DEEPFUZZer - Using Deep Learning to Write a Fuzzer

Burak Akgül[1], Suat Tarık Demirel[1]

July 20, 2021

Abstract

In this paper presents our work and plan about writing a jpeg fuzzer to expose bugs that exists in programs by providing inputs such that program to cause a crash in order to help to find and fix critical bugs. It will be capable of fuzzing UNIX commands and codes written in different programming languages. The fuzzer was written from scratch in C++ language. We also have written a text generator model to predict the fuzzed file. We added machine learning input generation model to be able to generate better inputs and improved detection rates. This way, our fuzzer model will be faster and be able to detect more unexpected behaviours. We reported the works and approaches that had been done. Our work can be seen in the github page: https://github.com/akgulburak/jpegfuzzer

1 Introduction

Fuzzing is a method of exploding bugs, crashes or memory leaks in software by providing randomized inputs to find unexpected behaviours and a fuzzer is a tool to make automatic fuzzing. Fuzzers are usually used to detect bugs, code coverage, crash detection, blackbox testing and security testing.

By fuzzing the input the right way, we can find different function blocks that is run by the program. These blocks gives the information about how the program works, without needing to inspect the program code.

The inputs that causes crash in the program can be detected by giving fuzzed inputs. Meaning that bugs can also be detected by these inputs.

By applying these methods, malicious users can also get information about how to crash and do malicious activities regarding the program. So, fuzzing can also be used to find the security vulnerabilities of the programs.

Fuzzers create randomized inputs and try to give them to programs to be able to crash or generate mappings from them. The creation of inputs can be done purely randomly or some kind of feedback loop can be used. If inputs were generated completely random this kind of fuzzer called as dummy fuzzer. If there were some methods, rules or knowledge for creating inputs this kind of fuzzer called as Smart fuzzer. There are three main types of fuzzers [1] which are Mutation based fuzzers, generation based fuzzer and Evolutionary fuzzer. Mutation based fuzzer is basically randomly mutates the valid inputs to create malformed inputs. Generation based fuzzer generate input from scratch. The inputs can be completely random or has some intelligence methods. Lastly evolutionary fuzzer has some kind of feedback loop so that it can learn over time. Even though those Smart fuzzers are used, fuzzing may take long time to generate expected results as generating and trying inputs takes time. To be able to get ahead of this performance issues and increase our success rates with fewer inputs, a machine learning model was used. Deep-Fuzzer is machine learning oriented smart fuzzer. It has a machine learning input generator. It uses machine learning input generator to predict inputs rather than trying randomized inputs. I this way deep-fuzzer perform faster than a ordinary fuzzer and cause unexpected behaviour more likely.

2 Related Works

There are fuzzers that uses machine learning models to be able to fuzz more efficiently. One example was described in the article V-Fuzz: Vulnerability-Oriented Evolutionary Fuzzing[2]. It mainly focuses on detecting vulnerabilities of binary programs for quickly finding bugs.V-Fuzz has two main components a vulnerability prediction model and a vulnerability-oriented evolutionary fuzzer. For a binary software, the prediction model will perform an analysis on which components of the software are more likely to have vulnerabilities. Then, the evolutionary fuzzer will leverage the prediction information to generate high-quality inputs which tend to arrive at these potentially vulnerable components. In this way, V-Fuzz decrease the waste of computing resources and time. Also it significantly improves the efficiency of fuzzing.

Another example was described in the article GANFuzz: A GAN-based industrial network protocol fuzzing framework [3]. It use the generative adversarial network(GAN) to train a generative model with the dataset of protocol messages in order to reveal the protocol grammar. In this way, it can generate fake but sensible messages that are perfect test cases to test implementations of INPs.

This and similar examples might be replicated in our own fuzzer to see the effect of machine learning.

3 Methodology

3.1 Writing initial fuzzer

We needed to create our own fuzzer from scratch. To get started, a fuzzer in C++ was beginned to be written in the "https://carstein.github.io/2020/04/18/writing-simple-fuzzer-1.html" link. This fuzzer gets .jpeg files and distorts them randomly. It preserves the magic numbers that exist at the beginning and the end. It was written in Python and was working slow, so we have written in C++ as the author recommends. Our fuzzer in this state is able to generate fuzzed .jpg images.

3.2 How initial fuzzer works

Our initial fuzzer takes a .jpeg file and flips some random bits or replaces the bits with magic numbers. In its current state, it does not use feedback mechanism. After generating our fuzzed data, we create a child process by using fork() method. The child will be execute our fuzzed data, and we will wait for the child to complete its execution. If the child returns an error code, we are appending this code to an array and we count number of crashed. Unfortunately, if the program goes into infinite loop, the program freezes at that point because the parent will wait the child to finish. So, we added a timeout to handle this condition. We have decided that if 2 seconds have passed after we tried the jpeg2hdf command with our fuzzed data, we would consider this as a timeout.

