Machine Learning for Black-Box Fuzzing of Network Protocols

Rong Fan(✉) and Yaoyao Chang

Beijing Institute of Technology, Beijing, China
fanrong_1992@163.com

Abstract. As the network services are gradually complex and important, the security problems of their protocols become more and more serious. Vulnerabilities in network protocol implementations can expose sensitive user data to attackers or execute arbitrary malicious code deployed by attackers. Fuzzing is an effective way to find security vulnerabilities for network protocols. But it is difficult to fuzz network protocols if the specification and implementation code of the protocol are both unavailable. In this paper, we propose a method to automatically generate test cases for black-box fuzzing of proprietary network protocols. Our method uses neural-network-based machine learning techniques to learn a generative input model of proprietary network protocols by processing their traffic, and generating new messages using the learnt model. These new messages can be used as test cases to fuzz the implementations of corresponding protocols.

Keywords: Black-box fuzzing · Proprietary network protocol
Machine learning

1 Introduction

Fuzzing is one of the most effective techniques to find security vulnerabilities in application by repeatedly testing it with modified or fuzzed inputs. State-of-the-art Fuzzing techniques can be divided into two main types: (1) black-box fuzzing [1], and (2) white-box fuzzing [2]. Black-box fuzzing is used to find security vulnerabilities in closed-source applications and white-box fuzzing is for open source applications. In terms of proprietary protocols, whose specification and implementation code are unavailable, black-box fuzzing is the only method can be conducted. There are two kinds of black-box fuzzing: (1) mutation-based fuzzing, and (2) generation-based fuzzing. Mutation-based fuzzing requires no knowledge of the protocol under test, it modifies an existing corpus of seed inputs to generate test cases. In contrast, generation-based fuzzing requires the input model to specify the message format of the protocol, in order to generate test cases. It has been proved that generation-based fuzzing performs much better, when compared to mutation-based fuzzing [3]. However, the input model of generation-based fuzzing can not be provided if neither the specification nor the

© Springer International Publishing AG, part of Springer Nature 2018
S. Qing et al. (Eds.): ICICS 2017, LNCS 10631, pp. 621–632, 2018.
https://doi.org/10.1007/978-3-319-89500-0_53

implementation code of the protocol are available. Therefore, it requires protocol reverse engineering to figure out the message format of the protocol.

There have been many approaches to find security vulnerabilities in protocol implementations. For example, static code analysis [4,5], white-box fuzzing [2,6], symbolic execution [7,8], and dynamic taint analysis [9] can help spotting vulnerabilities of the protocol, if the source code of the protocol is available. And if the specification of the protocol is already known, there are several modern fuzzers such as Sulley [10], Peach [11] and SPIKE [12] can be used. However, if the specification and implementation code of the protocol are both unavailable, things have become completely different. In this situation, only a few methods [13,14] can be applied for protocol vulnerability discovery. These methods provided first solution for automatically fuzzing proprietary protocols if a program analysis is not possible or hard to carry out. But they have used variants of traditional clustering algorithm and n-gram based approaches that are limited by contents of finite length.

In contrast with previous work, we make the first attempt at applying neural-network-based machine-learning techniques for black-box fuzzing of proprietary network protocol. Our method combine the concepts from fuzzing with the techniques from natural language processing. In specifically, we capture sufficient network traffic of an unknown protocol, then use seq2seq model with LSTM cells to learn a generative input model that can be used to generate test cases. Finally, we use the generative model to communicate with the implementation of unknown protocol.

The rest of the paper is organized as follows: Sect. 2 gives a brief introduction to neural-network-based machine-learning techniques. We introduce our method for black-box fuzzing of proprietary protocols in Sect. 3. Section 4 presents results of fuzzing experiments with our method. Related work is discussed in Sect. 5. We conclude in Sect. 6.

2 Preliminaries

We now give a brief introduction to neural-network-based machine-learning techniques.

2.1 Recurrent Neural Networks

Recurrent neural networks (RNNs) address the issue of information persistence, which traditional neural networks can't do. They are networks with loops in them, operating on a variable length input sequence $(x_1, x_2, ..., x_T)$ and consist of a hidden state h_t and an output y.

