Learn to Accelerate Identifying New Test Cases in Fuzzing

Weiwei Gong(✉), Gen Zhang, and Xu Zhou

School of Computer, National University of Defense Technology, Changsha 410073, China
IssacGong@outlook.com, zhanggen12@hotmail.com, zhouxu@nudt.edu.cn

Abstract. Fuzzing is an efficient testing technique to catch bugs early, before they turn into vulnerabilities. Without complex program analysis, it can generates interesting test cases by slightly changing input and find potential bugs in programs. However, previous fuzzers either are unable to explore deeper bugs, or some of them suffer from dramatic time complexity, thus we cannot depend on them in real world applications. In this paper, we focus on reducing time complexity in fuzzing by combining practical and light-weight deep learning methods, which fundamentally accelerate the process of identifying new test cases and finding bugs. In order to achieve expected fuzzing coverage, we implement our method by extending stage-of-the-art fuzzer AFL with deep learning methods and evaluate it on several wide-used and open source executable programs. On all of these programs, efficiency of our method is witnessed and significantly better outcomes are generated.

Keywords: Accelerate · New test case · Fuzzing · Deep learning Security

1 Introduction

Security of networks and softwares is attracting increasing attention these days. Despite efforts to increase the resilience of software against security flaws, vulnerabilities in software are still commonplace [1]. During the past few decades, security specialists and researchers spared no effort in finding vulnerabilities and fixing them. However, many classes of vulnerabilities, such as functional correctness bugs, are difficult to find without executing a piece of code [2]. With regard to the problem of code executing, there has been much debate about the efficiency of symbolic execution versus more lightweight fuzzers [3]. Symbolic execution tools, such as KLEE, EXE and DART, are capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs [2,4,5]. While fuzzing is the process of finding security vulnerabilities in input-parsing code by repeatedly testing the parser with modified, or fuzzed, inputs [6].

G. Wang et al. (Eds.): SpaCCS 2017, LNCS 10656, pp. 298–307, 2017.
https://doi.org/10.1007/978-3-319-72389-1_24

Symbolic execution is very effective because each test case typically execute the target program along a certain path [7]. However, this effectiveness comes at the cost of spending significant time doing program analysis and constraint solving. It triggers a large number of paths in the target program and will result in path explosion [8]. However, with regard to fuzzing, today most vulnerabilities are exposed by particularly lightweight fuzzers that do not leverage any program analysis [9]. It turns out that even the most effective symbolic execution is less efficient than fuzzing if the time spent generating a test case takes relatively too long [10]. For the above reasons, in this paper, we give up classic symbolic execution and focus on extending a stage-of-the-art fuzzer American Fuzzing Lop (AFL) [11].

There are three main types of fuzzing techniques in use [12]: black-box random fuzzing [13], white-box constraint-based [14] fuzzing and grey-box fuzzing [15]. Black-box fuzzing is a technique of software testing without any knowledge of the internal architecture of the target program. It only examines the fundamental aspects of the system, which treats the software as a Black Box [16]. White-box fuzzing is based on analysis of internal structure of the target program and is very effective and efficient in validating design and assumptions. White-box fuzzing is performed based on the knowledge of how the system is implemented [17]. Grey-box fuzzing tests the program with limited knowledge of the structure of an application. It provides combined benefits of black-box and white-box fuzzing techniques and the tester can design excellent test scenarios [15].

However, since grey-box fuzzing is efficient for real world program and its combined benifits, it is widely applied in software testing. In the range of grey-box fuzzing, there have been several well-known fuzzers, such as automatic generation [18], fuzz [19] and semantic model [20]. Although these methods do give up the time-consuming program analysis, they suffer from low testing accuracy: either take a non-crash point as a crash or ignore the real significant crash. Furthermore, existing grey-box fuzzers have been effective mainly in discovering superficial bugs, close to the surface of software, while struggling with more complex ones [21,22]. On top of all these reasons, we focus on extending a state-of-the-art grey-box fuzzer AFL, which employs evolutionary algorithms to operate valid input generation and a simple feedback loop to assess how good an input is [23]. And in previous works, AFL is proved to have high accuracy and able to reveal deeper bugs in programs [3,23].

