CS701 Project Report: Forward-mode Automatic Differentiation for Angora

Yuhao Zhang [1] **Rui Huang** [1]

Abstract

The state-of-the-art fuzzer Angora leverages gradients descent to solve the non-linear numerical path constraints which are expensive for symbolic execution to solve. However, Angora computes partial derivatives by divided difference which is imprecise and inefficient. We implement forward-mode automatic differentiation in place of the divided difference inside Angora to get the precise partial derivatives more efficiently. The results on 10 toy programs show that automatic differentiation can help Angora solve some constraints in less time.

1. Introduction

Fuzzing testing (Takanen et al., 2008) is a technique that generates random test inputs to the target program in order to trigger errors such as crashes, built-in assertion failures, and memory leaks in the program. Branch coverage is the evaluation metric for the test inputs generated by fuzzing testing tools. More branch coverage usually means higher possibility to find potential errors in the program. To increase the branch coverage, the state-of-the-art fuzzer Angora (Chen & Chen, 2018) adopts principled search without symbolic execution. One of the key techniques proposed by Angora is gradient descent, which is used to solve the non-linear numerical path constraints which are expensive for symbolic execution to solve.

Angora computes partial derivatives used in gradient descent by divided difference:

$$\frac{f(\mathbf{x}+\Delta_{x_i}\mathbf{v_i})-f(\mathbf{x})}{\Delta_{x_i}},$$

where $\mathbf{v_i}$ is the unit vector in the ith dimension. This approximation can get the precise partial derivatives when Δ_{x_i} approaches zero. However, Angora cannot set Δ_{x_i} to floating-point values that are closer to zero, because it cannot change the integer variables $f, x_1, \ldots, x_n$ to the floating-point type. Thus, Angora computes the partial derivatives by setting $\Delta_{x_i} = \pm 1$, which leads to bad approximation of the partial derivatives.

Angora requires to compute two divided differences for each set of input $x_1, \ldots, x_n$:

$$f(\mathbf{x}+\mathbf{v_i})-f(\mathbf{x})$$

and

$$f(\mathbf{x})-f(\mathbf{x}-\mathbf{v_i}).$$

So Angora needs to execute the target program three times to get the values of f under three sets of input: $\mathbf{x}+\mathbf{v_i}$, $\mathbf{x}$, and $\mathbf{x}-\mathbf{v_i}$. These triple executions create a large overhead and slow down the fuzzing testing.

In this paper, we improve Angora by implementing forward-mode automatic differentiation (AD) in place of divided difference used in gradient descent. With the assistance of AD, Angora can get the precise partial derivatives by only executing the target program once.

Our forward-mode AD involves three steps: First, we create custom classes for AD and replace the original data types in the target programs. Second, we override all the arithmetic operations in these classes. These operations will not only carry the original arithmetic semantics but also calculate partial derivatives. Third, we create other function *get_dx* to retrieve the partial derivatives from the class. Whenever the partial derivatives are needed, we will insert a proxy call to this function.

We evaluate the AD-assist Angora on 10 toy programs. These programs are manually created by two authors on the bias of covering diverse program structures, e.g., for loops, function calls, and nested branches. Experimental results show that our approach can perform at least as well as Angora in most test cases. We also have some test cases where our approach significantly outperforms Angora, and one test case where our approach performs much worse. We will analyze these results in detail in Section 5.

Two main contributions of this paper are an implementation of AD-assisted Angora fuzzer and an evaluation on 10 toy programs.

*Equal contribution [1]Department of Computer Science, University of Wisconsin-Madison, Madison, USA. Correspondence to: Yuhao Zhang <yuhaoz@cs.wisc.edu>.

Proceedings of the 36^{th} International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019. Copyright 2019 by the author(s).