As Fig. 1 shows, a block of neural network, A, looks at some input x_t and outputs a value h_t. A loop is able to pass information from one step of the network to the next. A RNN can be thought of as multiple copies of the same network, each passing a message to a successor as Fig. 2.

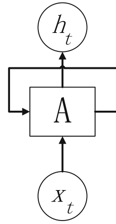

Fig. 1. Recurrent neural network with loops

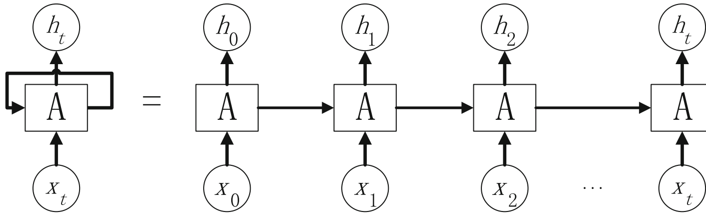

Fig. 2. Unrolled recurrent neural network

The RNN processes the input sequence in a series of time stamps. For a particular time stamp t, the hidden state h_t and the output y_t at that time stamp has equations as Eqs. 1 and 2 show.

$$h_t = f(h_{t-1}, x_t) \tag{1}$$

$$y_t = \phi(h_t) \tag{2}$$

In Eq. 1, f is a non-linear activation function such as sigmoid, tanh etc., which is used to introduce non-linearity into the network. And ϕ in Eq. 2 is a function such as softmax that computes the output probability distribution over a given vocabulary conditioned on the current hidden state. RNNs can learn a probability distribution over a character sequence $(x_1, x_2, ..., x_{t-1})$ by training to predict the next character x_t in the sequence.

In theory, RNNs are absolutely capable of handling long-term dependencies, where the predictions need more context. Unfortunately, in practice, RNNs become unable to learn to connect the information in cases shown in Fig. 3,

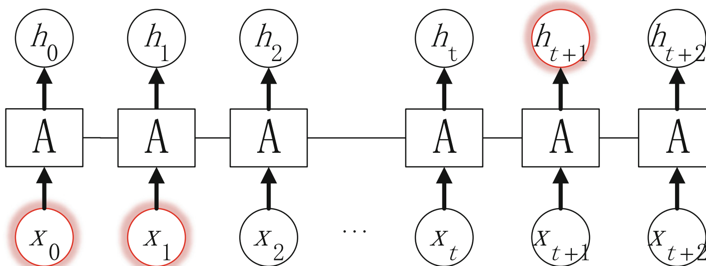

Fig. 3. RNN long-term dependencies

where the distance between the relevant information and the place that it is needed becomes very large.

2.2 Long Short-Term Memory Networks

Long short-term memory networks (LSTMs) are a special kind of RNN, explicitly designed to avoid the long-term dependency problem. They also have the form of a chain, which has repeating modules of neural networks. But instead of having a single neural network layer, the repeating module has a different structure as Fig. 4 shows.

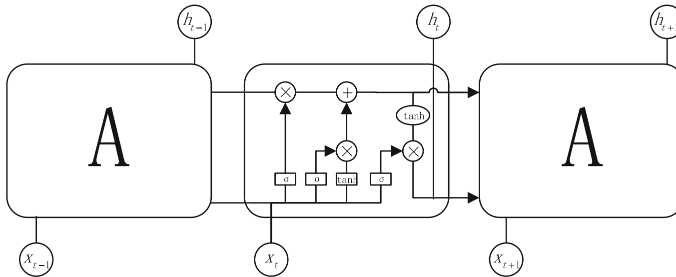

Fig. 4. LSTM repeating module with four interacting layers

The horizontal line crossing through the top of Fig. 4 is the cell state, which is the key of LSTMs. LSTMs are able to remove or add information to cell state with structures called gates, which composed out of a sigmoid neural net layer and a point wise multiplication operation.

2.3 Sequence to Sequence

A basic sequence-to-sequence (seq2seq) model, as introduced by Cho et al. [15], consists of two recurrent neural networks, an encoder RNN that processes a variable dimensional input sequence to a fixed-size state vector, and a decoder RNN that takes the fixed-size state vector and generates the variable dimensional output sequence. The basic architecture is depicted as Fig. 5.

