Improvements to graph construction in AFLGO

ZHANG Zhexiang
Supervisor: Professor MENG Wei

UG CSCIN Student
The Chinese University of Hong Kong

December 3, 2023

Table of Contents

Table of Contents

Introduction

Fuzz[12] 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)[5] 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.

Introduction

However, maximizing code coverage is not always suitable for all situations. AFLGO[6], 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[7] using simulated annealing[9].
However, it cannot handle indirect calls[8].

Introduction

Indirect calls are extensively used in C/C++ programs to offer dynamic program behaviors[11]. 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[7].

Introduction

To enable the recognition of indirect calls in AFLGO, an architecture derived from AFLGO is presented in this report. It aims at constructing more comprehensive graphs.

Introduction

The remainder of this presentation is structured as follows.
Section **Motivating Example** showcases an example that greatly undermines the effectiveness of simulated annealing.
Section **Project Architecture** examines the architecture of AFLGO and proposes a modified iterative one.
Section **Improved Distance Calculation Algorithms** presents algorithms for calculating basic block distances efficiently.
Section **Construct More Complete Graphs** introduces some implementations.

Table of Contents

Function-Level Distance Calculation (D.C.) in AFLGO

AFLGO offers a method to quantify the distance between given function and a set of target functions.

Function-Level Distance Calculation (D.C.) in AFLGO

AFLGO offers a method to quantify the distance between given function and a set of target functions.
Let us consider a function n and a set of target functions T_f. The function-level distance is computed as follows[6]

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

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 CG.

Function-Level Distance Calculation (D.C.) in AFLGO

AFLGO offers a method to quantify the distance between given function and a set of target functions.
Let us consider a function n and a set of target functions T_f. The function-level distance is computed as follows[6]

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

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 CG.

Remark

If $R(n, T_f)$ is empty, the result is undefined.

Basic Block Distance Calculation (D.C.) in AFLGO

Similarly, the distance between basic blocks is defined. It is based on function-level DC.

Basic Block Distance Calculation (D.C.) in AFLGO

Similarly, the distance between basic blocks is defined. It is based on function-level DC.

$d_b(m_1, m_2)$ represents the shortest distance between two basic blocks within the same CFG G_i. Define the neighbor set $N(m)$ for a basic block m, where $N(m) = \{f | f \text{ is called by } m \text{ and } |R(f, T_f)| > 0\}$. Define the transfer block set T, which collects all basic blocks $b_i \in G_i$ for which the neighbor set $|N(b_i)| > 0$. Define the distance between a basic block m and all other target basic blocks T_b as follows[6]

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

Please notice that different $CFG_i s$ will have different Ts.

Basic Block Distance Calculation (D.C.) in AFLGO

Similarly, the distance between basic blocks is defined. It is based on function-level DC.

$d_b(m_1, m_2)$ represents the shortest distance between two basic blocks within the same CFG G_i. Define the neighbor set $N(m)$ for a basic block m, where $N(m) = \{f | f \text{ is called by } m \text{ and } |R(f, T_f)| > 0\}$. Define the transfer block set T, which collects all basic blocks $b_i \in G_i$ for which the neighbor set $|N(b_i)| > 0$. Define the distance between a basic block m and all other target basic blocks T_b as follows[6]

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

Please notice that different $CFG_i s$ will have different Ts.

Remark

If the transfer block set T is empty for all $CFGs$, the result is undefined.

An Example

Consider the following C code snippet called "main.c". Suppose our objective is to test the code located at line 2.

```c
1 #include <stdio.h>
2 void f1() {printf("Contains vulnerabilities\n");}
3 void f2() {printf("f2\n");} // target
4 int main() {
5     void (*fp)();int var;scanf("%d",&var);
6     fp = (var&1) ? &f1 : &f2;
7     (*fp)(); // indirect call here
8     return 0;
9 }
```

Observation

$f1$ is unreachable for *main* in CG.

An Example

Consider the following C code snippet called "main.c". Suppose our objective is to test the code located at line 2.

```c
1 #include <stdio.h>
2 void f1() {printf("Contains vulnerabilities\n");}
3 void f2() {printf("f2\n");} // target
4 int main() {
5     void (*fp)();int var;scanf("%d",&var);
6     fp = (var&1) ? &f1 : &f2;
7     (*fp)(); // indirect call here
8     return 0;
9 }
```

Observation

$f1$ is unreachable for *main* in CG.

Observation

What will happen if we try to set line 2 as target?

An Example

In this context, the function-level distance is undefined and will not provide any meaningful information. Additionally, AFLGO also consider it as the absence of transfer basic block set. Because it uses CG generated by LLVM[10] to determine the reachability. Thus $d_b(b_i, B_i)$ will be undefined.

