Hybrid Way of Code Coverage Tracking in Fuzz

Hanyi Nie, Xu Zhou and Junnan Zhang
National University of Defense Technology
Changsha, Hunan Province, China
{niehanyi17 & zhouxu & zhangjunnan12}@nudt.edu.cn

Abstract—**In software testing, code coverage can be one of the major metrics for evaluating the effectiveness of a test. Among all existing software testing methods, coverage-guided fuzzing is widely used nowadays, but the way it uses to obtain path coverage is mostly based on code instrumentation or emulation. However, a tester cannot take targeted measures if have no information about where the progress of the test is stuck. This paper proposes a method to record precise code coverage in a hybrid way which combining static program analysis and dynamic tracing. This work is on the basis of previous work that leverages hardware mechanism (Intel Processor Trace) to collect branch information and implement a tool called CovFuzz. Our approach can achieve an accurate coverage track that can reversibly find the corresponding source code or assembly code to assist program analysis and break through the bottleneck when the progress of software testing gets stuck. Our experiments show that the code coverage can be improved with the help of accurate path information.**

Keywords-fuzz; code coverage; Intel PT; software testing

I. Introduction

Software security is getting more and more attention due to the wide use of software, even little mistake like a wrong letter can make huge lost for users, it's really necessary to test software thoroughly and find vulnerability before it causes real problems. Software testing techniques can be classified according to the knowledge acquired from program, and they are white-box, grey-box and black-box testing separately. The black-box testing knows nothing about the architecture of target program. The tester cannot control program process but to change inputs to observe the differences of outputs to evaluate the effect of execution. In the white-box testing, all the source code information needs to be obtained first, and the code is directly audited by manual or automated tools like pattern matching to detect vulnerability thoroughly, which often causes high overhead. The grey-box testing considers the internal properties of programs while maintaining the simplicity of black-box to make a compromise between cost and accuracy. The coverage-guided fuzzing is a famous one in grey-box testing and uses code coverage information to guide the way to generate input.

One of the most state-of-the-art tools of coverage-guided fuzzing is American Fuzzy Lop(AFL) [1] , which will collect some simple program execution information to decide which new inputs have more potential to find new branches and would be keep for further mutated. But this kind of path record is in a tough way that cannot reflect the precise state of program execution. However, when the test has been executed to a certain extent, it may be difficult to find a new path, and the full progress may be stuck. In this situation, continuing the test would not bring anything useful. But with some more fine-grained coverage information, things will be much more different. One can directly search for the branch which connects executed areas and never executed areas in program and generates an input directly based on the condition to execute the branch to breakthrough this bottleneck.

However, it would be impossible for AFL to record such precise path because its way to track code coverage is to instrument a random number in each basic block when compiling the source code, and to calculate the hash value of branch between two basic blocks like A->B as (1). But there may be collisions due to the limited random range, and the greater size of the program is, the higher possibility of collisions becomes[11]. Therefore, for one thing, the hash value for a branch may not be unique due to the collisions. For another, testers cannot find out which part in the program is corresponding as all we can get is how many times a hash value has been hit in all executions.

$$\mathrm{cur} \oplus (\mathrm{prev} \gg 1) \tag{1}$$

There is a way to avoid the collision due to the random mark of basic block(BB) and keep the relation between BB and its mark by using Intel Processor Trace to get feedback. Intel PT is a new hardware mechanism of Intel Processor[3] that can trace the accurate address of every basic block and detailed control flow information. kAFL uses it to fuzz OS kernels [4] and PTfuzz [2] uses the address of instruction to mark BB instead of random number. But they keep the other mechanism of AFL and the collision in the calculation of branch's hash value cannot be avoided and the relation between BB and mark get lost in the final output.

In summary, this paper makes the following contributions.

- We propose a hardware-based program control flow tracking method to accurately record the code coverage

This work is partially supported by the The National Key Research and Development Program of China (2016YFB0200401), by program for New Century Excellent Talents in University, by National High-level Personnel for Defense Technology Program (2017-JCJQ-ZQ-013) , by the HUNAN Province Science Foundation 2017RS3045.

status of the tested program, to completely avoid collisions and loss of information.

- We propose a method to record the execution status of branches in a hybrid way which combines static program analysis and dynamic execution information. The method can reversibly locate the related basic blocks in the corresponding program source code or assembly code according to the collected information.
- We implement a tool called CovFuzz and conduct experiments to prove that this accurate information record can assist the program analysis when the fuzz process or other kinds of test tool are stuck to quickly break through the bottleneck that block the code exploring and improve code coverage.

