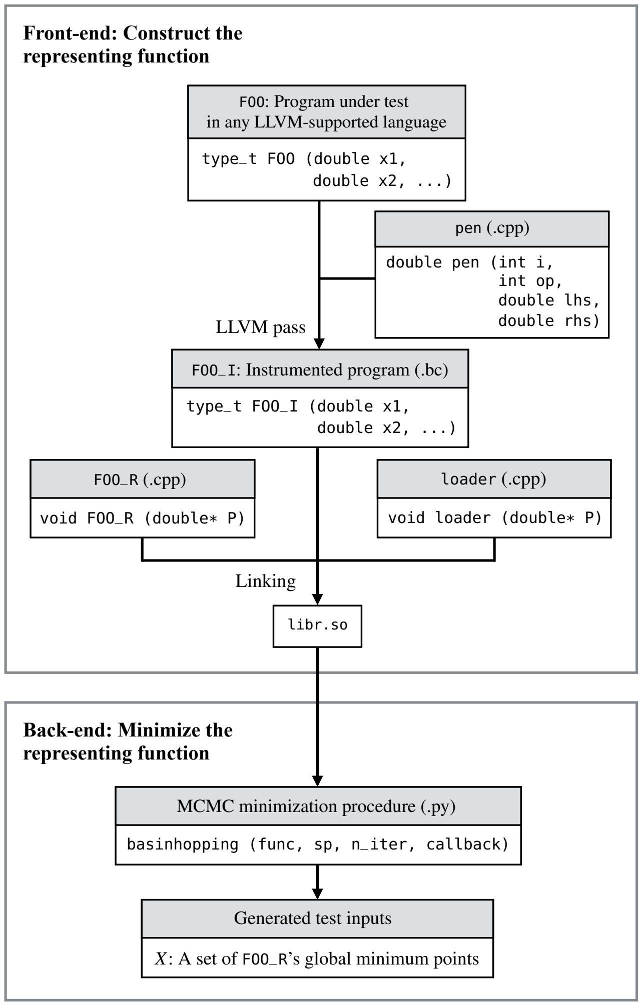
As a proof-of-concept demonstration, we have implemented Algo. 1 into the tool CoverMe. This section presents its implementation and technical details.

5.1 Frontend of CoverMe

The frontend implements Steps 1 and 2 of Algo. 1. CoverMe compiles the program under test FOO to LLVM IR with Clang [2]. Then it uses an LLVM pass [9] to inject assignments. The program under test can be in any LLVM-supported language, *e.g.*, Ada, the C/C++ language family, or Julia, *etc.* Our implementation has been tested on C code.

Fig. 4 illustrates FOO as a function of signature `type_t FOO (type_t1 x1, type_t2 x2, ...)`. The return type (output) of the function, `type_t`, can be any kind of types supported by C, whereas the types of the input parameters, `type_t1, type_t2, ...`, are restricted to `double` or `double*`. We have explained the signature of *pen* in Def. 4.2. Note that CoverMe does not inject *pen* itself into FOO, but instead injects assignments that invoke *pen*. We implement *pen* in a separate C++ file.

The frontend also links FOO_I and FOO_R with a simple program `loader` into a shared object file `libr.so`, which is the outcome of the frontend. It stores the representing function in the form of a shared object file (.so file).



Front-end: Construct the representing function
FOO: Program under test in any LLVM-supported language
type_t FOO (double x1, double x2, ...)
pen (.cpp)
double pen (int i, int op, double lhs, double rhs)
LLVM pass
FOO_I: Instrumented program (.bc)
type_t FOO_I (double x1, double x2, ...)
FOO_R (.cpp)
void FOO_R (double* P)
loader (.cpp)
void loader (double* P)
Linking
libr.so
Back-end: Minimize the representing function
MCMC minimization procedure (.py)
basinhopping (func, sp, n_iter, callback)
Generated test inputs
X: A set of FOO_R's global minimum points

Figure 4: CoverMe implementation.

5.2 Backend of CoverMe

The backend implements Step 3 of Algo. 1. It invokes the representing function via `libr.so`. The kernel of the backend is an external MCMC engine. It uses the off-the-shelf implementation known as *Basinhopping* from the Scipy Optimization package [10]. Basinhopping takes a range of input parameters. Fig. 4 shows the important ones for our implementation `basinhopping(f, sp, n_iter, call_back)`, where `f` refers to the representing function from `libr.so`, `sp` is a starting point as a Python `Numpy` array, `n_iter` is the iteration number used in Algo. 1 and `call_back` is a client-defined procedure. Basinhopping invokes `call_back` at the end of each iteration (Algo. 1, Lines 24-34). The call_back procedure allows CoverMe to terminate if it saturates all branches. In this way, CoverMe does not need to wait until passing all *n_start* iterations (Algo. 1, Lines 8-12).

5.3 Technical Details

Sect. 3 assumes Def. 3.1(a-c) for the sake of simplification. This section discusses how CoverMe relaxes the assumptions when handling real-world floating-point code. We also show how CoverMe handles function calls at the end of this section.

Handling Pointers (Relaxing Def. 3.1(a)) We consider only pointers to floating-point numbers. They may occur (1) in an input parameter, (2) in a conditional statement, or (3) in the code body but not in the conditional statement.