$\varphi(\mathbf{x})$	$f(\mathbf{x})$	$\psi(\mathbf{x})$
$a < b$	$a - b$	$f(\mathbf{x}) < 0$
$a \leq b$	$a - b$	$f(\mathbf{x}) \leq 0$
$a > b$	$b - a$	$f(\mathbf{x}) < 0$
$a \geq b$	$b - a$	$f(\mathbf{x}) \leq 0$
$a = b$	$\|a - b\|$	$f(\mathbf{x}) \leq 0$
$a \neq b$	$-\|a - b\|$	$f(\mathbf{x}) < 0$

Table 1. The conversion from each type of comparison to its numerical function and its constraint.

$\frac{\partial f}{\partial x_i}^+$	$\frac{\partial f}{\partial x_i}^-$	$\lvert\frac{\partial f}{\partial x_i}^+\rvert < \lvert\frac{\partial f}{\partial x_i}^-\rvert$	partial derivative
≥ 0	≤ 0	-	0
≥ 0	> 0	-	$\frac{\partial f}{\partial x_i}^+$
< 0	≤ 0	-	$\frac{\partial f}{\partial x_i}^-$
< 0	> 0	true	$\frac{\partial f}{\partial x_i}^-$
< 0	> 0	false	$\frac{\partial f}{\partial x_i}^+$

Table 2. The discussion of different combinations of $\frac{\partial f}{\partial x_i}^+$ and $\frac{\partial f}{\partial x_i}^-$. - means no constraint.

The rest of the paper is organized as follows: In section 2 and 3, we provide a background of Angora and automatic differentiation. In section 4, we present our approach for automatic differentiation and how we incorporate it into Angora. In section 5, we show the experiment and results.

2. Background for Angora

In this section, we provide a background for gradient descent technique in Angora fuzzer and discuss the drawbacks of the divided difference for calculating the partial derivatives.

2.1. Gradient Descent in Angora

To increase the branch coverage, the Angora adopts principled search without symbolic execution. One of the key techniques proposed by Angora is gradient descent, which is used to solve the non-linear numerical path constraints which are expensive for symbolic execution to solve.

Angora applies gradient descent to all integer comparisons $\varphi(\mathbf{x})$ shown in the first column in Table 1, where $\mathbf{x} = x_1, \ldots, x_n$ is a set of inputs. Each type of comparison is converted to a numerical function $f(\mathbf{x})$ and a constraint ψ on $f(\mathbf{x})$ such that $\varphi(\mathbf{x})$ holds iff $\psi(\mathbf{x})$ holds. There are two type of ψ: $f(x) < 0$ and $f(x) \leq 0$. As such, to satisfy $\varphi(\mathbf{x})$ is equivalent to minimize $f(\mathbf{x})$ to satisfy corresponding ψ. Such optimization problem can be solved by gradient descent effectively. Table 1 summarizes the conversion from each type of comparison to its numerical function and its constraint. If the predicate of a conditional statement contains logical operators *and* or *or*, Angora splits the statement into multiple conditional statements and tries to solve them separately.

2.2. Partial Derivatives Calculation in Angora

Calculating the partial derivatives with regard to each input x_i is the essential step of gradient descent. Angora computes partial derivatives by divided difference:

$$\frac{f(\mathbf{x} + \Delta_{x_i}\mathbf{v_i}) - f(\mathbf{x})}{\Delta_{x_i}},$$

where $\mathbf{v_i}$ is the unit vector in the ith dimension. Angora first chooses the value of Δ_{x_i} to be ± 1 and then executes the target program three times to get $f(\mathbf{x} + \mathbf{v_i})$, $f(\mathbf{x})$, and $f(\mathbf{x} - \mathbf{v_i})$. These three values enable Angora to compute partial derivatives in two directions:

$$\frac{\partial f}{\partial x_i}^+ = f(\mathbf{x} + \mathbf{v_i}) - f(\mathbf{x})$$

and

$$\frac{\partial f}{\partial x_i}^- = f(\mathbf{x}) - f(\mathbf{x} - \mathbf{v_i}).$$

The calculation of the partial derivative involves discussion (shown in Table 2) on $\frac{\partial f}{\partial x_i}^+$ and $\frac{\partial f}{\partial x_i}^-$. It is possible that in the second execution, the program fails to reach the program point where $f(\mathbf{x} \pm \mathbf{v_i})$ is calculated because the program takes a different branch at an earlier conditional statement. When this happens, Angora set the partial derivative in that direction to zero, instructing the gradient not to move $\mathbf{x}$ in that direction.