II. Background and Related Work

A. *Coverage-guided Fuzz*

Since first developed in early 1990s, fuzzing has evolved much and achieved great success in vulnerability detect. One of the most popular fuzz is coverage-guided fuzz, which can get simple coverage information by using some light weight instrumentation. AFL is a very efficient coverage-guided fuzz tool that enlighten many other works. The way it uses to record coverage information is by setting up a 64KB shared memory called bitmap [4]. Each byte in the bitmap represents a branch, which is updated according to the hash value calculated in (1). The limitation of bitmap size means the number randomly instrumented to basic block should be within the range of 「0,64k」. The work process of AFL shows in Fig. 1. The first step is using the gcc offered by AFL to compile and instrument the source code [3], then mutate the seeds to generate as many as testcases and collect coverage information when executing them. For testcases that can cover new branches will be added to seed pool for further mutation. Most of the improvement works are focused on using code coverage information. AFLFast [7]and Fairfuzz [8] have used the frequency at which the branch is hit, make code that are hard to be executed get more attention. By using coverage, we could also implement a directed fuzz that concentrate to some specified code. The papers [5] and [6] considered the distance between basic blocks to modify the direction of coverage exploration.

However, for programs without source code or cannot be instrumented like OS kernel, AFL would lose its property of coverage-guided and work blind, cause loss of efficiency. The paper[10] first introduced a feedback mechanism based on hardware by using Intel PT to collect execution information and achieved AFL fuzz in the OS kernel. Then we will briefly introduce PT.

B. *Intel Processor Trace*

Intel PT is a part of Intel Architecture that offers control flow tracing relying on hardware facilities with low overhead. It hardly has an influence on the performance of software being traced. Instructions in program that could change the program flow are called Change of Flow Instructions (COFI). The target of where the flow turn to is in stored instruction or in registers.

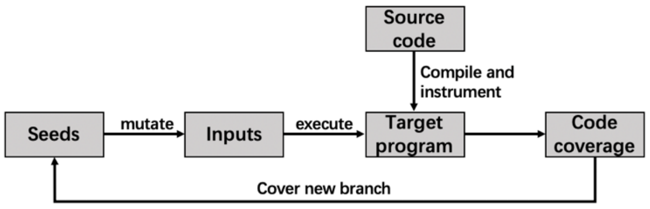

Figure 1. The work process of AFL

According to the addressing mode, COFI are divided in the following three categories:

1) *Direct transfer COFI:* The target is an IP that embedded in the instruction bytes.

2) *Indirect transfer COFI:* The IP is stored in registers or memory location.

3) *Far transfer COFI:* Operations that change the flow are far jumps, like exceptions, interrupts or traps.

To handle different kinds of COFI, the PT will collect execution information in different types of data packets. The most relevant types to our works are Taken Not-Taken (TNT) packets and Target IP (TIP) packets. TNT packets are designed to track the direction of the direct conditional branch by record whether the jump is Taken or Not-Taken. If taken, combining with the IP embedded in instruction, we could know whether the target or the next instruction in order would be executed. TIP packets record all the IP of indirect branches or events that can't get target IP from assembly code.

PTfuzz is a grey-box fuzzing approach which leverages PT to collect branch information and uses the start address of a COFI to represent a basic block instead of instrumentation. It achieves great performance improvement compared to AFL extended with QEMU in the binary-only test. In this paper, we propose a tool based on PTfuzz to further use the information collected in PT and make it possible for testers to have a detailed understanding of execution status.

III. Design

In this section, we will introduce our method and how it assists fuzzing or other kinds of software testing. There are two main parts of CovFuzz: one is fuzz and the other is coverage track. Since the fuzz part is almost the same as PTfuzz, we will introduce how the coverage track works. For the different types and characteristics of instructions and in order to get all the execution information completely, we use a hybrid method combining hybrid static analysis and dynamic tracking to get the accurate coverage.

A. *Static Anaysis*

In assembly code, instructions have no influence on control flow means that they won't cause jump and would be executed in order until meeting a COFI instruction that will change the control flow. A basic block is composed of a COFI with all the non-COFI instructions before it and represented by the address of the first instruction in coverage. Those basic blocks are classified by the COFI, and the number of possible branches is calculated according to the type.

As the number of COFI in different programs differs a lot, a structure with fixed size cannot be worked well. The size would either so large as to be a waste of space or too small to record completely. The size of HashMap is dynamically variable, fitting for recording the coverage information and flexible to adapt to all programs. The initial HashMap is build based on the static analysis of tested programs. For all basic blocks, each has a corresponding structure containing jump information including TargetEdge and NextEdge. Then the structure would be added to HashMap with the address of the first instruction to be its key value. Different types of COFI instructions would have different number of branches.

1) *Direct jump branch:* For conditional ones, there are two possible branches from it, we record the id of next basic block as NextEdge and the operand embedded in COFI instruction as TargetEdge. The total edges plus 2 accordingly. For unconditional ones, it would directly jump to the target edge, therefore the total edges plus 1 and just set TargetEdge.