CoverMe inherently handles case (3) because it is execution-based and does not need to analyze pointers and their effects. CoverMe currently does not handle case (2) and ignores these conditional statements by not injecting *pen* before them.

Below we explain how CoverMe deals with case (1). A branch coverage testing problem for a program whose inputs are pointers to doubles, can be regarded as the same problem with a simplified program under test. For instance, finding test inputs to cover branches of program `void FOO(double* p) {if (*p <= 1)... }` can be reduced to testing the program `void FOO_with_no_pointer (double x) {if (x <= 1)... }`. CoverMe transforms program `FOO` to `FOO_with_no_pointer` if a `FOO`'s input parameter is a floating-point pointer.

Handling Comparison between Non-floating-point Expressions (Relaxing Def. 3.1(b)) We have encountered situations where a conditional statement invokes a comparison between non floating-point numbers. CoverMe handles these situations by first promoting the non floating-point numbers to floating-point numbers and then injecting *pen* as described in Algo. 1. For example, before a conditional statement like `if (xi op yi)` where `xi` and `yi` are integers, CoverMe injects `r = pen (i, op, (double) x, (double) y));`. Note that such an approach does not allow us to handle data types that are incompatible with floating-point types, *e.g.*, conditions like `if (p != Null)`, which CoverMe has to ignore.

Handling Infeasible Branches (Relaxing Def. 3.1(c)) Infeasible branches are those that cannot be exercised by any input. Determining which branch is infeasible is difficult in general. CoverMe uses a heuristic to detect infeasible branches. When CoverMe finds a minimum that is not zero, it deems the unvisited branch of the last conditional to be infeasible and adds it to Saturate, the set of unvisited and deemed-to-be infeasible branches.

Imagine that we modify l_1 of the program `FOO` in Fig. 3 to the conditional statement `if (y == -1)`. Then the branch 1_T becomes infeasible. We rewrite this modified program below and illustrate how we deal with infeasible branches.

```
l0: if (x <= 1) {x++};
    y = square(x);
l1: if (y == -1) {...}
```

where we omit the concrete implementation of `square`.

Let `FOO_R` denote the representing function constructed for the program. In the minimization process, whenever CoverMe obtains x^* such that `FOO_R`$(x^*) > 0$, CoverMe selects a branch that it regards infeasible. CoverMe selects the branch as follows: Suppose x^* exercises a path τ whose last conditional statement is denoted by l_z, and, without loss of generality, suppose z_T is passed through by τ, then CoverMe regards z_F as an infeasible branch.

In the modified program above, if 1_F has been saturated, the representing function evaluates to $(y+1)^2$ or $(y+1)^2 + 1$, where y equals to the non-negative `square(x)`. Thus, the minimum point x^* must satisfy `FOO_R`$(x^*) > 0$ and its triggered path ends with branch 1_F. CoverMe then regards 1_T as an infeasible branch.

CoverMe then regards the infeasible branches as already saturated. It means, in line 12 of Algo. 1, CoverMe updates Saturate with saturated branches and infeasible branches (more precisely, branches that CoverMe regards infeasible).

The presented heuristic works well in practice (See Sect. 6), but we do not claim that our heuristic always correctly detects infeasible branches.

Handling Function Calls By default, CoverMe injects `r` = pen_i only in the entry function to test. If the entry function invokes other external functions, they will not be transformed. For example, in the program `FOO` of Fig. 3, we do not transform `square(x)`. In this way, CoverMe only attempts to saturate all branches for a single function at a time.

However, CoverMe can also easily handle functions invoked by its entry function. As a simple example, consider:

```
void FOO (double x) { GOO(x); }
void GOO (double x) { if (sin(x) <= 0.99) ... }
```

If CoverMe aims to saturate `FOO` and `GOO` but not `sin`, and it sets `FOO` as the entry function, then it instruments both `FOO` and `GOO`. Only `GOO` has a conditional statement, and CoverMe injects an assignment on `r` in `GOO`.

6. Evaluation

This section presents our evaluation of CoverMe. All experiments are performed on a laptop with a 2.6 GHz Intel Core i7 running a Ubuntu 14.04 virtual machine with 4GB RAM. The main results are presented in Tab. 2, 3 and Fig. 5.

6.1 Experimental Setup

Benchmarks Our benchmarks are C floating-point programs from the Freely Distributable Math Library (Fdlibm) 5.3, developed by Sun Microsystems, Inc. These programs are available from the network library netlib. We choose Fdlibm because it represents a set of floating-point programs with reference quality and a large user group. For example, the Java SE 8's math library is defined with respect to Fdlibm 5.3. [3], Fdlibm is also ported to Matlab, JavaScript, and has been integrated in Android.

Fdlibm 5.3 has 80 programs. Each program defines one or multiple math functions. In total, Fdlibm 5.3 contains 92 math functions. Among them, we exclude 36 functions that have no branch, 11 functions involving input parameters that are not floating-point, and 5 static C functions. Our benchmarks include *all* remaining 40 functions. Sect. A of [5] lists all excluded functions in Fdlibm 5.3.

Parameter Settings CoverMe supports three parameters: (1) the Monte-Carlo iteration number *n_iter*, (2) the local optimization algorithm `LM`, and (3) the number of starting points *n_start*. They correspond to the three input parameters of Algo. 1. Our experiment sets *n_iter* = 5, *n_start* = 500, and `LM` = "powell" (*i.e.*, Powell's algorithm [37]).

