Swift: Efficient Fuzzing with Neural Program Smoothing and Seed Selection

Abstract

Fuzzing is a popular technique for finding software bugs. For genetic algorithm-based fuzzing, it can mutate the seed files provided by users to obtain a number of inputs. These inputs are later used to test the objective application in order to trigger potential crashes. Such algorithms, while simple to implement, often get stuck in fruitless sequences of random mutations. We propose **Swift**, a efficient fuzzing tool that has a novel seed selection strategy and incorporates neural network model to generating test cases in a short amount of time. The main goal of **Swift** is to apply the current research approach on machine learning to accelerate fuzzing process. During seed selection, we use Long short-term memory **(LSTM)** to take the execution path as the input, and predict if the given seed is more likely to trigger vulnerability. As comparison, the existing fuzzers usually treat all seeds equally, ignoring the fact that certain seeds have a high chance to trigger vulnerability. We then use a smooth surrogate function to approximate the target program's discrete branching behavior, and apply neural network for generating test cases to increase the efficiency of the fuzzing process. Our evaluations demonstrate that **Swift** outperforms state-of-the-art fuzzers on popular real-world programs at triggering crashes.

1 Introduction

Vulnerabilities often refer to the flows or weaknesses in hardware, software, protocol implementations, or system security policies that allow an attacker to access or compromise the system without authorization, and have become the root cause of the threats toward network security. WannaCry ransomware attack outbroke on May 2017, and more than 150 countries and 300,000 users were attacked, causing more than $8 billion in damage[6]. The virus spread widely by utilizing the "Eternal Blue" vulnerability of the NSA (Nation Security Agency) leak. The number of vulnerabilities announced by CVE (Common Vulnerabilities and Exposures) began to explode in 2017, from the original highest 7946 vulnerabilities in 2014 to the publication of 16555 vulnerabilities in 2018 (CVE 2019)[3].

Considering the increasing number and the severe damages of vulnerabilities, vulnerability discovery technology has attracted widespread attention. Fuzzing technology is an efficient method to discover weaknesses, which was first proposed by Miller et al. in 1990 [11]. It is an automatic testing technique that covers numerous boundary cases using invalid data (e.g., files, network packets, program codes) as to automate the testing process. The fuzzing process is about generating random test inputs and run these inputs to see if they trigger any potential security vulnerabilities, for example buffer overflow, assertion crashes, or memory leak.[11] Most of the current fuzzers use evolutionary guidance to generate inputs. This type of fuzzer gradually increases the code coverage and amplifies the probability of finding vulnerabilities. For example, American Fuzzy Lop (AFL)[2], libfuzzer[5], and honggfuzz[4] have drawn attention from both industry and academia and have discovered hundreds of vulnerabilities.

The logic behind these fuzzers is that, it instruments the program to observe which inputs explore new program branches, and keeps these inputs as seeds for further mutation. One important limitation, however, is that most of the inputs that it creates are ineffective, especially when the input corpus become larger, this algorithm becomes less efficient to produce quality input, in this case the input that can find bugs of the given system. One reason is the seed selection, since different seeds will trigger different paths that have different probabilities of being vulnerable. As reported in [12], the bug distribution in programs is often unbalanced, i.e., approximately 80% of bugs are located in approximately 20% of program code. Another reason is the quality of the test cases. Some fuzzers use symbolic execution to solve path constraints[7], but symbolic execution is slow and cannot solve many types of constraints efficiently. One optimization technique is to use gradient based search rather than the evolutionary guidance algorithm, as it has been shown to be able to perform good results in domains like machine learning. However, gradient-guided optimization algorithms cannot be directly applied to fuzzing real-world programs as they often contain significant amounts of discontinuous behaviors (cases where the gradients cannot be computed accurately) due to widely different behaviors along different program branches.[8]. She Et al.[14] proposed a smoothing surrogate function and a neural network to approximate the target behavior and thus find the critical path. However, they didn't explore the effect of the selection on hyperparameters, for example the number of hidden layer, the size of the neural network model, and the number of neurons, given different data sets.

