Neural Fuzzing: Software-Agnostic Line Coverage Improvement using AFL Fuzzing and Neural Networks pipeline

E6121 Spring 2018
I. Michael, Y. Schaal, J. Yang
Columbia University

Abstract
Debugging has a huge role in software development cycles. Fuzzing is an effective software testing technique to find bugs. Up to 50% of the development time is dedicated to writing test cases and get more unique crashes and code coverage, [2]. Real-world software are very complex, has massive size and multiple modules. Modern fuzzers are not very effective in exploring bugs that lie deeper in the execution and usually hit to a certain threshold where no new bugs are being discovered, [3].

In this paper, we present a combination of a Neural Network architecture and Fuzz testing that doesn't require any prior knowledge about the given software or its inputs. In order to maximize code coverage and find deeper bugs within the code, we take existing seeds generated by the AFL Fuzzer and mutate them to include new test-cases to the given software[4]. This system offers a better debug process with higher code coverage and deep bug exposure at the same given time compared to fuzzing alone.

1. Overview

1.1 Introduction

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a software. The software is then monitored for exceptions such as crashes, or failing built-in code assertions or for finding potential memory leaks early in the development process. Existing fuzzers have been effective mainly in discovering superficial bugs, close to the main module of software (low-hanging bugs) [5], [6], while having difficulties with complex and deeper ones. Many of today's applications have a complex input format and the code execution heavily depends on input values aligning to this format. Most modern fuzzers blindly mutates values to generate new inputs to get higher code coverage. In this (pessimistic) scenario, most of the resulting inputs do not conform to the input format and are rejected in the early stages of the execution. This makes a traditional random fuzzer often ineffective in finding bugs that hide deep in the execution. It can also get the debug process to a certain threshold that no new bugs can be found after crossing it.

State-of-the-art fuzzers such as AFL, [4], employ evolutionary algorithms to maintain a valid input generation. Such algorithms contain a simple feedback loop to assess how good an input is. In detail, AFL retains any input that discovers a new path and mutates that input further to check if doing so leads to new basic blocks. While simple, this strategy cannot effectively select the most promising inputs to mutate from the discovered paths. In addition, mutating an input involves answering two questions: where to mutate (which offset in the input) and what value to use for the mutation? The problem is that AFL is completely application-agnostic and employs a blind mutation strategy. It simply relies on generating a huge amount of mutated inputs in the hope of discovering a new basic block. Unfortunately, this approach yields a slow fuzzing strategy, which can only discover deep execution paths by sheer luck. Fortunately, we can increase the efficiency of an AFL-like fuzzers manifold by adding another step to the pipeline of processing those inputs through a calibrated neural network, [3].

In this paper we present a combination of a Neural Network architecture and Fuzz testing that doesn't requires any prior knowledge about the given software or its inputs. In contrast to approaches that optimize the input generation process with symbolic linking to produce inputs better inputs of higher quality, our work explores a new scalable way in the design space, where we do more work after the fuzzing phase to get more high-quality inputs given some input corpus. The key method here is that we can improve the efficiency of general-purpose fuzzers with a "smart" regenerative pipeline based on control- and data-flow application features without having to resort to less scalable symbolic execution and other solutions. By feeding the fuzzer's input into the Neural Network pipeline a new and better seeds are being generated to include higher coverage and more deeper code that would not be reached otherwise. This methodology can be scalable and run simultaneously to the fuzzer run creating a continuous pipeline. Thanks to this approach we have achieved a massive improvement of input's code coverage. We have evaluated our full pipeline with AFL Fuzzer, [4], on a binary utility tool named Objdump [7]. Our main measurement of improvement was based on the code coverage percentages as well as the number of unique crashes found in our pipeline compared to the AFL Fuzzer.

