Vol.2022-SE-210 No.32
2022/3/12

Efficient Machine Learning Method for Protocol Fuzz Testing: Improvement of Sequence-to-Sequence Model and Refined Training Data

Bo Wang[1,a)] Toshihiro Maruyama[1,b)] Ako Suzuki[1,c)] Yuichi Kaji[2,d)]

Abstract: Fuzz testing is one of software testing methods for finding software vulnerabilities and is used as a technology for finding unknown security vulnerabilities as a black-box test. Although many fuzz testing methods that are based on machine learning have been investigated, they cannot analyze and learn the real-time status of communication protocol. We focus on the method of efficient machine learning for protocol fuzzing, and present major problems of current fuzzing tools, and introduce techniques to get around the problems with an improvement of Sequence-to-Sequence model and refined training.

Keywords: Fuzz Testing, Protocol, Machine Learning, Seq2Seq

1. Introduction

With the practical use and spread of IoT (Internet of Things) devices, there are concerns about security threats to IoT devices, many of which are caused by software vulnerabilities and erroneous packets in a communication protocol. *Fuzz testing* has been used as one of the most widely-deployed techniques to discover software "security vulnerabilities" since its introduction in the early 1990s [1]. At a high level, fuzz testing or *fuzzing* refers to a process of repeatedly running a program with machine-generated inputs that may be syntactically or semantically malformed. In practice, attackers routinely deploy fuzz testing in scenarios such as exploit generation and penetration testing. Generally, fuzz testing is one of the most popular software testing methods for finding software vulnerabilities in a device and a computer system and is used as a technology for finding unknown security vulnerabilities in the software that is treated as a "black box," or a "grey box," or a "white box" depending on the situation of the test.

Although many fuzz testing methods have been investigated so far (see [2], [3] for extensive surveys of fuzz testing tools), they cannot grasp the real-time status of a communication protocol. Moreover, there are still some serious challenges to be solved. Although many fuzz testing methods based on machine learning have been investigated, they cannot analyze and learn the real-time status of communication protocol.

The authors focus on efficient machine learning for protocol fuzz testing. This paper gives a brief review of the status of communication protocol that was attacked and exposed to security vulnerabilities. We present the major problems of current fuzz testing tools (*fuzzers*) that have not yet been solved, and then give the purpose of this paper, which is an approach for efficient machine learning for protocol fuzz testing.

[1] JVCKENWOOD Corporation, 3-12, Moriyacho, Kanagawa-ku, Yokohama-shi, Kanagawa, 221-0022 Japan
[2] Nagoya University, Furo-cho, Chikusa-ku, Nagoya, 464-8601 Japan
[a)] wang.bo@jvckenwood.com
[b)] maruyama.toshihiro@jvckenwood.com
[c)] suzuki.ako@jvckenwood.com
[d)] kaji@icts.nagoya-u.ac.jp

2. Background

2.1 Fuzz Testing

Fuzz Testing is a technique in which a large number of machine-generated inputs, both valid and invalid, are fed into a program to search for flaws and vulnerabilities.

Fuzzers are automated by processes, feeding data (initial program knowledge) and reporting it out that any discovered interesting program states. On the other hand, users without automated fuzzers are required to provide feeding data and to analyze output program states. Traditionally, "interesting program states" were program crashes that identified flaws and vulnerabilities in the program, but more complex program monitoring techniques now allow for the identification of other types of interesting states.

The goal of a fuzzer is to create inputs that cause the program to execute program paths, discovering those that lead to interesting program states. Thus, fuzzers are repeatedly measured by various types of explored program paths, termed in test coverage [4]. A previous research, by Gorbunov et al. [5], was the fuzzer, providing protocol fuzz testing with machine generated random data and preparing them before the test.

2.2 Fuzzing and Machine Learning

In [6], Wang et al. explained the reasons why machine learning techniques can be used for fuzz testing scenarios and identified several different stages in which machine learning has been used.

