WEI2302 Improvements to graph construction in AFLGO

Zhexiang ZHANG

April 9, 2024

Abstract

AFLGO, as a directed graybox fuzzer (DGF), confronts difficulties in effectively handling indirect calls within the program. During this semester, the efforts are directed towards implementing the distance algorithm proposed and examining its practicality. Simultaneously, we validated the assertion that a more comprehensive call graph and control flow graphs can assist AFLGO in reaching its target. Furthermore, we try to integrate AFLGO with the `llvm-pass-icg` and verified its capability to identify and report hidden edges.

1 Review of Fuzz

Fuzzing [MFS90] is a method used for security testing, which involves generating random inputs and monitoring the behavior of software during runtime to detect any abnormal occurrences. It is categorized into three levels: black box, gray box, and white box, depending on the availability of the target program's source code. American Fuzzy Lop (AFL)[14] falls under the gray box fuzzing category. AFL instruments the target program to gather code coverage information at runtime, with minimal impact on overall performance. By mutating seed inputs, AFL maximizes code coverage, enabling comprehensive testing of the target program. However, AFL and other gray box fuzzers lack effective guidance in their approach. Specifically, when it comes to patch testing, AFL's capabilities are limited to testing specific APIs or the entire program. To overcome this limitation, the concept of Directed Graybox Fuzzing (DGF) was introduced, and AFLGO[Böh+17] was developed as an implementation of DGF. AFLGO aims to provide a directed graybox fuzzing capability, allowing for more targeted and efficient testing compared to traditional gray box fuzzers.

AFLGO is a variant of AFL that prioritizes specific code segments rather than focusing solely on overall code coverage. It achieves this by combining static and dynamic analysis techniques. AFLGO introduces two key concepts: basic block distance and seed distance. Basic block distance refers to the distance between any two basic blocks, which can be determined during the compilation phase. Seed distance, on the other hand, represents the distance from the input to the target code segment and is obtained during runtime.

The AFLGO workflow consists of two steps. In the first step, AFLGO instruments the program to gather basic block distance information using static analysis techniques. This information is then used to guide the subsequent fuzzing process. In the second step, AFLGO calculates the seed distance through dynamic analysis during runtime. It formulates the problem as an optimization task, aiming to minimize the seed distance and generate inputs that closely approach the marked code segment. To address this optimization problem, AFLGO employs an algorithm known as simulated annealing [KGV83]. Simulated annealing is utilized to control and improve the quality of seeds. It effectively deals with the power scheduling problem in directed fuzzing by determining which seeds should receive higher mutation weights and which seeds should be disregarded. This approach enhances the effectiveness of the fuzzing process [Che+18]. By combining static and dynamic analysis, AFLGO introduces a directed fuzzing capability that allows for more targeted and efficient testing, focusing on specific code segments of interest. The utilization of simulated annealing as an optimization strategy further enhances the effectiveness of AFLGO in discovering potential vulnerabilities and improving the overall quality of the fuzzing process. Simulated annealing is a powerful algorithm for energy scheduling; however, its effectiveness relies on accurate distance information. In scenarios where distance information cannot be readily calculated, such as in the presence of function pointers, simulated annealing **may** prove ineffective [Che+18]. Due to the nature of these indirect calls, the objects they reference cannot be

easily determined through static analysis. By default, the LLVM [LA04] utilized by AFLGO does not include an analysis of objects referenced by indirect calls. Consequently, the resulting call graph (CG) and inter-procedural control flow graph (iCFG) lack comprehensive information in this regard. Since the construction of these graphs is crucial for calculating the basic block distances [21] in AFLGO, the absence of accurate distance information poses challenges to the simulated annealing algorithm employed by AFLGO in navigating indirect call instructions.

2 Review of Last Semester

In the previous semester, we introduced an iterative architecture, as shown below, aiming to uncover new call graph or control flow graph structures. This architecture has been adapted from the AFLGO architecture, wherein direct calls are designated as the target. The process involves iteratively executing AFLGO and gathering program function call details during runtime. The output encompasses both crashing inputs and additional indirect calls as part of the CG. The modified architecture incorporates several key functionalities. The **TargetGeneration** functionality identifies potential indirect call sites during the compile-time phase, aiming to generate comprehensive CG and CFG information. The **ModifiedDistanceCalculator** improves distance calculation efficiency, while the **ModifiedInstrumentor** enhances the existing instrumentation mechanism to provide additional runtime information for reporting indirect calls. The **IterationGraphConstruction** functionality collects runtime information for constructing the CG and CFG. Together, these functionalities enhance the architecture's performance in generating comprehensive CG and CFG information, optimizing distance calculation, and facilitating runtime analysis.