1.2 Prior Work

Our mentor D. She was testing this pipeline on another GNU binary utility called Readelf, [8]. He had similar results on Readelf software with similar code coverage and unique functions. We also found a homologous project by Microsoft related to fuzzing and Security Vulnerability detection tool called Neural Fuzzing, [9]. In this article, they describe multiple ways to optimize the different modern fuzzing techniques. In this article, the use the AFL[4], and applied 4 different Neural Networks architectures to improve the input's quality is described. They come to a conclusion that LSTM networks are the most effective ones producing about 10% better results compared to other networks.
Another key paper describing a similar idea using AFL[4], using a Generative Adversarial Network (GAN) to initialize the system with novel seed files and improve the performance by about 14% [10]. Moreover, their experiments showed that the GAN model was faster and more effective than the LSTM model (mentioned above), and outperformed a random augmentation strategy, as measured by the number of unique code paths discovered.
Besides traditional Neural Networks, reinforcement learning can be used to produce even better results and outperform baseline random fuzzing, [11]. They have formalized fuzzing as a reinforcement learning problem using the concept of Markov decision processes. Then this allowed them to apply deep Q-learning algorithms that optimize rewards, which they have defined from runtime properties of the program under test. By observing the rewards caused by mutating with a specific set of actions performed on an initial program input, the fuzzing agent learns a policy that can next generate new higher reward inputs and improve the input's quality.

2. Background

In this section, we cover the background required for our discussion of Neural Fuzzer in subsequent sections. Fuzzing is a software testing technique aimed at finding bugs in an application [12], [13]. The distinction of a fuzzer is its ability to generate bug triggering inputs. From an input generation perspective, fuzzers can be mutation- or generation based. Mutation-based fuzzers start with a set of known inputs for a given application and mutate these inputs to generate new inputs. In contrast, generation-based fuzzers first learn or acquire the input format and generate new inputs based on this format. Neural Fuzzer is a mutation-based fuzzer. With respect to input mutation, fuzzers can be classified as white box, black box and grey box. A whitebox fuzzer [14], [15], [17] assumes access to the application's source code—allowing it to perform advanced program analysis to better understand the impact of the input on the execution. A blackbox fuzzer [18], [19] has no access to the application's internals. A greybox fuzzer aims at a middle ground, employing lightweight program analysis (mainly monitoring) based on access to the application's binary code. VUzzer is a greybox fuzzer.

Furthermore, another factor that influences input generation is the application exploration strategy. A fuzzer is directed if it generates inputs to traverse a particular set of execution paths [14]–[16], [17], [20]. A coverage-based fuzzer, on the other hand, aims at generating inputs to traverse different paths of the application in the hope of triggering bugs on some of these paths [21], [22], [23], [24], [4]. Neural Fuzzer is a coverage-based fuzzer.

2.1. Objectives and Technical Challenges

The most important goal for us, while working on this project, was to generate better test cases in the same amount of time by keeping the fuzzing agnostic to the input program. Moreover, we wanted to create a continuous process in which the new mutated inputs are being fed back into the fuzzer again to reproduce newer inputs which are more inclusive.

We have met quite a few technical challenges:

1. Generating enough inputs to get the network to work Eventually we ended up using about 14K inputs for both training and testing.
2. Creating the correct network architecture to produce the maximal results. The final architecture we ended up using is described in the next section (2.2).
3. Mutating the network outputs with gradient descent and feeding it back to the fuzzer.

2.2. Problem Formulation and Design

We have created the system to be as modular as possible in order to create scalability and flexibility to the input software. This architecture in Figure 1 solves the above objectives as follows:

1. The used fuzzer keeps on running regardless of the network and can keep running forever. After it reached a certain threshold, the network should be operable and start its main role.
2. The network architecture setup was provided for us by our mentor Dongdong. This had to be fed new inputs and needed to run on a completely different software, Objdump (Binutils). Ultimately, this network for the machine learning mutating of seeds should be universal so it would best generate seeds to gain maximum coverage for any file.
3. After the training phase has ended we have seperated the last phase of mutating the network outputs and feeding it back to the fuzzer.

2.3 Software Design

A. **Data Generation**

As stated before, we have decided to work with the Objdump GNU Binary Utility. We have chosen to run it with the -D flag because it is including all of the other flags that Objdump offers. This will result in a better function coverage later on when we will fuzz. We compiled the Binutils source code with afl-gcc to create AFL instrumentations within the Objdump code. We have also created a simple hello world app in C language, this is our initial seed that is given to the AFL Fuzzers. There is no point to fuzz with large files as it will produce similar results with a much longer runtime (less efficient). We needed more than 10K seeds for both training and testing in order to run the NN successfully. We used the AFL multiprocess feature to accelerate the process. We had one main thread which was configured with -M flag that stands for master:

./afl-fuzz -i ./afl_in/ -o ./afl_out/ -M fuzzer01 ../binafl/binutils-2.30/binutils/objdump -D @@

The other 7 processes ran with -S flag which stands for slave process. The master is responsible to synchronize the data once in a while with the slave processes. In total, we ran 8 fuzzer processes for about 6 and a half hours which resulted in 6652 seeds from buzzer01, those seeds will use as the training data. We took the 2nd process output seeds as testing data, resulting in 8448 seeds and a total of 15100 seeds.