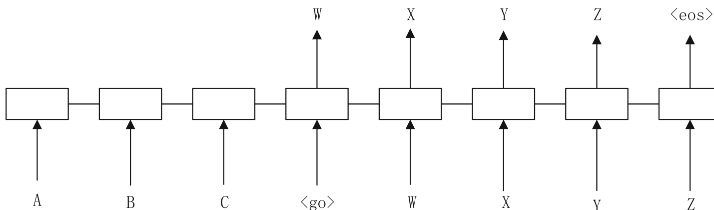

Fig. 5. Basic sequence to sequence

Each box in Fig. 5 represents a cell of the RNN, in our method an LSTM cell. Encoder and decoder can share weights or, as is more common, use a different set of parameters. We train the seq2seq model using a corpus of network recordings, treating each one of the message as a sequence of characters. Before training, we concatenate all the messages into a single file.

3 Methodology

The main idea of our method is to learn a generative input model over the set of network protocol messages. We use a seq2seq model that has been historically proved to be very successful at many automatic tasks such as speech recognition and machine translation. Traditional n-gram based approaches are limited by contexts of finite length, while the seq2seq model is able to learn arbitrary length contexts to predict next sequence of characters. The seq2seq model can be trained in an unsupervised mode to learn a generative input model, which can be used to generate test cases.

3.1 Training the Model

Before training the seq2seq model, we need to preprocess the corpus. Firstly, we count the non-repeating characters in the corpus, and sort them in a list according to their frequency of occurrence. Then, take each character as key and its order in list as value, storing in a dictionary. Finally, create a tensor file which replace all characters with its value in list. The main purpose of preprocessing is to calculate the number of batches N_b,

$$N_b = \frac{S_t}{S_b * L_s} \tag{3}$$

where S_t is the size of tensor file, S_b is the size of one batch, which is set to 50 by default. And L_s is the length of each sequence in batches.

After the preprocessing, we train the seq2seq model in an unsupervised learning mode. Due to the absent of training dataset labels, we are not able to accurately determine how well the trained models are performing. We instead train several models with different *epochs*, which is the number of learning algorithm execution. Therefore, an *epoch* is defined as an iteration of the learning algorithm to go over the complete training dataset. We train the seq2seq models M_s as shown in Algorithm 1 with five different numbers of epochs N_e: 10, 20, 30, 40 and 50. We use an LSTM model with 2 hidden layers, and each layer consists of 128 hidden states.

I_p is the initial path, where the checkpoints file stored in. N_s is the number of training steps to save intermediate result and the default setting is 1000.

Algorithm 1. Pseudocode for training

Input: I_p, N_b, N_e, N_s.
Output: M_s
1. **if** I_p is not None **then**
2. restore checkpoints from file
3. **end if**
4. new M_s
5. **for** e = 0,1,...,N_e **do**
6. initialize training session
7. **for** b = 0,1,...,N_b **do**
8. run training session with M_s
9. **if** (e*N_b+b) % N_s == 0 or
10. e == N_e-1 **and** b == N_b-1 **then**
11. **Output** M_s
12. **end if**
13. **end for**
14. **end for**

3.2 Test Case Generation

We use the trained seq2seq model to generate new protocol messages. At the beginning of the fuzzing, we always connect to the server, and take the received message for initial sequence I_s. Then request the seq2seq model to generate a sequence until it outputs one protocol message terminator like CRLF in ftp. Based on sampling strategy, there are three different strategies for message generation. Now, we give the details of these three different sampling strategies we make experiments with.

Max at Each Step: In this sampling strategy, we pick the best character in the predicted probability distribution. This strategy will generate protocol messages which are most likely to be well-formed. But this feature just makes the strategy unsuitable for fuzzing. Because we need test cases which are not quite the same as well-formed messages for fuzzing.

Sample at Each Step: In this sampling strategy, we don't pick the best predicted next characters in the probability distribution. As a result, this strategy is able to generate multifarious new protocol messages, which combines various templates the seq2seq model has learnt from the protocol messages. Due to sampling, the generated protocol messages will not always be well-formed, which is of great use for fuzzing.