AFL is a brute-force fuzzer coupled with an exceedingly simple but rock-solid instrumentation-guided genetic algorithm. However, it would take hours and days to find a unique crash with AFL tool. And the main drawback of AFL is that it actually runs the target program for once with the input to decide weather there is a state transition or new state: in other words, the process of identifying new test cases. When the target program is complex and contains millions of parsing or condition code, running the program for once would take considerable time and the whole process of AFL would take hours and days to find a unique crash (we will discuss this in detail in Sect. 2). To tackle this

problem, in this paper, we integrate efficient deep learning methods such as neural networks into AFL to accelerate the process of identifying new tesecases and finding bugs. On all of our experiments, efficiency of our method is witnessed and significantly better outcomes are generated.

The main contribution of this paper is as follows: (1) we apply grey-box fuzzing, which is efficient and easy to implementation; (2) we extend AFL with deep learning methods, which significantly accelerate the process of identifying new tesecases and finding bugs.

2 Background

2.1 American Fuzzy Lop

American Fuzzy Lop (AFL) is a brute-force fuzzer coupled with an exceedingly simple but rock-solid instrumentation-guided genetic algorithm. It uses a modified form of edge coverage to effortlessly pick up subtle, local-scale changes to program control flow [24].

The overall algorithm can be summed up as:

(1) Load user-supplied initial test cases into the queue,
(2) Take next input file from the queue,
(3) Attempt to trim the test case to the smallest size that doesn't alter the measured behavior of the program,
(4) Repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies,
(5) If any of the generated mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new entry in the queue,
(6) Go to 2.

American Fuzzy Lop does its best not to focus on any singular principle of operation and not be a proof-of-concept for any specific theory. The tool can be thought of as a collection of hacks that have been tested in practice, found to be surprisingly effective, and have been implemented in the simplest, most robust way people could think of at the time [9]. Thus extension on AFL can give us a high level platform to start with.

More specifically, when we look deeply into the running process of AFL, we will figure out there are several procedures that are time consuming, which we actually aim at to make improvement. As described above, AFL takes initial test cases into the queue, trim it down, makes mutations, and actually runs the target program for once with the input to decide weather this test case will cause state transition or new state. If this test case do cause state transition or new state, we can identify it as a new test case. When the target program is complex and contains millions of parsing or condition code, running the program for once would take considerable time and the whole process of AFL would take hours and days to find a unique crash. So we extend AFL with deep learning techniques, without actually running the target program, to accelerate the whole process of identifying new test cases and finding bugs, which is significant improvement of original version of AFL.

2.2 Deep Learning

Machine learning and deep learning is under spot light these days, attracting thousands of researchers to work hard on it. Deep and recurrent neural networks (DNNs and RNNs, respectively) are powerful models that achieve high performance on difficult pattern recognition problems in vision, and speech [25]. Deep Learning in Neural Networks (NNs) is relevant for Supervised Learning (SL), Unsupervised Learning (UL), and Reinforcement Learning (RL) [26]. Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction [27].

Deep learning comes with a lot brand new characteristics and the combination of software testing and deep learning would make sense. In this paper, we apply deep learning techniques to improve the performance of AFL and accelerate the process of new test case identification and bug finding, which is a brand-new and interesting research field (to the best of our knowledge so far).

3 Model Overview

In this section, we will describe our proposed model in detail and the whole process of our method will be presented.

3.1 Train the AFL Results with Deep Learning Methods

As described in Sect. 2, when the target program is complex and contains millions of parsing or condition code, running the program for once would take considerable time and the whole process of AFL would take hours and days to find a unique crash. So we focus on eliminating or cutting down these procedures using deep learning techniques.

In AFL, one test case is mutated with certain fuzzing techniques, such as flipping a certain bit, adding a certain integer, replacing with a certain number and so on. In total, there are 16 kinds of fuzzing techniques in AFL: flip2, flip4, flip8, flip16, flip32, arith8, arith16, arith32, int8, int16, int32, ext-UO, ext-UI, ext-AO, havoc and splicing. After one test case is mutated by one of the above techniques, it will be thrown into the target program to run for once to see if there will be a new state of program or state transition. If there is something new, the mutated test case will be identified as a new test case and saved for later use.