Tools for Comparison We have compared CoverMe with three tools that support floating-point coverage-based testing:

- Rand is a pure random testing tool. We have implemented Rand using a pseudo-random number generator.
- AFL [1] is a gray-box testing tool released by the Google security team. It integrates a variety of guided search strategies and employs genetic algorithms to efficiently increase code coverage.
- Austin [26] is a coverage-based testing tool that implements symbolic execution and search-based heuristics. Austin shows to [27] be more effective than a testing tool called CUTE [39] (which is not publicly available).

We have decided to not consider the following tools:

- Klee [16] is the state-of-the-art implementation of symbolic execution. We do not consider Klee because its expression language does not support floating-point constraints. In addition, many operations in our benchmark programs, such as pointer reference, dereference, type casting, are not supported by Klee's backend solver STP [22], or any other backend solvers compatible with the Klee Multi-solver extension [35].[1]
- Klee-FP [18] is a variant of Klee geared toward reasoning about floating-point value equivalence. It determines equivalence by checking whether two floating-point values are produced by the same operations [18]. We do not consider Klee-FP because its special-purpose design does not support coverage-based testing.
- Pex [42] is a coverage tool based on dynamic symbolic execution. We do not consider Pex because it can only run for .NET programs on Windows whereas our tested programs are in C, and our testing platform is Linux.
- FloPSy [28] is a floating-point testing tool that combines search based testing and symbolic execution. We do not consider this tool because it is developed by the same author of Austin and before Austin is released, and the tool is not available to us.

Coverage Measurement Our evaluation focuses on branch coverage. Sect. C also shows our line coverage results. For CoverMe and Rand, we use the Gnu coverage tool Gcov [7].

[1]In the recent Klee-dev mailing list, the Klee developers mentioned that Klee or its variant Klee-FP only has basic floating-point support and they are still "working on full FP support for Klee" [8].

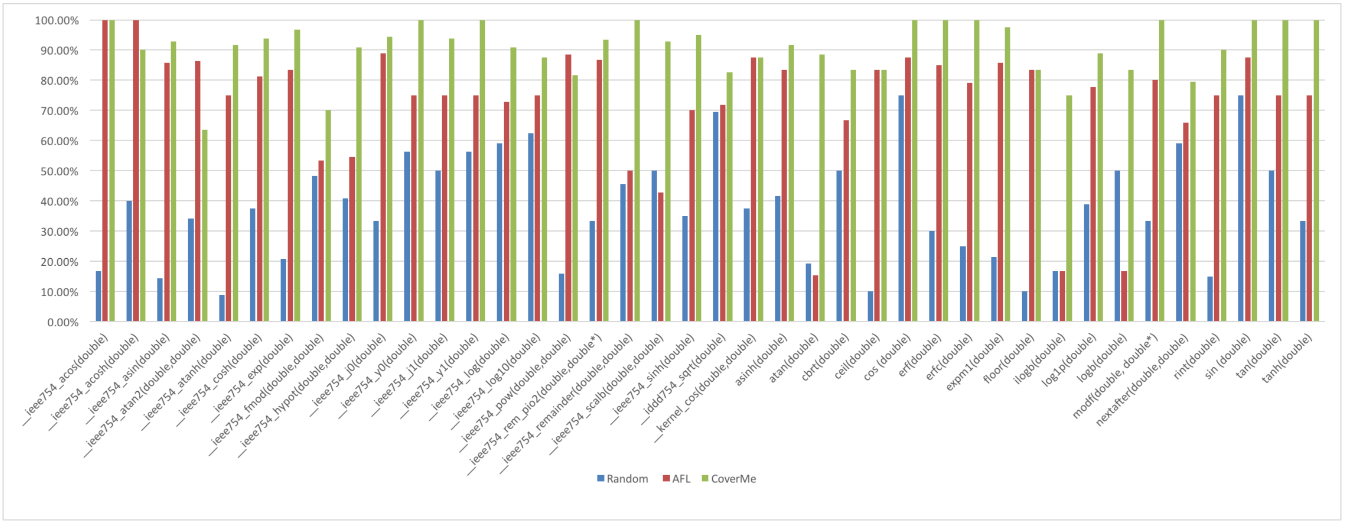


Figure 5: Benchmark results corresponding to the data given in Tab. 2. The y-axis refers to the branch coverage; the x-axis refers to the benchmarks.

For AFL, we use AFL-cov [4], a Gcov-based coverage analysis tool for AFL. For Austin, we calculate the branch coverage by the number of covered branches provided in the `csv` file produced by Austin when it terminates.

Time Measurement To compare the running time of CoverMe with the other tools requires careful design. CoverMe, Rand, and AFL all have the potentials to achieve higher coverage if given more time or iteration cycles. CoverMe terminates when it exhausts its iterations or achieves full coverage, whereas Random testing and AFL do not terminate by themselves. In our experiment, we first run CoverMe until it terminates using the parameters provided above. Then, we run Rand and AFL with ten times of the CoverMe time.

Austin terminates when it decides that no more coverage can be attained. It reports no coverage results until it terminates its calculation. Thus, it is not reasonable to set the same amount of time for Austin as AFL and Rand. The time for Austin refers to the time Austin spends when it stops running.