B. **Combining and converting bitmaps**

The next step in our process was to clean up the data, we decided to remove seeds that were bigger than 10Kb to reduce and limit the size of the files we were working with later on it will affect the runtime of the neural network and the network generated seeds. Then we needed to convert the bitmaps we generated, by running the AFL Fuzzer tool called afl-showmap, to get the unique bitmap description. This was also needed to remove duplicates from our data. These steps were completed by using shell commands to create a file which we named new_int, this file contained all of the bitmaps concatenated for both testing and verifying, to be used in our python script convert_bitmap.py. The convert_bitmap.py script converted all the given bitmap files into a numpy readable arrays and stored them in a newly created convert_bitmaps folder.

C. **Training the network**

Once the converted bitmaps were set and the data was clean of noise, we were ready to begin training the network. Figure 1 shows the architecture of the network. This sequential network model involved four total layers (two hidden layers), as described in the figure. The seeds and bitmaps were fed into this network as a tuple.

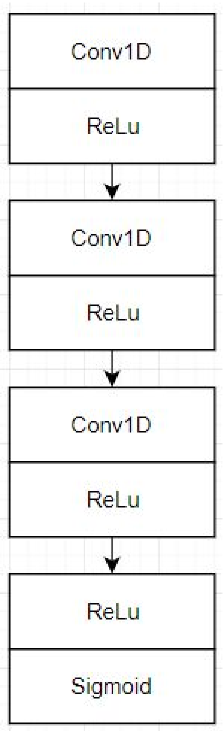


Figure 1: Network Architecture

The network was trained for 70 epochs, ran by using the script cnn_afl.py, for a duration of several hours. After the training has been completed, the validation accuracy was shown as 97%. From this point we could use the model generated, which we saved as final_model_reduce.h, to mutate the seeds further so that the goal of more code line coverage over Objdump can be achieved.

D. **Generating seed mutations**

To generate seed mutations after training the network, we used the saved model along with the cnn_afl.py script (specifically the function gen_mutate). This function would take the trained model along with the gen seeds and create a new folder, called adv_gen_seeds. This is where the mutated seeds are being kept throughout multiple folders.

E. **Running Objdump with Gcov on seeds**

Once the seed mutations have been set up, now we can run Objdump with the -D flag, using gcov and feeding all the mutated seeds to it. We used the script named gen_gcov.py, which goes through each one of the mutated seeds in the adv_gen_seeds folders and subfolders and calls Objdump with the flag -D on the seed files. We compiled this version of the Binutils with gcov flags to create instrumentations(gcov flags: -fprofile-arcs -ftest-coverage). Gcov keeps a track of which lines get called in a generated Objdump.gcda file. This process is very time-consuming. As shown in the next section of the results, we ended up putting timeouts for files first running longer than 60 minutes, then lowered it to 5 minutes. These timeouts caused our results to be a bit inconsistent due to timeouts occurring at different times the program is running and Objdump not being able to complete on some of the seeds we used.

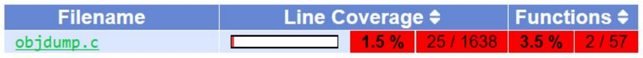
F. **Generating HTML using Lcov**

After running Objdump on all of the mutated seeds, we were eager to see our results from the instrumented Objdump. Gcov is a bit difficult to read by humans so we have used Lcov which is a tool that generates an HTML to uncover the gcov results. We ran Lcov on the gcov results from

Objdump to retrieve the HTML visual results shown in the next section, results.

3. Results

The results generated with Lcov discussed in this section vary. Some of the tests ran had many different parameters which will be discussed. We will also mention why we think the results varied from experiment to experiment. In the first experiment, we ran size.c, which is not the Objdump file we are focusing on. As can be seen in Figure 2, size.c utilizes Objdump.c. The use of Objdump is not very big considering the line coverage was only 1.5% for this experiment. This gave us a baseline of how gcov can be used.