The use of machine learning techniques in fuzz testing requires three prior conditions:

- Training requires massive samples
- Supervised learning requires the labeled data
- The inputs require to be converted to vectors

For the first and second conditions, the fuzz testing process is sufficient because the fuzz testing can produce a large number of test samples and crash samples, which can be labeled during sample generation (e.g., whether code coverage increases during execution).

Since many of the input files of fuzz testing can be treated directly as textual data, natural language processing provide an effective means of converting various data into vectors. Therefore, the third prior condition is also satisfied. The satisfaction of these prior conditions and the advantages of machine learning have led to rapid growth in the research of machine learning-based fuzz testing.

Machine Learning has been used to generate new inputs in the fuzz testing process and, to a lesser extent, to improve post-fuzz testing [4]. Unsupervised learning has seen the most successful applications to input generation, with fuzz testing tools such as AFL [7] integrating genetic algorithms (GAs) into the input generation process. There are also recent applications of both supervised and reinforcement learning (RL) to input generation. Additionally, all three types of ML have been applied to symbolic execution, primarily to reduce constraint equation solve times, such as *supervised learning*, *unsupervised learning*, and *reinforcement learning*. Both supervised and unsupervised learning have been applied to post-fuzz testing processes primarily for crash triage and root cause categorization. These pieces of the fuzz testing process tend not to be bottlenecks, which may account for the lack of research. Fan et al. have a method to automatically generate test cases for black-box fuzz testing of proprietary network protocols based on machine learning techniques to learn a generative input model [8]. Zhao et al. proposed a method to learn the protocol frame structures from communication traffic and generates fake but plausible messages as test case, but it is not suitable for fuzz testing [9].

2.3 Security Vulnerabilities

Security vulnerabilities mean that "any mechanism that could lead to a breach of the security of a system in the presence of a threat. Vulnerabilities may arise unintentionally due to inadequacy of design or incomplete debugging. Alternatively, the vulnerability may arise through malicious intent, e.g. the insertion of a Trojan horse [10]."

Software vulnerabilities are weaknesses or flaws present in a program code. Unfortunately, conventional testings and manual code reviews cannot always find every vulnerability. Left alone, vulnerabilities can impact the performance and security of software. They could even allow untrustworthy agents to exploit or gain access to products and data.

2.4 Security Vulnerabilities of Communication Protocol

Protocols are pervasive in computer systems, as they enable communication among parties over a local network or the internet. Therefore, protocol implementations are an appealing target for malicious actors, as a vulnerability in an implementation may be remotely exploited. However, it is challenging to test the security of protocol implementations, since protocols are most often stateful in nature. This means that, compared to stateless programs, the input space for testing is not limited to the format of individual messages (which can be very large by its own), but is further enlarged by the potential combinations of several messages. Therefore, protocol security testing is in need of techniques that can efficiently test such input space, by taking into account the protocol states [11].

There is an effort that tries to get over the security vulnerabilities of a communication protocol through the approach of fuzz testing. *Protocol Fuzz Testing* [12] is a fuzz testing to send forged packets to the tested application, or eventually acts as a proxy, modifying requests on the fly and replaying them.

3. Proposed Scheme

3.1 Sequence-to-Sequence Model

In protocol fuzzing, we need to model the state transition of a target software that is regarded as a black box. The state transition is brought by a protocol message in a predetermined format, where the data is interpreted according to the current state of the communication system. This makes the relation between the states and the protocol messages more complicated and less obvious, making the protocol fuzzing extremely difficult.

One possible approach to the issue is not to try to control everything. Recently there is wide recognition that machine learning algorithm contributes to constructing a model of complicated data and phenomenon, even though the theoretical mechanism of the contribution is not understood well. We would like to take a similar approach; use a widely recognized Sequence-to-Sequence (Seq2Seq) model to learn the state transition of a communication protocol.