6.2 Quantitative Results

6.2.1 CoverMe versus Random testing

As a sanity check, we first compare CoverMe with Rand. In Tab. 2, we sort all programs (Col. 1) and functions (Col. 2) by their names and give the numbers of branches (Col. 3).

Tab. 2, Col. 6 gives the time spent by CoverMe. The time refers to the wall time reported by the Linux command `time`. Observe that the time varies considerably, ranging from 0.1 second (`s_erf.c`, `erfc`) to 22.1 seconds (`e_fmod.c`). Besides, the time is not necessarily correlated with the number of branches. For example, CoverMe takes 1.1 seconds to run `s_expm1.c` (42 branches) and 10.1 seconds to run `s_floor.c` (30 branches). It shows the potential for real-world program testing since CoverMe may not be very sensitive to the number of lines or branches.

Tab. 2, Col. 4 gives the time spent by Rand. Since Rand does not terminate by itself, the time refers to the timeout bound. As mentioned above, we set the bound as ten times of the CoverMe time.

Tab. 2, Col. 7 and 9 show the branch coverage results of Rand and CoverMe respectively. The coverage is reported by the Gnu coverage tool Gcov [7]. CoverMe achieves 100% coverage for 11 out of 40 tested functions with an average of 90.8% coverage, while Rand does not achieve any 100% coverage and attains only 38.0% coverage on average. The last row of the table shows the mean values. Observe that all values in Col. 9 are larger than the corresponding values in Col. 7. It means that CoverMe achieves higher branch coverage than Rand for every benchmark program. The result validates our sanity check.

Col. 10 is the improvement of CoverMe versus Rand. We calculate the coverage improvement as the difference between their percentages. CoverMe provides 52.9% coverage improvement on average.

Remark 6.1. Tab. 2 shows that CoverMe achieves partial coverage for some tested programs. The incompleteness occurs in two situations: (1) The program under test has unreachable branches; (2) The representing function fails to confirm $\texttt{FOO\_R} = 0$ when it in fact holds (Thm. 4.3). The latter can be caused by a weak optimization backend, which produces sub-optimal minimum points, or by floating-point inaccuracy when evaluating `FOO_R`. Sect. D of the paper's extended version [5] illustrates these two situations.

6.2.2 CoverMe versus AFL

Tab. 2 also gives the experimental results of AFL. Col. 5 corresponds to the "run time" statistics provided by the

Table 2: CoverMe versus Rand and AFL. The benchmark programs are taken from Fdlibm [6]. The coverage percentage is reported by Gcov [7]. The time for CoverMe refers to the wall time. The times for Rand and AFL are set to be ten times of the CoverMe time.

Benchmark			Time (s)			Branch (%)			Improvement (%)	
File	Function	#Branches	Rand	AFL	CoverMe	Rand	AFL	CoverMe	CoverMe vs. Rand	CoverMe vs. AFL
e_acos.c	ieee754_acos(double)	12	78	78	7.8	16.7	100.0	100.0	83.3	0.0
e_acosh.c	ieee754_acosh(double)	10	23	23	2.3	40.0	100.0	90.0	50.0	-10.0
e_asin.c	ieee754_asin(double)	14	80	80	8.0	14.3	85.7	92.9	78.6	7.1
e_atan2.c	ieee754_atan2(double,double)	44	174	174	17.4	34.1	86.4	63.6	29.6	-22.7
e_atanh.c	ieee754_atanh(double)	12	81	81	8.1	8.8	75.0	91.7	82.8	16.7
e_cosh.c	ieee754_cosh(double)	16	82	82	8.2	37.5	81.3	93.8	56.3	12.5
e_exp.c	ieee754_exp(double)	24	84	84	8.4	20.8	83.3	96.7	75.8	13.3
e_fmod.c	ieee754_fmod(double,double)	60	221	221	22.1	48.3	53.3	70.0	21.7	16.7
e_hypot.c	ieee754_hypot(double,double)	22	156	156	15.6	40.9	54.5	90.9	50.0	36.4
e_j0.c	ieee754_j0(double)	18	90	90	9.0	33.3	88.9	94.4	61.1	5.6
	ieee754_y0(double)	16	7	7	0.7	56.3	75.0	100.0	43.8	25.0
e_j1.c	ieee754_j1(double)	16	102	102	10.2	50.0	75.0	93.8	43.8	18.8
	ieee754_y1(double)	16	7	7	0.7	56.3	75.0	100.0	43.8	25.0
e_log.c	ieee754_log(double)	22	34	34	3.4	59.1	72.7	90.9	31.8	18.2
e_log10.c	ieee754_log10(double)	8	11	11	1.1	62.5	75.0	87.5	25.0	12.5
e_pow.c	ieee754_pow(double,double)	114	188	188	18.8	15.8	88.6	81.6	65.8	-7.0
e_rem_pio2.c	ieee754_rem_pio2(double,double*)	30	11	11	1.1	33.3	86.7	93.3	60.0	6.7
e_remainder.c	ieee754_remainder(double,double)	22	22	22	2.2	45.5	50.0	100.0	54.6	50.0
e_scalb.c	ieee754_scalb(double,double)	14	85	85	8.5	50.0	42.9	92.9	42.9	50.0
e_sinh.c	ieee754_sinh(double)	20	6	6	0.6	35.0	70.0	95.0	60.0	25.0
e_sqrt.c	iddd754_sqrt(double)	46	156	156	15.6	69.6	71.7	82.6	13.0	10.9
k_cos.c	kernel_cos(double,double)	8	154	154	15.4	37.5	87.5	87.5	50.0	0.0
s_asinh.c	asinh(double)	12	84	84	8.4	41.7	83.3	91.7	50.0	8.3
s_atan.c	atan(double)	26	85	85	8.5	19.2	15.4	88.5	69.2	73.1
s_cbrt.c	cbrt(double)	6	4	4	0.4	50.0	66.7	83.3	33.3	16.7
s_ceil.c	ceil(double)	30	88	88	8.8	10.0	83.3	83.3	73.3	0.0
s_cos.c	cos (double)	8	4	4	0.4	75.0	87.5	100.0	25.0	12.5
s_erf.c	erf(double)	20	90	90	9.0	30.0	85.0	100.0	70.0	15.0
	erfc(double)	24	1	1	0.1	25.0	79.2	100.0	75.0	20.8
s_expm1.c	expm1(double)	42	11	11	1.1	21.4	85.7	97.6	76.2	11.9
s_floor.c	floor(double)	30	101	101	10.1	10.0	83.3	83.3	73.3	0.0
s_ilogb.c	ilogb(double)	12	83	83	8.3	16.7	16.7	75.0	58.3	58.3
s_log1p.c	log1p(double)	36	99	99	9.9	38.9	77.8	88.9	50.0	11.1
s_logb.c	logb(double)	6	3	3	0.3	50.0	16.7	83.3	33.3	66.7
s_modf.c	modf(double, double*)	10	35	35	3.5	33.3	80.0	100.0	66.7	20.0
s_nextafter.c	nextafter(double,double)	44	175	175	17.5	59.1	65.9	79.6	20.5	13.6
s_rint.c	rint(double)	20	30	30	3.0	15.0	75.0	90.0	75.0	15.0
s_sin.c	sin (double)	8	3	3	0.3	75.0	87.5	100.0	25.0	12.5
s_tan.c	tan(double)	4	3	3	0.3	50.0	75.0	100.0	50.0	25.0
s_tanh.c	tanh(double)	12	7	7	0.7	33.3	75.0	100.0	66.7	25.0
MEAN		23	69	69	6.9	38.0	72.9	90.8	52.9	17.9

