INSTRCR: Lightweight instrumentation optimization based on coverage-guided fuzz testing

Cao Zhang
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, China
e-mail: 271038091@qq.com

Wei Yu Dong, Yu Zhu Ren
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, China

***Abstract*—In Fuzzing facing binary coverage, the main role of instrumentation is feedback code coverage (in the case of Fuzz for binary, instrumentation can provide coverage information, which plays an important role in guiding the operation of seeds in Fuzz) . The current instrumentation optimization technique mainly relies on the control flow graph (CFG) to select key basic blocks at the basic block level, but the accuracy of this method is not high enough. Considering that the actual path in the actual operation of the binary may be different from the CFG generated in advance, this paper is based on the indirect jump that cannot be accurately analyzed in the CFG, and some of the basic blocks that can be optimized for high-frequency interpolation. According to the algorithm proposed in this paper, The combination of static analysis and dynamic analysis is used to continuously adjust and select key basic block nodes for instrumentation. It is verified by experiments that this kind of instrumentation method can effectively improve the coverage rate and reduce the overhead, and provide effective guidance for Fuzzing, which can effectively reduce the Fuzzer's false negatives.**

Keywords-instrumentation; binary; fuzzing; control flow graph

I. INTRODUCTION

Fuzz testing based on coverage feedback has been considered an effective and important branch of fuzz testing technology in recent years. It usually performs instrumentation on the target program to obtain path coverage information. If a new path is found, the current test case is added to the seed pool as a seed for subsequent fuzzing, otherwise it is thrown away. Because of the feedback of coverage, fuzz testing can continuously improve the coverage of the path, which greatly increases the probability of triggering bugs[1][2][3]. The process is shown in Figure 1.

Feedback on coverage is therefore critical for such Fuzzers. In general, coverage feedback is by inserting detection code into the target program so that Fuzzer can know where the input is running in the program and know which program paths have been executed and which paths have not been executed. However, the behavior of the instrumentation is not perfect. It provides us with coverage feedback information, which brings additional overhead and inevitably reduces the speed of the Fuzzer. Because the program instrumentation will need to run additional instrumentation detection code than the program without the instrumentation, and the more instrumentation points, the more the instrumentation code, the greater the overhead cost.

Therefore, reducing the overhead of instrumentation while improving coverage information is an unavoidable contradiction. Most existing techniques for reducing the cost of instrumentation either rely on simple heuristics or become inefficient when applied directly to overlay-guided fuzzing[4][5][6].INSTRIM applies a CFG-sensitive instrumentation algorithm, which can reduce the overhead cost by selecting some basic blocks for instrumentation while improving the acceleration for fuzzing [7]. But this method is based on the assumption that the CFG is 100% identical to the actual path of the program, that is, it ignores the fact that the CFG may not match the real path of the program.

Therefore, we have studied how to select key instrumentation points more accurately and less accurately, taking into account the impact of the inaccuracy of existing CFG recovery techniques on the accuracy of instrumentation. We designed and implemented CFG sensitive instrumentation optimization algorithm and dynamic adjustment mark algorithm.

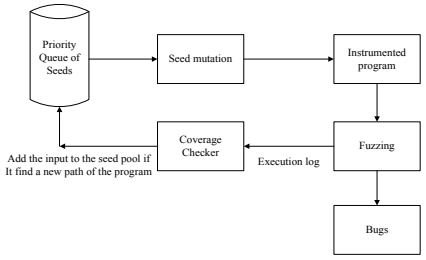

Figure 1. Instrumentation and Fuzzing process.

II. BACKGROUND AND RELATED WORK

A. Coverage-guided fuzzing test

The fuzzing test technique based on coverage feedback is usually a gray box fuzzing test, which performs instrumentation on the target program to obtain useful information such as path, code, edge, and the like. If it finds that a seed has acquired a new path, edge, or overwrites more

978-1-7281-2871-9/19/$31.00 ©2019 IEEE

code on the run, or it may have been running for a longer period of time, the current test case is considered to be likely to trigger a bug. The seed will be added to the seed pool as a seed for subsequent fuzzing, otherwise the next seed test will be taken directly. Because of the constant feedback of useful information such as coverage, such fuzzing can detect bugs more effectively than fuzzing techniques that do not use this strategy. The fuzzing test based on coverage feedback has Randoop [8] in the early days. It does not aim to discover vulnerabilities, but its method is similar to the method we introduced today. The more popular tool is AFL [9] , AFL The instrumentation strategy used is full instrumentation and random instrumentation. The cost is high in the case of full instrumentation, and sufficient code coverage information cannot be guaranteed in the case of random instrumentation.