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[Böh+17] itself operates as an iterative architecture, consuming a significant amount of time during its execution. Consequently, if the fuzzer is subjected to further iteration, the time required could extend to several days or even weeks. In comparison to the identification of hidden edges using pointer analysis programs[Jez23], this extended time frame is not considered efficient.

Require: program p
1: $targets$ = **TargetGeneration(p)**,CG = **LLVMCG(p)**,CFG_i = **LLVMCFG(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

Despite its limited effectiveness, the architecture yielded valuable insights. We confirmed that different distance data have varying effects on AFLGO's ability to reach the target, with a more comprehensive Call Graph proving particularly helpful4.3. Additionally, we experimented with integrating llvm pass icg into AFLGO to enable feedback on caller call data4.4.

Furthermore, we have successfully implemented the distance algorithm that was discussed in the previous semester. Review the definition of function level distance in AFLGO:

$$d_f(n, T_f) = (\sum_{t_f \in R(n,T_f)} (d_f(n, t_f)^{-1})^{-1}, |R(n, T_f)| \neq 0$$

, where $R(n, T_f)$ is a set of target functions that can be reached from function n. And $d_f(n, t_f)$ is defined as the shortest path between n and t_f in the LLVM CG[LA04]. AFLGO[Böh+17] also offers a method for calculating the distance between basic blocks by leveraging the distances of the preceding function components. After calculating function-level distance, we can use it to calculate basic block

distance. Another basic block distance is defined as follows:

$$d_b(m, T_b) = \begin{cases} 0, & m \in T_b \\ 10 \min(d_f(n, T_f)), & m \in T, n \in N(m) \\ [\sum_{t \in T}(d_b(m,t) + d_b(t, T_b))^{-1}]^{-1}, & otherwise \end{cases}$$

, where $d_b(m_1, m_2)$ represents the shortest distance between two basic blocks, m_1 and m_2, within the same LLVM CFG G_i[LA04]. Additionally, the neighbor set $N(m)$ comprises functions that are called by basic block m. For a function f to be included in the neighbor set, it must satisfy the condition $|R(f, T_f)| > 0$. A function is considered a neighbor of a basic block if it is called by that block and has the can reach the target functions. The transfer block set T collects all basic blocks $b_i \in G_i$ for which the neighbor set $|N(b_i)| > 0$.

The old distance algorithm, that operates on a brute force calculation involving the computation of distances between almost all possible pairs of nodes in the graph, is as follows:

Algorithm 1 AFLGO Function-Level Distance Calculation Algorithm

1: **for** $n \in N$ **do**
2: **for** $t \in T$ **do**
3: $Dist(n) = 1/(1/Dist(n) + 1/Get\_Dist(G, n, t))$ /*Initialize $Dist(n)$ with $+\infty$ and call algorithm $|T|$ times for each node n*/
4: **end for**
5: Add $(n, Dist(n))$ to distance records R /* For each element in R, it needs to be multiplied by n if necessary */
6: **end for**
7: **return** R

It is easy to see that the time complexity of preceding algorithm is $|N| \cdot |T| \cdot D(|G|)$, where $D(|G|)$ is the time complexity of Dijkstra's algorithm with input CG G. Last semester, we 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.