frontend of AFL. Col. 8 shows the branch coverage of AFL. As mentioned in Sect. 6.1, we terminate AFL once it spends ten times of the CoverMe time. Sect. B of [5] gives additional details on our AFL settings.

Our results show that AFL achieves 100% coverage for 2 out of 40 tested functions with an average of 72.9% coverage, while CoverMe achieves 100% coverage for 11 out of 40 tested functions with an average of 90.8% coverage. The average improvement is 17.9% (shown in the last row).

The last column is the improvement of CoverMe versus AFL. We calculate the coverage improvement as the difference between their percentages. Observe that CoverMe outperforms AFL for 33 out of 40 tested functions. The largest improvement is 73.1% with program `s_atan.c`. For five of tested functions, CoverMe achieves the same coverage as AFL. There are three tested functions where CoverMe achieves less (`e_acosh.c`, `e_atan2.c`, and `e_pow.c`).

Remark 6.2. We have further studied CoverMe's coverage on these three programs versus that of AFL. We have run AFL with the same amount of time as CoverMe (rather than ten times as much as CoverMe). With this setting, AFL does not achieves 70.0% for `e_acosh.c`, 63.6% for `e_atan2.c`, and 54.4% for `e_pow.c`, which are less than or equal to CoverMe's coverage.

That being said, comparing CoverMe and AFL by running them using the same amount of time may be unfair because AFL usually requires much time to obtain good code coverage — the reason why we have set AFL to run ten times as much time as CoverMe for the results reported in Tab. 2.

6.2.3 CoverMe versus Austin

Tab. 3 compares the results of CoverMe and Austin. We use the same set of benchmarks as Tab. 2 (Col. 1-2). We use the time (Col. 3-4) and the branch coverage metric (Col. 5-6) to evaluate the efficiency and the coverage. The branch coverage of Austin (Col. 5) is provided by Austin itself rather than by Gcov. Gcov needs to have access to the generated test inputs to report the coverage, but Austin does not provide a viable way to access the generated test inputs.

Austin shows large performance variances over different benchmarks, from 667.1 seconds (`s_sin.c`) to hours. As shown in the last row of Tab. 3, Austin needs 6058.4 seconds on average for the testing. The average time does not include the benchmarks where Austin crashes[2] or times out. Compared with Austin, CoverMe is faster (Tab. 3, Col. 4) with 6.9 seconds on average.

CoverMe achieves a higher branch coverage (90.8%) than Austin (42.8%). We also compare across Tab. 3 and Tab. 2. On average, Austin provides slightly higher branch coverage (42.8%) than Rand (38.0%), but lower than AFL (72.9%).

[2]Austin raised an exception when testing `e_sqrt.c`. The exception was triggered by `AustinOcaml/symbolic/symbolic.ml` from Austin's code, at line 209, Column 1.