Algorithm 1 Pseudocode of the mutation engine

```
1: procedure MUTATE
2:     method ← rand_choice(0,1)
3:     for $i <$ flip_number do random_indexes[i] ← rand()%(data_size-8)
4:     if method = 0 then
5:         $data[random\_index[i]]) \leftarrow flip\_bit(data[random\_index[i]])$
6:     if method = 1 then
7:         $data \leftarrow magic(data, random\_index[i]]$
```

3.3 Machine Learning Input Generation

The tensorflow documentation in the link "https://www.tensorflow.org/text/tutorials/text_generation" describes that by giving the text as x label and giving the next characters as y label, a text generator can be trained. In the same documentation, it can also be seen that a "Romeo and Juliet" script can be created by the recurrent neural network (RNN) model. It encodes strings as integers and creates a vocabulary from them, and then trains the model.

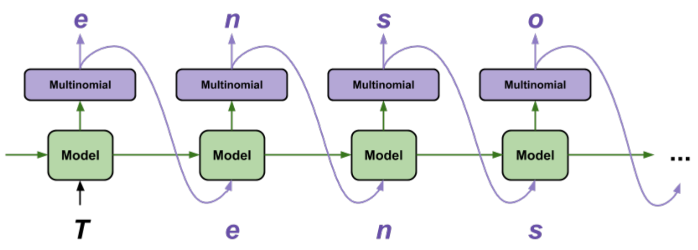

Figure 1: Visualization of sampling for text generation model

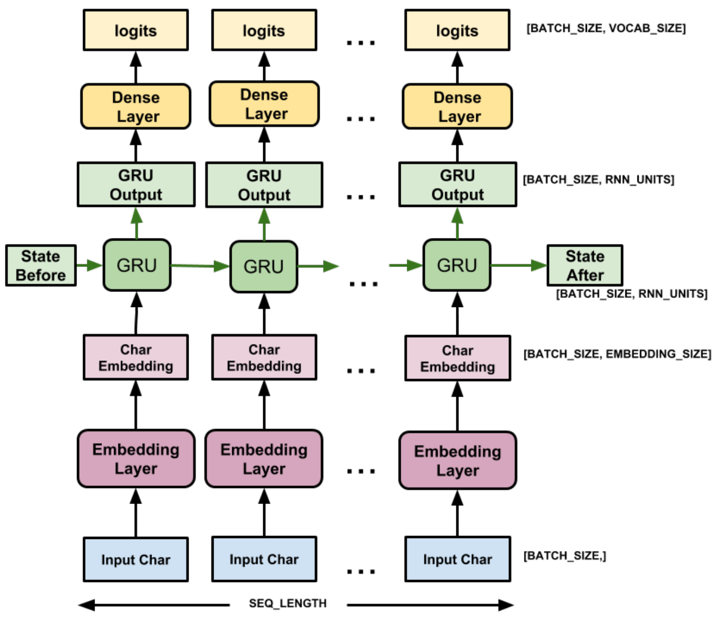

Figure 2: Visualization of training loop for text generation model.

As mentioned in the related works section in the paper, GAN's can also be used as a form of generating inputs by looking at the training data. GANs have two parts; generator and discriminator. Generator tries to create a fake input by looking at the training data and presents this data to discriminator, which in turn tries to classify the said data as real or fake. The generator and discriminator; thus, trains each other. This approach could also be used to generate the machine learning model.

3.4 Machine Learning Oriented Fuzzer

We thought that the input generation with text generator can be applied to our case too, as the inputs string are treated as integers and we can represent our fuzzed .jpg files as a list of integers to feed them to our model. We have taken the base image as the training data and the fuzzed images as training labels. By employing this strategy, we expected that the common parts in the data that makes the "jpeg2hdf" command wait, would be present for the generated data by our model. We have acquired 7000 data by using our simple fuzzer and trained our model with 10 epochs with 100 of the data.

After 9th epoch, we have seen that the training would not go any further, so we concluded that for 100 data, 10 epochs is enough. Training with the subset of 100 data was needed at that point because the training for one epoch took more than 3 hours for all the data and it was not that essential to train with all of the data, as we ended of with high enough accuracy, with 98%, at the end of the training loop.

We have used text generation model, but if we were to employ the GAN approach for creating the model, we should have written a code for the discriminator to actually try the given input and return whether the input has been successfully fuzzed or not, but we had to wait for 2 two seconds as we have decided that 2 seconds is the maximum time for the "jpeg2hdf" command to finish running. For 100 data, this would mean that 200 seconds of waiting, and that alone would make our progress much slower, so this approach was discarded and text generation model approach has been taken.

3.5 Testing the fuzzer

We tested our fuzzer with two different ways. Firstly, we wanted to see if we can fuzz some UNIX commands. jpeg2hdf command was used for this purpose. Our fuzzer created a distorted .jpg file and when it was tried to be converted to .hdf file with the jpeg2hdf command, the command could not finish. It was concluded that the command was either in an infinite loop or was crashed unexpectedly.