B. Instrumentation for Coverage-guided Fuzz Testing

The Dyninst API uses the properties of the dominating tree and a heuristic algorithm to select node instrumentation: all leaf nodes and some non-leaf nodes, but this method basically selects all the basic blocks, resulting in low efficiency [5]. Ohmann et al. proved that determining the optimal coverage pull detection problem is an NP-hard (non-deterministic polynomial), and proposed a solution based on mixed integer linear programming and two approximation methods, but this method needs to scan the program in order to know all execution paths [6] in advance for static analysis. INSTRIM uses the incoming edge knowledge to distinguish different paths by distinguishing the marked basic blocks by storing the node labels of the last marked basic block in the thread local storage and recording the path segment pairs (the last node label on the overlay map, And the corresponding incoming edge label) to record the code coverage, but it can only instrument the source code, andIt is based on the fact that the CFG is exactly the same as the real runtime path of the program.

C. Control flow graph recovery

Many program analysis work relies on CFG [10], which is the basis of much work. At present, the mainstream CFG recovery technologies are IDA pro and Angr. The CFG recovery technology disassembly strategy in IDA pro is conservative, and its code coverage needs to be improved [11]. Angr computes the CFG by simulating each basic block, but encounters a basic block with different representations in different contexts, although setting the context sensitivity parameter to set the caller of the function caller saved in the call stack The number can make the recovered CFG relatively more accurate, but the larger the value, the more the time cost will increase exponentially. On the whole, CFG recovery technology is still not completely accurate, that is, it is completely consistent with the possible real path.

This is because: First of all, for a simple program, it is relatively easy to accurately restore the control flow graph. But for complex, large-scale software, even a software developer can't do it to complete a complete and accurate control flow graph. Because there may be some errors in the developer, although this situation may be rare, we must consider this situation, because the purpose of our Fuzzing is to finally find out the developer's mistakes. Secondly, in the absence of source code, the use of disassembly techniques for large programs to recover control flow graphs involves recovering complex indirect jumps, which have multiple types: Computed, Context-sensitive, and Object-sensitive, these types of indirect jumps, it is difficult to solve all of them effectively. The most advanced control flow graph recovery technologies, such as IDA pro and Angr, have also confirmed our views [11].

```
1  def  mark (G, t , s，  B_Lj ) :
2    blocks={}
3    blocks=binary.identify()
4    G=binary.Angr.Emulate()
5    Lj=binary.Angr.Emulate.Lj()
6    for x in Lj
7      for (i=0,i<=size(blocks),i++)
8        if x in blocks{i}
9          marked+=blocks(i)
10     return marked
11   G = G. cut_off_subgraph (s , t )
12   marked = {v for ( u , v )  in  G.E.backedges }
13   G.E erases out backedges
14   # G becomes DAG
15   T = topolog(G)
16   for x in T :
17     need_marked = False
18     P( x ) ={}
19     for u in G.E.to ( x ) :
20       P( x ) = P( x ) ∪ P( u )
21       for v in G.E.to ( x ) :
22       if u == v : continue
23       if size (common_point ( P(u), P(v))) > 0 :
24       need_marked = True
25     if need_marked or x in marked :
26       marked += x
27       P(x) = {x}
28   return marked
29 def  main (G) : # G := control flow graph
30   marked ={}
31   D = dominator_tree (G)
32   #D. idom ( v ) := immediate dominator of v
33   current _state = G.final_state
34   while current _state != G.initial_state :
35     marked += mark (G, current_state , D. idom ( current_state ) )
36    current _state = D. idom (current _state )
37   return marked
```

Listing 1: Psecudocode of INSTRCR

Therefore, when we use the control flow diagram for instrumentation, we must fully consider the premise that the control flow graph is not accurate. However, the existing instrumentation technique for coverage-guided fuzz testing does not take into account the premise that CFG is inaccurate.