More specifically, the content of one test case can always be translated into a sequence of 0s and 1s. Likewise, the fuzzing techniques can also be seen as a 4-bit binary sequence. For example, "0000" stands for "flip2" and "1111" stands for "splicing". And the position and value of mutation can be translated in the same way. At last, weather this is a new test case is simply 1 and 0. Table 1 illustrates how we translate the original input of AFL problem into a binary sequence. For example, we have a string test case "3", and fuzzing technique is "int8", which will add an 8-bit integer to test case. Moreover, the 8-bit integer

is 4 and the position is just the 0 bit. This mutated test case will cause state transition in AFL so it is a NEW test case. Thus we can translate this situation into our defined binary sequence. "00...0011" for test case, "1000" for fuzzing technique, "00...000" for mutation position, "00...00100" for mutation value and "1" for new.

Table 1. Transition to binary sequence

32-bit	4-bit	10-bit	32-bit	1-bit
Test case	Fuzzing techniques	Mutation position	Mutation value	New

After we run AFL for a certain time, the data we collect from the process of AFL will be divided into training data and testing data according to a certain proportion, which will be discussed in detail in Sect. 4. After translating all the training data into binary sequence, the preceding 78-bit can be taken as input and the last 1-bit can be seen as tag. And this binary sequence can perfectly fit into a neural network model, which is a widely-used division of deep learning techniques. A neural network model take a sequence of 1s and 0s as input and simply out put 1 or 0 after calculating in networks. We will use this process to predict weather a mutated test case is a new test case in AFL. More specifically, for training the network, $78 + 1$ bits are needed as training data and as for testing, 78 bits are taken as input, while the last 1 bit is what we need the network to calculate.

3.2 Integrate the Learning Results into AFL

As discussed above, we will eliminate or cut down the time-consuming new test case identifying procedure of AFL by integrating the learning results into AFL. The original version will run the program with the mutated test case, but our new method will simply throw the translated binary sequence into a neural network model and predict weather this test case will result in a new state. To be more specific, when AFL pulls out a test case to make mutation on it and at this time we manually stop further executing on this test case. On the contrary, we put this test case into the neural network in our defined form in Table 1, and make prediction on weather this is a new test case. The result will be sent back to AFL for further running: if this test case is new and will cause state transition, it will be saved for later use; if not, discarded. One thing need to be declared is that: although train the neural network model will take some time, but we only have to train once for one target program. And time complexity of predicting the result in a simple neural network is only $O(n_1 * n_2 + n_2 * n_3 + \ldots)$ [28]. So when the number of nodes is in a small range, such as in our model, the predicting process takes much less time than actually running the target program for once. As a result, our method can accelerate the process of identifying new test case and finding bugs of the original version of AFL.

So the whole process of our method of extending AFL runs as follows:

(1) Run AFL for a certain amount of time to collect data we need for further training. (One thing has to be made clear, compared to running AFL for hours and days, the time spent in this procedure is much less.)
(2) Train the data we collect in our neural network.
(3) Go back to the AFL with the training results to make predictions for each test case, and finally find potential bugs in programs.

Algorithm 1 also illustrates the process of our extended AFL:

Algorithm 1. Our proposed method of extending AFL

Input:
$\mathcal{T}_0$: set of user initial test cases
P: target program
t: fixed time of collecting data
k: k-fold division of data
NN: Neural networks with certain parameters
$T^{'}$: a mutated test case waiting to be identified
Output:
NEW: weather $T^{'}$ is a new test case and cause state transition
Procedures:
1: Run AFL for time t with $\mathcal{T}_0$ as input in P, to collect data D we need for further training
2: Divide D into training data D_{train} and testing data D_{test} with k-fold
3: Train the data D_{train} we collect in the neural networks NN and we get results R
4: Go back to AFL with the training results R to make predictions for test case $T^{'}$
5: return NEW

4 Experiments

In this section, we will present the experiment results of our method and discuss the improvement over the original version of AFL. We ran our experiments on an Ubuntu 14.04 LTS system equipped with a 64-bit 4-core Intel CPU and 32 GB RAM. For our experiment, we select 8 target programs: bmp2tiff, pal2rgb, tiff2pdf, tiff2ps, gif2png, readelf, nm-new and cxxfix [29–31], which are all widely-used in their areas and also have been tested in previous papers [3,23].

4.1 Training Results of Neural Networks

For each of our target programs, we run AFL with it for a certain time and collect over 500K test cases for later training process. And all of them are stored in binary sequence as shown in Table 1. And the neural network we apply is sklearn.neural-network.MLPClassifier [32], which is a widely-used multi-layer perceptron classifier. The hidden layer size is (5, 2) and the solver is “adam”.