Sample on Spaces: This sampling strategy combines the two strategies described above. It uses the best predicted character in the probability distribution when the last character of the input sequence is not a space. And it samples distribution to generate next character when the input sequence ends with a space, similar to the second strategy. More well-formed protocol messages compared to the second strategy can be generated by this strategy.

Algorithm 2. Pseudocode for sampling

Input: I_s, M_s, N, strategy type T.
Output: generated sequence S
1. put I_s into M_s
2. c = last character in I_s
3. **for** n = 0,1,...,N **do**
4. use M_s to predict the D_p
5. **if** $T == 0$ **then**
6. pick the best character in D_p
7. **else if** $T == 1$ **then**
8. weighted pick character in D_p
9. **else**
10. **if** c is space **then**
11. weighted pick character in D_p
12. **else**
13. pick the best character in D_p
14. **end if**
15. **end if**
16. append picked character to S
17. c = picked character
18. **end for**
19. **Output** S

N is the number of characters in the generated sequence, and we set it randomly to generate messages of arbitrary length.

4 Experimental Evaluation

4.1 Experiment Setup

In this section, we present results of fuzzing experiments with two ftp applications WarFTPD 1.65 and Serv-U build 4.0.0.4. We establish these two ftp applications on two servers, which run Windows Server 2003. The seq2seq models is trained on a personal computer, which has a Ubuntu 16.04 operating system. We implement a client program to communicate with ftp server, using the test cases generated by trained seq2seq model as input. If the program detects any error reports from ftp server, it records error messages in an error log. And we can validate whether the recorded error messages are indeed able to trigger vulnerabilities. Moreover, it is also feasible to implement a server program of the protocol to fuzz the client applications.

We use three working standards to evaluate fuzzing effectiveness:

Coverage: A basic demand shared by random and more advanced grammar-based fuzzers is that the instruction coverage should be as high as possible. In the case of our method, the fuzzer is able to fuzz the communication both ends but its coverage is highly depend on the network recordings.

Bugs: During the fuzzing process, we take the advantage of tool AppVerifier to monitor the running of ftp server. AppVerifier is a free runtime monitoring tool which can catch memory corruption bugs like buffer overflows, and it is widely used for fuzzing on Windows.

Performance Comparison: We record the statistical data when our fuzzer and existing fuzzer Sulley and SPIKE running with Serv-U build 4.0.0.4 for performance comparison. The statistical data include *Times*, *Time* and *Speed*. *Times* is the number of test cases sent, and *Time* means how many minutes was taken to find the bug. *Speed* indicates the number of test cases sent per second.

4.2 Corpus

We extracted about 10,000 messages for WarFTPD and 36,000 messages for Serv-U from network recordings. Most of the network recordings are generated by normal access to ftp server. And part of the traffic is generated by Sulley. Using Sulley is to improve the instruction coverage, because normal access may not include some less commonly used commands like MDTM, which is used to get the modification time of the remote file.

These 10,000 messages for WarFTPD and 36,000 messages for Serv-U which have both client and server side data are the training corpus for the seq2seq model we used in this work. We generate protocol messages using the trained seq2seq model, but the input data for ftp server should be transfered from network. Therefore we implement a client program to send the generated messages to ftp server.

4.3 Result

In order to obtain a reasonable explanation of coverage results, we select the network recordings of normal access to ftp server, and measure their coverage of the ftp application, to be used as a baseline for following experiments. When training the seq2seq model, an important parameter is the number of epochs. The results of experiments obtained after training the seq2seq model with 10, 20, 30, 40 and 50 epochs is reported here.

Coverage. Figure 6(a) and (b) show the instruction coverage obtained with **sample at each step** and **sample at spaces** from 10 to 50 epochs for WarFTPD and Serv-U. The figures also show the coverage obtained with the corresponding baseline.

We observe the following:

- The coverage for **sample at each step** and **sample on spaces** are above the **baseline** coverage for most epoch results.
- The trend for the coverage of WarFTPD and Serv-U from 10 to 50 epochs is quite unstable and unpredictable.
- The best coverage obtained with **sample at each step** and **sample on spaces** are both with 40-epochs.

