Improvements to graph construction in AFLGO

ZHANG Zhexiang
Supervisor: Professor MENG Wei

UG CSCIN Student
The Chinese University of Hong Kong

April 22, 2024

Table of Contents

Table of Contents

Review of Fuzz and Last Semester

Fuzz[10] is a security testing method that generates random inputs and monitors software behavior for abnormal occurrences during runtime.
As a fuzzer, American Fuzzy Lop (AFL)[4] operates by instrumenting the target program to gather code coverage information during runtime. By applying input mutation techniques, it aims to maximize code coverage, thereby increasing the likelihood of triggering potential vulnerabilities.

Review of Fuzz and Last Semester

However, maximizing code coverage is not always suitable for all situations. AFLGO[5], as a variant of AFL, is developed and it focus on specific code segments rather than overall code coverage.
It properly handles the power scheduling problem in directed fuzzing[6] using simulated annealing[8].
However, it cannot handle indirect calls[7].

Review of Fuzz and Last Semester

Indirect calls are extensively used in C/C++ programs to offer dynamic program behaviors[9]. Though simulated annealing is a powerful algorithm for energy scheduling, it relies on proper distance information. In cases where distance information cannot be calculated, such as with indirect calls, simulated annealing will be ineffective[6].

Review of Fuzz and Last Semester

To overcome this shortcoming, in the previous semester, we introduced an iterative architecture as shown below.

Require: program p

1: $targets$ = **TargetGeneration(p)**, CG = **LLVMCG(p)**, CFG_i = **LLVM-CFG(p)**
2: **while** NOT *timeout* **do**
3: $BBDistance$ = **ModifiedDistanceCalculator(**CG**,**CFG_i**,**$targets$**)**
4: $Instrumented\_Binary$ = **ModifiedInstrumentor(p,**$BBDistance$**)**
5: $report$ = **AFLGoFuzzer(**$Instrumented\_Binary$**,**$seed\_inputs$**)**
6: **ReportCrashingInputs(**$report$**)**
7: CG, CFG_i= **IterationGraphConstruction(**$report$**,**CG**,**CFG_i**)**
8: **end while**
9: **return** CG, CFG_i

Review of Fuzz and Last Semester

Require: program p

1: ...
2: **while** NOT *timeout* **do**
3: ...
4: CG, CFG_i= **IterationGraphConstruction(***report*,*CG*,CFG_i**)**
5: **end while**
6: **return** CG, CFG_i

Remark

We put the fuzzer in the iteration. By collecting runtime information, we can produce more completed CG. Then augmented CG will help it to reach the deeper targets.

Review of Fuzz and Last Semester

However, during the testing conducted this semester, we observed that this approach proved to be time-consuming. The reason behind it is that the AFLGO[5] itself operates as an time-consuming iterative architecture!

Remark

The architecture has been proven to be too slow, even much much more slower than some static analysis algorithms. i.e. slower than $O(n^3)$.

Although this architecture is slow, we can try to verify its correctness. We will discuss it in part 3.

Review of Fuzz and Last Semester

Last semester, we also noticed a property of digraph:

$$\delta_G(t, f) = \delta_{G_R}(f, t)$$

, where $\delta_G(a, b)$ represents the shortest path length between node a and node b in digraph G. Based on this property, we improved the distance calculation algorithm in AFLGO. Part 2 will show the evaluation results of the new algorithm.

Table of Contents

Evaluation

The algorithm has already been implemented in `distance.py`[3]. It has been used to test the program: `libxml2_ef709ce2`[1]. Its call graph consists of 2541 function nodes and 8220 function call edges. And there are 7482 CFGs.

Remark

There are two types of distance calculation, say, FLDC and BBDC, in AFLGO. FLDC requies the CG. And BBDC requires FLDC and CFGs.

Function Level Distance Calculation Evaluation

With the call graph consisting of 2541 function nodes and 8220 function call edges, the running time is as follows. Please note that we only present the `real` time instead of the `user` time.

LoC That Set As Target	Original `real` Time	New `real` Time
1	0m17s	0m16s
4	0m18s	0m16s
10	0m18s	0m17s
20	0m18s	0m17s
50	0m20s	0m18s
100	0m33s	0m18s
200	0m34s	0m18s

Function Level Distance Calculation Evaluation

LoC That Set As Target	Original `real` Time	New `real` Time
1	0m17s	0m16s
4	0m18s	0m16s
10	0m18s	0m17s
20	0m18s	0m17s

Observation

For smaller target size, the time difference between the two algorithms is not significant. Further analysis indicates that a considerable amount of time (About 13 to 15 seconds) is spent on reading the call graph via library function.