In addition, based on some high-frequency interchangeable plug-in nodes in the program structure (see Section 2, Part 2 for details), We mainly optimize the existing instrumentation technology in two aspects. First, based on the inaccurate CFG, how to accurately select the instrumentation point statically, and secondly, dynamically adjust the instrumentation point in real time.

III. ALGORITHM

Marking the instrumented nodes on the CFG is summarized as distinguishing between different path problems, and distinguishing different paths can be represented by marking the least points with different paths.

Our optimization algorithm utilizes general program structure knowledge [7]. Before applying our optimization algorithm, we need to do the following work: firstly, static analysis of the program generates CFG, then establish the dominant relationship of CFG basic blocks, mark the basic blocks with back edges, erase the edges, and integrate CFG and The dominant relationship between the basic blocks divides the CFG into subgraphs, and performs topological sorting on all the basic blocks in the subgraph. The basic blocks sorted by the topology are sequentially used to calculate whether the node needs to be marked.

INSTRCR mainly consists of two parts:

A. CFG-aware Instrumentation Optimization Algorithm

We used Angr's CFG.Emulate technique when refactoring CFG. An indirect jump list Lj, Lj in CFG.Emulate terminates runtime maintenance saves unresolved indirect jumps. We believe that these unresolved indirect jumps are easily overlooked and valuable. Based on this, the first point optimization is generated: firstly, the basic blocks containing the unresolved indirect jumps are recovered by using Angr, and then the basic blocks are initially marked(line 3 - 10). Shown. Then, the sub-picture is divided into the CFGs that have been initially marked (line 29 - 37), and the sub-pictures are finally marked (line 11 - 28).

The pseudo code is shown in Listing 1.

B. Real-time dynamic adjustment mark algorithm

By calculating the execution frequency of the runtime basic block, the basic block with high execution frequency is replaced in the alternative case with the basic block with low execution frequency, so that we can optimize the selection of the marked basic block. Based on this, we made a second improvement, as shown in the pseudo code list 2.

The structure shown in Figure 2 is a common branch structure in the general program. After the CFG split subgraph, if the subgraph with the structure shown in Fig. 2 is detected, we will monitor the Seed through the B2 path through our previous marker points. After the frequency of the B2 path, the marker point adjustment is performed according to the comparison of the Seed through the B2 and B3 frequencies.

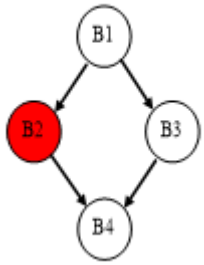

Figure 2. A common program structure

```
1  If ( P=P_B2/P_B3 >X)
2    If (B2 in marked)
3      marked=Marked.delete(B2)
4      marked=marked.add(B3)
5  else
6  continue
```

Listing 2

Taking Figure 2 as an example, suppose that the instrumentation node we originally selected is B2. By marking B2, we can identify two different paths: the path of B1-B2-B4 is represented by {B2}, the path of B1-B3-B4 is represented by { } (empty set). For example, in the Fuzzing process, if there are 99 seeds in 100 seeds passing through B2 and only one seed path passing B3, then we will consider B2 to be a high-frequency instrumentation trigger basic block. There will be a significant overhead in B2 instrumentation, because in 100 runs, there will be 99 runs running the instrumentation code here. At this point, changing the instrumented node to B3 will greatly reduce the cost of the instrumentation. Since the instrumentation node is changed to B3, in the same 100 runs, only one Fuzzing will run the detection code of this instrumentation node.

Although in theory, depending on the seed, the execution path may be different, and the execution frequency of each basic block is unstable. However, the results of our experiments show little difference.

Assume that P_{B2} is the probability that the seed passes through B2, and P_{B3} is the probability that the seed passes B3. Our dynamic adjustment labeling algorithm will adjust the basic block mark after $P=P_{B2}/P_{B3}$ is certain after the program runs X times.

$$\begin{cases} P_{B2}+P_{B3}=100\% \\ P_{B3}>P_{B2} \end{cases}$$

The cost reduction efficiency can be derived from $(P_{B2} - P_{B3}) / P_{B2}$ with a minimum cost reduction of 3.9% and a maximum of 100%.