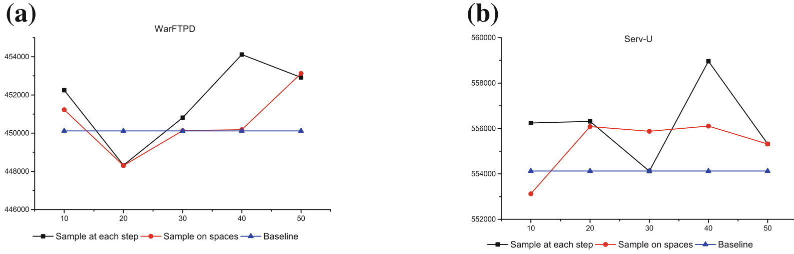

Fig. 6. Coverage for WarFTPD and Serv-U from 10 to 50 epochs.

Table 1. Bugs found by fuzzing

Application	Test results	Bug type
WarFTPD 1.65	CWD, CDUP, DELE, NLST, LIST	DoS
	USER	Buffer overflow
Serv-U build 4.0.0.4	SITE, CHMOD, MDTM, LIST	Buffer overflow
	XCRC, STOU, DSIZ	DoS

Bugs. Another working standard is of course the number of bugs found. Our method has been tested on WarFTPD and Serv-U two ftp applications, and after a nearly 4-days experiment, we found almost all of the already known vulnerabilities in these two ftp applications as Table 1 shows.

There is a $SMNT$ buffer overflow vulnerability in Serv-U not found, because of the incompleteness of network traffic we used to train the seq2seq model.

Performance Comparison. In addition to coverage and bugs, a third working standard of interest is performance of our method. We compared our fuzzer with existing fuzzer Sulley [10] and SPIKE [12]. As Table 2 shows, the efficiency of our method is slightly lower than that of the existing methods. This is because

Table 2. Performance comparison

Command	Sulley			SPIKE			Our fuzzer		
	Times	Time	Speed	Times	Time	Speed	Times	Time	Speed
SITE	987	1	16	3123	2	26	1307	3	9
CHMOD	1322	1	22	1212	1	20	3426	5	11
MDTM	1453	1	24	3127	2	24	1688	2	13
LIST	922	1	15	1562	1	26	987	2	8
XCRC	1348	1	22	2043	1	34	3366	5	12
STOU	2897	2	23	3031	2	25	1590	2	14
DSIZ	3188	2	26	3875	2	29	3425	6	10

that the generation of test cases by seq2seq model takes a lot of time. However, Sulley and SPIKE can only be used when the specification of the protocol is available, but our method is able to fuzz proprietary network protocols, whose specification and implementation code are both unavailable.

5 Related Work

Protocol Reverse Engineering. Over a decade ago, the process of reverse engineering a network protocol was a tedious, time-consuming and manual task. Nowadays, there are plenty of methods proposed for automating the process of protocol reverse engineering. The methods can be divided into two branches: On the one hand, methods that utilize the protocol implementation [16,17], and on the other hand, those extract protocol specification from network recordings only. The Protocol Informatics Project [18] uses a bioinformatics method to implement byte sequence alignment of similar message formats. The Discoverer tool [19] present a recursive clustering approach of tokenized messages. Biprominer [20] and ProDecoder [21] presented by Wang et al. focused on binary protocols, they retrieve statistically relevant keywords and sequencing. Based on data mining techniques, AutoReEngine [22] reveal keywords and their position within messages. It is particularly difficult to extract protocol specification in case the protocol implementation code can not be available for network security staff, but network recordings only. These approaches provide first means for automatically identify message field boundaries and formats, but unfortunately, they are not able to relate variable fields over temporal states.

Protocol Fuzzing. Fuzzing is one of the most effective techniques to uncover security flaws in application by generating test case in an automated way. Two types of fuzzing can be discriminated here: (1) black-box fuzzing [1] which a tester can only seeing what input and output of an application, and white-box fuzzing [2] that allows the tester to inspect the implementation code (either binary or source code) and for instance, take advantage of static code analysis and symbolic execution. This classification is obviously applicable to protocol fuzzing as well. Most well-known black-box random fuzzers today support generation-based fuzzing, e.g. Peach [11] and SPIKE [12], can be used to fuzz protocol implementation when the specification of the protocol is available, but can do no more when the protocol is unknown. Only few approaches can fuzz protocol in situation where specification and implementation code are both unavailable. AutoFuzz [13] and PULSAR [14], which both infer the protocol state machine and message formats from network traffic alone.