In this paper, we propose a hybrid fuzzing solution, Swift, to alleviate the aforementioned limitation. It applies a new seed selection strategy that prioritizes seeds that are more likely to exercise vulnerable paths, and also a fine-tune neural network to improve the efficiency of vulnerability detection. The core challenge is how to determine whether a path is likely to be vulnerable, and how to apply gradient-guide algorithms on real-world programs. Inspired by the substantial success in image and speech recognition, we use a deep neural network to learn the hidden pattern of vulnerable program paths. We also implement a surrogate function to effectively model the target program's branching behaviors. When compared with AFL, Swift achieved a 100% increase in the number of crashes triggered given a short amount of time.

2 Background

2.1 Fuzzing Procedure

The fuzzing process, as shown in Figure 1, begins by choosing a corpus of "seed" inputs with which to test the target program. The fuzzer then repeatedly mutates these inputs and evaluates the program under test. If the result produces "interesting" behavior, the fuzzer keeps the mutated input for future use and records what was observed. Eventually the fuzzer stops, either due to reaching a particular goal (e.g., finding a certain sort of bug) or reaching a timeout. Different fuzzers record different observations when running the program under test. In a "black box" fuzzer, a single observation is made: whether the program crashed. In "gray box" fuzzing, observations also consist of intermediate information about the execution, for example, the branches taken during execution as determined by pairs of basic block identifiers executed directly in sequence. "White box"

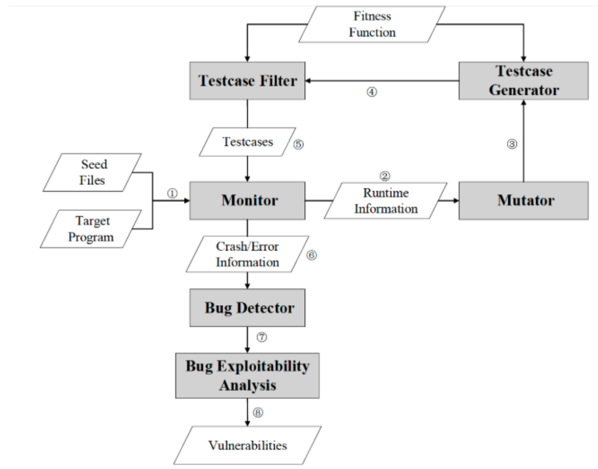


Figure 1: Fuzzing, in a nutshell

fuzzers can make observations and modifications by exploiting the semantics of application source (or binary) code, possibly involving sophisticated reasoning. Gathering additional observations adds overhead. Different fuzzers make different choices, hoping to trade higher overhead for better bug-finding effectiveness. Usually, the ultimate goal of a fuzzer is to generate an input that causes the program to crash.

2.2 Neural Networks

Neural Networks (NNs) model takes vectors X as inputs and outputs vectors y by applying trainable weights matrix W in a defined way F. Variations of F derives different type of NNs models, e.g. Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Fully Connected Neural Networks (FCNNs) etc. Neural Networks usually consist of multiple layers, we refer $F^{(l)}$ as computation way at layer l with corresponding trainable weights $W^{(l)}$. In each type of model, there are still multiple factors controlling the way model processing intermediate results in between different layers, like activation functions, dropouts etc. In our work, we use Rectified Linear Unit (ReLU) as our activation function.

2.3 Seed Selection Stragey

The seed selection strategy is critical for fuzzing. A good seed selection strategy can improve the ability of path traversal and vulnerability detection. AFL takes a simple seed selection strategy, i.e., preferring smaller and faster seeds, to generate and test more test cases in a given amount of time. Rawat et al. [13] prioritize seeds that exercise deeper paths, de-prioritizes seeds exercising error-handling blocks and high-frequency paths, and thus it is likely that hard-to-reach paths could be tested and useless error-handling paths will be avoided, and prioritized the valid inputs that do not contain these basicblocks. AFLFast [1] prioritizes seeds exercising low frequency paths and being selected fewer. Angora prioritizes seeds whose paths contain conditional statements with unexplored branches, which enables focus on low-frequency paths after exploring high-frequency paths. The existing seed selection strategies focus mainly on execution speed, path frequency, path depth and path branches that are not traversed.