Though there are many types of network algorithms, taking time and other factors into consideration, in this paper we only introduce a single neural network to accomplish our experiment. We take training data as input for our neural network and in an acceptably short time it will calculate the results. We obtain training results and they are illustrated in Table 2.

The first column "Target" are the 8 target programs we experiment on. The next column "Total" is the number of test cases we collect for each target program. Then we divide all the collected test case into training and testing data with 5-fold, which means training data is $\frac{4}{5}$ of the total test case and testing data is $\frac{1}{5}$ accordingly. After fitting training data into our neural networks, we can predict a new coming test case. And the third column "Error" is the number of wrong predictions and the last column is predicting accuracy of our neural networks. As Table 2 illustrates, our predicting accuracy is between around 85% to 90%. With such high accuracy, our proposed method of identifying new test cases can accurately predict weather a mutated test case will cause state transition and perform almost the same as the orginal version in accuracy, and perform much better in speed.

Table 2. Training results of neural networks

Target	bmp2tiff	pal2rgb	tiff2pdf	tiff2ps	gif2png	readelf	nm-new	cxxfix
Total	718392	1409466	2797254	1282716	1395384	1768752	566442	543078
Error	12320	32415	48820	17220	28880	29175	11814	9780
Accuracy	89.71%	86.20%	89.53%	91.95%	87.58%	90.10%	87.49%	89.19%

4.2 Time Performance of Our Method

As described above, after training our neural networks with the data we collect, it is time to predict weather a mutated test case will cause state transition in the process of AFL. And the high accuracy of our prediction is definitely a solid basis for later operation. Again, we run the 8 target programs for a certain time, and the only difference compared to the original AFL is that a test case doesn't have to go through the target program to decide weather it will be saved or discarded. And the only thing need to be done is to throw this test case to our already trained neural networks to make predictions. Table 3 illustrates comparison with the original version of AFL.

The first column are the 8 target programs as described above. And the second column is the number of crashes and hangs AFL and our method find in a certain time. So the comparison between AFL and our method is with regard to the time of finding a given number of crashes and hangs. The next two columns are the time AFL and our method need to find those crashes and hangs respectively. (All of our execution time is in accordance with the work of Böhme et al. [3] and Rawat et al. [23].) As presented in the table, our method do lead the original AFL in execution time and on average at 5% in advance. As detailed

Table 3. Time (in minutes) performance of our method

Target	bmp2tiff	pal2rgb	tiff2pdf	tiff2ps	gif2png	readelf	nm-new	cxxfix
Crash/Hang	27/47	39/12	64/8	48/22	17/11	0/3	0/2	0/1
Before	12.03	14.08	21.05	11.05	9.03	167.03	20.10	25.80
After	11.51	13.40	19.82	10.60	8.68	156.84	18.91	24.25
Improvement	4.41%	4.83%	5.94%	4.07%	3.88%	6.10%	5.90%	5.99%

in our theoretical analysis in the previous section, our method does NOT have to actually run the target program and it only depend on our neural networks to make predictions to see weather a test case will cause state transition. Thus both theoretical analysis and experiments show our method does accelerate AFL in the process of identifying new test cases and finding bugs.

5 Conclusion

Facing the severe security issue nowadays, in our proposed method, we first select fuzzing over complicated symbolic execution to fucus on. Additionally, in all the implementation of fuzzing, we concentrate on state-of-the-art AFL to make extension for its promised characteristics. In order to accelerate the process of AFL and identifying new test cases, in this paper, we focus on reducing time complexity in fuzzing by combining practical and light-weight deep learning methods. In order to achieve expected fuzzing coverage, we implement our method by extending AFL with deep learning methods and evaluate it on 8 wide-used and open source target programs. On all of these programs, our method to extend AFL shows prominent improvement over the original version. However, our method does not focus on improving code coverage to find more bugs. Thus in later work, we will try to make research on these fields. Moreover, though there are many types of network algorithms, taking time and other factors into consideration, in this paper we only introduce a single neural network to accomplish our experiment. In the future we would use different neural networks on different situations to evaluate the performance of different neural networks.

Acknowledgments. The work is supported by The National Key Research and Development Program of China (No. 2016YFB0200401).