3 Quick Distance Calculation Algorithm

3.1 Introduction

Based on the property $\delta_G(t, f) = \delta_{G_R}(f, t)$, we prepared following algorithm:

Algorithm 2 Quick Function-Level Distance Calculation Algorithm

1: $G^R = reverse(G)$
2: **for** $t \in T$ **do**
3: $L = Get\_Dist\_List(G^R, t)$
4: **for** $n \in N$ **do**
5: $R[n] = 1/(1/R[n] + 1/L[n])$ /* Initialize R[n] with $+\infty$ for all n */
6: **end for**
7: **end for**
8: **return** R /* For each element in R, it needs to be multiplied by n if necessary */

$Get\_Dist\_List(G^R, t)$ be a function that returns a list of shortest paths from the target function t to all other nodes in the reverse graph G^R. Actually, in unweighted graphs, the shortest path can be efficiently and simply calculated using the Breadth-First Search. The method to get $Dist\_List(G^R, t)$ is based on BFS instead of $dijkstra\_predecessor\_and\_distance(G^R, t)$ API[23b] proposed last term. This section2 encompassed a comprehensive evaluation of our algorithm, revealing significant advantages when dealing with scenarios involving a large number of targets. However, in cases where the number

of targets is small and there is a prevalence of small-sized basic blocks, our algorithm does not exhibit a substantial advantage. Detailed analysis and explanations for this situation are presented.

3.2 *Get_Dist_List*(G^R, t) and BFS

It should be noted that if we are able to solve the Single-Source Shortest Path (SSSP) problem, we can also address the distance calculation problem in AFLGO. This is due to the fact that once we have generated the results for all single source nodes in the reversed graph, we can subsequently calculate the corresponding harmonic mean in the orignal graph. In the previous term, the focus was primarily on Dijkstra's Algorithm[23a][23b]. However, given that the call graph is unweighted, the following theorem holds.

Theorem 1 *SSSP Problem can be solved in* $O(|V|+|E|)$ *time if all edges in the directed simple graph have the same positive weight* 1*.*

The preceding theorem can be easily proven. When all edges have the same positive weight of 1, the distance between the source node s and a node f is equivalent to the minimum number of edges traversed from s to reach f. The problem can be solved using a Breadth-First Search (BFS) algorithm with a time complexity of $O(|V| + |E|)$. Notice that there are t targets, thus we need to perform t times BFS and total time complexity is $O(t(|V| + |E|))$.

Theorem 2 *The distance calculation in AFLGO can be solved in* $O(t(|V| + |E|))$ *time.*

3.3 Evaluation

The code for calculating the efficiency has already been implemented. It has been used to test the efficiency of the target program, which is `libxml2_ef709ce2`[23c]. The call graph of this program consists of `2541` function nodes and `8220` function call edges. And there are `7482` CFGs.

3.3.1 Function Level Distance Calculation Evaluation

The code in `distance/gen_distance_orig.sh`[21] has been modified to calculate the running time of performing function-level distance calculation. Our algorithm is implemented in `distance.py`[21], where we manually specify the target functions for analysis. Due to the use of `memorize`[21] decorators and dictionaries in the original code, the complexity of the algorithm cannot be easily analyzed. Additionally, the cost of reading `.dot` files using `networkx`[24b] library functions is not explicitly known, making it challenging to provide a precise analysis. As a result, we can only calculate the total time consumed and attempt to analyze it based on the observed results.

```
time ($AFLGO/distance/distance_calculator/distance.py -d $TMPDIR/dot-files/callgraph.
    dot -t $TMPDIR/Ftargets.txt -n $TMPDIR/Fnames.txt -o $TMPDIR/distance.callgraph.txt
     > $TMPDIR/step${STEP}.log 2>&1 || FAIL=1)
```

Listing 1: Function Level Distance Calculation Time Measurement

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

For smaller target sizes, 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 `read_dot`[24a], which is a process that cannot be optimized. As the

target size increases, we have observed that the old algorithm starts to take a significant amount of time (approximately 10 seconds more) due to the repeated use of the Dijkstra algorithm. However, the time taken by the new algorithm did not increase significantly, indicating that, apart from the image reading process, it only requires an additional 1-2 seconds of execution time.

This experiment confirms that algorithms play a crucial role in improving time efficiency, particularly when dealing with larger target sizes. The impact of algorithmic improvements becomes more significant as the target size increases.