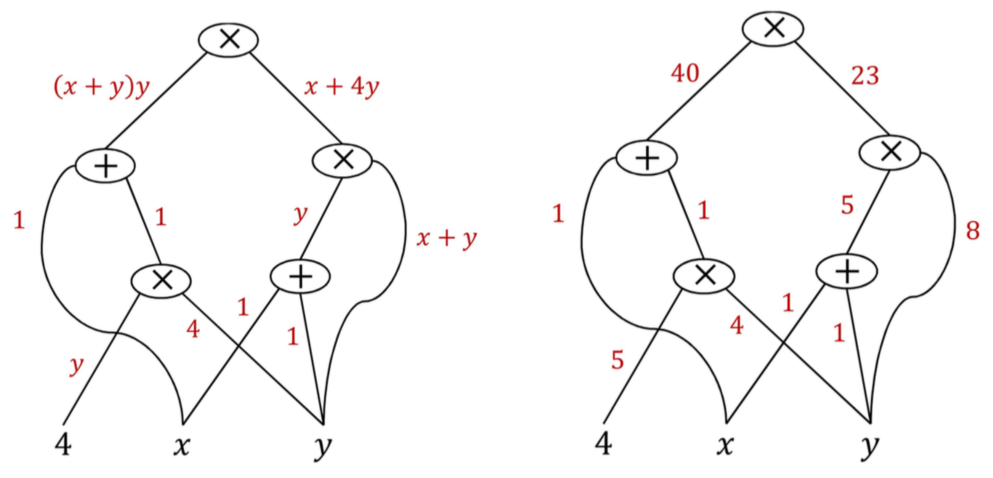
Divided difference can get the precise partial derivatives when Δ_{x_i} approaches zero. However, Angora cannot set Δ_{x_i} to other floating-point values that are closer to zero than ± 1, because it cannot change the integer variables $f, x_1, \ldots, x_n$ to the floating-point type. Setting Δ_{x_i} to ± 1 leads to bad approximation of the partial derivatives.

Angora needs to execute the target program three times to get the values of f under three sets of input: $\mathbf{x} + \mathbf{v_i}$, $\mathbf{x}$, and $\mathbf{x} - \mathbf{v_i}$. These triple executions create a large overhead and slow down the fuzzing testing.

3. Background for Automatic Differentiation

In this section, we give a background [1] for two types of automatic differentiation (AD): forward-mode AD (Reps & Rall, 2003) and reverse-mode AD (Griewank et al.; Iri, 1984;

[1]The background of AD is adapted from CS701 lecture notes `http://pages.cs.wisc.edu/~cs701-1/LectureNotes/trunk/cs701-lec-12-1-2015/cs701-lec-12-01-2015.pdf`.



(a) The symbolic computation graph of the function $h(x, y) = (x + 4y) * ((x + y) * y)$.

(b) The labeled computation graph of the function $h(x, y) = (x + 4y) * ((x + y) * y)$ when $x = 3$ and $y = 5$.

Figure 1. The symbolic and labeled computation graphs of $h(x, y) = (x + 4y) * ((x + y) * y)$.

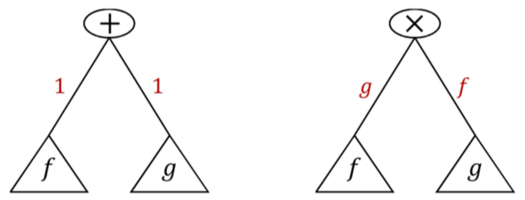


Figure 2. The edge label scheme.

Griewank, 1991; Speelpenning, 1980). Then we discuss why we choose forward-mode AD to calculate the partial derivatives.

