ATLANTIS PRESS
Advances in Computer Science Research (ACSR), volume 76
7th International Conference on Education, Management, Information and Mechanical Engineering (EMIM 2017)

Improvement of Seed Selection Strategy for Graybox Fuzzing

Huabin Tang[1, a] and Wei Wang[2, b]
[1]Capital Normal University, Beijing 100048, China
[a]tanghuabin@ict.ac.cn, [b]wangwei@ict.ac.cn

Keywords: Software security; Fuzzing; American fuzzy loop; Operation system; Performance;

Abstract. According to AFL(American fuzzy lop) author, the famous hacker Michal Zalewski (lcamtuf)said that most of the current bugs were found by the fuzzing, rather than symbolic execution and other technical analysis based on the discovery of the program. The reason is that fuzzing is faster (no analysis program is needed, no need to constrain), and more extensible (the effect of path explosion is smaller). At present, the commonly used fuzzing tool AFL is a security-oriented fuzzing device, which uses a new compile-time instruments and genetic algorithms to automatically find clear and interesting test cases, triggering the target binary file in the new internal state. In the process of using AFL, we found that its genetic algorithm can not guide the direction of fuzzing. Through the design to control the number of variation of each fuzzing seed, to guide it to the low frequency path direction variation, could get a better fuzzing effect. In the same time, the improved AFL was 3 times faster than the old AFL.

Introduction

Fuzzing is a type of safety test between a complete manual penetration test and a complete automated test [1]. It takes full advantage of the capabilities of the machine: randomly generating and sending data, and at the same time trying to introduce security experts' experience into security. As an important fuzzing tool, the AFL incorporates an extremely simple but absolutely reliable, pin-oriented genetic algorithm that captures the branch (edge) coverage and the run-time branch execution count by writing some of the instructions during the compilation [2]. When a mutated input produces an execution trace containing a new path, the corresponding input file will be saved and used to the next new fuzzing process. For those input seed who doesn't have a new path, even if their instrumentation output mode is different, will be discarded. This algorithm is effective but also inefficient, because in most cases, each time you select the next seed input only in the marked as the input selection. The so-called favorite means that the input corresponds to the presence of such a fragment on the execution path, and all other inputs that can be executed to its seed input do not have the input to execute fast or the input volume is minimal. Therefore, the number of times each seed is input to fuzz is determined by the execution time, the path fragment coverage, and the time taken to generate the new input, that is, regardless of the number of times the route was selected before the seed input and the number of times the path was executed[3].The experience shows that if fuzz input is selected when the number of times previously selected or the implementation of the lower frequency of the path will be able to improve the efficiency of the test. If an algorithm can be improved so that the AFL can take into account the number of times the path is selected before the seed is entered before the selection of the seed for the next round and the number of times the path has been executed by the input will increase the efficiency of fuzzing.

Fuzzing

Fuzzing is not really a new technology. It was discovered in 1989 by Professor Barton Miller of Madison University in Wisconsin. Because people are currently focusing on developing more secure software, fuzzing is more widely used and become a recognized code test method. In the software testing process [3].A program is randomly generated by a large number of data validation, known as fuzzing. If a program fails to respond to any of these data, it begins to clash, lock,

Copyright © 2017, the Authors. Published by Atlantis Press.
This is an open access article under the CC BY-NC license (http://creativecommons.org/licenses/by-nc/4.0/).

consume a lot of memory, or generate uncontrollable program errors, and the developer knows that there is a bug somewhere in the code[4]. The implementation of fuzzing is a simple process:
1. Prepare a correct file in the inserted program.
2. Replace some parts of the file with random data.
3. Open the file with the program.
4. Observe what has been destroyed.