3.3.2 Basic Block Calculation Evaluation

The code in `distance/gen_distance_orig.sh`[21] has been modified to calculate the running time of performing basic block distance calculation. It contains `7482` CFG dot files. In general, this particular part of the process is the most time-consuming and can potentially require several hours to complete.

```
time(
  for f in $(ls -1d $TMPDIR/dot-files/cfg.*.dot); do
  # code for calculate basic block distance, there are 7482 files
  done
  echo ""
  # combine the results
  cat $TMPDIR/dot-files/*.distances.txt > $TMPDIR/distance.cfg.txt
)
```

Listing 2: BB Distance Calculation Time Measurement

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

It can be frustrating when the algorithm does not demonstrate any improvement in distance calculation for basic blocks. However, upon further research, it has been discovered that most control flow graphs have relatively small sizes with only a few ($\tilde{20}$) vertices and ($\tilde{30}$) edges. 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. In addition, reading all CFGs takes approximately 9m34s.

This observation provides an explanation for the lack of improvement observed in the algorithm. In many cases, there is not a significant variation in the size of basic blocks. As demonstrated by the following two graphs, 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. When the CFGs do not exhibit substantial differences in size, the advantages of the new algorithm may not be fully realized.

This experiment demonstrates that in scenarios where there are numerous but small control flow graph calculations, the algorithm does not enhance efficiency. This finding is likely applicable to other programs in real-world settings, where the majority of basic blocks are small in size but occur in large quantities.

3.4 Conclusion

Taking into account the architecture we proposed for running the fuzzer iteratively in the previous semester, if we aim to reduce the execution time, it becomes necessary to modify the distance parameter during the runtime of the fuzzer. However, upon considering that the computation time for the distance calculation has not exhibited a significant reduction, particularly in the case of the basic block distance. Consequently, we are compelled to relinquish the iteration architecture.

4 Hidden Function Discovery

4.1 Introduction

We consider a function f as **hidden** if, for any possible function call chains ending with f and starting with *main*, there is at least one indirect call within each chain. The experiment has revealed that

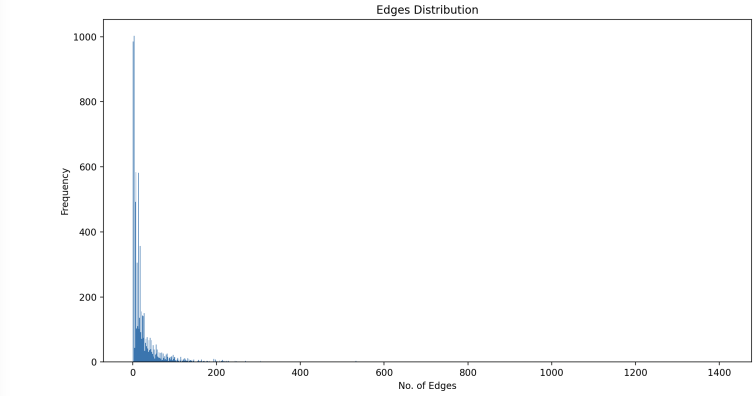

Figure 1: Edges Average: 28

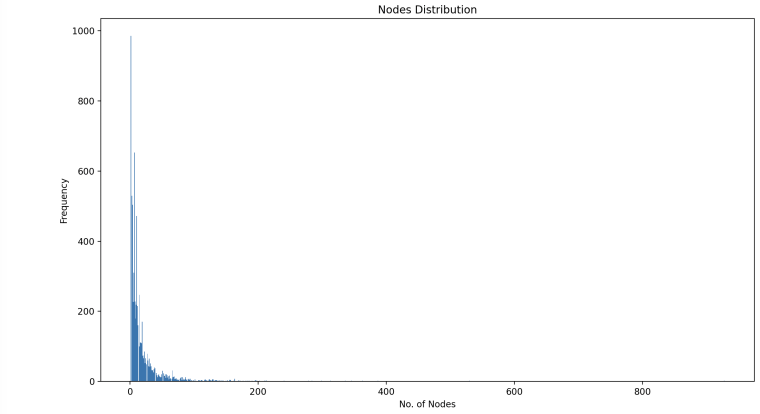

Figure 2: Nodes Average: 20

attempting to set a hidden function as the goal, as initially anticipated in the previous semester, does not yield effective information through BB/CG distance calculations.