3.1. Forward-mode Automatic Differentiation

Forward-mode automatic differentiation provides a way to compute the derivative of a function defined by a program. A forward-mode AD tool transforms a program that computes a numerical function $f(x)$ into a related program that not only computes $f(x)$ but also computes the derivative $f'(x)$. Forward-mode AD can be illustrated by the following code example:

```
1  int f(int x) {
2      int ans = 1;
3      for (int i = 1; i <= k; i++) {
4          ans = ans * g_i(x);
5      }
6      return ans;
7  }
8
9  int f_dx(int x) {
10     int ans = 1;
11     int ans_dx = 0;
12     for (int i = 1; i <= k; i++) {
13         ans_dx = ans_dx * g_i(x) + ans *
       g_i_dx(x);
14         ans = ans * g_i(x);
15     }
16     return ans_dx;
17 }
```

The first function $f(x)$ (lines 1-7) calculates and returns

$$\prod_{1 \leq i \leq k} g_i(x),$$

where $g_i(x)$ is a set of differentiable numerical functions. The second function $f_{dx}(x))$ (lines 9-17) resembles the calculation of $f(x)$. Besides, $f_{dx}(x))$ calculates and returns ans_{dx}, which can be proved as

$$\sum_{1 \leq i \leq k} g_{dx}(i) \prod_{j \neq i} g_j(x).$$

In the language that supports operator overloading, such as C++, AD can be carried out by defining a new data type that has field for both the values and the derivative, and overloading the arithmetic operators to calculate both fields (Rall, 1983; 1984). In Section 4, we will present using operator overloading to transform the program to support forward-mode AD.

3.2. Reverse-mode Automatic Differentiation

The reverse-mode automatic differentiation can be seen as solving a path program on computation graphs. A computation graph is a labeled directed-acyclic graph (DAG).

Figure 1(a) shows a computation graph of the following function:

$$h(x, y) = (x + 4y) * ((x + y) * y)$$

To formulate AD as a path problem, we need to define the semantic of two operators *combine* and *extend*, and the values that label the computation graph's edges. In AD, we define *combine* as *addition* and *extend* as *multiplication*. The label scheme is shown in Figure 2.

We present how to compute $\frac{\partial h(x,y)}{\partial x}$ on the computation graph when $x = 3$ and $y = 5$. First, we label the symbolic computation graph shown in Figure 1(a) to Figure 1(b). Then, we solve the path problem st rating from the root node $h(x, y) = (x + 4y) * ((x + y) * y)$ to the leaf node x:

$$\frac{\partial h(x,y)}{\partial x}\bigg|_{x=3,y=5} = 40 * 1 + 23 * 5 * 1 = 155.$$

3.3. Discussion on Forward- and Reverse-mode Automatic Differentiation

In Angora's gradient descent, we only care about the partial derivative of the numerical function $f(\mathbf{x})$ at current fuzzing branch. Thus, the partial derivatives are all single-target and multi-sources path problems. Theoretically, the reverse-mode AD will have better performance than forward-mode AD.

However, we choose to use forward-mode AD because it is more capable of carrying partial derivatives through loops, function calls, and nested branches.

4. Approach

In this section, we present our forward-mode automatic differentiation for single input. Thus, we refer partial derivatives as gradients to ease presentation. We implement gradient descent using forward-mode AD within Angora's framework in the following 3 steps:

- We implement an AD-assisted Int class in place of the original int data type in the program source code. This AD-assisted class will enable us to automatically calculate gradients when we run the compiled binaries.
- Inside the Angora Pass, for each conditional statement, we insert a proxy call to a member function (*get_dx*) in the Int class to retrieve the gradients of the arguments in the conditional statement.
- We send the gradients extracted from Angora Pass back to Angora and use these gradients to replace the original gradients approximated by Angora.

4.1. AD-assisted Int Class

In order to automatically calculate gradients of every conditional statement in a forward fashion when running the program, we propose to implement AD-assisted classes to replace the original data types for which we have gradients to compute. We override the arithmetic operations in these AD-assisted classes so that they will compute gradients alongside performing original operations.