Function Level Distance Calculation Evaluation

LoC That Set As Target	Original `real` Time	New `real` Time
50	0m20s	0m18s
100	0m33s	0m18s
200	0m34s	0m18s

Observation

As the target size increases, we have observed that the old algorithm starts to take a significant amount of time. However, the time taken by the new algorithm did not increase significantly, indicating that, apart from the reading process, it only takes extra 1-2 seconds.

Function Level Distance Calculation Evaluation

Conclusion

This experiment confirms that the algorithm play a crucial role in improving time efficiency of FLDC when dealing with larger target sizes.

Basic Block Calculation Evaluation

The code in `distance/gen_distance_orig.sh`[3] has been modified to calculate the running time of basic block distance calculation. It contains 7482 CFG dot files.

Basic Block Calculation Evaluation

The test results are as follows.

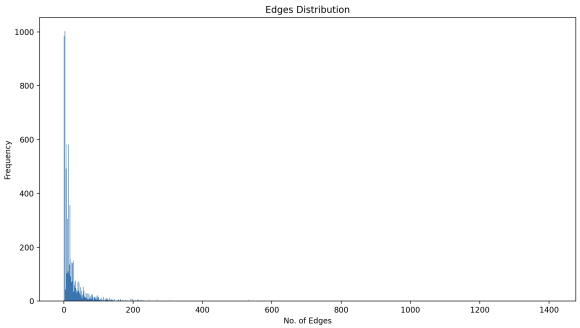
LoC That Set As Target	Original `real` Time	New `real` Time
1	32m4s	31m16s
4	32m3s	32m1s
100	34m	34m

Observation

It can be surprising when the algorithm does not demonstrate any improvement in distance calculation for basic blocks. Why?

Basic Block Calculation Evaluation

Further research shows that most CFGs have relatively small sizes with only a few ($\tilde{20}$) vertices and ($\tilde{30}$) edges. As demonstrated by the following two graphs.



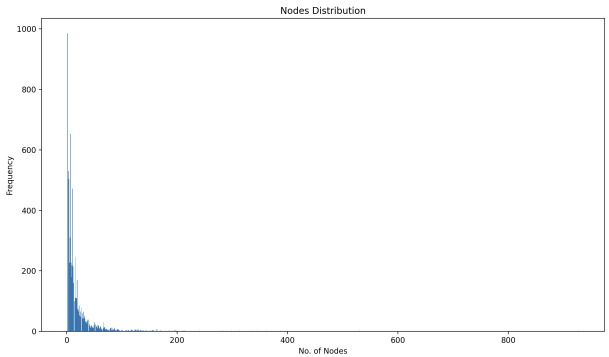


Figure: Edges/Nodes Distribution

Question

What impact will this have on the running time?

Basic Block Calculation Evaluation

The number of nodes in BB is primarily concentrated around 20, while the number of edges is concentrated around 30. Only a small fraction of CFGs have sizes exceeding 100. As a result, the calculation of basic block distances in each CFG can be considered to have a **constant time complexity** for each target. Consequently, the total time spent can be roughly estimated by multiplying the number of CFGs by the number of targets.

Conclusion

This experiment demonstrates that in scenarios where there are numerous but small CFGs, the algorithm does not enhance the efficiency of BBDC.

Conclusion

Conclusion

Our algorithm partially improves the DC performance of AFLGO.

Table of Contents

Introduction

Require: program p

1: ...
2: **while** NOT *timeout* **do**
3: ...
4: $CG, CFG_i =$ **IterationGraphConstruction(**$report, CG, CFG_i$**)**
5: **end while**
6: **return** CG, CFG_i

Despite the impracticality of executing the slow architecture within a feasible timeframe, we will try to **validate** its correctness. The verification process comprises two distinct parts.

* **We will test whether furnishing more comprehensive distance data aids AFLGO in reaching the hidden function.**
* **We will verify that, following the LLVM instrumentation, hidden edge information can be reported.**

Introduction

* **We will test whether furnishing more comprehensive distance data aids AFLGO in reaching the hidden function.**

Remark

In other words, we will provide AFLGO with more proper distance values with respect to the hidden target. Such proper distance values are generated from augmented CG/CFGs. We hope that those values will improve the performance in reaching the hidden target.

* **We will verify that, following the LLVM instrumentation, hidden edge information can be reported.**

Remark

In other words, we want to collect each possible caller-call function pairs during runtime. By examining these pairs, we can detect if any indirect call occurs. The detection log will be used to create augmented CG/CFGs.

Introduction

* **We will test whether furnishing more comprehensive distance data aids AFLGO in reaching the hidden function.**
* **We will verify that, following the LLVM instrumentation, hidden edge information can be reported.**

Remark

By combining two parts above, it is **reasonable** to draw the following conclusion: By putting AFLGO into our iteration arch, it can approach to hidden targets. And more completed CG/CFGs can be generated.