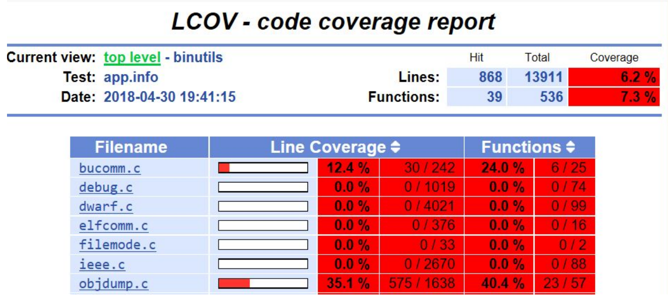


Filename	Line Coverage ⬍			Functions ⬍	
objdump.c		1.5 %	25 / 1638	3.5 %	2 / 57

Figure 2: Results from not running seeds on Objdump

Results for base fuzzed seeds

The next experiment we performed involved the base seeds before mutation occurs. We ran Objdump over all the seed files (testing and verifying). After running lcov to see the results of the line coverage, as seen in Figure 3 it came out to be 35.1%. This is a much higher line coverage from the previous experiment which is what we expected. This means that the fuzzed seeds did well at covering lines that may not have been covered by the original input. Unfortunately for this experiment, objdump was running on one of the seeds for over 24 hours so we needed to stop the experiment at around 70% through the seeds.



LCOV - code coverage report

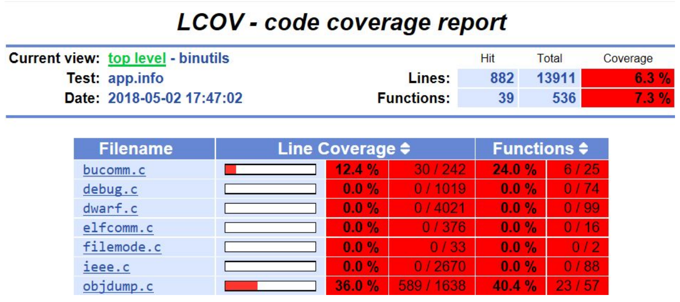
			Hit	Total	Coverage
Current view:	top level - binutils				
Test:	app.info	Lines:	868	13911	6.2 %
Date:	2018-04-30 19:41:15	Functions:	39	536	7.3 %

Filename	Line Coverage ⬍			Functions ⬍	
bucomm.c		12.4 %	30 / 242	24.0 %	6 / 25
debug.c		0.0 %	0 / 1019	0.0 %	0 / 74
dwarf.c		0.0 %	0 / 4021	0.0 %	0 / 99
elfcomm.c		0.0 %	0 / 376	0.0 %	0 / 16
filemode.c		0.0 %	0 / 33	0.0 %	0 / 2
ieee.c		0.0 %	0 / 2670	0.0 %	0 / 88
objdump.c		35.1 %	575 / 1638	40.4 %	23 / 57

Figure 3: Base seed results

Results for Neural Network mutated seeds

Since we saw that objdump took a really long time to run on some of the seeds, we decided to set a timeout so that any objdump process running more than 30 minutes, we would just skip running objdump for that specific seed. The runtime for this ended up being much faster, though the true line coverage was not discovered since some of the mutated seeds were skipped using this process. As can be seen in Figure 4, using this method we got 36% line coverage for Objdump.c. This is a good result since it the best line coverage percentage we have seen thus far in our experiments.



LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - binutils				
Test:	app.info	Lines:	882	13911	6.3 %
Date:	2018-05-02 17:47:02	Functions:	39	536	7.3 %

Filename	Line Coverage ⬍			Functions ⬍	
bucomm.c		12.4 %	30 / 242	24.0 %	6 / 25
debug.c		0.0 %	0 / 1019	0.0 %	0 / 74
dwarf.c		0.0 %	0 / 4021	0.0 %	0 / 99
elfcomm.c		0.0 %	0 / 376	0.0 %	0 / 16
filemode.c		0.0 %	0 / 33	0.0 %	0 / 2
ieee.c		0.0 %	0 / 2670	0.0 %	0 / 88
objdump.c		36.0 %	589 / 1638	40.4 %	23 / 57

Figure 4: Mutated seed results

4. Discussion and Further Work

There are many improvement and changes that can be made on this system. Some of the improvements include running on programs other than Objdump, using different data amount or parameters than what was described in this work, or trying to use different network architectures.