In this work, we consider to implement AD for the int data type in the source program, because Angora only applies gradient descent to integer comparisons. The following code shows an overview of the AD-assisted Int Class we created:

```

class Int {
private:
    int val;
    long long dx;
public:
    Int (int);
    Int (int, long long);
    long long get_dx(int) const;
    operator int();
    Int& operator=(const int&);
    Int operator *(const Int&) const;
    Int operator *(const int&) const;
    friend Int operator *(const int&, const
     Int&);
    //omit functions for +, -, /
};

Int Int::operator *(const Int&b) const {
    return Int(val * b.val, b.dx * val + dx
     * b.val);
}

Int Int::operator *(const int&b) const {
    return Int(val * b, dx * b);
}

Int operator *(const int&a, const Int&b) {
    return b * a;
}

long long Int::get_dx(int arg_id) const {
    return dx;
}

//omit functions for +, -, /
```

The field *val* stores the original value of this integer and the field *dx* stores the forward-computed gradient with respect to this integer.

For the multiplication operation, we have 3 overloaded functions where we calculate not only the result of multiplication, but also the gradients with respect to the multiplication. An important member function in this class is *get_dx*, which is used to extract the computed gradients from this class. Therefore, whenever we want to obtain a gradient for some

```
1.  %39 = load i64, i64* %4, align 8, !tbaa.struct !11
2.  %40 = call i64 @_Z3foo3Int(i64 %39)
3.  store i64 %40, i64* %9, align 8
4.  %41 = call i32 @_ZN3IntcviEv(%class.Int* nonnull %10)
5.  %42 = icmp slt i32 %41, 12596
6.  br i1 %42, label %45, label %51
```

Figure 3. LLVM's IR before inserting the proxy call

```
1.  %39 = load i64, i64* %4, align 8, !tbaa.struct !11
2.  %40 = call i64 @_Z3foo3Int(i64 %39)
3.  store i64 %40, i64* %9, align 8
4.  %XX = call i32 @get_dx(%class.Int* nonnull %10)
5.  %41 = call i32 @_ZN3IntcviEv(%class.Int* nonnull %10)
6.  %42 = icmp slt i32 %41, 12596
7.  br i1 %42, label %45, label %51
```

Figure 4. LLVM's IR after inserting the proxy call

argument of conditional statements, we can simply insert a proxy call to this member function using this argument, as we will describe in the next subsection.

4.2. Retrieving Gradients inside a LLVM Pass

The previous subsection talks about how to compute gradients automatically in a forward fashion by overriding arithmetic operations. However, it is still non-trivial to retrieve the gradients computed in the Int class at run-time. In this subsection, we describe how we retrieve the gradients for some arguments in a specific conditional statement by inserting proxy calls in the LLVM pass.

Since we already have a *get_dx* function to return the computed gradients implemented in the Int class, the only question left is where to insert a proxy call to this function and what argument to use. Figure 3 shows a code segment of LLVM's intermediate representation with one conditional statement at line 5. This conditional statement makes a comparison between the integer in Reg41 and a constant integer 12596, while the integer stored in Reg41 is cast from an Int object at line 4. The mangled function colored at line 4 is a cast member function of Int class whose only argument Reg10 is a *this* pointer which points to the Int class itself.

As shown in Figure 4, we insert a proxy call to the *get_dx* function right before the cast function and take as input the argument of that cast function, i.e., the *this* pointer. Notice that there is no cast function call for the second arguments, i.e., constant integer 12596, which indicates that there is no gradient for this argument. Since Angora has already created a LLVM pass called Angora Pass, we implemented our proxy call insertion inside the Angora Pass for simplification.

4.3. Sending Gradients to Angora Fuzzer

After inserting a proxy call in the Angora Pass, we will be able to retrieve the gradients from the compiled binary. Then the final step is to send the gradients retrieved from the compiled binary to the Angora Fuzzer in the process of gradient descent. Note that, originally Angora also needs to interact with the compiled binary in order to receive the outputs and arguments of conditional statements given different inputs. Therefore, the sending and receiving of gradients can be easily integrated into Angora's original interactions with the executable.