PART I: Given the More Completed Graph To Reach Target

* Given the More Completed Graph To Reach Target

Due to time constraints, we cannot test many different programs. In this section, our experiment focuses on a practical program `libxml2_ef709ce2`[1], wherein we introduce a **hidden function** that will crash with probability of p. This function is designated as the target for AFLGO.

Remark

The `libxml2` is commonly used for parsing, manipulating, and generating XML documents. By applying code insertion and crash detection , our conclusions gain increased credibility.

Configuration

In the experimental group, we need set that **hidden function** as target and provide proper recalculated distance values. In the control group, the target is the same but there is no proper distance values. We also need a special background group, where **hidden function** will never crash.

* Code Insertion for Three Groups

For all three groups, we modified `valid.c` in `libxml2_ef709ce2` project by adding following two functions.

```
static void good_function(){return;}
static void bad_function(){
    int* p = NULL;
    *p = 0;// target!
    return;
}
```

We have added a function invocation within the `xmlAddID` function. If the program reaches this function, there is a probability of p that it crashes.

```
// AFLGO
srand(time(NULL));int r = rand();void (*fp_aflgo)();
if (r % p == 0) fp_aflgo = bad_function; // different p here
else fp_aflgo = good_function;
fp_aflgo();
```

Listing 1: function invocation

* Distance Recalculation for Experimental Group

For the experimental group, we need to overwrite `distance.callgraph.txt` and `BBsites.txt` carefully. In the AFLGO[5], the target program is compiled twice. The first compilation generates CG/CFGs for distance calculation, so we will modify the scripts here. The overwriting is divided into two steps, as follows.

1. Add **two** edges to callgraph after `distance.py` reading `dot` file. Recalculate the distance and output the result to `distance.callgraph.txt`.
2. Locate the basic block corresponding to the function invocation and add the record of {`bbname, functionname`} pairs to the `BBsites.txt` file. This ensures that the BB distances are correctly computed.

* Different p

Then we conducted three experiments corresponding to different values of p. When p is set to $+\infty$, it represents the program in background group is unable to execute the `bad_function`.

```
// AFLGO
srand(time(NULL));int r = rand();void (*fp_aflgo)();
if (r == 0) fp_aflgo = bad_function; // p = $+\infty$
else fp_aflgo = good_function;
fp_aflgo();
```

Listing 2: function invocation

* Given the More Completed Graph To Reach Target

We have recorded the respective data in the table provided below:

p(crash probability%)	Fuzzing Time	Total Crashes	Unique Crashes
10(10%,with data) E	1d8h	4.5k	97
10(10%,without data) C	1d8h	4.1k	97
$+\infty$(0%,without data) B	1d8h	4.2k	97

Slow Execution in E Group

We notice that in the groups with data, the execution time was high. It sometimes takes several seconds to complete an instance at the beginning. These issues could potentially be attributed to the introduction of a substantial volume of basic block distance and the calculation of seed distance.

* Given the More Completed Graph To Reach Target

p(crash probability%)	Fuzzing Time	Total Crashes	Unique Crashes
10(10%,with data) E	1d8h	4.5k	97
10(10%,without data) C	1d8h	4.1k	97
$+\infty$(0%,without data) B	1d8h	4.2k	97

Observation

The total crashes in the control group is not higher than the total crashes in the background group. This indicates that AFLGO without proper data guidance cannot find the hidden function at all.

* Given the More Completed Graph To Reach Target

p(crash probability%)	Fuzzing Time	Total Crashes	Unique Crashes
10(10%,with data) E	1d8h	4.5k	97
10(10%,without data) C	1d8h	4.1k	97
$+\infty$(0%,without data) B	1d8h	4.2k	97

Observation

Comparing the total crashes in the control group to the experimental group, we can conclude that providing proper distance information benefits AFLGO in reaching the target function since it produces more total crashes.

Observation

All three groups produces the same numbers of unique crashes. This outcome may be attributed to the criteria used by AFLGO to determine crash uniqueness, which relies on executed edges.

* Given the More Completed Graph To Reach Target

p(crash probability%)	Fuzzing Time	Total Crashes	Unique Crashes
10(10%,with data)	1d8h	4.5k	97
10(10%,without data)	1d8h	4.1k	97
$+\infty$(0%,without data)	1d8h	4.2k	97

Conclusion

We verified that more comprehensive distance data aids AFLGO in reaching the hidden function.

Conclusion

The experiment was divided into three groups, and their comparison provided strong evidence. And the experimental subject is a practical software. The probability of crash is $p = 10\%$, which is not even high, in order to show the fact that AFLGO has truly approached the hidden target.

PART II: Given the Target To Get More Completed Graph

PART II: Given the Target To Get More Completed Graph

To uncover additional hidden calls, we need to collect runtime information during fuzzing.