2) *Indirect jump branch:* As the target address is unknow, we need to set NextEdge and record the target while in execution.

3) *Far jump:* Target is unknown and next address is unreachable. The direction would be in register or memory.

Once a COFI instruction is being decoded, the structure of this basic block is added to HashMap. The HashMap would be updated in execution to record coverage. The dynamic part is described in the next part.

B. *Dynamic Tracing*

To record the code coverage in testing, the HashMap should be updated in a dynamic way. However, the overhead is much higher when updating a HashMap, which definitely has a side effect on the performance of fuzzing. To balance accuracy and efficiency, the HashMap is only updated when the seed is discovered. The speed of fuzzing is very fast and could run hundreds of testcases each second, the proportion of seeds is so small that such a low frequency of updating HashMap would not have significant influence.

When starting a fuzz, a seed is always necessary and dry-run first to get initial information. Only a newly explored seed could have another run to update HashMap. When a basic block is hit by the seed, the fuzzer would check the node of the last basic block to see whether the address of the current one is in the node. If it is, update its hit number. Otherwise it would be called Hidden edges that are come from indirect jump or far transfer instructions and be stored in a link list of hidden edges. Information of hidden edges could only be known when being executed.

The way used by AFL to record coverage is completely dynamic without any prior knowledge like how many branches are exist, therefore testers cannot know how many branches are waiting to be explored. However, a control flow graph(CFG) build based on static program analysis will miss those hidden edges make the path be incomplete. For example, function A call function B are shown in Fig. 2, the control flow should be basic block chain 1->2->4->5->4. Actually, this chain would be scattered in the static analysis because the last instruction of function B would be 'ret' and the target address can only get from an execution stack. Thus, the only way to collect completely code coverage is by combining both static analysis with dynamic tracing.

Function A:	Basic block 1
If (FunctionB()){	Basic block 2
FunctionC();}	Basic block 3
Function B:	Basic block 4
Return a;	Basic block 5

Figure 2. Example of Basic Blocks Structure in a Function Call

TABLE I. RESULTS OF RUNING TEST CASE IN COVFUZZ AND AFL

tool	*Covered seeds*	*Covered branches*	*Covered condition*	*New seeds*	*New branches*
CovFuzz	45	55	2	11	7
AFL	16	48	1	6	8

C. *Case Study*

To illustrate how CovFuzz works, we write a test program and run it on AFL and CovFuzz with the same seed. The test program is simple and reads text from .txt file to see if there are strings compared to "0", "ab", "crash". Both two tools get stuck after one hour and cannot make any progress in two hours. The detail is shown in Table. 1. The first two columns mean the number of the seeds and branches found when each tool is stuck respectively. The third column lists the number of conditions that have been satisfied. We analyze the coverage information about the missed branches and compare them with the instructions in assembly code. The result shows that in AFL only "0" has been satisfied while CovFuzz only misses "crash". Then we manually generate a testcase that meet the criteria. The last two columns are the number of the newly found seeds and branches. This test program is very small with dozens of code lines in total, so the cost of manually analysis is much lower than running fuzz for countless hours. To verify the availability of our work, we test it on some real-world program in the next section.

IV. EVALUATION

In this section, we will discuss the result of using our tools on real-world programs. We only test it on our work because the major contribution of the work is accurate coverage and its help in testing. This tool is based on PTfuzz and the previous work has proved that PTfuzz has a relatively higher code coverage than AFL[2]. Furthermore, we have illustrated how this work could help other test tools in the last section by comparing our tool with AFL.

We chose four binaries to test, including djpeg and jpegtran from libjpeg 9c, pngfix from libpng 1.6.35 and tiffinfo from libtiff 4.0.9. We tested them all for at least 24 hours. Due to the constraints of computation resources and time, we only waited three hours or a little bit more after the progress be struck if the running time was out of 24 hours. The results are shown in Table. II. The first column lists the number of visible edges in the binary, we could get it after building the HashMap at the

TABLE II. RESULTS OF RUNING REAL WORLD PROGRAM AND COVERAGE IMPROVEMENT

Target binary	*Visible branches*	*Covered seeds*	*Covered branches*	*Time with no find*	*Check points*	*No.1 newly covered branches*	*No.2 newly covered branches*	*No.3 newly covered branches*
djpeg	811	56	109	4 hours	3	47	19	65
jpegtran	1367	26	84	3 hours	3	56	50	87
pngfix	1432	276	751	3 hours	3	14	19	24
tiffinfo	453	12	28	15hours	3	5	38	68

begin of the test. The next three columns contain information about execution before human intervention. The covered seeds and covered branches represent the number of seeds and branches found before stuck. The covered branch excludes hidden edges. Time with no find means how long has the progress been in stuck. After a test to binary gets stuck for hours, we would analyze the coverage information got from CovFuzz to see branches that are never been hit during the test. Normally, the number varies by binary size but would be much more than a hundred. We randomly chose three branches that connect reachable basic block and untouched one as breakpoints to overcome the bottleneck for each binary.

