Binary Fuzz Testing Method Based on LSTM

Xiaoxiao Yuan*, Limin Pan* and Senlin Luo*

Information System & Security and Countermeasures Experiments Center, Beijing Institute of Technology, Beijing 100081, China

*Corresponding author e-mail: 269179817@qq.com, panlimin2016@gmail.com, luosenlin2012@gmail.com

Abstract. Fuzzing is an effective software testing technique to find bugs. In the binary fuzzing, the attempt of generating test data mainly focuses on the improvement of the mutation algorithm, which lacks further screening of the test data. This paper proposes a binary fuzz testing method based on long short-term memory. This method records the executed path information by QEMU. Then we assign values to different code branches according to the frequency of executed paths. The path weights can be calculated. The LSTM model will be trained by test data and the path weights. Finally, we can take the test data as input and filter the data by referring to the path weights. This method can improve the efficiency of fuzz testing. Experiments on a variety of different types of binary programs show that compared to the state-of-the-art fuzzer American fuzzy lop, this method achieves higher code coverage and time efficiency in the same time.

1. Introduction

Fuzz testing is an automated vulnerability mining method proposed by B.P. Miller of the University of Wisconsin [1]. This method discovers software bugs by providing unexpected inputs and monitoring for abnormal results. A typical fuzzing process generally involves the following steps:

a) identifying the fuzzing target;
b) identifying the input of target program;
c) generating mutated data;
d) executing mutated data and monitoring the execution process;
e) assessing the availability of bugs;

Generating mutated data is the key step of fuzzing. However, existing methods focus on the way of mutation, ignoring the importance of data screening. This paper proposes a data screening method based on LSTM model to reduce the execution time of invalid data. We make the following contributions:

1) We present a way of calculating the value of code branches and path weights;
2) We show that fuzzers with data sieve performs better than general ones. Our data screening method improve the validity of data;
3) We evaluate L-AFL, a fully functional fuzzer that implements our approach, on three different programs and show that it is highly effective.
2. Background
Typically, a fuzzer is started with an initial set of seed input files, which are continuously transformed to generate malicious inputs either by random mutations, constraint-solving, or using a set of manually-defined heuristics.

The current fuzzing techniques can be broadly categorized into three main categories: Blackbox fuzzing [2], Whitebox fuzzing [3] and Greybox fuzzing. Blackbox fuzzers treat the target program as a black box with no internal inspection inside the program. While whitebox fuzzers require knowledge of the structure of the program being tested to generate input mutations to specifically target certain code fragments. Greybox fuzzers form a middle ground where they perform limited source code inspection. L-AFL is a greybox fuzzer.

Furthermore, the generation of input is influenced by the application exploration strategy. The directed fuzzer generates inputs to traverse a particular set of execution paths [4]. And the coverage-based fuzzer aims at generating inputs to traverse different paths of the application which can trigger bugs on some of these paths. L-AFL is a coverage-based fuzzer.

In this paper, we explore how to use machine learning to learn a strategy for data screening with the help of coverage. We first run the traditional fuzzing techniques for a limited time to obtain data which lead to new code coverage, and then use this data to learn a model to guide input sieve. Although our technique is applicable to any fuzzing system, we instantiate it on the current state-of-the-art AFL fuzzer, which is a genetic algorithm based greybox fuzzer.

3. Overview
To address the targets mentioned in previous section, we propose a way to predict whether test data is valid. The core idea of the algorithm is to assign a value to the code coverage based on the frequency of the code branch. Test data with a code coverage value below the threshold will then be filtered out. How we assign a value to the code branch value is based on a common sense, that the less frequent the frequency, the more interesting the branch. And the more interesting the branch, the higher weight we assign it. The value of the code coverage can be calculated according to the weight of the branch. With the code coverage value, we can screen out the test data by limiting the threshold.

Our framework consists of two parts: training and application. The model’s training part is shown in Figure 1. The mutation-based fuzzer mutate the seed taken from the seeds queue to obtain test data. Then the fuzzer executes the target program which takes test data as input. Collecting code coverage information by using the instrumentation technique during program execution. Calculating the code branch weight and the coverage path weights of the test data by referring to the statistical information of the frequency of occurrence of code branches. Segmenting and padding the test data and taking it as the input of the model. While taking the path weights as the model’s output. Training the LSTM model to get the model parameters.

![Figure 1. Model’s training part](image-url)
The model’s application part is shown in Figure 2. After mutation, the test data will be entered into the LSTM model. The LSTM model will output the code coverage weights of the test data. If the code coverage weights is greater than the threshold, the test data is allowed to execute. Otherwise, the test data is skipped. If the test data trigger something interesting during execution, it will be taken as the input gain and input into the seeds.