Col. 7-8 are the improvement metrics of CoverMe against Austin. We calculate the Speedup (Col. 7) as the ratio of the time spent by Austin and the time spent by CoverMe, and the coverage improvement (Col. 7) as the difference between the branch coverage of CoverMe and that of Austin. We observe that CoverMe provides 3,868X speed-up and 48.9% coverage improvement on average.

Remark 6.3. Three reasons contribute to CoverMe's effectiveness. First, Thm. 4.3 (namely, each minimum point found in the minimization process corresponds to a new branch until all branches are saturated) allows CoverMe's search process to target the right test inputs only. Most random techniques do not have such guarantee, so they can waste time searching for irrelevant test inputs. Second, SMT-based methods run into difficulties in certain kinds of programs, *e.g.*, those with nonlinear arithmetic. It makes sense to use unconstrained programming in programs that are heavy on floating-point computation, in part because we have designed the representing function of CoverMe to be smooth to some degree, *e.g.*, the branch distances defined in Def. 4.1 are quadratic expressions; the smoothness allows CoverMe to leverage the power of local optimization and MCMC. Third, CoverMe only has to minimize a single representing function, whereas symbolic execution usually needs to solve a large number of path conditions.

Since CoverMe has achieved high code coverage on most tested programs, one may wonder whether our generated inputs have triggered any latent bugs. Note that when no specifications are given, program crashes have frequently been used as an oracle for finding bugs in integer programs. Floating-point programs, on the other hand, can silently produce wrong results without crashing. Thus, when testing floating-point programs, program crashes cannot be used as a simple, readily available oracle as for integer programs. Our experiments, therefore, have focused on assessing the effectiveness of CoverMe in solving the problem defined in Def. 3.1 and do not evaluate its effectiveness in finding bugs, which is orthogonal and exciting future work.

7. Related Work

Many survey papers [31, 40, 43] have reviewed the algorithms and implementations for coverage-based testing.

Random Testing The most generic automated testing solution may be to sample from the input space randomly. Pure random testing is usually ineffective if the input space is large, such as in the case of testing floating-point programs. AFL [1] is an improved random testing approach that incorporates a set of heuristics. It starts from a random seed and repeatedly mutates it to attain more program coverage.

Symbolic Execution Most branch coverage based testing algorithms follow the pattern of symbolic execution [24]. It selects a target path τ, derives a path condition Φ_τ, and calculates a model of the path condition with an SMT solver. The

Table 3: CoverMe versus Austin. The benchmark programs are taken from the Fdlibm library [6]. The "timeout" refers to a time of more than 30000 seconds. The "crash" refers to a fatal exception when running Austin.

Benchmark		Time (second)		Branch coverage(%)		Improvement	
Program	Entry function	Austin	CoverMe	Austin	CoverMe	Speedup	Coverage (%)
e_acos.c	ieee754_acos(double)	6058.8	7.8	16.7	100.0	776.4	83.3
e_acosh.c	ieee754_acosh(double)	2016.4	2.3	40.0	90.0	887.5	50.0
e_asin.c	ieee754_asin(double)	6935.6	8.0	14.3	92.9	867.0	78.6
e_atan2.c	ieee754_atan2(double, double)	14456.0	17.4	34.1	63.6	831.2	29.6
e_atanh.c	ieee754_atanh(double)	4033.8	8.1	8.3	91.7	495.4	83.3
e_cosh.c	ieee754_cosh(double)	27334.5	8.2	37.5	93.8	3327.7	56.3
e_exp.c	ieee754_exp(double)	2952.1	8.4	75.0	96.7	349.7	21.7
e_fmod.c	ieee754_frmod(double, double)	timeout	22.1	n/a	70.0	n/a	n/a
e_hypot.c	ieee754_hypot(double, double)	5456.8	15.6	36.4	90.9	350.9	54.6
e_j0.c	ieee754_j0(double)	6973.0	9.0	33.3	94.4	776.5	61.1
	ieee754_y0(double)	5838.3	0.7	56.3	100.0	8243.5	43.8
e_j1.c	ieee754_j1(double)	4131.6	10.2	50.0	93.8	403.9	43.8
	ieee754_y1(double)	5701.7	0.7	56.3	100.0	8411.0	43.8
e_log.c	ieee754_log(double)	5109.0	3.4	59.1	90.9	1481.9	31.8
e_log10.c	ieee754_log10(double)	1175.5	1.1	62.5	87.5	1061.3	25.0
e_pow.c	ieee754_pow(double, double)	timeout	18.8	n/a	81.6	n/a	n/a
e_rem_pio2.c	ieee754_rem_pio2(double, double*)	timeout	1.1	n/a	93.3	n/a	n/a
e_remainder.c	ieee754_remainder(double, double)	4629.0	2.2	45.5	100.0	2146.5	54.6
e_scalb.c	ieee754_scalb(double, double)	1989.8	8.5	57.1	92.9	233.8	35.7
e_sinh.c	ieee754_sinh(double)	5534.8	0.6	35.0	95.0	9695.9	60.0
e_sqrt.c	iddd754_sqrt(double)	crash	15.6	n/a	82.6	n/a	n/a
k_cos.c	kernel_cos(double, double)	1885.1	15.4	37.5	87.5	122.6	50.0
s_asinh.c	asinh(double)	2439.1	8.4	41.7	91.7	290.8	50.0
s_atan.c	atan(double)	7584.7	8.5	26.9	88.5	890.6	61.6
s_cbrt.c	cbrt(double)	3583.4	0.4	50.0	83.3	9109.4	33.3
s_ceil.c	ceil(double)	7166.3	8.8	36.7	83.3	812.3	46.7
s_cos.c	cos (double)	669.4	0.4	75.0	100.0	1601.6	25.0
s_erf.c	erf(double)	28419.8	9.0	30.0	100.0	3166.8	70.0
	erfc(double)	6611.8	0.1	25.0	100.0	62020.9	75.0
s_expm1.c	expm1(double)	timeout	1.1	n/a	97.6	n/a	n/a
s_floor.c	floor(double)	7620.6	10.1	36.7	83.3	757.8	46.7
s_ilogb.c	ilogb(double)	3654.7	8.3	16.7	75.0	438.7	58.3
s_log1p.c	log1p(double)	11913.7	9.9	61.1	88.9	1205.7	27.8
s_logb.c	logb(double)	1064.4	0.3	50.0	83.3	3131.8	33.3
s_modf.c	modf(double, double*)	1795.1	3.5	50.0	100.0	507.0	50.0
s_nextafter.c	nextafter(double, double)	7777.3	17.5	50.0	79.6	445.4	29.6
s_rint.c	rint(double)	5355.8	3.0	35.0	90.0	1808.3	55.0
s_sin.c	sin (double)	667.1	0.3	75.0	100.0	1951.4	25.0
s_tan.c	tan(double)	704.2	0.3	50.0	100.0	2701.9	50.0
s_tanh.c	tanh(double)	2805.5	0.7	33.3	100.0	4075.0	66.7
MEAN		6058.4	6.9	42.8	90.8	3868.0	48.9