We set the program running times X and P as adjustable according to the requirements, that is, every X times, it is dynamically adjusted according to the probability of each marked point until it is optimal.

IV. IMPLEMENT

Under the Linux system, combined with QEMU and Angr, on the Fuzz framework made by our team, we implemented the INSTRCR instrumentation algorithm with python.

A. Fuzz framework

Fuzzer is written by Python and is mainly used to test binary. The code coverage rate is used to select a high-level test sample to trigger the vulnerability. It maintains a pool of test cases, also known as a seed pool. Each time you select a file from the pool, it will be mutated a lot, then use the mutated file as the input to run the detected program, and finally see if the running result will cause the target to crash, according to the instrumentation collection path information to see if it finds new Paths, etc., if any, treat such seeds as valuable and continue to put them into the seed pool for the next round of variation and testing. Variations on the seed include: (1) bitflip; (2) arithmetic; (3) interest; (4) dictionary; (5) havoc; (6) splice.

B. Mark basic Block

GDB is a program debugging tool under UNIX. It can identify the target architecture, set the size and end, and implement basic block markup by setting breakpoints with GDB. Breakpoints are divided into two types: temporary breakpoints and permanent breakpoints. Permanent breakpoints are pre-marked basic blocks with unrecognized indirect jumps and basic blocks with back edges. Temporary breakpoints are other basic blocks.

C. Determine the Basic Block where the Indirect Jump is not Recognized

Angr.CFG.Emulate uses a variety of techniques in generating CFG: (1)enforcement; (2)lightweight backwardsing; (3) symbolic execution; (4) value set analysis. The CFG is restored by these technologies. When CFG.Emulate terminates the run, an indirect jump list L_j is generated. At this time, L_j stores the indirect jumps that are not parsed, by comparing each indirect in all basic blocks. The jump instruction can determine the basic block where these indirect jumps are located.

D. Coverage monitoring

Our method differs from AFL in that the paired combination of source basic blocks and destination basic blocks is used to record edges [12]. After we have calculated the basic blocks that need to be instrumented for all the basic blocks, we use GDB to set breakpoints. Our algorithm ensures that all the path information on the CFG can be recorded by the marked basic blocks.

V. EVALUATION

Our algorithm is designed to serve coverage-guided ambiguity, so we selected execution time and labeled basic block scale as indicators to evaluate our algorithm. In order to compare with the previous algorithm, we chose 8 source programs and the binary they generated.

A. Target programs

In order to be comparable with previous algorithms, we compiled and generated binary for libfreetype, libxml2, libcapstone, lame-silent-preset standard, objdump-dg, libpypy, coreclr, and libwireshark.

B. Execution time

The execution time is the average value obtained by testing 8 execution programs with AFL, INSTRIM, and INSTRCR three times. It should be noted that AFL and INSTRIM test the source code of 8 test cases, and INSTRCR tests the binary corresponding to the source code. Although this comparison does not seem to be very strict, we believe that based on the same program (source code and its corresponding binary), after applying different instrumentation techniques, for our algorithm - INSTRCR comparison, it Still has a certain meaning.

C. Proportion of marked basic blocks

Table 2 shows the proportions of the basic blocks of AFL, INSTRIM, and INSTRCR markers. The data shows that the number of basic blocks marked by INSTRCR in small programs is not much different from that of INSTRIM. The number of basic blocks marked in medium-sized programs is slightly higher than that of INSTRIM. This is mainly because, as the program code increases, complex indirect jumps also occur. As a result, the basic blocks that we mark are also increasing, which is the same as we expected. We believe that the increase of the basic block of the mark does not mean that the value of the instrumentation algorithm is reduced, because reducing the overhead cost is only one aspect of the value of the instrumentation algorithm, and on the other hand, the code coverage. If the increase in the number of marked basic blocks helps to discover new paths or improve code coverage, we believe that such costs are worthwhile.

D. Code Coverage

Our record code coverage is achieved by recording path coverage. The path recorded by INSTRIM is the same as the path when the AFL is fully instrumentation, so we only compare it with the full AFL instrumentation. As shown in Table 3, our algorithm INSTRCR found more paths than in the case of AFL full instrumentation, which was mainly due to the indirect jump basic block that we chose to mark out.