Question

We need to set a hidden function as target, then report any indirect call edges during fuzzing. Then how to collect runtime information?

PART II: Given the Target To Get More Completed Graph

We use `llvm pass` and harness.

* `llvm-pass-icg`, as discussed last semester, is used to report runtime information.

** Harness is utilized to redirect the program's output to some specific output files for recording purposes. The reason for why we need harness is that AFLGO does not analyze the program's output. As a result, any output sent to the stdout will be disregarded during the fuzzing process.

Remark

Although AFLGO will fork multiple child processes and OS will ensure the atomicity of the writing, we do not want different processes write to the same file simultaneously. To distinguish the outputs generated by different child processes, we will require them to output to different files.

PART II: Why we redirect outputs to different files

If we write to the same file, we may not get the correct results. Imagining that if two child processes p_1, p_2 are writing to the same file simultaneously, the output of p_1 is $f_{indirect\_call}, f_{callee}$. We should add an edge between them. But the file may contain $f_{indirect\_call}, f_{p_2}, f_{callee}$, which can cause an edge to be added between $f_{indirect\_call}, f_{p_2}$.

Remark

To output to different files, We use `getpid()` API. The output filename,e.g. *p.txt*, may be reused later after the termination of process with pid p. This will not have negative consequences, because given the running process that currently occupies pid p, no other process can write to *p.txt* simultaneously.

Remark

The number of files will be very large.

PART II: Given the Target To Get More Completed Graph

At the beginning of the harness, the `freopen()` is used to open the file and associate it with the `stdout` stream in harness. This allows us to capture and analyze the program's output:

```
#include ...
int main() {
        // before the first time compliation:
        char filename[20];
        snprintf(filename, sizeof(filename), "%d.txt",
    getpid());
        FILE *file = freopen(filename, "a", stdout);
        ...
}
```

Listing 3: test a programme

PART II: Given the Target To Get More Completed Graph

```
#include ...
int main() {
        ...
        FILE *file = freopen(filename, "a", stdout);
        ...
}
```

Remark

Such modifications may lead to unexpected behavior for some programes.

Not using harness?

In `llvm-pass-icg`, we output caller-callee using `printf` API. It looks like we do not need harness anymore if we replace this API with redirection API. However, the time cost of file open/close will be very high if we try to do so.

PART II: Given the Target To Get More Completed Graph

And some assumptions need to be made. It is assumed that we have prior knowledge of at least one path to access the hidden function. Based on this assumption, we can utilize the recalculated distance values and analyze the information generated during runtime.

PART II: Given the Target To Get More Completed Graph

In the simple project [2] (comprising several hundred LoC), we identify three functions, denoted as `known`, `unknown_a`, and `unknown_b`, and insert the following code at the beginning of those functions:

```
1 // AFLGO
2 srand(time(NULL));int r = rand();void (*fp_aflgo)();
3 if (r % 10 == 0) fp_aflgo = bad_function; // different p
      here
4 else fp_aflgo = good_function;
5 fp_aflgo();
```

Listing 4: function invocation

Relying on the assumption, we know that two edges connecting `known` with both `good_function` and `bad_function` already exist. Consequently, set `bad_function` as the AFLGO's target and recalculate relevant basic block distance. Finally, we start fuzzing process and analyze the outcomes.

PART II: Given the Target To Get More Completed Graph

We need to analyze and clear the files from time to time, thereby preventing the accumulation of large files for subsequent analysis.

Insertion Point	Fuzzing Time	Found Hidden Edges
Three random functions	30min	4
Three random functions	30min	4
Three random functions	30min	4

Conclusion

We conducted three rounds of fuzzing and observed a consistent discovery of four hidden edges within half an hour. We verified that, following the LLVM instrumentation, hidden edges can be reported.

PART II: Given the Target To Get More Completed Graph

Conclusion

It is feasible to infer that the utilization of AFLGO, harness and LLVM can contribute to the identification of indirect call edges during runtime.

Conclusion

Considering the insights derived from previous parts, it becomes apparent that a more comprehensive distance values holds the potential to facilitate AFLGO in approaching its objectives.

Conclusion

Building upon the findings, we may assert that a more completed graph structure not only aids AFLGO in the exploration of additional hidden edges, but also augments the graphs. Meanwhile, we verify the correctness of the iterative structure.

Table of Contents

Conclusion & Review the Whole Project

During this semester, efforts were made to implement the proposed distance algorithm and test its practicality. Additionally, the assertion that a more comprehensive call graph and control flow graphs could enhance AFLGO's effectiveness in reaching its target was validated. Furthermore, the integration of AFLGO with the llvm-pass-icg and harness verifies its ability to identify and report hidden edges.