Seq2Seq can process input and output sequences of various lengths and is good at handling sequential data such as continuous values. In addition, Pytorch supports Encoder and Decoder processing in the Seq2Seq model [13], [14], which is helpful for the realization of the investigated approach. Seq2Seq model is a learning model that converts an input sequence into an output sequence [15]. In this context, the sequence is a list of symbols, corresponding to the words in a sentence. The seq2seq model has achieved great success in fields such as machine translation, dialogue systems, question answering, and text summarizing. All of these tasks can be regarded as the task to learn a model that converts an input sequence into an output sequence. To achieve the task, the Seq2Seq model employs an encoder-decoder model as its component. The encoder has a mechanism that converts input data (such as image, text, audio, video, etc.) into a (fixed-length) feature vector. The decoder receives the feature vector from the input and generates the output data that corresponds to the input data. In a sense, the input is encoded in the feature vector, and the encoded information is decoded and processed by the decoder.

3.2 Possible Problems and Our Approach

3.2.1 Motivation

In the conventional approach of fuzz testing, we prepare a massive quantity of machine-generated fuzz data, feed the data to a target software, and try to detect vulnerabilities. This approach is however not suitable for protocol fuzz testing because the data used in communication protocol is strongly context-sensitive. Randomly generated fuzz data often bring early termination of the communication, which makes it difficult to increase the test coverage of the target software. For efficient protocol fuzz testing, it is essential to filter out fuzz data that seems not like protocol data, and machine learning techniques such as Seq2Seq might be contributing for the sake; we construct a model of a communication protocol by machine learning and use the model to qualify randomly-generated fuzz data.

In this context, the sequence is a list of symbols, corresponding to the words in a sentence. We will use the model to gain efficient protocol fuzz testing. Generally, for modeling, the natural language translation of the Seq2Seq model is processed by pairing massive input language sentences and target (output) language sentences, where sentences are encoded as a fixed-length feature vector. However, this machine learning model is not suitable to learn communication protocols because communication data in a protocol are dynamic, long, and variable both in length and in contents.

3.2.2 Problems

Although many fuzz testing methods have been investigated, there are still some problems that need to be solved.

Problem 1: preparation of fuzz data. Conventional fuzz testing tools cannot perform fuzz testing on the communication protocol of target devices, because we must prepare massive fuzz data in advance. The preparation of massive fuzz testing data is costly, and efficient tests cannot be achieved during fuzz testing. It is difficult to prepare these test data on the communication protocol, since a protocol sequence is a long and context-sensitive bit stream. For fuzz testing protocol, we need to make a bit conversion for any bit, which is costly and useless most time.

Problem 2: context in the data. When fuzz testing the communication protocol, the random protocol data (bitstream) cannot gain efficient machine learning as input of the Seq2Seq model. The reason is that random protocol data has no intention of crashing a system. The tentative goal of fuzzing is to increase the test coverage, as it contributes to finding out new and/or unexpected information about the system specifically unidentified bugs. This suggests that it is not sufficient to let the machine learn random data, as such random data are not intended to expose unknown behavior of the system and thus not effective for producing a good learning effect.

3.2.3 Contribution

To fit the Seq2Seq model to communication protocols, we consider focusing on a certain part of communication data instead of trying to learn the entire communication data. In the case of Bluetooth protocol, for example, the “operations field data” can be a good candidate to be learned.

Another concern of the approach is that it is not sufficient to let the model distinguish seemingly correct/incorrect protocol data. To boost the efficiency of fuzz testing, we attach training data with a heuristically determined tag that indicates if the data is likely to bring software failure or not. This makes the learned model more informative and should contribute to improving the efficiency of protocol fuzz testing.

3.3 Refined Training Data

3.3.1 Random Fuzz Generation and its Limit

Generation of random inputs is a focal point in fuzz testing, but the truly random generation of inputs is useless in protocol fuzzing.