![Figure 2. Model’s application part](image)

### 3.1. Collecting code coverage

Code coverage information is obtained through dynamic instrumentation technique of binary programs. The L-AFL’s instrumentation operation is achieved by means of the QEMU tool. QEMU is an open source simulator. It uses a dynamic code translation mechanism to divide the target program into basic blocks for translation and execution. A basic block is a sequence of statements that are executed sequentially in a program. Each basic block has only one entry statement and one exit statement. The basic block can only be executed from the entry statement to the exit statement. The last statement of the basic block is a jump statement that jumps to the first statement of another basic block.

Based on the above properties of the basic block, the execution path information of the program can be recorded by the jump order of the basic block. By hooking the core function cpu_tb_exec of the accel/tcg/cpu-exec.c file in QEMU, an output function for recording the current basic block information is called each time the simulator executes the basic block. Finally we can get the basic block sequence in the program execution process.

The basic block sequence of program execution can be obtained in the following four steps:

- For convenience of description, each basic block in the program is marked with the symbol $b_i$. Then, by tracking the execution process of the binary program, a set of basic block sequences can be recorded as $(b_1, b_2, b_3, ..., b_n)$.

- The execution of a binary program is a process of jumping from one basic block to another. The different jump directions of the basic block can be called different branches. New branches are often more likely to cause security issues than new blocks. We use the symbol $(b_{src}, b_{dst})$ to mark the branch of the source block $b_{src}$ to the destination block $b_{dst}$. We will simply mark this branch as $e$, ie $e = (b_{src}, b_{dst})$. Then the execution path of the program can be described by a set of branch sequences $(e_1, e_2, e_3, ..., e_m)$.

- Considering that during the execution of the program, some branches will appear multiple times. In order to quickly compare the execution paths of different test data, and to distinguish the rarity of different branches, a simple classification of the above branch sequences is made. We count and merge the same branch $e_i$ and record it as $\tilde{e}_i$. Then, $\tilde{e}_i = (e_i, cnt(e_i))$, Where $cnt(e_i)$ represents the number of times that branch $e_i$ appears during this execution.

- The more times branch $e_i$ appears, the less meaningful it is to record it in detail. In order to further weaken the influence of the non-sensitive branch on the path count, the $cnt$ value is divided into eight categories according to the size, which are 1, 2, 3, 4-7, 8-15, 16-31, 32-127, $\geq$128. According to the interval in which the cnt value falls, we perform the set operation and get an eight-bit value, which is recorded as a bucket. For example, when the $cnt$ value is 1, it falls into the first interval. So the last position is 1, and the bucket is 1000 0000 (Binary), and so on. Rewrite the value of $t$ with bucket to $t_i =$...
Finally, we get the statistical set of the branch sequence \((t_1, t_2, t_3, ..., t_k)\). This statistical information is used as the code coverage collected in our experiment.

### 3.2. Calculating path weights

The path weights can be calculated from the branch sequence mentioned in the previous section. For each test data, there is a corresponding branch sequence. That is to say, different test data’s \(e_i\) corresponds to different bucket values. Traverse the branch sequence of all test data and record the minimum bucket value of \(e_i\). Then, the minimum bucket value is taken as the weight \(k_i\) of the branch. Path weights can be calculated from \(k_1 + k_2 + \cdots + k_k\).

### 3.3. Model training

The test data is a set of variable length binary files. The test data is first segmented with a fixed length. Then padding the portion of the undersized size. The test data can be used as input data for the model after preprocessing.

Due to the varying length and sequential nature of the input, Recurrent Neural Network (RNN) [5] was the obvious choice. RNNs are known to have problems with longer sequences. Due to this reason we chose Long Short-Term Memory (LSTM) [6] as our base recurrent unit. LSTMs extend the memory capability of a recurrent unit to longer sequences.

The state \(h_t\) and the output \(o_t\) of the LSTM unit can be expressed by the following formula using the input \(x_t\) and the state \(h_{t-1}\) of the previous moment:

\[
h_t, o_t = f(x_t, h_{t-1})
\]

The above formula can also be expressed by splitting the LSTM unit into three gates that increase or decrease the control of the unit state information, namely the forget gate, the input gate and the output gate. The forget gate determines how much information to discard (retain) by entering the last state \(h_{t-1}\) and the current input \(x_t\) into the \(\sigma\) function to produce a value between 0 and 1. 0 means completely discarded while 1 means completely reserved. The forget gate can be expressed as:

\[
f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)
\]

Where \(\sigma\) represents the Sigmoid activation function, \(W\) represents the weight vector, and \(b\) represents the output vector.

The input gate generates a value \(i_t\) between 0 and 1 by entering the previous state \(h_{t-1}\) and the current input \(x_t\) into the \(\sigma\) function to determine how much new information to retain:

\[
i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)
\]

A \(\text{tanh}\) layer obtains a candidate information to be added to the cell state through the previous state \(h_{t-1}\) and the current input \(x_t\):

\[
\hat{C}_t = \text{tanh}(W_c \cdot [h_{t-1}, x_t] + b_c)
\]

Multiplying \(i_t\) by the candidate information \(\hat{C}_t\) gives us the information we want to add to the cell state.