In real-world scenarios, the file distance.cfg.txt may not always be empty. However, the fuzzer may still lose valuable information on how to approach certain functions. For instance, consider a situation where a function f_t can only be called through the chain $f_1, f_2, f_{indirect\_call\_site}, f_3, f_4, f_t$. AFLGO might be able to obtain basic block distances for functions f_3 and f_4, but it will never acquire this information for functions f_1 and f_2. Consequently, AFLGO is unable to properly approach the function $f_{indirect\_call\_site}$.

To enhance AFLGO's performance[Che+18], additional distance data is provided to guide the fuzzer towards target functions that may be hidden behind indirect call sites. This is achieved by generating extra call graph or control flow graph distance results and overwriting the existing distance values. The rationale behind this approach is that the fuzzer can only perceive the distance values inserted in the code, irrespective of how they were calculated. By overwriting these values, it is possible to direct the fuzzer towards the desired destination. Different generated distance values within the same graph can guide the fuzzer to different stages, and by overwriting them, it can be guided to the proper target function.

4.2 Evaluation Method

Despite the impracticality of executing this cumbersome architecture within a feasible timeframe, we will endeavor to validate its efficacy. The verification process comprises two distinct components. Firstly, we will ascertain whether furnishing more comprehensive distance data aids AFLGO in reaching

the hidden function. Secondly, we will verify that, following the LLVM instrumentation (utilizing the LLVM Pass employed in the previous semester) to uncover concealed paths, supplementary edge information can be reported. The following sections will further elaborate on these two parts. Through the consolidation of these two components, it is **plausible** to draw the conclusion that, when provided with a function, it can expedite its approach towards this function by covering the distance parameter after reporting more comprehensive path information at runtime. Or, at the very least, it may not hinder AFLGO's proximity to the target function.

4.3 PART I: Given the More Completed Graph To Reach Target

4.3.1 Introduction

Due to time constraints, we cannot test many different programs. In this section, our experimental focus on a practical program `libxml2_ef709ce2`[23c], wherein we introduce an additional hidden function that carries a certain probability p of inducing program crashes. This function is designated as the target.

The `libxml2` library is commonly used for parsing, manipulating, and generating XML documents. Its widespread adoption among developers in handling XML-related tasks makes it a typical and essential component. By employing code insertion and crash detection techniques within this practical library, our conclusions gain increased credibility.

To ascertain the potential benefits of providing AFLGO with more comprehensive distance data in approaching the target, we have devised multiple sets of experiments. The control group employs AFLGO without acquiring any distance data, while the experimental group's AFLGO incorporates varying values of p. The verification process involves running the experiments for a specified duration and examining the resulting total crash occurrences to draw definitive conclusions.