2.4 Function smoothness and Optimization.

Optimization algorithms usually operate in a loop beginning with an initial guess of the parameter vector x and gradually iterating to find better solutions. The key component of any optimization algorithm is the strategy it uses to move from one value of x to the next. The ability and efficiency of different optimization algorithms to converge to the optimal solution heavily depend on the nature of the objective and constraint functions. In general, smoother functions can be more efficiently optimized than functions with many discontinuities. Intuitively, the smoother the objective/constraint functions are, the easier it is for the optimization algorithms to accurately compute gradients or higher-order derivatives and use them to systematically search the entire parameter space.

3 Overview

In this section, we further explain the problems we need to solve through a motivating example and describe an overview of our approach.

3.1 Motivating Example

As mentioned above, the seed selection strategy of current fuzzers doesn't reach the optimal efficiency. To help better illustrate the problem here, consider the code in Figure 2. The overflow function will cause a stack overflow if the input string is longer than 32 characters. In the main function, only the input start with character '2' will call our overflow function. Assuming that test cases that generated by the fuzzer are "1xxxxxx", "2yyyyyy", and "3zzzzzz". Then AFL will perform mutation on each test case as shown in Figure 3.

However, the two paths that are exercised by the seed inputs "1xxxxxx" and "3zzzzzz" are clearly unlikely to have vulnerabilities. Thus, the importance of fuzzing them is low. We should put more energy, in other words fuzzed more times than others, on the path generated by "2yyyyyy" in order to trigger the potential vulnerability in the path. The example we gave here is pretty simple, but in real world the conditions can be much more complicated. Our motivations are, a) how can we select the best seed, and b) how can we generate the test path that are good at trigger vulnerabilities given the seed.

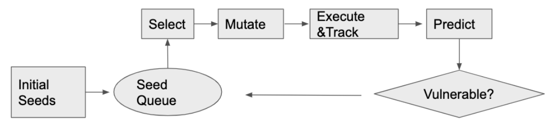


Figure 2: Seed selection Algorithm

3.2 Seed Selection Algorithm

The key idea is to use a seed priority strategy to guarantee that better seeds will be tested and mutated with more time and more iterations, thus improving the overall efficiency and effectiveness. We first obtain our training set by running AFL on our target data. Once we have enough seed (roughly run the AFL for about an hour), we can run the machine learning algorithm on our seeds. More specifically, we want to find the pattern in the seeds that could trigger crashes, and we want to prioritize the seed with same pattern later during fuzzing. The detailed algorithm is documented in Figure 4.

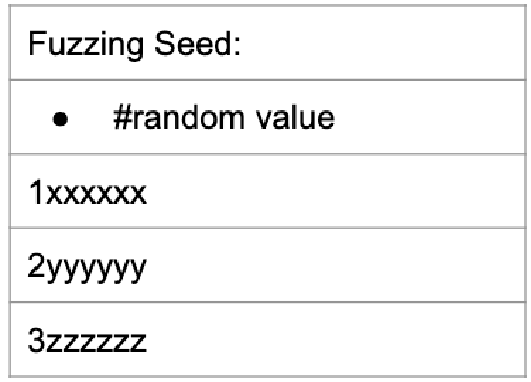
3.3 Gradient-guided Optimization

After we rank/select the seed, we want to mutate the seed and generate test cases that could trigger the vulnerabilities. To apply the neural network, we need to approximate a program's discontinuous branching behavior smoothly in order to calculate the gradient and perform optimization. Without such smoothing, the gradient-guided optimization process may get stuck at different discontinuities/plateaus. We use a feed-forward neural network for this purpose indicated in Figure 5 [14].