TABLE 1. THE EXECUTION TIME

Program	AFL	INSTRIM		INSTRCR	
	Time (s)	time	speedup	time	speedup
libfreetype	343	281	1.22	283	1.21
libxml2	147	125	1.18	126	1.16
libcapstone	79	73	1.08	73	1.07
lame	2503	2361	1.06	2383	1.05
objdump	1367	1278	1.07	1289	1.06
libpypy	17141	10143	1.69	11427	1.5
coreclr	19602	13902	1.41	15078	1.3
libwireshark	1549	1192	1.3	1290	1.2

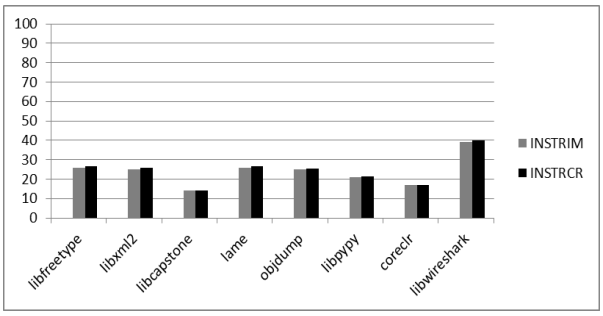

TABLE 2. PROPORTION OF MARKED BASIC BLOCKS

Program	AFL	INSTRCR
libfreetype	1	1.03
libxml2	1	1.02
libcapstone	1	1.01
lame	1	1.09
objdump	1	1.08
libpypy	1	1.09
coreclr	1	1.10
libwireshark	1	1.03

TABLE 3. PATH COVERAGE

VI. CONCLUSION

This paper explores the lightweight instrumentation optimization for coverage-guided fuzz testing. This paper starts from the application basis of general instrumentation - the incompleteness of CFG and the problems in the middle and late stages of instrumentation.

Our experimental results show that our algorithm can shorten the execution time (relative to AFL) while improving the coverage and finding more interesting seeds (relative to INSTRIM) by fully considering the incompleteness of CFG and adjusting the high-frequency basic blocks of the instrumentation, effectively reduce the cost of Fuzz. The contribution of this paper is mainly to provide a new idea for optimizing the instrumentation technology. We hope that the development of the instrumentation technology for coverage-guided fuzzy will be more broad, thus providing better service for coverage-guided fuzzy.

REFERENCES

[1] Klees G, Ruef A, Cooper B, et al. Evaluating fuzz testing[C]//Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2018: 2123-2138.

[2] Hongliang L , Xiaoxiao P , Xiaodong J , et al. Fuzzing: State of the Art[J]. IEEE Transactions on Reliability, 2018:1-20.

[3] Manes V J M , Han H S , Han C , et al. Fuzzing: Art, Science, and Engineering[J]. 2018.

[4] T. Ball and J. R. Larus, “Efficient Path Profiling,” in IEEE/ACM International Symposium on Microarchitecture, 1996.

[5] Tikir M M, Hollingsworth J K. Efficient instrumentation for code coverage testing[C]//ACM SIGSOFT Software Engineering Notes. ACM, 2002, 27(4): 86-96.

[6] P. Ohmann, D. B. Brown, N. Neelakandan, J. Linderoth, and B. Liblit,“Optimizing Customized Program Coverage,” in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016.

[7] Hsu C C, Wu C Y, Hsiao H C, et al. Instrim: Lightweight instrumentation for coverage-guided fuzzing[C]//Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research. 2018.

[8] Pacheco C, Lahiri S K, Ernst M D, et al. Feedback-directed random test generation[C]//Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, 2007: 75-84.

[9] M. Zalewski, “American Fuzzy Lop,”http://lcamtuf.coredump.cx/afl/.

[10] Xu L, Sun F, Su Z. Constructing precise control flow graphs from binaries[J]. University of California, Davis, Tech. Rep, 2009.

[11] Yan S, Kruegel C, Vigna G, et al. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis[C]// Security & Privacy. 2016.

[12] Website. 2017. American Fuzzy Lop (AFL) Fuzzer. http://lcamtuf.coredump.cx/ afl/technical_details.txt. (2017). Accessed: 2018-06-13