Table 1 shows the taxonomy (classification) of common operations that are used in generating fuzz in a protocol fuzz test [16]. The generation of fuzz can be classified into three kinds of taxonomy, 1) *fuzz testing messages*, 2) *fuzz testing payload*, and 3) *fuzz testing fields* that 3) is classified into two data types 3a) *fuzz testing fields* (*numerical*) and 3b) *fuzz testing fields* (*string*), depending on the data type of contents in the targeted field.

Fuzz testing messages. Fuzz testing messages tries to bring confusion on the sequence of messages that are sent to the target device. Typically this fuzz testing takes a well-formed message (e.g. captured from a previous session, or an older message in the current session of the protocol) and inserts this message at a random position in the message sequence. As a result, the target device receives several valid messages followed by a message with a valid structure, but at an unexpected position in the message sequence.

Fuzz testing fields and payloads. Fuzz testing fields and fuzz testing payloads mutate the internal structure of individual messages. Fuzz testing payloads adds, removes, or duplicates payloads. This results in missing or extra payloads in the message. Fuzz testing fields alters the values that are stored in the fields of a payload. Numerical fields

Table 1 Taxonomy of Protocol Fuzz Testings.

	Taxonomy	Approach of Fuzz Testing	Description
1)	Fuzz Testing Messages	Insert random message	Insert a well-formed message at a random position in the messages sequence
2)	Fuzz Testing Payloads	Insert random payload	Inserts a random payload at a random position in the list of payloads
		Duplicate random payload	Duplicates a randomly chosen payload
		Remove random payload	Removes a randomly chosen payload
3a)	Fuzz Testing Fields(numerical)	Set to random number	Set the field to a randomly chosen number
		Set to zero	Set the field to zero
3b)	Fuzz Testing Fields(string)	Append random bytes	Append a sequence of random bytes
		Modify random byte	Replace a randomly chosen byte with a random byte
		Set to the empty string	Set the field to the string of length zero
		Insert string termination	Insert the string termination character at a randomly chosen position

often indicate the type or length of other fields/payloads. In contrast, string fields hold values that are typically used as inputs to the system's internal functions, and hence they often have specific syntax, for example, a string representing a date.

Above mentioned operations are effective for most fuzz tests such as simple file fuzzing, but sole utilization of random operations is not effective in protocol fuzzing. It is common that a system aborts a communication session once an inconsistency to an underlying protocol is detected because digital communication is erroneous in nature. Even if there is no malicious entity in the communication, unfortunate accidents and system failures easily corrupt the communication. Therefore, a communication system usually has a mechanism that tries to detect errors extensively. Most randomly generated fuzz data are detected by the error checking mechanism and bring controlled abortion of the communication session. This means that it is difficult to increase the test coverage of the target software even if we prepare a massive quantity of random fuzz data that are generated by the above-described taxonomy.

In a machine-learning based fuzzing, we train a learning model by providing training data and letting the model distinguish good fuzz data that are likely to detect software vulnerabilities and bad fuzz data that are less likely to do that. The performance of the learning model is affected by the quality of the provided training data. Better training data makes the learning model more effective and informative, boosts the efficiency of fuzz testing. With this regard, it is easily understood that randomly-generated fuzz data is not suitable as the training data.

It is essential to prepare certain amounts of training data that have better quality than randomly generated fuzz data.

3.3.2 Refined Training Data with Tag

It is ideal if we can prepare a sufficient amount of training data that indeed makes the target software malfunction. However, such training data are hardly obtained in practice. Therefore, we consider preparing training data by slightly modifying rather small amounts of quality fuzz data. The obtained training data have better quality than randomly generated ones and are called as *refined training data* in this paper.

To generate refined training data, we focus on operation fields in a protocol packet. The data in the focused operation fields are processed and/or changed by tactics. For example, we can change the data structure of a header of one operation field, or we can change a specific value in a header of one operation field to special values such as the maximum, null, or out-of-the-boundary value.