An Example

In this context, the function-level distance is undefined and will not provide any meaningful information. Additionally, AFLGO also consider it as the absence of transfer basic block set. Because it uses CG generated by LLVM[10] to determine the reachability. Thus $d_b(b_i, B_i)$ will be undefined.

Observation

This leads to the loss of meaningful seed distance and makes the simulated annealing algorithm ineffective.

An Example

In this context, the function-level distance is undefined and will not provide any meaningful information. Additionally, AFLGO also consider it as the absence of transfer basic block set. Because it uses CG generated by LLVM[10] to determine the reachability. Thus $d_b(b_i, B_i)$ will be undefined.

Observation

This leads to the loss of meaningful seed distance and makes the simulated annealing algorithm ineffective.

Solution

Try to provide AFLGO with more comprehensive graphs.

An Example

However, a seemingly compromise solution is to set line 7 as the target, with the hope of triggering the execution of f1.

```
    (*fp)(); // Set target
```

An Example

However, a seemingly compromise solution is to set line 7 as the target, with the hope of triggering the execution of f1.

```
    (*fp)(); // Set target
```

This may present a paradox.

Remark

This approach requires prior knowledge that f1 might be called from the line 7.

Remark

Our focus should be on obtaining some more complete graphs before evaluating that if f_1 could be analyzed by AFLGO's or not.

An Example

Whitebox fuzzing techniques are useful for path exploration. However, we will not use it. We decide to run AFLGO iteratively. And it becomes possible to gather more comprehensive graphs and distance information during runtime.

New Architecture

In the next section, we will present the proposed architecture.

Table of Contents

AFLGO Architecture

```
Require: program p
1: $targets$ = TargetSelection(p), $CG$ = LLVMCG(p), $CFG_i$ = LLVM-CFG(p)
2: $BBDistance$ = AFLGoDistanceCalculator($CG$, $CFG_i$, $targets$)
3: $Instrumented\_Binary$ = AFLGoInstrumentor(p, $BBDistance$)
4: $report$ = AFLGoFuzzer($Instrumented\_Binary$, $seed\_inputs$)
5: ReportCrashingInputs($report$)
```

Compilation information collection

The program undergoes a two-step compilation process. During the first compilation, CG and CFG are generated. Those graphs are utilized to calculate the basic block distance via **AFLGoDistanceCalculator**. In the second compilation, distance information is inserted into the program through **AFLGoInstrumentor**.

AFLGO Architecture

```
Require: program p
1: $targets$ = TargetSelection(p), $CG$ = LLVMCG(p), $CFG_i$ = LLVM-CFG(p)
2: $BBDistance$ = AFLGoDistanceCalculator($CG$, $CFG_i$, $targets$)
3: $Instrumented\_Binary$ = AFLGoInstrumentor(p, $BBDistance$)
4: $report$ = AFLGoFuzzer($Instrumented\_Binary$, $seed\_inputs$)
5: ReportCrashingInputs($report$)
```

Runtime seed distance optimization

AFLGoFuzzer offers a comprehensive set of execution methods that monitor the program, gather runtime information, and apply simulated annealing to select inputs. The principle is to steer the execution "closer" to the target, thereby increasing the likelihood of executing the objective functions.

Modified AFLGO Architecture

In order to enhance its capability by providing it with more comprehensive graphs, we present the algorithm below. We add four components, namely ModifiedDistanceCalculator, ModifiedInstrumentor, TargetGeneration, and IterationGraphConstruction.

```
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 $CG$, $CFG_i$
```

Modified AFLGO Architecture

TargetGeneration aims to identify and report all potential **indirect calls sites**. These indirect call sites will be considered as targets for AFLGO.
ModifiedDistanceCalculator aims to enhance the efficiency of distance calculation.
ModifiedInstrumentor aims to furnish additional runtime information to report indirect calls.
IterationGraphConstruction aims to collect runtime information in order to construct the corresponding CG and CFG_i.

Remark

The following sections will present more details about these parts.

Table of Contents

ModifiedDistanceCalculator

AFLGO focuses on offloading resource-intensive analyses to the compile-time to preserve runtime efficiency. The proposed iteration architecture needs a different approach, because in this context, it may be impractical to allocate a significant portion of time in compilation, i.e., several hours[6] for each iteration.

Remark

The approach of first calculating the function-level distance and then calculating the basic block distance may be a compromise[6] compared to the directly calculation, as reported in their paper.

ModifiedDistanceCalculator

Previously, when the *iCFG* was generated, AFLGO would calculate the target distance within the *iCFG* for each basic block directly. But it is time-consuming. Thus it chooses to entirely omits the *iCFG* computation step[6]. Although now uses F.L. and B.B., the time is still unacceptable.

Question

What algorithms they used to calculate distance? Is that possible to improve them?

An Improved Algorithm for F.L.D.C.

Function-Level D.C.