An aspect that would be very helpful for this type of system is the creation of a continuous, automated process can be achieved with LSTMs (Long-short term memory), which would train a model and on a certain interval feed it with new seeds that the fuzzer has generated. Because of the heavy complexity of the pipeline we didn't get to meet the limitations of the fuzzer or the network. An interesting future experiment would be to test the limits of the system and see when the effectiveness of each part of the system drops.

5. Conclusion

We worked on combining a Neural Network architecture and Fuzz testing in order to maximize code coverage and find deeper bugs within the code. Our results showed that though there was a small increase in line coverage for the neural network mutated seeds, it was not significant enough to say that this can work 100% of the time. We also saw that the time process of this system is very long. Since this is not yet automated, a user would need to go through many steps to accomplish this type of work on their software. We highly recommend this system to be automated for future work since it would make this process more easily testable on broader spectrums of programs and parameters.

6. Acknowledgements

We would like to thank our mentor for this research, Dongdong She. As well as our professor Junfeng Yang for his help and guiding us through this complex topic which is discussed in this paper.

7. References

[1] Link to your bitbucket.
[2] T. Britton, L. Jeng, G. Carver, P. Cheak, T. Katzenellenbogen, Reversible Debugging Software, University of Cambridge Judge business school.
[3] S. Rawat, V. Jain , A. Kumar , L. Cojocar, C. Giuffrida, H. Bos, VUzzer: Application-aware Evolutionary Fuzzing, Computer Science Institute, Vrije Universiteit Amsterdam, NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA Copyright 2017 Internet Society, ISBN 1-1891562-46-0.
[4] M. Zalewski, "American fuzzy lop," at: http://lcamtuf.coredump.cx/afl/.
[5] S. Clark, S. Frei, M. Blaze, and J. Smith, "Familiarity breeds contempt: The honeymoon effect and the role of legacy code in zero-day vulnerabilities," in ACSAC'10. New York, NY, USA: ACM, 2010, pp. 251–260.
[6] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in IEEE S&P'16. IEEE Press, 2016.
[7] "Linux GNU Binary Utilities, Objdump" at: http://man7.org/linux/man-pages/man1/objdump.1.html
[8] "Linux GNU Binary Utilities, Readelf" at: http://man7.org/linux/man-pages/man1/readelf.1.html
[9] W. Blum, "Neural fuzzing: applying DNN to software security testing" Microsoft Research Blog, November 13, 2017
[10] N. Nichols, M. Raugas, R. Jasper, N. Hilliard, Faster Fuzzing: Reinitialization with Deep Neural Models, arXiv:1711.02807v1 [cs.AI] 8 Nov 2017
[11] K. Bottinger, P. Godefroid , R. Singh, Deep Reinforcement Fuzzing, arXiv:1801.04589v1 [cs.AI] 14 Jan 2018
[12] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," Commun. ACM, vol. 33, no. 12, pp. 32–44, 1990.
[13] A. Takanen, J. DeMott, and C. Miller, Fuzzing for Software Security Testing and Quality Assurance, 1st ed. Norwood, MA, USA: Artech House, Inc., 2008.
[14] P. Godefroid, "Random testing for security: blackbox vs. whitebox fuzzing," in Int. workshop on Random testing. New York, NY, USA: ACM, 2007, pp. 1–1.
[15] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," SIGPLAN Not., vol. 40, no. 6, pp. 213–223, 2005.
[16] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in ICSE'09. IEEE Computer Society, 2009, pp. 474–484.
[17] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in USENIX SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 49–64.
[18] "Peach fuzzer," http://www.peachfuzzer.com/.
[19] OpenRCE, "Sulley fuzzing framework," https://github.com/OpenRCE/sulley.
[20] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos, "The borg: Nanoprobing binaries for buffer overreads," in CODASPY '15. New York, NY, USA: ACM, 2015, pp. 87–97.
[21] B. Copos and P. Murthy, "Inputfinder: Reverse engineering closed binaries using hardware performance counters," in PPREW'15. New York, NY, USA: ACM, 2015, pp. 2:1–2:12.
[22] U. Kargen and N. Shahmehri, "Turning programs against each other: ´ High coverage fuzz-testing using binary-code mutation and dynamic slicing," in FSE'15. New York, NY, USA: ACM, 2015, pp. 782–792.
[23] K. Serebryany, "Libfuzzer: A library for coverage-guided fuzz testing (within llvm)," at: http://llvm.org/docs/LibFuzzer.html.
[24] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in NDSS'16. Internet Society, 2016, pp. 1–16.