Once the refined training data is prepared, we may be given an empirical tag to each datum of the training data. The tag indicates if the generated datum is likely to cause any malfunction of the target software. Of course, such tags are not always available, and some empirical work should be needed to determine the tag of a specific datum. This point will be described in a later section.

The refined training data with Tag are then used to train a learning model.

3.4 Improvement of Seq2Seq Model

To fit the Seq2Seq model to communication protocols, we improve the Seq2Seq model. We consider focusing on a certain part of communication data instead of trying to learn the entire communication data. In the case of Bluetooth protocol, for example, the "operations field data" can be a good candidate to be learned.

As described previously, a Seq2Seq model consists of an encoder and a decoder. Both the encoder and decoder consist of several long short-term memory (LSTM) blocks concatenated. In the typical training phase of the model, the LSTM block of the encoder is provided with an input symbol, and the LSTM block of the decoder is provided with an output symbol. Upon this framework, we consider to provide the LSTM block of the encoder a piece of operation field data as its input and consider providing the LSTM block of the decoder the tag that has been attached to the corresponding input.

4. Case Study and Evaluation

4.1 The Target of the Case Study

Bluetooth is a wireless technology that is designed for data exchange between two devices. The technology supports wireless communication with a physical range of 10 to 100 meters and is widely used in many devices including smartphones, personal computers and gaming consoles, etc [17]. Even though Bluetooth IoT (Internet of Things) de-

Table 2 Operation Fields of Major Bluetooth Profiles.

BT Profile	Operation Field
OBEX	Connect
	Disconnect
	Put
	Get
	Abort
PBAP	Connection ID
	Name
	Type
	Application Parameters
MAP	Connection ID
	Type
	Application Parameters
	Body/EndOfBody
	Flags
	Name
FTP	Connect
	Disconnect
	Put
	Get
	Abort
	SetPath
	Action
	Session
OPP	Count
	Name
	Type
	Length
	Time
	Description
	HTTP
	Body/EndOfBody
	Single Response Mode
	Single Response Mode Parameters
	Session Parameters
	Session Sequence Number

vices are widely spreading recently, there are concerns about security threats to Bluetooth IoT devices [18], [19]. The security threats to Bluetooth should be challenged and solved, the authors consider.

According to the discussion in previous sections, we conduct a case study for the Bluetooth protocol. In order to use Bluetooth, a device must be compatible with the subset of Bluetooth profiles (often called services or functions) necessary to use the desired services. A Bluetooth profile is a specification regarding an aspect of Bluetooth-based wireless communication between devices. Bluetooth has various types of Bluetooth profiles of communication protocols [20], including OBEX, PBAP, MAP, FTP, and OPP. Every Bluetooth profile has its own operation fields as Table 2.

In this case study, we focus OBEX (OBject EXchange) profile of Bluetooth. The profile defines a communication protocol that facilities the exchange of binary objects between devices. The profile is versatile and almost all Bluetooth devices support the OBEX profile.

4.2 Training the Seq2Seq Model

As discussed in previous sections, the training data set is made by some tactics and attached with heuristically determined tags.

The training data is generated by modifying the operation field data of the OBEX protocol. Table 3 shows examples of operations that are used to prepare the refined training data. The operations include, for example, changing the data structure to a binary, and to change specific field values to unexpected ones such as the maximum, null, and out-of-range values.

Then the fuzz data are attached heuristically tags which are either of "e0" or "f0." The tag "e0" means that the fuzz data are less likely to cause any malfunction of the target device, while the tag "f0" means that the fuzz data are more likely to cause some malfunction of the target. The tags are computed empirically according to the preceding works of the authors.

