The Stacked Seq2seq-attention Model for Protocol Fuzzing

1st Zicong Gao
Cyberspace Security Department
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, China
gzcrdfzf@126.com

2nd Weiyu Dong
Cyberspace Security Department
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, China
dongxinbaoer@163.com

3rd Rui Chang
Cyberspace Security Department
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, China
crix1021@163.com

4th Chengwei Ai
Cyberspace Security Department
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, China
chengwei.ai@qq.com

***Abstract*—Fuzzing is an effective approach to discover vulnerabilities in software by generating the amount of unexcepted data as inputs to a program. It is difficult to fuzz the protocol automatically because it is necessary to manually construct a template that satisfies the protocol specification to generate test cases. In this paper, we establish stacked seq2seq-attention models to generate protocol test cases automatically. Seq2seq-attention is a machine learning technique which has an encoder-decoder structure to output text sequences based on context. We evaluate the training effect of seq2seq-attention models with different layers of LSTM and point out that the highest correctness of test cases is achieved by 3 layers LSTM. Besides, we implement a fuzzer based on stacked seq2seq attention model and compare with grammar-based fuzzer, which result indicates the test cases generated by our fuzzer discover more unique basic blocks.**

***Keywords*—Machine Learning, Security, Seq2seq Attention, Fuzzing**

I. Introduction

Fuzzing is one of the widely used technology to find software vulnerabilities by repeatedly testing it with distinct inputs. There are two main types of fuzzing according to its method to input generation: (1)mutation-based fuzzing (2)grammar-based fuzzing. Mutation-based fuzzing mutates the initial test inputs automatically via different strategies. In contrast, grammar-based fuzzing requires the specific format to generate test inputs and is hard to be fully automatic for the grammar specification is written by hands usually. However, grammar-based fuzzing is more effective than mutation-based fuzzing [1], especially in constructing the complex structure input format such as PDF format files and webpages including HTML and Javascript documents [2].

In terms of protocol fuzzing, both grammar-based fuzzers and mutation-based fuzzers take important roles. SPIKE [3], Boofuzz [4] use the specification of the protocol to construct the test inputs. While Pulsar [5] and autofuzz [6] use clustering algorithm and n-gram based approaches to recover protocol automatically to generate inputs. The grammar-based fuzzers consume times to manually construct an initial template based on grammar for test case generation. On the other hand, the generation of Pulsar and autofuzz is limited by contents of finite length and the time overhead of traditional clustering algorithm. Therefore, rapidly automatic to generate inputs satisfied with the protocol is a great concern to protocol fuzzing.

In this paper, we explore whether it is probable to use machine learning to automatically generate the fuzzing test cases satisfied with the protocol specification. Specifically, we establish different layers Sequence to Sequence attention (seq2seq-attention) [7] model with Long Short Term Memory (LSTM) [8] unit to learn the probability distribution of the character in protocols. We collect the test cases produced by traditional grammar-based fuzzers as the input data and take unsupervised training. And we implement a machine learning-based fuzzer that use previously trained model to generate test cases. The evaluation of two specific FTP programs shows that it is feasible to use the machine learning to automatically construct inputs that conform to the protocol specification. And we investigate the effect of different layers of LSTM on test cases generation. Besides, we observe that our machine learning-based fuzzer is better than traditional grammar-based fuzzers in code coverage.

The contribution of this paper includes:

• We establish a stacked seq2seq-attention model to learn the protocol specification automatically and implement a machine learning-based fuzzer with it.

• We compare the correctness of test cases generated by seq2seq-attention model used different layers LSTM.

• We take evaluation between machine learning-based fuzzer and traditional grammar-based fuzzers on two FTP server programs in metric of code coverage, demonstrating our fuzzer is able to outperform than grammar-based fuzzers.

The rest of the paper is organized as follows: section II gives an introduction to the background knowledge related to the seq2seq-attention. Section III describes our model design. We present our experiment results in Section IV. And we conclude the paper in Section V.