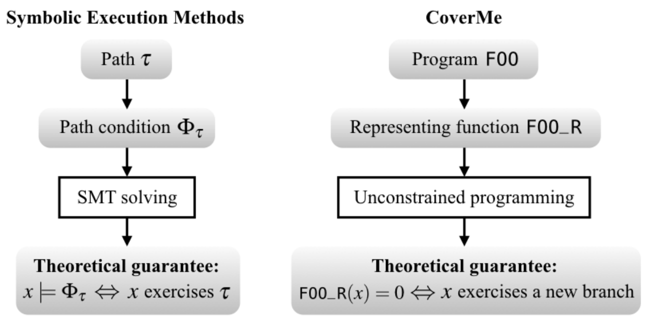

Symbolic Execution Methods
Path τ
Path condition Φ_τ
SMT solving
Theoretical guarantee:
$x \models \Phi_\tau \Leftrightarrow x$ exercises τ
CoverMe
Program FOO
Representing function FOO_R
Unconstrained programming
Theoretical guarantee:
FOO_R$(x) = 0 \Leftrightarrow x$ exercises a new branch

Figure 6: Symbolic execution methods versus CoverMe.

symbolic execution approach is attractive because of its theoretical guarantee, that is, each model of the path condition Φ_τ necessarily exercises its associated target path τ. In practice, however, symbolic execution can be ineffective if there are too many target paths (a.k.a. path explosion), or if the SMT backend has difficulties in handling the path condition. When analyzing floating-point programs, symbolic execution and its DSE (Dynamic Symbolic Execution) variants [15, 41] typically reduce floating-point SMT solving to Boolean satisfiability solving [36], or approximate the constraints over floats by those over rationals [28] or reals [13].

Fig. 6 illustrates a comparison between symbolic execution methods and CoverMe. While symbolic execution solves a path condition Φ_τ with an SMT-based backend for each target path τ, CoverMe minimizes a *single* representing function FOO_R with execution-based unconstrained programming. Therefore, CoverMe does not have path explosion issues, and in addition, it does not need to analyze program semantics. Similar to symbolic execution, CoverMe also comes with a theoretical guarantee, namely, each minimum attaining 0 necessarily trigger a new branch, a guarantee that contributes to its effectiveness (Sect. 6).

Search-based Testing Miller *et al.* [32] reduce the problem of testing straight-line floating-point programs into *constrained programming* [44], that is, optimization problems in which one needs to both minimize an objective function and satisfy a set of constraints. Korel [21, 25] extends Miller *et al.*'s to general programs, leading to the development of search-based testing [31]. It views a testing problem as a sequence of subgoals where each subgoal is associated with a fitness function; the search process is then guided by minimizing those fitness functions.

CoverMe's representing function is similar to the fitness function of search-based testing in the sense that both reduce testing into function minimization. However, CoverMe uses the more efficient unconstrained programming rather than the constrained programming used in search-based testing. Moreover, CoverMe's use of representing function comes up with a theoretical guarantee, which also opens the door to a whole suite of optimization backends.

8. Conclusion

We have introduced a new branch coverage based testing algorithm for floating-point code. We turn the challenge of testing floating-point programs into the opportunity of applying unconstrained programming. Our core insight is to introduce the representing function so that Thm. 4.3 holds, which guarantees that the minimum point of the representing function is an input that exercises a new branch of the tested program. We have implemented this approach into the tool CoverMe. Our evaluation on Sun's math library Fdlibm shows that CoverMe is highly effective, achieving 90.8% of branch coverage in 6.9 seconds on average, which largely outperforms random testing, AFL, and Austin.