References

1. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: augmenting fuzzing through selective symbolic execution. In: Proceedings of the Network and Distributed System Security Symposium (2016)
2. Cadar, C., Dunbar, D., Engler, D.R., et al.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, vol. 8, pp. 209–224 (2008)

3. Böhme, M., Pham, V.-T., Roychoudhury, A.: Coverage-based greybox fuzzing as markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1032–1043. ACM (2016)
4. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. ACM Trans. Inf. Syst. Secur. (TISSEC) **12**(2), 10 (2008)
5. Godefroid, P.: Compositional dynamic test generation. In: ACM SIGPLAN Notices, vol. 42, pp. 47–54. ACM (2007)
6. Godefroid, P., Peleg, H., Singh, R.: Learn&fuzz: machine learning for input fuzzing. arXiv preprint arXiv:1701.07232 (2017)
7. Zalewski, M.: Symbolic execution in vulnerability research (2016)
8. Boonstoppel, P., Cadar, C., Engler, D.: RWset: attacking path explosion in constraint-based test generation. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 351–366. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_27
9. Zalewski, M.: American fuzzy lop (AFL) fuzzer-technical details (2016)
10. Böhme, M., Paul, S.: A probabilistic analysis of the efficiency of automated software testing. IEEE Trans. Softw. Eng. **42**(4), 345–360 (2016)
11. Zalewski, M.: American fuzzy lop (AFL) fuzzer (2016)
12. Khan, M.E.: Different forms of software testing techniques for finding errors. Int. J. Comput. Sci. Issues **7**(3), 11–16 (2010)
13. Sutton, M., Greene, A., Amini, P.: Fuzzing: Brute Force Vulnerability Discovery. Pearson Education, London (2007)
14. Godefroid, P., Levin, M.Y., Molnar, D.A., et al.: Automated whitebox fuzz testing. In: NDSS, vol. 8, pp. 151–166 (2008)
15. Khan, M.E., Khan, F., et al.: A comparative study of white box, black box and grey box testing techniques. Int. J. Adv. Comput. Sci. Appl. (IJACSA), **3**(6) (2012)
16. Khan, M.E.: Different approaches to black box testing technique for finding errors. Int. J. Softw. Eng. Appl. **2**(4), 31 (2011)
17. Khan, M.E.: Different approaches to white box testing technique for finding errors (2011)
18. Bird, D.L., Munoz, C.U.: Automatic generation of random self-checking test cases. IBM Syst. J. **22**(3), 229–245 (1983)
19. Forrester, J.E., Miller, B.P.: An empirical study of the robustness of windows NT applications using random testing. In: Proceedings of the 4th USENIX Windows System Symposium, Seattle, pp. 59–68 (2000)
20. Offutt, A.J., Hayes, J.H.: A semantic model of program faults. In: ACM SIGSOFT Software Engineering Notes, vol. 21, pp. 195–200. ACM (1996)
21. Clark, S., Frei, S., Blaze, M., Smith, J.: Familiarity breeds contempt: the honeymoon effect and the role of legacy code in zero-day vulnerabilities. In: Proceedings of the 26th Annual Computer Security Applications Conference, pp. 251–260. ACM (2010)
22. Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., Whelan, R.: LAVA: large-scale automated vulnerability addition. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 110–121. IEEE (2016)
23. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: Vuzzer: application-aware evolutionary fuzzing (2017)
24. Zalewski, M.: American fuzzy lop (AFL) fuzzer-readme (2016)
25. Sutskever, I., Martens, J., Dahl, G.E., Hinton, G.E.: On the importance of initialization and momentum in deep learning. ICML **3**(28), 1139–1147 (2013)

26. Schmidhuber, J.: Deep learning in neural networks: an overview. Neural Netw. **61**, 85–117 (2015)
27. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436–444 (2015)
28. Orponen, P.: Neural networks and complexity theory. Math. Found. Comput. Sci. **1992**, 50–61 (1992)
29. Welles, M., Warmerdam, F., Kiselev, A., Kelly, D.: Tag image file format. http://www.libtiff.org/
30. GNU. GNU binutils. http://www.gnu.org/software/binutils/
31. Raymond, E.S.: GIFs to PNGs. http://www.catb.org/esr/gif2png/
32. sklearn. sklearn.neural_network.mlpclassifier. http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html