4 Design

In this section,we present the detailed design of SWIFT. As shown in Overview, Swift consists two modules to work coordinately. It ranks the seeds by the seed selection module. Then it feeds the seeds

Figure 3: A motivating example



Fuzzing Seed:
• #random value
1xxxxxx
2yyyyyy
3zzzzzz

Figure 4: A seed example

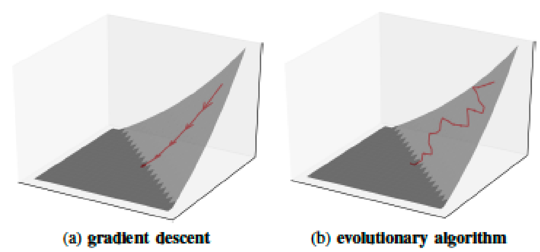


Figure 5: Feed-forward neural network

into the neural network module to generate mutated test cases and feed into the program.

4.1 Seed Selection

This module aims at prioritizing the seeds for the following fuzzing. We use the AFL to collect the raw seeds then store them into the seed folder. After run the raw seeds, we can create our initial training dataset that maps the character of the seeds to the fuzzing outcome. Note the seed here is similar to a sequence of command, giving instructions on to identify the input bytes and the direction of the mutation (e.g., increment or decrement their values) in order to maximize the possibility of trigger crash. This type of data structure is an ideal input for LSTM. We can define the seed sequence as a NumPy array. We can then use the reshape() function on the NumPy array to reshape this one-dimensional array into a three-dimensional array with sample number,time steps, and features at each time step. Our goal is to classify that sequence data into one of the two category, easy-to-trigger or hard-to-trigger. The results are in Figure.6

4.2 Neural Network

Algorithm 1 shows the outline of our Neural Network module. The key idea is to identify the input bytes with highest gradient values and mutate them, as they indicate higher importance to the fuzzer. Starting from a seed, we iteratively generate new test inputs. As shown in Algorithm 1, at each iteration, we use value of the gradient to identify the input bytes that will cause the maximum change. Next, we try both directions/signs of the gradient for operation. Conceptually, our usage of gradient sign is similar to the adversarial input generation methods[10]. We also give a range (0-255) for the mutation considering the range of given byte.

[t]

[1]

Gradient-guided mutation $Seed \leftarrow TopPrioritySeed$ $byterange \leftarrow 255$

$Iter \leftarrow numberofiteration$ $perm \leftarrow parameterforneuralnetwork$ $g \leftarrow computedgradientofseed$ $range(Iter)$ $gradient \leftarrow top(perm, seed)$ $range(byterange)$ $v \leftarrow gradient + j * g[location]$ $testcases \leftarrow mutate - seed(v, perm)$ $range(byterange)$ $v \leftarrow gradient - j * g[location]$ $testcases \leftarrow mutate - seed(v, perm)$ Return $testcases$

Gradient-guided mutation

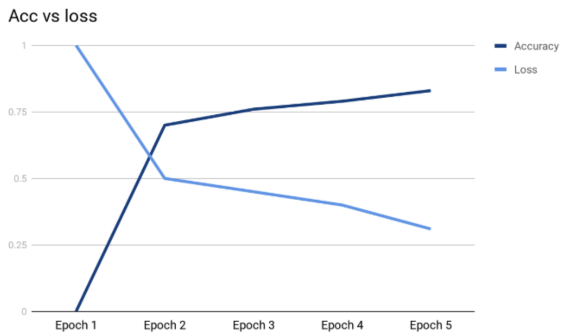


Figure 6: LSTM Results

5 Evaluation

Our model training, containing approximately 400 lines of python code, is developed based on keras with TensorFlow as the backend.