6 Conclusion

It is a challenging problem of computer security to find vulnerabilities in the implementations of proprietary protocols. To the best of our knowledge, this is the first attempt to do black-box protocol fuzzing using neural network learning algorithm, which is able to find vulnerabilities in protocol implementations,

whether or not the code nor specification are available. We presented and evaluated algorithms with different sampling strategies to automatically learn a generative model of protocol messages.

Although we have applied our method on very common network protocols, the method is also able to find vulnerabilities in unusual implementations, such as in embedded devices and industrial control systems. Moreover, we are considering adding some form of reinforcement learning in our future work to guide the fuzzing process with coverage feedback from the application.

References

1. Sutton, M., Greene, A., Amini, P.: Fuzzing: Brute Force Vulnerability Discovery. Pearson Education, London (2007)
2. Godefroid, P., Levin, M.Y., Molnar, D.A., et al.: Automated whitebox fuzz testing. In: NDSS, vol. 8, pp. 151–166 (2008)
3. Miller, C., Peterson, Z.N.: Analysis of mutation and generation-based fuzzing. Technical report, Independent Security Evaluators (2007)
4. Sotirov, A.I.: Automatic vulnerability detection using static source code analysis. Ph.D. thesis, University of Alabama (2005)
5. Chess, B., McGraw, G.: Static analysis for security. IEEE Secur. Priv. **2**(6), 76–79 (2004)
6. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: ACM Sigplan Notices, vol. 43, pp. 206–215. ACM (2008)
7. Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice: preliminary assessment. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 1066–1071. ACM (2011)
8. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)
9. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: 2010 IEEE Symposium on Security and Privacy (SP), pp. 317–331. IEEE (2010)
10. Amini, P., Portnoy, A.: Sulley: pure Python fully automated and unattended fuzzing framework (2013)
11. Eddington, M.: Peach fuzzing platform. In: Peach Fuzzer, p. 34 (2011)
12. Spike fuzzing platform. http://www.immunitysec.com/resourcesfreesoftware.shtml
13. Gorbunov, S., Rosenbloom, A.: Autofuzz: automated network protocol fuzzing framework. IJCSNS **10**(8), 239 (2010)
14. Gascon, H., Wressnegger, C., Yamaguchi, F., Arp, D., Rieck, K.: Pulsar: stateful black-box fuzzing of proprietary network protocols. In: Thuraisingham, B., Wang, X.F., Yegneswaran, V. (eds.) SecureComm 2015. LNICST, vol. 164, pp. 330–347. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-28865-9_18
15. Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y.: Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078 (2014)
16. Comparetti, P.M., Wondracek, G., Kruegel, C., Kirda, E.: Prospex: protocol specification extraction. In: 2009 30th IEEE Symposium on Security and Privacy, pp. 110–125. IEEE (2009)

17. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 317–329. ACM (2007)
18. Beddoe, M.: The protocol informatics project (2004)
19. Cui, W., Kannan, J., Wang, H.J.: Discoverer: automatic protocol reverse engineering from network traces. In: USENIX Security Symposium, pp. 1–14 (2007)
20. Wang, Y., Li, X., Meng, J., Zhao, Y., Zhang, Z., Guo, L.: Biprominer: automatic mining of binary protocol features. In: 2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), pp. 179–184. IEEE (2011)
21. Wang, Y., Yun, X., Shafiq, M.Z., Wang, L., Liu, A.X., Zhang, Z., Yao, D., Zhang, Y., Guo, L.: A semantics aware approach to automated reverse engineering unknown protocols. In: 2012 20th IEEE International Conference on Network Protocols (ICNP), pp. 1–10. IEEE (2012)
22. Luo, J.Z., Yu, S.Z.: Position-based automatic reverse engineering of network protocols. J. Netw. Comput. Appl. **36**(3), 1070–1077 (2013)