II. Background

A basic sequence to sequence(seq2seq) [9] is an encoder-decoder architecture in figure 1 to receive input from a sequence and generate an output of variable length. The recurrent neural network(RNN) is used in encoder and decoder to handle sequence. And RNN unit is usually composed of an LSTM [8] or a GRU [10] which have various logic gate operations. The hidden semantic information is encoded to context vector *C* to transferred to the decoder hidden state. The seq2seq has achieved great success in machine translation, text summaries, and speech recognition because its encoder-decoder architecture does not limit the sequence length of input and output. However, seq2seq encodes all input sequences into a uniform semantic feature and then decodes them. Therefore, *C* must contain all the information in the original sequence, and its length becomes a bottleneck limiting the performance of the model.

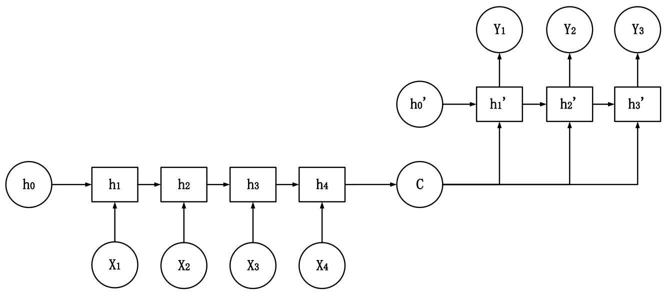

Fig. 1. Schematic diagram of sequence to sequence

The Attention mechanism [7] solves this problem by entering a different C_t at each time. The following figure 2 is a decoder with an Attention mechanism: Each C_t automatically selects the most appropriate context information for the Y_t that is currently being output. Specifically, we use a_{tj} to measure the correlation between h_j in the *j* th stage of the encoder and the *t* th stage in decoding. The context information C_t of the input of the *t*th stage in decoder comes from the weighted sum of all h_j to a_{tj}.

The a_{tj} is a output to softmax function of e_{tj}. Their representations are as follows:

$$a_{tj} = softmax(e_{tj}) \tag{1}$$

$$e_{tj} = a(h_t{}', h_j) \tag{2}$$

$$a(h_t{}', h_j) = V_a{}^{\intercal} \tanh(W_a[h_t{}'; h_t]) \tag{3}$$

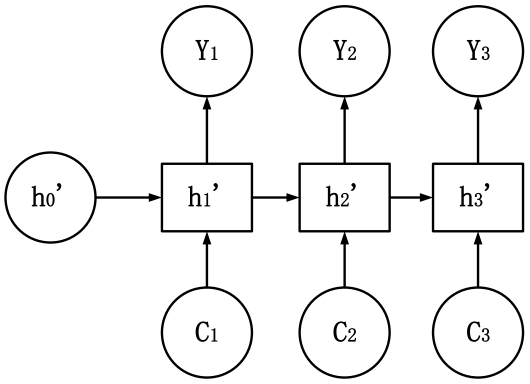

Fig. 2. Schematic diagram of attention mechanism in decoder

The W_a and V_a are weight matrices whose parameters are learned in the encoder and decoder. The calculation process of a_{tj} is also called the dot product matrix method.

III. Model design

In figure 3, we give an overview of stacked seq2seq-attention model which consists of three modules: encoder module, attention module and decoder module.

Firstly, the encoder reads an input text and transforms each character in the text into a one-hot coded vector X_t. One-hot encoding is used to represent each feature as a mutually exclusive binary element vector. So X_t = $(x_1$, x_2, ..., $x_m)^{\intercal}$. (x_i = 0 or 1) Then encoder uses LSTM with a forward hidden state h_t receive the one-hot coded vector X_t and output the next step hidden state h_{t+1}. The output state h_{t+1} is not only used for the next input word but also transferred to the attention module. To measure the effect of LSTM stacking, we separately establish 2 to 5 hidden layers LSTM in the encoder stage, and each layer consists of 128 hidden states. Because our data for training is lack of label, we take an unsupervised mode. And we observe models in different epochs which represent the number of learning produce execution in order to determine how well the training models under unsupervised mode.