5.1 Study Subjects

We evaluate Swift on two different real-world applications: libjpeg and readelf. We compare the amount of triggering crashes by Swift to AFL.

5.2 Evaluation Setup

Our experimental setup includes the following two steps: First, we run AFL on each given program for an hour to generate the initial seed collections. Then, we run AFL and our fuzzer for a fixed time budget with the same initial seed corpus and compare the number of crashes they triggered.

5.3 Results

The number of crashes is an important factor in measuring the effectiveness of a fuzzer. Some crashes may be caused by the same root cause (i.e., duplicated) or may not be security-related; however, in general, the greater the number of crashes found, the higher the probability that more vulnerabilities can be identified. We compare the results of Swift and AFL on real-world applications in this section. Based on Figure 7 and Figure 9, it is easy to tell that Swift greatly outperform AFL in terms of crashes triggered in a very short amount of time (one hour). Figure 8 gives a closer look on the bug we triggered

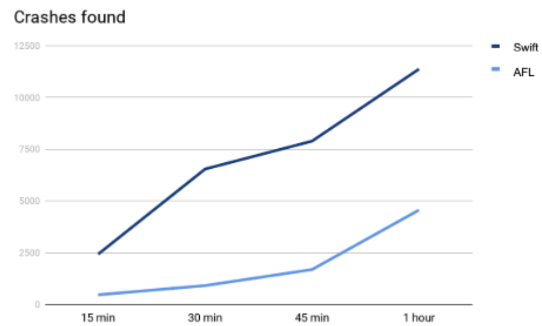


Figure 7: Compare between Swift and AFL on libjpeg

Figure 8: Crash found by Swift

6 Discussion

6.1 Limitations

First, Swift can't detect the vulnerability behind magic bytes, because our model can only predict the existing path to guide seed selection. This problem is a popular topic for improving code coverage, and we leave it as our future work

Second, the quality of our algorithm is largely depend on the initial quality of the seed generated by AFL. Since AFL adapt a random algorithm, there is chance that Swift performs badly based on the given bad quality of seeds.

Third, the goal for Swift is to find as many crashes as possible. However, that doesn't necessarily means more vulnerabilities, as more crashes can be triggered by one specific bug.

6.2 Other Experiments We tried

Transfer Learning: The initial idea is to apply transfer learning on fuzzing. In Computer Vision,

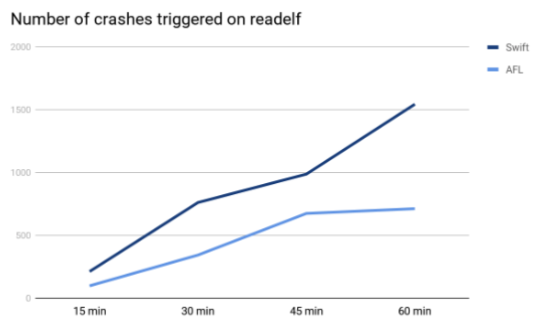


Figure 9: Compare between Swift and AFL on readelf

transfer learning is widely used from general to task-specific cases. In addition, most of the works achieve good performance over the target model just by transferring the first few layers of the source model. In fuzzing, we could use transfer learning to reduce data size by transferring previously trained models between programs. Such methods might eliminate or reduce the amount of model training required for fuzzing previously unexplored programs. However, because of the randomness natural of our mutation program, we are unable to transfer this random low-level features, thus not getting a good result.

Fuzzing on IoT firmware: More and more end devices, such as laptops, pads, smartphones, smart-home devices, and wearable devices, are used in social life. However, it is hard to apply the fuzzing software on the IoT firmware, as they usually share a different operating system or encryption/protection mechanism. It is also a very hot research topic and we will stay tune for the lastest research results to see if there is any future colloboration oppertunity.