After testing our fuzzer with jpeg2hdf, we have written a very simple program in C++ that can read .jpg images. It uses *stb_image.h* library which is found in the following "https://github.com/nothings/stb" link. Our fuzzer crashed the program and program returned SIGSEGV signal to us.

After obtaining these two mentioned results, we concluded that our fuzzer works to some extent.

After writing this simple fuzzer, we have also tested our fuzzer model that we have trained with 100 different image files. The results can be seen in the results section of the paper.

4 Results

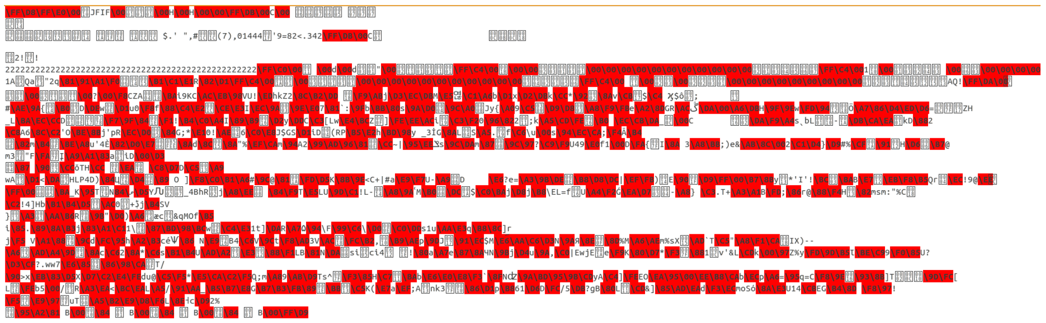

Figure 3: Original file

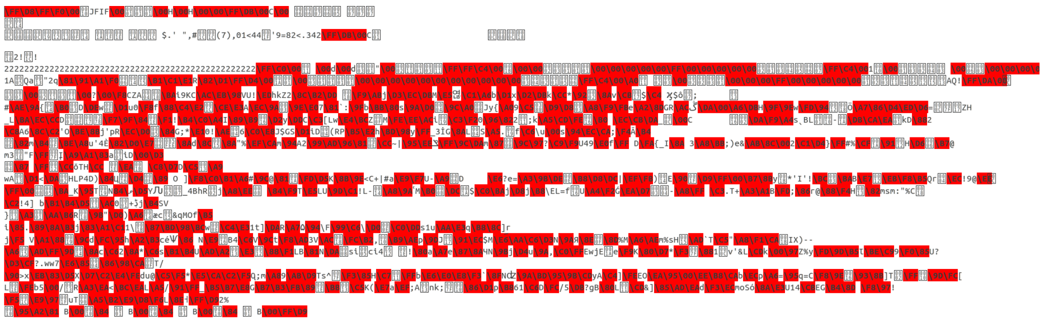

Figure 4: Fuzzed version with the simple fuzzer with 2% bit flipping rate

Figure 5: Prediction of the model

Our original file can be seen from the figure 3, and the fuzzed version of it by the simple fuzzer that we have written in C++ can be seen in the figure

4. Our model predicts the fuzzed version of the file from the original file in the figure 3 and outputs the file seen in figure 5. Unchanged parts of the predicted file is 50%, compared to the simple fuzzer which was 98%.

After testing our model with 100 data, we have found that 89% has successfully induced the behavior that we have expected, compared to the 17% that was found by the simple fuzzer.

5 Conclusion

We have learned about the fuzzers and how they work and why they are used. We have learned how and why the fuzzers improve their testing stages, so that they overcome the difficulty of testing the program or code millions of times to get a meaningful results. So, we have thought that, predicting the fuzzed file with a machine learning model by training it with inputs that already cause different behaviors for the code, would help us find the common code portions that causes this behavior.

We have written a simple input fuzzer that randomly flips given percentage of bits. This fuzzer was written in C++ to ensure that our program works fast as we may need to try the fuzzer for millions of times. We have used this simple fuzzer to generate data that is required for our machine learning model training.

We have also created a machine learning fuzzer model that creates a fuzzed input given an unaltered .jpg file. The fuzzed file has a 89% success rate to exhibit the timeout behavior that we have discussed. The written fuzzer model can be further improved; because in its current state, it changes 50% of the data, whereas we have used the training data were we have randomly changed 2% of the data. We also need to use more data to train the model as we could only used 100 data because of the time constraints.

References

[1] "Our guide to fuzzing." [Online]. Available: https://www.f-secure.com/en/consulting/our-thinking/15-minute-guide-to-fuzzing

[2] Y. Li, S. Ji, C. Lv, Y. Chen, J. Chen, Q. Gu, and C. Wu, "V-fuzz: Vulnerability-oriented evolutionary fuzzing," 01 2019.

[3] Z. Hu, J. Shi, Y. Huang, J. Xiong, and X. Bu, “Ganfuzz: A gan-based industrial network protocol fuzzing framework,” in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, ser. CF ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 138–145. [Online]. Available: https://doi.org/10.1145/3203217.3203241