4.3.2 Methodology

4.3.2.1 Code Modification 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;
}
```

Listing 3: functions added

We have added a function invocation within the `xmlAddID` function. If the server 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 4: function invocation

In our experiment, we have designated `*p = 0;` as the target for AFLGO. Throughout the operation, we have implemented a series of steps to replace the distance numbers generated by AFLGO. Instead, we have inserted our own calculated distances into the program. In the following sections, we will provide a comprehensive overview of the process and present the results.

4.3.2.2 Distance Overwrite In the AFLGO[Böh+17] process, the target program is compiled twice. The first compilation generates dot files for distance calculation. This calculation involves two steps: CG distance calculation and CFG distance calculation using the CG distance. After the calculation is complete, the program undergoes a second compilation, wherein the relevant CFG distances are

inserted to enable runtime path information collection. To achieve our objectives, we need to overwrite the results of distance calculation (D.C.).

In the first step of D.C., when calculating the control flow graph distance, we introduce additional edges to the call graph G. This is necessary because `xmlAddID` indirectly invokes `good_function` and `bad_function`, and thus there are no edges in G that connect `xmlAddID` to them. This can result contains only one record `bad_function,1.0`. After adding those edges, we recalculate the distance and output the result to `distance.callgraph.txt`.

In the second step of D.C., AFLGO computes the distances between basic blocks. To perform this calculation, a complete `BBsites.txt` file is required, which specifies the functions that each basic block may invoke (if applicable). Our task is to locate the basic block corresponding to the function invocation and add the record of `bbname` and `functionname` to the `BBsites.txt` file. This ensures that the BB distances are accurately computed.

After overwriting the distance, we will start the fuzzing process.

4.3.3 Fuzzing Result

We conducted three experiments corresponding to different values of p and the existence of BB distance data. It is important to note that when p is set to $+\infty$, it represents the program in control group is unable to execute the `bad_function`. We notice that in the experimental group, the execution time was notably sluggish, taking several seconds to complete at the beginning. Moreover, instances of startup timeouts and failures were observed. These issues could potentially be attributed to the introduction of a substantial volume of basic block distance and the calculation of seed distance. We have recorded the respective data in the table provided below:

p(crash probability%)	Fuzzing Time	Total Crashes	Unique Crashes	Cycle Done
10(10%,with data)	1d8h	4.5k	97	2
10(10%,without data)	1d8h	4.1k	97	2
$+\infty$(0%,without data)	1d8h	4.2k	97	2

Despite the expectation that the group with $p = +\infty$ would have fewer unique crashes, all three groups exhibited 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. Further investigation is required to understand the underlying reasons for this observation.

The total crashes in the group with ($p = 10$, without data) is not higher than the group with ($p = +\infty$,without data). This could be due to the slower execution of test cases in the former. Additionally, we can deduce that the negative impact of the modified distance insertion on the process of identifying total crashes is relatively minor. Comparing the total crashes in the group with ($p = 10$, without data) to the group with ($p = 10$, with data), we can conclude that providing appropriate distance information benefits AFLGO in reaching the target function.

It is important to note that we cannot definitively conclude that providing more comprehensive distance data will help AFLGO reach the target function faster. While the experiments showed that the first group discovered more crashes within the same time frame, this is primarily due to the "hidden" nature of the `bad_function`. As a hidden function, directly setting it as the target would result in an **empty** result for the BB distance calculation. Hence, AFLGO guided by data has a reasonable advantage in discovering the `bad_function` faster compared to AFLGO without data guidance. However, this does not establish a direct relationship between the completeness of distance data and the speed of AFLGO's approach to the target function. If the `bad_function` is not hidden but can be found through indirect call sites, in such case, the computational resources may be allocated to discovering and executing these call sites. However, if these indirect call sites are difficult to execute, it is possible for AFLGO's speed to slow down. In summary, if it is not possible to find the target function (no distance data exists), we can help AFLGO locate it. However, if AFLGO is already capable of finding the target function (distance data exists), our intervention will not affect its discovery process.

4.3.4 Weight Edges

The experiment provided evidence that data can effectively guide AFLGO. Based on this, we speculate that by assigning appropriate weights to the edges in the control graph, **AFLGO can be guided towards the same target through different paths**. Specifically, the distance between the indirect

call site and the callee can be considered infinite. If a function `bad_function` can be reached through both `a` and `b`, increasing the weight of the edge (`a`,`bad_function`) would give AFLGO a reason to favor reaching `bad_function` through `b` rather than `a`. This weight assignment mechanism can influence AFLGO's decision-making process and steer it towards specific paths with the same target.

4.4 PART II: Given the Target To Get More Completed Graph

4.4.1 Introduction

To uncover additional hidden calls, certain steps need to be followed. These steps involve setting the hidden function as the target, collecting runtime information about function calls, and enhancing the control graph with the gathered data. The process relies on the `llvm-pass-icg`[23d], which was discussed in a previous semester. A harness is utilized to redirect the program's output to a specific output file 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.

4.4.2 Some Assumptions

In this context, certain assumptions need to be made. It is assumed that we have prior knowledge of at least one path to access the hidden function, which can be seen as employing an empty basic block distance directly. Building upon this assumption, we utilize the modified distance metric to designate the hidden function as the target and analyze the information generated during its runtime.

4.4.3 Methodology

4.4.3.1 LLVM Pass The `llvm-pass-icg` was discussed in the previous semester, there are certain pitfalls that need to be highlighted in the context of the current discussion.

Two significant issues need to be addressed in relation to the `llvm-pass-icg`. Firstly, there is a challenge regarding distance calculation during the initial compilation phase. `llvm-pass-icg` is designed to insert output statements at the beginning of each basic block without considering any global flag. Since the evaluation of any global flag would introduce branch statements, which will change basic blocks. The program does not generate new basic blocks after passing through `llvm-pass-icg`, distance calculation could be based on the original program.

The second issue concerns the order of passing during the second compilation in AFLGO. The target program should undergo `llvm-pass-icg` first, followed by AFLGO's pass. This sequence enables the program to output `caller_callee` information during runtime, while fuzzer can capture the corresponding distance data to do optimazation.

4.4.3.2 AFLGO Harness A harness refers to the test case execution framework used to run the target program under fuzzing. The harness is responsible for monitoring its execution and feeding it with various inputs or test cases to discover potential vulnerabilities or explore different code paths. If we are testing a function, we should create the harness. The `freopen()` is used to open the file `caller_callee` and associate it with the `stdout` stream.