As shown in Figure 5, Angora Fuzzer implemented a trace function (*__angora_trace_cmp_tt*) to receive information about conditional statements from the binary. This trace function has been registered into the binary via the Angora Pass. Therefore, for each conditional statement, we will insert a proxy call to this trace function in the Angora Pass and pass all the required information about this conditional statement as arguments, such as cmpid and the two operands.

We add two parameters in the trace function, i.e. the gradients w.r.t. the first operand and the second operand respectively, as shown in Figure 5. Then we can send the gradients retrieved from the previous step as the additional arguments of the proxy call in the Angora Pass.

After Angora Fuzzer receives the AD gradient from the compiled binary, we will be able to use this gradient to replace the original numerical approximation approach Angora used in the process of gradient descent.

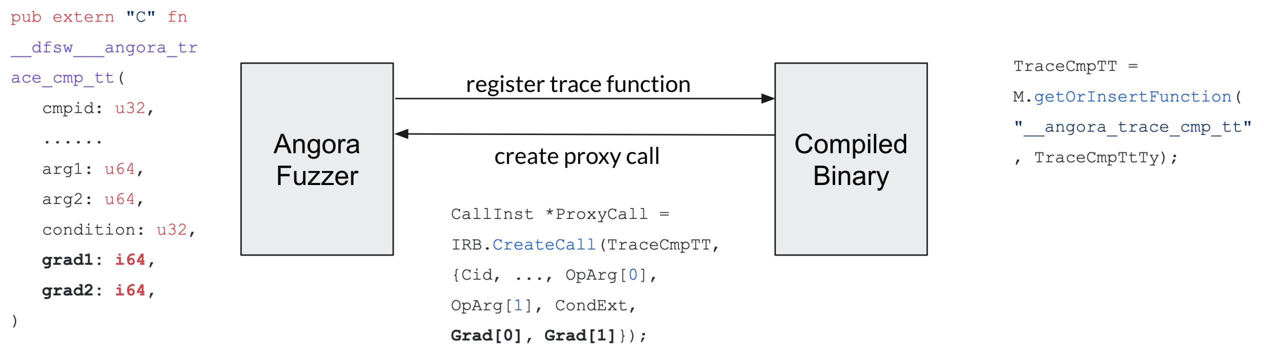
5. Experiment

5.1. Experiment Setups

We run all our experiments on a Google Compute Engine with 2 Intel Xeon CPUs running on a 64-bit Debian GNU/Linux 9.12 system. Our llvm/clang version is 7.0.0. We have modified GD_ESCAPE_RATIO from 1.0 in the original Angora to 0.5, because the original ratio could cause the process of gradient descent to be stuck in a local minimum and not able to escape from there. Because our approach currently still requires manual modification of source code to inject the Int class we created and to replace the original int data type, we carried out our experiments on 10 toy examples instead of large benchmark datasets. Our implementation and test cases are available online[2].

We describe what each test case looks like as Features in Table 3, including how many if-statements there are, how many function calls, whether there exists for loop, how many total paths in the program, and so on. The goal here

[2]https://github.com/ForeverZyh/Angora



```
pub extern "C" fn
__dfsw___angora_tr
ace_cmp_tt(
   cmpid: u32,
   ......
   arg1: u64,
   arg2: u64,
   condition: u32,
   grad1: i64,
   grad2: i64,
)
```

```
CallInst *ProxyCall =
IRB.CreateCall(TraceCmpTT,
{Cid, ..., OpArg[0],
OpArg[1], CondExt,
Grad[0], Grad[1]});
```

```
TraceCmpTT =
M.getOrInsertFunction(
"__angora_trace_cmp_tt"
, TraceCmpTtTy);
```