Once the refined training data with empirical tags are prepared, we start training the Seq2Seq model. The encoder is given with the fuzz data as its input, and the decoder is given with the empirical tag as its input. For training the Seq2Seq model, we have six training sets as Table 4; a hundred pairs of Connect field data, a hundred pairs of data that is composed of 20 pieces of data for each of five OBEX operations, two hundred pairs of data that is composed of 40 pieces of data for each of five OBEX operations, three hundred pairs of data that is composed of 60 pieces of data for each of five OBEX operations, four hundred pairs of data that is composed of 80 pieces of data for each of five OBEX operations, and five hundred pairs of data that is composed of 100 pieces of data for each of five OBEX operations. Each empirical fuzz data has its tag, such that it is tagged with "e0" or "f0." The Seq2Seq model is trained to "translate" fuzz data which are prepared by modifying the Bluetooth OBEX operation field data to a tag that indicates if the fuzz data is good or not. Thus, after training the Seq2Seq model, we could input a new fuzz data to the model, and then give a machine learning perspective result, which is its tag.

As Table 4, we trained the Seq2Seq model. As a result, we could input a new fuzz data to the model, which is not training data, and then the model outputs a machine learning perspective result, which is its tag, such that the new fuzz data can be translated to a tag "e0" or "f0."

4.3 Evaluation and Discussion

To evaluate the Seq2Seq efficient machine learning model, we focus on the volume of operations and *epochs* (learning data) with learning loss that we change the condition in the cases of Operation Fields and Total Epochs by operations. We count one Epoch, is as a training data set of each Operation Field, vary in fuzz data: for example, we count one for complete one Connect operation, then count two for the next repetition or another operation

Table 4 shows the Training Patterns of Case Study. The first and the second rows are compared by the volume of Operation Fields with one-hundred learning data to the Connect operation only and 20 learning data each to five Operations Fields, including Connect and Disconnect/Put/Get/Abort, in which total of one-hundred learning data. And from the second row and blow, the Total Epochs are added

Table 3 An Example of Preparing Refined Training Data of OBEX Operations.

Operation	Processing Type	Tactic	Fuzz data	Tag	Description
Connect	Data structure change	Binary	0x8A	e0	Change the data structure of Connect's HeaderWho Header Encoding/Header ID with a binary data
Disconnect	Change specific value	Maximum	65535	f0	Change the Disconnect's Maximum Packet Length to the maximum value data
Put	Change specific value	Null	0x00	f0	Change the Put's Response Code to NULL
Get	Change specific value	Outside	254	f0	Change the Get's Maximum Packet Length a value outside the boundary
Abort	Change specific value	Indeterminate	0xA1	f0	Change the Abort's Response Code to an indeterminate value

Table 4 Training Patterns of Case Study.

Color	Total Epochs	Operation Fields	Epochs/Operation	Description
Black	100	Connect	100	Total of 100 Epochs to Connect Operation only.
Green	100	Connect	20	Total of 100 Epochs, 20 each to 5 Operations.
		Disconnect	20	
		Put	20	
		Get	20	
		Abort	20	
Navy	200	Connect	40	Total of 200 Epochs, 40 each to 5 Operations.
		Disconnect	40	
		Put	40	
		Get	40	
		Abort	40	
Red	300	Connect	60	Total of 300 Epochs, 60 each to 5 Operations.
		Disconnect	60	
		Put	60	
		Get	60	
		Abort	60	
Blue	400	Connect	80	Total of 400 Epochs, 80 each to 5 Operations.
		Disconnect	80	
		Put	80	
		Get	80	
		Abort	80	
Orange	500	Connect	100	Total of 500 Epochs, 100 each to 5 Operations.
		Disconnect	100	
		Put	100	
		Get	100	
		Abort	100	

by each one hundred of total learning data, which is evenly distributed to the five operations.