```
#include <stdio.h>
int main() {
        char filename[20];
        snprintf(filename, sizeof(filename), "%d.txt", getpid());
        FILE *file = freopen(filename, "a", stdout);
        test_function();// test function
        fclose(file);
}
```

Listing 5: test a function

To test a program and redirect its output to a file, simply add a file opening operation at the beginning of the program. This allows us to capture and analyze the program's output easily:

```
#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 6: test a programme

Indeed, the methods mentioned above for redirecting stdout to a file may encounter issues in certain scenarios. One potential problem arises when the program itself performs operations that modify stdout. Such modifications may lead to unexpected behavior or disruption of the harness's functionality. Furthermore, the introduction of new code can impact distance calculations in AFLGO, which may affect the accuracy of basic block distance. It is important to carefully consider the specific requirements and constraints of testing scenario to determine wheather we should use those approachs for redirecting stdout and ensure it does not interfere with the program's intended behavior or analysis processes.

Finding the solution for getting the output of the target programme seems not easy. Attempting to modify the `llvm-pass-icg` to redirect output from stdout to a file will introduce potential drawbacks. This becasue it requires performing file I/Os for each basic block, and will significantly slowing down the fuzzer's execution speed. Though this approach is technically correct, but the performance impact makes it unreasonable for practical. While some methods require modifications to AFL's driver code[14] that cannot easily figure out. So the feasible choice at present is to employ a temporary harness for redirecting the program's output.

4.4.4 Fuzzing

In the project [23d] (comprising several hundred LoC), we explicitly identify three functions, denoted as `known`, `unknown_a`, and `unknown_b`, and subsequently proceed to insert the provided code snippet as follows:

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

Listing 7: function invocation

Relying on the assumption, we know that an edge connecting `known` with both `good_function` and `bad_function` already exists. Consequently, set `bad_function` as the AFLGO's target and computate relevant basic block distance, which is non-empty. Subsequently, we start fuzzing process and analyze the outcomes.

```
$AFLGO/afl-2.57b/afl-fuzz -m none -z exp -c 45m -i in -o out ./target_harness --valid
    --recover @@
# We need to clear the content from time to time
cat *.txt | grep "__ICG_STDOUT__:" | grep ",I:1" | sort | uniq > parser.py
```

Listing 8: fuzzing and analyzing

Considering the relatively compact nature of the tested program, the fuzzer has the capability to execute hundreds of test cases per second. However, this leads to a problem where the `*.txt` file, responsible for recording every function call of all processes, becomes excessively large. To mitigate the issue, we employ a strategy of periodically analyzing the file and subsequently clearing its contents, thereby preventing the accumulation of large files for subsequent analysis. It is recommended to clear the file at least once every minute. Given the compact size of the program, we conducted the fuzzing for half an hour. The result are summarized as follows:

Insertion Point	Fuzzing Time	Found Hidden Edges
Random_1	30min	4
Random_2	30min	4
Random_3	30min	4

We conducted three rounds of fuzzing and observed a consistent discovery of four hidden edges within the span of half an hour.

It is crucial to acknowledge that, given the abbreviated duration of the testing period and the diminutive scale of the program itself, drawing direct conclusions from this experiment regarding the efficacy of the simulated annealing algorithm[KGV83] in uncovering the hidden functions is unwarranted. However, it is feasible to infer that the synergistic utilization of fuzzy testing and LLVM can indeed contribute to the identification of concealed edges during runtime.