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

Algorithm 1 D.C. Algorithm in AFLGO

```
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 $R$
```

An Improved Algorithm for F.L.D.C.

```
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 $R$
```

Observation

AFLGO uses *dijkstra_path_length(G, n, t)* API[1] to get the distance. Thus the time complexity of preceding algorithm is $O(|N||T|(|N|^2))$, which is extremely bad for huge size programs. Is that possible to improve it?

An Improved Algorithm for F.L.D.C.

Dijkstra algorithm returns a list of shortest path to all nodes for a given source node n. i.e., $dpl(G, n)$ returns a list of shortest paths between nodes n in G. And it is trivial that for any digraph, the following fact holds true:

Observation

$dpl(G, n, t) = dpl(G^R, t, n)$

An Improved Algorithm for F.L.D.C.

Dijkstra algorithm returns a list of shortest path to all nodes for a given source node n. i.e., $dpl(G, n)$ returns a list of shortest paths between nodes n in G. And it is trivial that for any digraph, the following fact holds true:

Observation

$$dpl(G, n, t) = dpl(G^R, t, n)$$

For arbitrary node n, we have following equation:

Observation

$$dpl(G, n, t) = dpl(G^R, t, n) = dpl(G^R, t)[n]$$

Based on this observation, we can improve D.C. Algorithm in AFLGO.

An Improved Algorithm for F.L.D.C.

Algorithm 2 Quick D.C.

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

Observation

Apply $dijkstra\_predecessor\_and\_distance(G^R, t)$ API[2]. The time complexity of preceding algorithm is $O(|T|(|N|^2)$.

An Improved Algorithm for B.B.D.C.

Recap the definition.

Basic Block D.C.

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

Since the calculation is performed independently for each CFG_i, it is possible to execute them in parallel. Please note that the transfer block set is NOT global. It is based on the given CFG_i

An Improved Algorithm for B.B.D.C.

Similar ideas can also be applied to this algorithm. This algorithm returns a list.

Algorithm 3 Quick Basic Block Distance Calculation Algorithm

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

An Improved Algorithm for B.B.D.C.

According to the AFLGO's algorithm, the time complexity of computing a shortest distance for basic blocks is denoted as $O(k^2 + k|N|^2)$, based on the fact that Djikstra's shortest-path algorithm which has a worst-case complexity of $O(|Nodes|^2)$. Here, we denote k as the number of intra-procedural CFGs in the program. $|N|$ represents the average number of nodes (basic blocks) within each CFG (CFG_i), and $|T|$ represents the number of transfer basic blocks. For our algorithm, the average time complexity is $O(\frac{k^2|T_f|}{k} + k|T|\frac{|N|^2}{|N|}) = O(k|T_f| + k|T||N|)$.

Fact

The time complexity of the oldest version is $O(k^2|N|^2)$.

ModifiedDistanceCalculator

Conclusion

Faster distance computation facilitates the implementation of an iterative architecture.

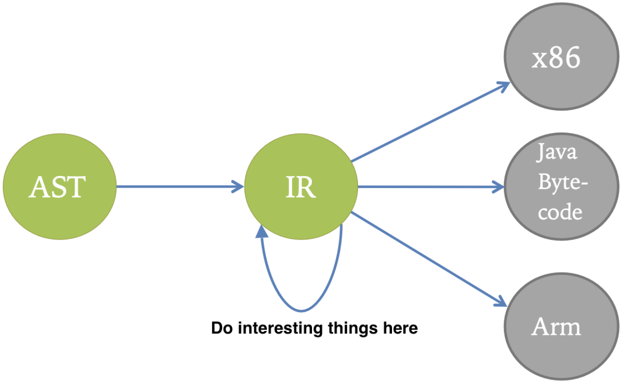
Table of Contents

Construct More Complete Graphs

AFLGO analysis CF and CFG at compile time using LLVM. However, some target functions may be determined dynamically based on runtime conditions. Those indirect calls will not be evaluated. Recap four parts: ~~ModifiedDistanceCalculator~~, TargetGeneration, ModifiedInstrumentor, and IterationGraphConstruction.

IR Modification: LLVM Pass

LLVM Pass[10] functionality gives us control over IR. TargetGeneration and ModifiedInstrumentor are based on transformation pass.



TargetGeneration

This functionality aims to identify and report all potential **indirect calls sites** during the compile-time phase. These indirect call sites will be considered as targets for AFLGO.

TargetGeneration

This functionality aims to identify and report all potential **indirect calls sites** during the compile-time phase. These indirect call sites will be considered as targets for AFLGO.
The llvm-pass-itargets tool is an implementation of **TargetGeneration**, in which it accepts a file awaiting compilation and generates indirect call sites as output. The output will be designated as the fuzzing target. The source code for this tool is now publicly available on GitHub[4].

llvm-pass-itargets

Throughout the compilation process, the dynamic cast operation is performed on each instruction. If the call instruction resulting from the dynamic cast operation lacks a callee, it is inferred that an indirect call site has been detected.