The result of learning efficiency is Figure 1, The Loss Curves by Epochs. The vertical y-axis shows Learning Loss and the horizontal x-axis shows the volume of Total Epochs. For the x-axis, it scales only one hundred maximum unless more Total Epochs in Table 4, since our concern about time, although it was not much difference around hundreds. The black, green, navy, red, blue, orange curve illustrates different practical training patterns, respectively. The pink additional dashed line is at y=0.0001, where we put a rough indication for a value of learning convergence, and it should be required to analyze an appropriate value later. A graph shifting left means less the training volume, and shifting below means less the training loss. As we see in Figure 1, by increasing the number of Operation Fields, it is quickly converging the Learning Loss, such that we are able to reduce the volume of preparing training data and to improve efficiency.

5. Conclusion and Future work

Efficient training is essential in machine learning-based protocol fuzzing. This study investigated the refinement of the training data and conducted a case study by focusing on the Bluetooth OBEX protocol. The results show that variations on the operation field items are effective in increasing the learning efficiency.

There are several directions to extend the research of this paper. It is needed to further refine training data and empirical tag so that the learning model shows better capabilities on finding good fuzz. It is also possible to apply our approach to other profiles of Bluetooth and to other protocols such as Wi-Fi, Ethernet, and so on. From a wider perspective, we need to evaluate how our approach contributes to the whole framework of protocol fuzzing.