The output gate obtains a value \(o_t\) between 0 and 1 from the previous state \(h_{t-1}\) and the current input \(x_t\) to determine how much of the cell state information needs to be output:

\[
o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)
\]

The unit state information is activated by the \(\text{tanh}\) layer and multiplied by the output \(o_t\) to obtain the current state:
\[ h_t = o_t \cdot \tanh(C_t) \]  

By inputting the training input data and the training output data into the LSTM model, the eight parameters in the model are learned, namely the weight matrix \( W_f \) and the bias term \( b_f \) of the forgetting gate, the weight matrix \( W_i \) and the bias term \( b_i \) of the input gate, and the output gate. The weight matrix \( W_o \) and the offset term \( b_o \), and the weight matrix \( W_c \) and the bias term \( b_c \) of the state of the calculation unit.

### 3.4. Test data discrimination

The determination of test data consists of two steps:

1. Predicting path weights based on test data and LSTM models.
2. Filter test data based on path weights and user-defined thresholds.

First, the test data is used as the input of the LSTM model, and the resulting output is the predicted code coverage path weights. In the process of screening test data, users can customize the threshold according to the required efficiency. We select the path weights of the seed file as the threshold. If the path weights of the test data is below the threshold, the probability that the test data triggers the interesting branch is less. We can discard this test data directly. Otherwise, the test data is sent to the subsequent execution engine to complete the entire fuzzing process.

### 4. Evaluation

Since the mutation and execution module can be provided by common fuzzers, our experiment is based on the state-of-the-art fuzzer AFL to build the prototype fuzzer L-AFL. We evaluate the code coverage of L-AFL and AFL at the same time. We select four different types of target programs as the objects to be tested, namely: xml document parser libxml, elf file format analysis tool readelf, jpeg image data format processing library libjpeg, pdf file parser mupdf. Our experimental environment is given in Table 1.

| Operating system       | CPU               |
|------------------------|-------------------|
| Linux Ubuntu14.04 server | Intel Xeon E5-2650 v4 |

To collect the data for training the models, AFL was run for 24 hours. Test data, code coverage pairs were collected at a uniform 1% sampling rate. The training data was heterogenous in length. So test data longer than 10kB was segmented into a set of 10kB segments. After segmenting, the data was binned according to length and padded to the nearest chunk sized boundary. Calculate the weight of different branches according to the code coverage. Sum and get the path weights. Then the LSTM model was trained for 20 hours and used for L-AFL. The four target programs were run for 24 hours using AFL and L-AFL, respectively. And the code coverage of the four target programs was recorded. The experimental results are shown in Figure 3-6.
Figure 3. Libjpeg

Figure 4. Libxml

Figure 5. Readelf
It can be seen from the experimental results that L-AFL achieved higher code coverage in the four groups of evaluations. For all 4 programs, L-AFL achieves $3.10\times, 1.49\times, 4.03\times, 1.84\times$ edge coverage than baseline AFL, respectively. For pdf files with more complex formats and larger file sizes, L-AFL did not perform as well as AFL in the beginning period. This may be due to the fact that the pre-processing for the pdf format file consumes too much time, and the process of querying the model has a negative impact on the entire fuzz testing process.

5. Conclusion
This paper argues that, in the binary fuzz testing, in addition to focusing on the mutation of the seed, the data filtering process should be performed for the mutated test data. To address the target, we propose a binary fuzz testing method based on long short-term memory. First we record the executed path information by QEMU. Then we assign values to different code branches according to the frequency of executed path. Calculate the path weights of each test data. Train the LSTM model with test data and the corresponding path weights. Finally, we can take the test data as input and filter the data by referring to the predicted path weights.

We have implemented our fuzzing technique in a prototype based on AFL, called L-AFL and evaluated it on several applications. We also compared its performance with that of AFL, showing that, in almost every target program, L-AFL can achieve higher code coverage within the same time. This concretely demonstrates that our method is effective.

References
[1] Godefroid P, Kiezun A, Levin M Y. Grammar-based whitebox fuzzing [C]. ACM Sigplan Notices. ACM, 2008, 43 (6): 206-215.
[2] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities [J]. Communications of the ACM, 1990, 33 (12): 32-44.
[3] Godefroid P, Levin M Y, Molnar D. SAGE: whitebox fuzzing for security testing [J]. Communications of the ACM, 2012, 55 (3): 40-44.
[4] Haller I, Slowinska A, Neugenschwandtner M, et al. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations [C]. Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13). 2013: 49-64.
[5] Paul Rodriguez, Janet Wiles, and Jeffrey L Elman. A recurrent neural network that learns to count [J]. Connection Science, 1999, 11 (1): 5–40.
[6] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory [J]. Neural computation, 1997, 9 (8): 1735–1780.