Furthermore, considering the insights derived from Part I4.3, it becomes apparent that a more comprehensive distance metric holds the potential to facilitate AFLGO in attaining its objectives. This principle holds true for larger-scale programs as well. Building upon the findings elucidated in Part II4.4, we may assert that a more exhaustive graph structure not only aids AFLGO in the exploration of additional hidden edges but also augments the overall graph. Consequently, this substantiates the validation of the correctness of the iterative structure.

5 Conclusion

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

6 Review the Whole Project

During our validation process, we encountered several instances where the effectiveness of certain ideas did not align with our expectations. For instance, while assessing the efficiency of distance calculation, we discovered that the algorithm did not perform as effectively as anticipated based on theoretical analysis conducted in the previous semester. Similarly, when validating our iterative architecture, we identified inefficiencies.

It is important to note that we did not conduct extensive testing on a wide range of programs to validate the conclusions presented in Part I4.3. Additionally, we did not quantify the notion of "efficiency" beyond using `Total Crash` as an explanation for our validation. As for Part II4.4, our attempt to implement a new feature of AFL, namely redirecting program output and incorporating it with LLVM, was accomplished in a rudimentary manner that may not be applicable to most programs.

Additionally, it has come to my attention that there are errors in the distance calculation process found in certain literature. Specifically, their LLVM pass modifies the basic blocks themselves (like trying to use any `br` instruction), which can lead to inconsistencies in distance calculation. This discovery was made towards the end of last year.

The progress of the project encountered some challenges along the way. For instance, when writing the LLVM pass, it was discovered that it was necessary to locate the entry block of `F` (a function) and perform the insertion at the entry block using the `getFirstNonPHI()` function. And to avoid concurrent modifications, we should push all required instructions into a list, etc. The related errors are very difficult to detect. Failure to do so resulted in crashes in the Clang compiler.

LLVM pass used by AFLGO also has a different API from the latest version, rendering the previously developed `llvm-pas-icg`. Modifications are needed to ensure compatibility with the current version of AFLGO.

References

[14] american fuzzy lop. https://lcamtuf.coredump.cx/afl/. 2014.

[21] AFLGo: Directed Greybox Fuzzing. https://github.com/aflgo/aflgo/blob/master/distance/distance_calculator/distance.py. 2021.

[23a] *dijkstra_path_length*. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.shortest_paths.weighted.dijkstra_path_length.html. 2023.

[23b] *dijkstra_predecessor_and_distance*. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.shortest_paths.weighted.dijkstra_predecessor_and_distance.html. 2023.

[23c] *libxml2*. https://github.com/aflgo/aflgo/blob/master/examples/libxml2-ef709ce2.sh. 2023.

[23d] *llvm − pass − icg*. https://github.com/LittleLerry/llvm-pass-icg. 2023.

[24a] *networkx.read_dot*. https://networkx.org/documentation/stable/reference/generated/networkx.drawing.nx_pydot.read_dot.html. 2024.

[24b] *networkx*. https://networkx.org/. 2024.

[Böh+17] Marcel Böhme et al. "Directed Greybox Fuzzing". In: Proceedings of the 2017 ACM SIGSAC Conference on C CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2329–2344. ISBN: 9781450349468. DOI: 10.1145/3133956.3134020. URL: https://doi.org/10.1145/3133956.3134020.

[Che+18] Hongxu Chen et al. "Hawkeye: Towards a Desired Directed Grey-box Fuzzer". In: Oct. 2018, pp. 2095–2108. DOI: 10.1145/3243734.3243849.

[Jez23] F. Jezuita. "Improving AFLGo27;s Directed Fuzzing by Considering Indirect Function Calls". In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (A Los Alamitos, CA, USA: IEEE Computer Society, Sept. 2023, pp. 143–150. DOI: 10.1109/ASEW60602.2023.00024. URL: https://doi.ieeecomputersociety.org/10.1109/ASEW60602.2023.00024.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing". In: Science 220.4598 (1983), pp. 671–680. DOI: 10.1126/science.220.4598.671. eprint: https://www.science.org/doi/pdf/10.1126/science.220.4598.671. URL: https://www.science.org/doi/abs/10.1126/science.220.4598.671.

[LA04] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis transformation". In: International Symposium on Code Generation and Optimization, 2004. CGO 2004. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[MFS90] Barton P. Miller, Lars Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities". In: Commun. ACM 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279.