Remark

It is essential to emphasize that the term dynamic in this context refers to the LLVM runtime during program compilation, rather than the runtime of the program itself.

```
CallInst *CI = dyn_cast<CallInst>(&I);
if (CI && !(CI->getCalledFunction())){
    // do something to report debug info here
}
```

llvm-pass-itargets

The following example serves to illustrate the detection of indirect call sites during complie-time. Consider the following source code of a program named "main.c".

```
1  #include <stdio.h>
2  void f1() {printf("f1\n");}
3  void f2() {printf("f2\n");}
4  int main() {
5      void (*fpa)();void (*fpb)();int var;scanf("%d",&var);
6      fpa = (var&1) ? &f1 : &f2;
7      fpb = (var&1) ? &f2 : &f1;
8      (*fpa)(); // an indirect call site here
9      (*fpb)(); // an indirect call site here
10 }
```

The output is provided below. It is served as the designated target for AFLGO.

```
main.c:8
main.c:9
```

ModifiedInstrumentor

The llvm-pass-icg tool is the implementation of **ModifiedInstrumentor**. This functionality aims to furnish additional runtime information to facilitate the reporting of caller and callee. The AFLGO's instrumentation mechanism falls short of meeting our specific requirements.

Question

Can we insert any valid code?

ModifiedInstrumentor

The llvm-pass-icg tool is the implementation of **ModifiedInstrumentor**. This functionality aims to furnish additional runtime information to facilitate the reporting of caller and callee. The AFLGO's instrumentation mechanism falls short of meeting our specific requirements.

Question

Can we insert any valid code?

Remark

Preserving the relationships between basic blocks is paramount, and thus, we should avoid inserting code that alters these relationships. **Branch instructions are strictly prohibited and should not be used**.

```
Value *icmp = builder.CreateICmpEQ(icfValue, zeroValue, "
    cmp_val");
Instruction *thenInst = SplitBlockAndInsertIfThen(icmp,
    firstInst, false); // not allowed
```

llvm-pass-icg

By inserting a global set instruction and a function call before each detected indirect call site. The global set instruction assigns the value 0 to a global variable *icf*, indicating the occurrence of an indirect jump. The inserted function call is responsible for printing the currently executing function to the standard output (or file).
Then for every basic block, we will not insert *CreateICmpEQ* instruction. Instead, we report *icf* value and reset it. We also printing the currently executing function to the standard output (or file).

llvm-pass-icg

By inserting a global set instruction and a function call before each detected indirect call site. The global set instruction assigns the value 0 to a global variable *icf*, indicating the occurrence of an indirect jump. The inserted function call is responsible for printing the currently executing function to the standard output (or file).
Then for every basic block, we will not insert *CreateICmpEQ* instruction. Instead, we report *icf* value and reset it. We also printing the currently executing function to the standard output (or file).

Remark

Magic string is attached to the *printf* function to identify the output originating from our inserted code. Filtering the changes in the *icf* from the output helps identify the caller and callee functions.

llvm-pass-icg

The source code for this tool is also publicly available on my GitHub[3].
Next, we provide an example for better illustration.

llvm-pass-icg

The source code for this tool is also publicly available on my GitHub[3]. Next, we provide an example for better illustration.

```c
#include <stdio.h>
void f1() {printf("f1\n");}
void f2() {printf("f2\n");}
void f3(){printf("f3\n");}
int main() {
    void (*fp)();int var; scanf("%d", &var);
    if (var == 1) {fp = &f1;}
    else if (var == 2) {fp = &f2;}
    else {fp = &f3;}
    (*fp)(); // indirect call here
}
```

Compile the file with our dynamic lib and run it.

```
echo 1 > input.txt
./main < input.txt | grep "__ICG_STDOUT__"
```

Example output:

```
__ICG_STDOUT__{main}{f1}
```

Construct More Complete Graphs

Recap four parts: ~~ModifiedDistanceCalculator~~, ~~TargetGeneration~~, ~~ModifiedInstrumentor~~, and IterationGraphConstruction. A prototype of iterators, without the involvement of simulated annealing, has been implemented.

IterationGraphConstruction

The tool will generate more complete graphs after a fixed number of iterations. Let's consider the *CG* and the above program as examples. Before the iteration process, the *CG* appears as follows:

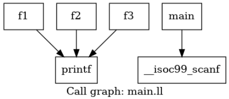


Figure: CG before the iteration

After 10 iterations, the CG probably appears as follows:

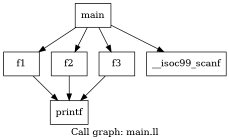


Figure: CG after the iteration

Table of Contents

Future Work

Future endeavors will concentrate on the implementation and evaluation of the proposed architecture.