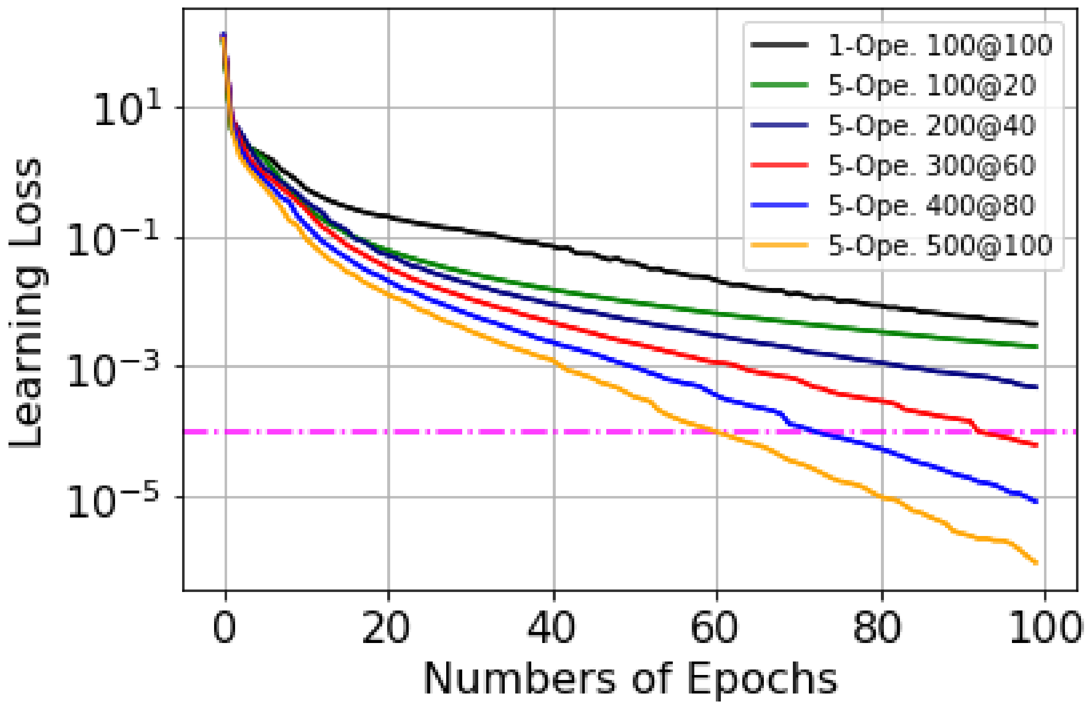

Fig. 1 The Loss Curves by Epochs

References

[1] B. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.
[2] V. Manès, H. Han, C. Han, S. Cha, M. Egele, E. Schwartz, and M. Woo, "The art, science, and engineering of fuzz testing: A survey," *IEEE Transactions on Software Engineering*, 2019.
[3] H. Wen, Z. Lin, and Y. Zhang, "FirmXRay: Detecting bluetooth link layer vulnerabilities from bare-metal firmware," Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 167–180, 2020.
[4] G. Saavedra, K. Rodhouse, D. Dunlavy, and P. Kegelmeyer, "A Review of Machine Learning Applications in fuzz testing," *CoRR*, vol. abs/1906.11133, 2019.
[5] S. Gorbunov and A. Rosenbloom, "Autofuzz: Automated network protocol fuzz testing framework," *IJCSNS International Journal of Computer Science and Network Security*, vol. 10, no. 8, pp. 239–245, 2010.
[6] Y. Wang, P. Jia, L. Liu, C. Huang, and Z. Liu, "A systematic review of fuzz testing based on machine learning techniques," Wang, Yan and Jia, Peng and Liu, Luping and Huang, Cheng and Liu, Zhonglin, *PloS one*, vol. 15, no. 8, e0237749, Public Library of Science San Francisco, CA USA, 2020.
[7] lcamtuf, "afl-fuzz: crash exploration mode," https://lcamtuf.blogspot.com/2014/11/afl-fuzz-crash-exploration-mode.html, accessed at Dec. 14, 2021.
[8] R. Fan and Y. Chang, "Machine learning for black-box fuzz testing of network protocols," International Conference on Information and Communications Security, pp. 621–632, Springer, 2017.
[9] H. Zhao, Z. Li, H. Wei, J. Shi, and Y. Huang, "SeqFuzzer: An industrial protocol fuzz testing framework from a deep learning perspective," 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pp. 59–67, IEEE, 2019.
[10] A. Butterfield, G. Ngondi, and A. Kerr, *A dictionary of computer science(7 ed.)*, Oxford University Press, 2016.
[11] R. Natella and V. Pham, "ProFuzzBench: A Benchmark for Stateful Protocol fuzz testing," *arXiv preprint arXiv:2101.05102*, 2021.
[12] owasp.org, "fuzz testing," https://owasp.org/www-community/fuzz testing, accessed at Dec. 14, 2021.
[13] pytorch.org, "NLP FROM SCRATCH: TRANSLATION WITH A SEQUENCE TO SEQUENCE NETWORK AND ATTENTION," https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html, accessed at Dec. 14, 2021.
[14] pytorch.org, FROM RESEARCH TO PRODUCTION: An open source machine learning framework that accelerates the path from research prototyping to production deployment, https://pytorch.org/, accessed at Dec. 14, 2021.
[15] I. Sutskever, O. Vinyals, and Q. Le, "Sequence to sequence learning with neural networks," *Advances in Neural Information Processing Systems*, Curran Associates, Inc., pp. 3104–3112, 2014.
[16] P. Tsankov, M. Dashti, and D. Basin, "SECFUZZ: Fuzz-testing security protocols," 2012 7th International Workshop on Automation of Software Test (AST), pp. 1–7, IEEE, 2012.
[17] ieee802.org, "IEEE 802.15 WPAN Task Group 1 (TG1)," https://www.ieee802.org/15/pub/TG1.html, accessed at Dec. 14, 2021.
[18] T. Panse and P. Panse, "A survey on security threats and vulnerability attacks on bluetooth communication," *International Journal of Computer Science and Information Technologies*, vol. 4, no. 5, pp. 741–746, 2013.
[19] K. Crawley, "Bluetooth security risks explained," AT&T Cybersecurity, https://cybersecurity.att.com/blogs/security-essentials/bluetooth-security-risks-explained, accessed at Dec. 14, 2021.
[20] Blutooth SIG, Inc., "Specifications List," https://www.bluetooth.com/specifications/specs/, accessed at Dec. 14, 2021.