In the attention module, each context information C_t is the sum up of the weight coefficient and the hidden state of the encoder. The weight coefficient is calculated by the dot product matrix method mentioned in section II.

Finally, the LSTM contains only one layer with 128 hidden in decoder stage, which is same as the origin seq2seq attention module [7]. The current LSTM cell takes the context vector C_t from the attention module, the previous hidden state of LSTM cell h_{t-1}' and the previous output vector Y_{t-1} as the inputs and output a vector Y_t after softmax function. The vector Y_t is the probability distribution of every character in time step t $(y_{1t}$, y_{2t}, y_{3t},..., $y_{mt})^{\intercal}$.

Because the output Y_t is the probability distribution of every character, there are three types of sampling for test cases generation: No sample, Sample, SampleSpace. No sample means to select the best character according to the distribution.

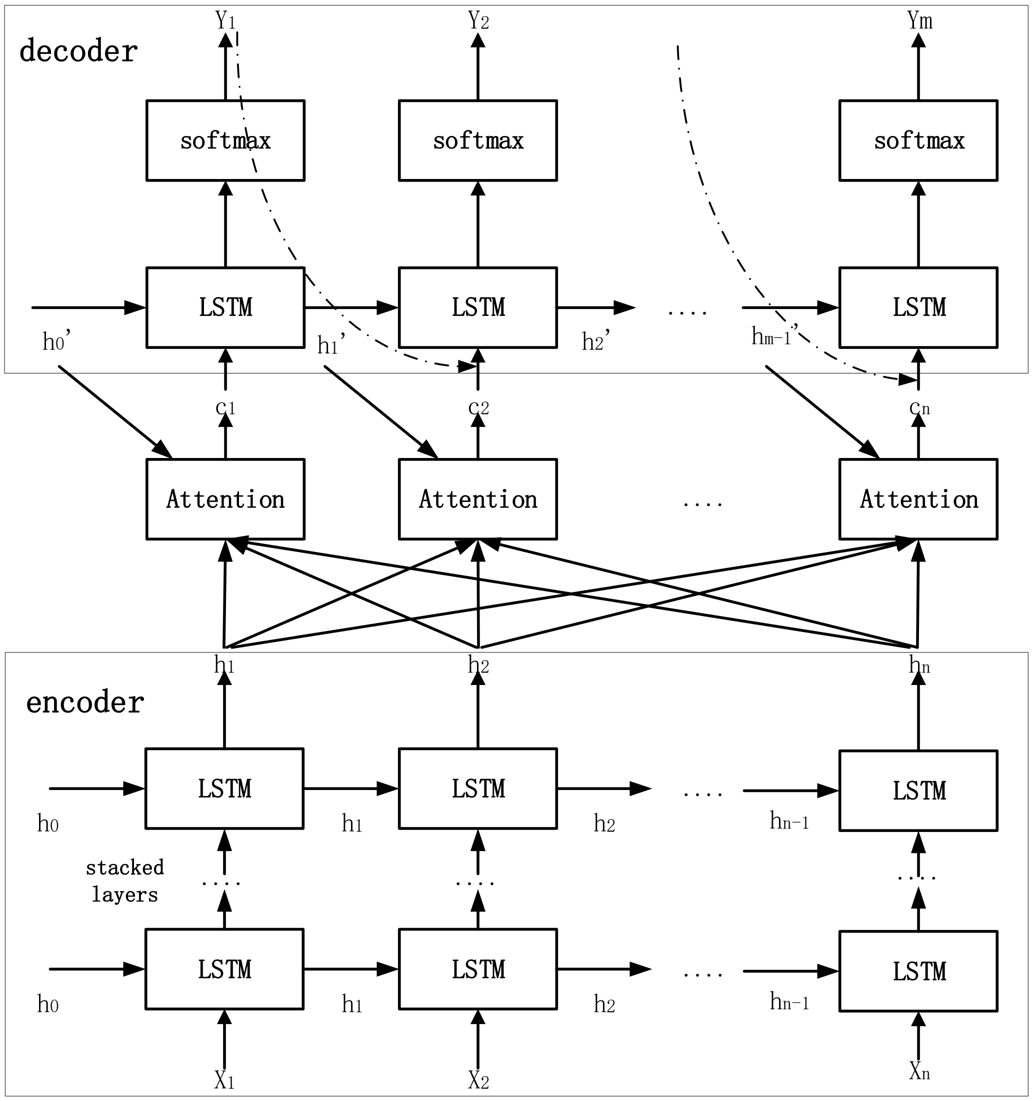

Fig. 3. Model overview of a stacked seq2seq-attention with LSTMs

The best predicted next character is not picked in the probability distribution under Sample strategy. SampleSpace is a combination of the above two methods and selects the best predicted character only when the current input sequence does not end with space. In our model, we select the SampleSpace strategy which is been proved to the most effective strategy in paper learn&fuzz [11].

As with general deep learning process, we calculate the cross-entropy loss function of our model during training and attempt to minimize the output L by Adam algorithm [12] in , where y_t' is the actual value and y_t is the predicted value.

$$L = -\frac{1}{N}\sum_{t=1}^{N} y_t \log(y_t{'}) + (1-y_t)\log(1-y_t{'}) \qquad (4)$$

Adam is an optimization algorithm that is different from random gradient drop. It calculates independent adaptive learning rates for different parameters by calculating the first-order moment estimation and second-order moment estimation of the gradient instead of maintaining a single learning rate to update all weights.

IV. Evaluation

A. Dataset and Experiment Setup