We selected and analyzed 12 breakpoints to figure out why those branches are never been hit in total. After associated the address with the assembly code, we observed that arguments set in the command line or the magic number to mark a special kind of inputs could hinder most testings to go deeper. We modified the inputs and changed the argument value according to the condition that blocks the corresponding breakpoint to see the coverage improvement. For each breakpoint, we only run it for 2 hours due to the time limitation. Then we manually counted the newly found branches and listed them in the last three columns in Table. II. The findings of solving different bottlenecks are listed separately with the serial number. However, the coverage of using different arguments could not be compared directly in bitmap. Our work shows exactly which branch is missed. Testers could compare the coverage of different tests and remove duplicate ones to get the coverage in total.The results of the experiment show that different ways to modify input or argument take different improvement.

Then we analyzed the CFG build according to information in HashMap, found out that changes of arguments could take much improvement. However, there are still missed branches even after trying every argument. Some branches could only be hit when under the premise of a specific group of arguments. We could get all the valuable sets by solving the breakpoints. For example, in the initial test of tiffinfo, there are dozens of basic blocks never been touched. We located those basic blocks and found that they belong to a same function which would only be called when three specific arguments are used together. We could hardly find this set by guessing.

V. DISCUSSION

The foremost limitation of CovFuzz is the fact that we do not know which breakpoint is capable of exploring more branches than others. The resource should be given to where we can gain more to discover more untouched code within limit time. This inspires us about future work to design an algorithm that could prioritize those branches lead to more unexplored branches and report them to testers. Another limitation of our work is that testers are hard to figure out what blocks the test when compiling the target program without using the -Wall argument. The magic numbers are difficult to be discovered when completely depending on assembly code.

VI. CONCLUSION

In this paper, we present CovFuzz, a binary compatible fuzz-testing tool featuring accurate coverage information collection capability. We analyze the drawbacks of previous works on coverage accuracy and demonstrate that accurate path trace could significantly help path exploration. Our work uses a hybrid way that combines static analysis with dynamic path tracing to collect code coverage and implements precise path coverage tracking by taking full advantage of Intel Processor Trace. We test it on a testcase and four real-world programs, proving that our work can help other test tools to cover more code.

REFERENCES

[1] M. Zalewski. American Fuzzy Lop (AFL) Fuzzer. Accessed: Jun. 1, 2019. [Online]. Available: http://lcamtuf.coredump.cx/afl

[2] Zhang G, Zhou X, Luo Y, Wu X. "PTfuzz: Guided Fuzzing with Processor Trace Feedback,"IEEE Access,vol. 6, 2018, pp.37302-37313.

[3] IntelCorporation.IntelProcessorTrace.Accessed:May.21,2019.[Online]. Available: https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing

[4] M. Zalewski. American Fuzzy Lop (AFL) Fuzzer-Technical Details. Accessed: Jan. 1, 2018. [Online]. Available: http://lcamtuf.coredump.cx/afl/technical_details.txt

[5] M.Böhme,V.-T.Pham,M.-D.Nguyen,andA.Roychoudhury,''Directed greybox fuzzing,'' in Proc. 24th ACM Conf. Comput. Commun. Secur. (CCS), 2017, pp. 1–16.

[6] Chen, H., Xue, Y., Li, Y., Chen, B., Xie, X., Wu, X., & Liu, Y. (2018, October). "Hawkeye: towards a desired directed grey-box fuzzer," In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.ACM,2018, pp. 2095-2108.

[7] Lemieux C, Sen K. "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage,"In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, 2018, pp.475-485.

[8] M. Böhme, V.-T. Pham, and A. Roychoudhury, ''Coverage-based greybox fuzzing as Markov chain,'' in Proc. ACM SIGSAC Conf. Comput. Commun. Secur., 2016, pp. 1032–1043.

[9] Zhao, L., Duan, Y., Yin, H., Xuan, J. "Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing," In Proceedings of the NDSS, 2019,pp.1-1.

[10] Schumilo S, Aschermann C, Gawlik R, et al. "kAFL: Hardware-Assisted Feedback Fuzzing for {OS} Kernels," In Proceedings of the 26th USENIX Security Symposium Security, 2017, pp.167-182.

[11] Gan, Shuitao, et al. "CollAFL: Path sensitive fuzzing." 2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018, pp.679-696.