7 Related Work

In the following, we first describe previous studies of applying machine learning at fuzzing and then existing studies on smoothing:

Seq2Seq: In this paper[12], they design a deep learning model that is able to trigger a higher code and behavior coverage of target programs by learning the correlation between seed inputs and the execution of the target program. In details, the original seeds are paths of PDF files and then they use machine learning models to predict the new paths of the target programs. During their experiments, they use PDF files as input which have more complex input format. According to the feature of paths of files: logic and sequence. This paper utilizes a sequence model: Recurrent Neural Network to generate new path of the target. After this process, they employ a Sequence-to-Sequence model to translate the new path to a real PDF file. In their experiments, not only can their models be used on other files like PNG and TTF files, but also their model can outperform other models. The first step of their model is data preparation. They utilize the Path Recorder to get the original path of execution sequence. Some of the paths are too lengthy to be handled by the RNN model so they also use a compression algorithm to compress the path. The second step is the Path Generator, and the generator model is based on Andrej Karpathy's Char-RNN, which is a two-layer, one for learning how basic blocks form the function and the other learn how these functions form the final paths. The final step is Seed Generation, during which the generated paths are translated into PDF files by using Sequence to sequence model.

In another paper[9], they tried to design an automatic grammar-based fuzzing model by using neural-network models. One main advantage of Sequence to Sequence model is that it allows for learning arbitrary length contexts to predict next sequences of characters as compared to traditional n-gram based as compared to traditional n-gram based approaches that are limited by contexts of finite length. Sequence model RNN can learn a probability distribution over a character sequence by training next character in the sentence. This paper trains the seq2seq model using a corpus of PDF objects treating each one of them as a sequence of characters. During training, they first concatenate all the object files into a single file resulting in a large sequence of characters. Then they use the learnt seq2seq model to generate new PDF objects. There are three strategies for object generation depending upon the sampling strategy used to sample the learnt distribution, including Nosample, sample and SampleSpace. Finally, they solve a tradeoff which is that a perfect learning technique would always generate well-formed objects that would not exercise any handing code, whereas a bad learning technique would result in ill-formed objects that wouldn't be quickly rejected by the parser upfront. After their experiments, they show that the learnt models are not only able to generate a large set of new well-formed objects, but also result in increased coverage of the PDF parser used in our experiments, compared to various forms of random fuzzing. This paper found there is an interesting relationship between model learning and fuzzing: the RNN models want to learn how the input file constructed while the fuzzing tools want to how to break the architecture into paths.

8 Conclusion

We present Swift, an efficient fuzzer that uses a seed selection approach combined with surrogate neural network to smoothly generate valid test cases. We further demonstrate how gradient-guided techniques can be used to generate new test inputs that can uncover different bugs in the target program. Our extensive evaluations show that Swift significantly outperforms AFL both in the numbers of crash triggered in a given amount of time. Our results demonstrate the vast potential of applying machine learning into the fuzzing process.

References

[1] AFLFast (extends AFL with Power Schedules). `https://github.com/mboehme/aflfast`.

[2] american fuzzy lop. `http://lcamtuf.coredump.cx/afl/`.

[3] Common Vulnerabilities and Exposures. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-2018`.

[4] Fuzzer developed by google. `https://github.com/google/honggfuzz`.

[5] LibFuzzer: A Library for Coverage-Guided Fuzz Testing. `https://llvm.org/docs/LibFuzzer.html`.

[6] WannaCry ransomware attack. `https://en.wikipedia.org/wiki/WannaCry_ransomware_attack`.

[7] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[8] S. Chaudhuri and A. Solar-Lezama. Smoothing a program soundly and robustly. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, pages 277–292, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[9] P. Godefroid, H. Peleg, and R. Singh. Learnfuzz: Machine learning for input fuzzing, 2017.

[10] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[11] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[12] T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. *SIGSOFT Softw. Eng. Notes*, 27(4):55–64, July 2002.

[13] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2 2017.

[14] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana. Neuzz: Efficient fuzzing with neural program smoothing. *2019 IEEE Symposium on Security and Privacy (SP)*, pages 803–817, 2018.