The dataset was collected by executing grammar-based fuzzer Boofuzz [4]. Boofuzz is one of benchmark of Sulley [13] and can construct network packets according to the protocol specification defined by users. We gave an RFC definition of FTP to Boofuzz and eventually got 36871 messages from it. These 36871 messages were corpus for stacked seq2seq-attention model and the max length of each messages was limited by 150 characters due to the overhead of training.

The stacked seq2seq-attention models training took place on a Ubuntu 16.04 system with NVIDIA GeForce 1080 Ti and were implemented using Keras [14] framework along with its python binding. In Keras, the training process split the validation set by default. After training, the model was called by a simple fuzzer that we implement to send the generation data.

In order to measure the effect of stacked seq2seq-attention model, we took experiments on two FTP server programs: Server-U 7.0.0.1 and Filezilla 2.23 which run on Windows server 2003. The first experiment counted how many test cases generated by our model conformed to FTP specification. The second experiment compared the code coverage by our machine learning-based fuzzer and Boofuzz by using instrumentation tool PIN [15].

B. Correctness

The principle for judging the correctness of the test case is that the header of the test case is the field specified by the FTP protocol. Each model of the different number of layers was given sufficient time to generate 10,000 test cases. We counted the number of correct test cases by content matching scripts based on the principle mentioned above and calculate the correct percentage. Figure 4 showed that as the epoch of training increased, the correct rate of test cases generated by different layers of model generation was improved. The experiment results also displayed the highest correctness of test cases generation is 3 layers stacked LSTM and the correctness of 4 layers and 5 layers decreased instead. It indicates using stacked LSTM over 3 layers does not enhance the effect of seq2seq-attention. The reason can be excepted from the general machine learning perspective that too many parameters used in complex models reduce the effect of training.

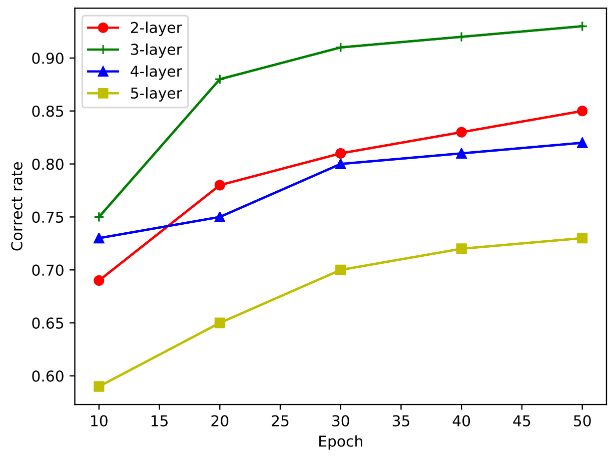

Fig. 4. Correct rate for different layers seq2seq-attention model from 10 to 50 epochs

C. Code Coverage

Unique basic block is used for measure the code coverage in this paper. And We have custom developed a pintool [15] for implementing statistics on unique basic blocks.