Figure 5. Interaction between Angora Fuzzer and compiled binaries

Test ID		0	1	2	3	4	5	6	7	8	9
Features	+	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	*	✓	✓	✓	✓	✓	✓	✓	✓	×	✓
	/	✓	✓	×	✓	✓	✓	✓	✓	✓	✓
	# of if-statements	3	1	1	1	2	2	1	1	1	1
	nested-if	✓	×	×	×	✓	✓	×	×	×	×
	# of conditions in each if-stats	2	2	2	2	2	2	2	2	2	4
	for-loop	×	×	×	×	×	✓	×	✓	✓	×
	# of functions	2	1	1	1	1	1	2	0	1	2
	# of arguments in each function	1	1	1	2	2	2	2	0	1	1
	# of total paths	4	2	2	2	3	3	2	2	2	4
Angora	success runs (out of 5)	5	5	5	3	4	5	5	5	5	5
	average steps (among success runs)	63	34	35	29k	17k	327.8	62.8	14	430.8	112.6
	standard deviation	0	0	0	30k	15k	38.1	1.6	0	591.0	8.7
Our Approach	success runs (out of 5)	5	5	5	3	5	5	2	5	5	5
	average steps (among success runs)	538.4	32	30	3.2k	32.2	1.5k	435	15	3.0k	153
	standard deviation	377.8	0	0	1.7k	1.6	1.1k	8.5	0	2.5k	84.8

Table 3. A summary of features and test results of 10 toy test cases. We first show different features included in these test cases, where +, -, *, / represent whether there is addition, subtraction, multiplication and division operation in the code respectively. Then we show the results from Angora and the results from our approach, respectively. We run each test case for 5 times and report how many times we succeed out of 5, the average steps among success runs and the standard deviation.

is to make these test cases as diverse as possible.

5.2. Experiment Results

We report the test results of Angora and the test results of our approach in Table 3. Because there is some randomness in the fuzzing process, for each test case on each approach, we run it for 5 times and report how many times we succeed in finding all the paths out of the 5 times, the average steps among success runs and the standard deviation. As we can see, in some test cases the fluctuation between different runs is very large (standard deviation is high), such as Test 3, Test 5 and Test 8.

On about half of the cases, Angora and our approach perform equivalently well, such as Test 1, Test 2, Test 7, Test 9. However, for some test cases, Angora will take significant steps to find all the paths or even fail, while our approach could solve the problem easily, such as Test 3 and Test 4. On the contrary, Angora performs much better than our approach on Test 6, while our approach fails 3 times out of 5 runs.

For the test cases we outperform Angora, we attribute it to that our AD-gradients are more precise than the numerical approximations used by Angora. Below is the source code of Test 3:

```
Int __attribute__ ((noinline)) foo(Int x,
    Int y) {
    return x * y + 123 * x * y * y - 321 *
    y;
}


int main (int argc, char** argv) {
    if (argc < 2) return 0;

    FILE *fp;
    size_t ret;

    fp = fopen(argv[1], "rb");

    if (!fp) {
          printf("st err\n");
            return 0;
    }

    int len = 1;
    int16_t _x = 0;
    ret = fread(&_x, sizeof _x, 1, fp);
    Int x = Int(_x, 1);
    x.get_dx(-1);
    fclose(fp);
    if (ret < len) {
          printf("input fail \n");
            return 0;
    }

    int d = x + 10;
    if (foo(x + 10, 100 / (x + 3)) < 125963
  && foo(x + 10, 100 / (x + 3)) > 120000)
    {
            abort();
    }

    return 0;
}
```

The computation of this conditional statement is complex and the constraint interval (120000 $\sim$ 125963) is narrow and hard to find. Angora's numerical approximation may not be precise when it involves complicated calculations, especially divisions. Meanwhile, the AD-gradients we computed in a forward fashion in the binary are precise enough for us to find the constraint interval in these cases.

For Test 6 where we perform worse than Angora, we do not have a conclusive explanation yet, but we do have two hypotheses. First, from the output log we could see the gradients computed from our approach are accurate, but when the input is descended towards the direction of gradients, this condition becomes not reachable any more. We think this could result from Angora's strategy for compound conditions. It will divide a compound condition into two simple conditions with a nested-if structure. The problem is that when the inner condition is satisfied, the outer condition may not be satisfied any more, and vice versa. Since our approach inherited Angora's framework, we cannot avoid this problem.

Second, Angora's gradient descent process and hyperparameters are specifically designed for its gradients computed from numerical approximation. These process and hyperparameters may not be suitable for our AD-gradients any more. Due to the time limitation, we haven't rebuilt our own framework for gradient descent yet. This could result in some unsatisfying performance.