American fuzzy lop

AFL is a violent fuzzer that captures the branch (edge) coverage and the run-time branch execution count by writing some of the instructions at compile time. The instructions inserted at the branch point are as follows:

1) cur_location = <COMPILE_TIME_RANDOM>;
2) shared_mem [cur_location ^ prev_location] ++;
3) prev_location = cur_location >> 1;

The variable cur_location identifies the current basic block.Its random identifier is generated at compile time[5]. Variable shared_mem [] is a 64 kB shared memory region. Every byte that is set in the array marks a hit for a particular tuple (A; B) In the instrumented code where basic block B is followed by basic block A. The shift is for the difference from block A to block B and from block B to block A.
The AFL uses the information recorded by the plunger to determine which generated inputs are kept blurred and which inputs are entered into the next blur and how long[6].The algorithm used is as follows:

```
Input: Seed Inputs S
1: T' = Ø
2: T = S
3: if T = Ø; then
4:         add empty file to T
5: end if
6: repeat
7:       t = chooseNext(T )
8:       p = assignEnergy(t)
9:       for i from 1 to p do
10:             t'= mutate input(t)
11:             if t'crashes then
12:                     add t'to T
13:             else if isInteresting(t') then
14:                     add t' to T
15:             end if
16:      end for
17: until timeout reached or abort-signal
Output: Crashing Inputs T'
```

AFL selection strategy:

For the path i,f (i) is the number of times the path is executed by the input that has been generated, s (i) is the number of times the seed corresponding to the path is entered; The energy of path i is a function of these two concepts. The AFL maintains a mapping of the path segments from the input to the input, identified by the cksum. S (i) and f (i) can modify the AFL to dynamically perform statistics at the time of program execution, and also maintain a mapping of cksum (i) to f (i). AFL In most cases, each time the next seed is selected, it is selected only in the input marked as favorite. The so-called favorite means that the input corresponds to the presence of such a fragment on the

execution path, and all other inputs that can be executed to its seed input do not have the input to execute fast or the input volume is minimal[7]. The number of times each seed is entered fuzz is determined by the execution time, the path fragment coverage, and the time it takes to generate the new input, that is, regardless of s (i) and f (i). If the newly generated input has experienced a new path fragment or the number of times the same fragment has experienced more than the number of existing input experiences, the new input is considered interesting and the input is placed in the input queue[8].

In the process of using AFL, we found that most of the input is performed with a few paths (called high-frequency paths), such as most of the generated inputs that perform the path of the failure of the input legitimacy check. Based on this, we have proposed some improved strategies:

For the strategy of selecting the next seed input, two strategies are proposed: one that takes precedence over the input of path i of s (i) (path i is the path of the current input execution); one is to give priority to f (i) The minimum execution path i is entered[9]. When selecting the favorite input for each path fragment, use policy 1, if more than one, apply policy 2; if more than one, use AFL's policy. The same process is used when selecting the next favorite input for the current input[10].

Experimental results:

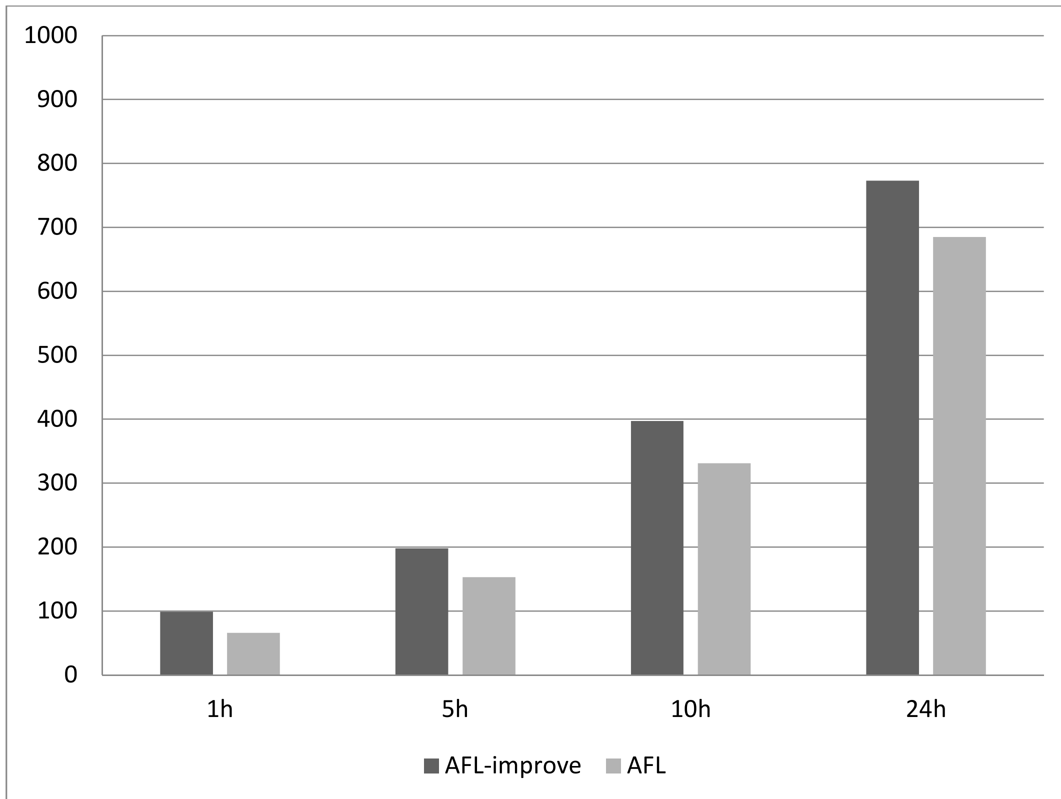

Figure 1. AFL-improved and AFL fuzz crushes over time

Summary

In the above figure, the horizontal axis represents the time used for fuzzing, the vertical axis represents the number of crashes out of fuzz, the test object is LibTIFF-3.8.2. In the same time ,the AFL-improve strategy significantly better than the original strategy , found more crashes. As we have shown in this experiment, in this work, We enhance the effectiveness and efficiency of AFL in producing crashes, as evidenced by our experiments and those of our collaborators. AFL-improve our extension of AFL exposes an order of magnitude more unique crashes than AFL in the same time budget.

References

[1] F. Bellard. Qemu, a fast and portable dynamic translator. In Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC' 05, pages 41–41, 2005.

[2] S. Brin and L. Page. The anatomy of a large-scalehypertextual web search engine. In Proceedings of the Seventh International Conference on World Wide Web7, WWW7, pages 107–117, 1998.

[3] R.S. Sandhu, P. Samarati. Access control: principle and practice[J]. IEEE Communications Magazine, 2002, 32(9): 40-48.

[4] F. Nielson, Hanne Riis Nielson, Chris Hankin. Principles of Program Analysis[M]. New York City, USA: Springer. 2005.

[5] Monirul I. Sharif, Wenke Lee, Weidong Cui, Andrea Lanzi. Secure in-VM monitoring using hardware virtualization[C]. Proceedings of the 16th ACM conference on Computer and communications security. USA. 2009: 477-487.

[6] Thiago Mattos Rosa, Altair Olivo Santin, Andreia Malucelli. Mitigating XML Injection 0-Day Attacks through Strategy-Based Detection Systems [J]. 2012, 11(4): 46-53.

[7] S. Kirkpatrick, C. Jr. Gelatt, and M. Vecchi. Optimization by simulated annealing. Science, 220(4598):671–680, 1983.

[8] George C. Necula, Scott McPeak, Westley Weimer. CCured: type-safe retrofitting of legacy code[J]. ACM SIGPLAN Notices - Supplemental issue. 2012, 47(4): 74-85.

[9] Ping Chen, Yi Fang, Bing Mao, Li Xie. JITDefender: A Defense against JIT Spraying Attacks[C]. IFIP International Information Security Conference. USA. 2011: 142-153

[10] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012, pages 166–176, 2012.