For future work, we plan to investigate the potential synergies between CoverMe and symbolic execution, and extend this work to programs beyond floating-point code.

Acknowledgments

This research was supported in part by the United States National Science Foundation (NSF) Grants 1319187, 1528133, and 1618158, and by a Google Faculty Research Award. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred. We would like to thank our shepherd, Sasa Misailovic, and the anonymous reviewers for valuable feedback on earlier drafts of this paper, which helped improve its presentation.

References

[1] AFL: American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. Retrieved: 01 Novermber, 2016.

[2] Clang: a C language family frontend for LLVM. http://clang.llvm.org/. Retrieved: 24 March 2016.

[3] Class StrictMath of Java SE 8. https://docs.oracle.com/javase/8/docs/api/java/lang/StrictMath.html. Retrieved: 09 Novermber, 2016.

[4] Code coverage analysis tool for AFL. https://github.com/mrash/afl-cov. Retrieved: 01 Novermber, 2016.

[5] An extended version of this paper. https://arxiv.org/pdf/1704.03394.

[6] Fdlibm: Freely Distributed Math Library. http://www.netlib.org/fdlibm/. Retrieved: 01 Nov, 2016.

[7] Gcov: GNU compiler collection tool. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html/. Retrieved: 24 March 2016.

[8] klee-dev mailing list. http://www.mail-archive.com/klee-dev@imperial.ac.uk/msg02334.html. Retrieved: 09 Novermber, 2016.

[9] llvm::pass class reference. http://llvm.org/docs/doxygen/html/classllvm_1_1Pass.html. Retrieved: 24 March 2016.

[10] Scipy optimization package. http://docs.scipy.org/doc/scipy-dev/reference/optimize.html. Retrieved: 24 March 2016.

[11] Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and Michael I. Jordan. An introduction to MCMC for machine learning. *Machine Learning*, 50(1-2):5–43, 2003.

[12] Arthur Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, and Tanja Vos. Symbolic search-based testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 53–62, Washington, DC, USA, 2011.

[13] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. Automatic detection of floating-point exceptions. In *POPL*, pages 549–560, 2013.

[14] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Syst. J.*, 22(3):229–245, 1983.

[15] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 443–446, 2008.

[16] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008.

[17] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[18] Peter Collingbourne, Cristian Cadar, and Paul HJ Kelly. Symbolic crosschecking of floating-point and simd code. In *Proceedings of the sixth conference on Computer systems*, pages 315–328, 2011.

[19] John E Dennis Jr and Robert B Schnabel. *Numerical methods for unconstrained optimization and nonlinear equations*, volume 16. 1996.

[20] Jonathan Eckstein and Dimitri P Bertsekas. On the Douglas—Rachford splitting method and the proximal point algorithm for maximal monotone operators. *Mathematical Programming*, 55(1-3):293–318, 1992.

[21] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.

[22] Vijay Ganesh and David L Dill. A decision procedure for bit-vectors and arrays. In *International Conference on Computer Aided Verification*, pages 519–531. Springer, 2007.

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

[24] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[25] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870–879, 1990.

[26] Kiran Lakhotia, Mark Harman, and Hamilton Gross. Austin: An open source tool for search based software testing of C programs. *Information and Software Technology*, 55(1):112–125, 2013.

[27] Kiran Lakhotia, Phil McMinn, and Mark Harman. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *Journal of Systems and Software*, 83(12):2379–2391, 2010.

[28] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan De Halleux. FloPSy: Search-based floating point constraint solving for symbolic execution. ICTSS'10, pages 142–157, Berlin, Heidelberg, 2010.

[29] DM Leitner, C Chakravarty, RJ Hinde, and DJ Wales. Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms. *Phys. Rev. E*, 56:363, 1997.

[30] Z. Li and H. A. Scheraga. 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):6611–6615, 1987.

[31] Phil McMinn. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, 2004.

[32] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Trans. Softw. Eng.*, 2(3):223–226, 1976.

[33] Glenford J. Myers. The art of software testing (2nd ed.). pages I–XV, 1–234, 2004.

[34] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. 2006.

[35] Hristina Palikareva and Cristian Cadar. Multi-solver support in symbolic execution. In *Computer Aided Verification*, pages 53–68. Springer, 2013.

[36] Jan Peleska, Elena Vorobev, and Florian Lapschies. Automated test case generation with smt-solving and abstract interpretation. NFM'11, pages 298–312, Berlin, Heidelberg, 2011.

[37] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. New York, NY, USA, 2007.

[38] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[39] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005.

[40] Chayanika Sharma, Sangeeta Sabharwal, and Ritu Sibal. A survey on software testing techniques using genetic algorithm. *arXiv:1411.1154*, 2014.

[41] Dawn Song, D Brumley, H Yin, J Caballero, I Jager, MG Kang, Z Liang, J Newsome, P Poosankam, and P Saxena. Bitblaze: Binary analysis for computer security, 2013.

[42] Nikolai Tillmann and Jonathan De Halleux. Pex: White box test generation for .net. TAP'08, pages 134–153, Berlin, Heidelberg, 2008.

[43] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[44] G. Zoutendijk. *Mathematical programming methods*. North-Holland, Amsterdam, 1976.