6. Discussion, Limitation, and Future Work

First, forward automatic differentiation needs to transform all original data types to new data types. This transformation is currently performed manually. We regard this manually transformation as our limitation which also restricts our evaluation on only toy programs instead of large benchmarks. We plan to perform this transformation on the abstract syntax tree (AST) during parsing. Second, gradient descent requires partial derivatives with regard to all inputs. Currently, our implementation only supports calculating the gradient with regard to a single input. We regard this incomplete implementation as one of our limitations. We plan to extend the new data type to store a list of partial derivatives with regard to all inputs. Moreover, the list of partial derivatives are sent to Angora together with a list of taint labels, which instructs the gradient descent to choose the corresponding partial derivative of certain input. Third, if the predicate of a conditional statement contains logical operators *and* or *or*, Angora splits the statement into multiple conditional statements and tries to solve them separately. We regard this design choice as a limitation of Angora, because we find that the gradient descent technique in Angora will get stuck in the separated conditional statements. We plan to improve this design choice in Angora to see a conditional statement that contains logical operators *and* or *or* as a whole and solve the statement together instead of separating it. This improvement requires the numerical function $f(\mathbf{x})$ over logical operators *and* or *or* and its corresponding constraint ψ, which have already been studied in Fischer et al. (2019); Shapiro (2007).

7. Conclusion

In this paper, we built on top of Angora Fuzzer and aimed to replace the original numerical approximation for gradient descent by automatic differentiation. We implemented an AD-assisted Int class to automatically compute gradients in a forward fashion and fed these gradients to Angora Fuzzer in place of the numerically-approximated derivatives in the process of gradient descent. Experimental results show that our approach can perform at least as well as Angora in most test cases. We also have some test cases where our approach significantly outperforms Angora, and one test case where our approach performs much worse. We attributed the incompetency of our approach to the mismatch between our AD-computed gradients and the original Angora framework, from where we also pointed out potential directions for future work.

References

Chen, P. and Chen, H. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pp. 711–725, 2018. doi: 10.1109/SP.2018.00046. URL https://doi.org/10.1109/SP.2018.00046.

Fischer, M., Balunovic, M., Drachsler-Cohen, D., Gehr, T., Zhang, C., and Vechev, M. T. DL2: training and querying neural networks with logic. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, pp. 1931–1941, 2019. URL http://proceedings.mlr.press/v97/fischer19a.html.

Griewank, A. The chain rule revisited in scientific computing. 1991.

Griewank, A. et al. On automatic differentiation.

Iri, M. Simultaneous computation of functions, partial derivatives and estimates of rounding errors : Complexity and practicality. *Japan Journal of Applied Mathematics*, 1, 12 1984. doi: 10.1007/BF03167059.

Rall, L. B. Differentiation and generation of taylor coefficients in pascal-sc. In *Proc. of the Symposium on A New Approach to Scientific Computation*, pp. 291–309, USA, 1983. Academic Press Professional, Inc. ISBN 0124286607.

Rall, L. B. Differentiation in pascal-sc: type gradient. *ACM Trans. Math. Softw.*, 10(2):161–184, May 1984. ISSN 0098-3500. doi: 10.1145/399.418. URL https://doi.org/10.1145/399.418.

Reps, T. W. and Rall, L. B. Computational divided differencing and divided-difference arithmetics. *High. Order Symb. Comput.*, 16(1-2):93–149, 2003. doi: 10.1023/A:1023024221391. URL https://doi.org/10.1023/A:1023024221391.

Shapiro, V. Semi-analytic geometry with r-functions. *Acta Numerica*, 16:239–303, 2007. doi: 10.1017/S096249290631001X.

Speelpenning, B. *Compiling Fast Partial Derivatives of Functions given by Algorithms*. PhD thesis, USA, 1980. AAI8017989.

Takanen, A., DeMott, J. D., and Miller, C. Fuzzing for software security testing and quality assurance. 2008.