The highest correctness model with 3 layers that trained in 50 epochs was selected to generate test cases for a simple fuzzer with a network packet sent. An 4-hours evaluation between our machine learning-based fuzzer and boofuzz took place on Serv-U 7.0.0.1 and Filezilla 2.23 in a host with Window server 2003.

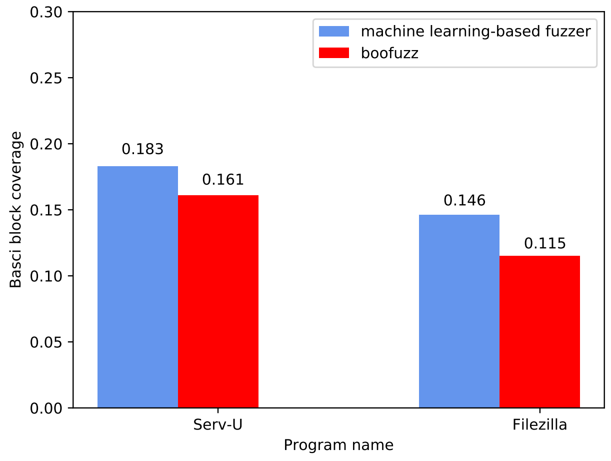

Fig. 5. Basic blocks coverage comparison

The basic block coverage of our proposed model is 3.1% and 2.2% higher than boofuzz on Serv-U and Filezilla in figure 5. The results demonstrate that the seq2seq-attention model can learn to generate test cases that discover new paths.

V. Conclusion

Test cases generation is the key to the protocol fuzzing. We establish stacked seq2seq-attention models with LSTM cells to learn the protocol format automatically. The model with 3 stacked layers that trained in 50 epochs generates test cases which meet the protocol specification in 93% correct rate. While more layer models do not better learn the format of the protocol in a limited time. Meanwhile, we implement a fuzzer based on trained seq2seq-attention model and compare with Boofuzz. The results of the comparison demonstrate that the machine learning-based approach can generate samples with higher code coverage. Future works include extracting data from actual network traffic and labeling the training data with code coverage tag to improve the model.

Acknowledgment

Thanks to project supported by the National Natural Science Foundation of China (No. 61802431).

References

[1] C. Miller, Z. N. Peterson *et al.*, "Analysis of mutation and generation-based fuzzing," *Independent Security Evaluators, Tech. Rep*, 2007.

[2] V. J. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "Fuzzing: Art, science, and engineering," *arXiv preprint arXiv:1812.00140*, 2018.

[3] D. Aitel, "Msrpc fuzzing with spike 2006," *Immunity Inc, August*, 2006.

[4] "boofuzz: A fork and successor of the sulley fuzzing framework," https://github.com/jtpereyda/boofuzz, 2015.

[5] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2015, pp. 330–347.

[6] S. Gorbunov and A. Rosenbloom, "Autofuzz: Automated network protocol fuzzing framework," *IJCSNS*, vol. 10, no. 8, p. 239, 2010.

[7] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[8] M. Sundermeyer, R. Schlüter, and H. Ney, "Lstm neural networks for language modeling," in *Thirteenth annual conference of the international speech communication association*, 2012.

[9] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[10] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[11] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 50–59.

[12] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[13] "sulley," https://code.google.com/archive/p/clusterfuzz.

[14] F. Chollet, *Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek*. MITP-Verlags GmbH & Co. KG, 2018.

[15] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "Pin: a binary instrumentation tool for computer architecture research and education," in *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*. ACM, 2004, p. 22.