Fuzzing Test Case Generation Based on Recurrent Neural Networks (Postprint)

Authors: Xu Peng, Liu Jiayong, Lin Bo, Sun Huiying, Lei Bin

Date: 2018-06-19T00:00:00+00:00

Abstract

The quality of fuzzing test cases constitutes a critical factor affecting testing effectiveness. Current conventional generation methods primarily rely on random mutation and manual protocol analysis-based construction, which respectively suffer from blind mutation with low efficiency and complex construction with high cost. To address these issues, we propose utilizing deep learning techniques to assist test case generation. By exploiting the advantages of recurrent neural networks in processing character text sequences, structural features are learned from sample data through training, and new data conforming to these structural features is predicted and generated. Combined with random mutation algorithms, this approach constructs an automatic generation model. Through the generation and effectiveness evaluation of PDF file input-type test cases using LSTM and GRU algorithm models, the generated test cases overall outperform conventional methods with superior validity rates and coverage. This method achieves the advantages of rapid, efficient generation and low construction difficulty through the assistance of recurrent neural networks, striking a balance between generation effectiveness and cost.

Full Text

Preamble

Title: Generation of Fuzzing Test Cases Based on Recurrent Neural Networks

Authors: Xu Peng[1], Liu Jiayong[1], Lin Bo[1], Sun Huiying[2], Lei Bin[3]

Affiliations: 1. College of Electronics & Information Engineering, Sichuan University, Chengdu 610065, China 2. Dept. of Data & Information, Sichuan Provincial Military District, Chengdu 610041, China 3. Troop 78100, Chengdu 610021, China

Abstract: The quality of fuzzing test cases is a critical factor affecting testing effectiveness. Current conventional generation methods primarily rely on random mutation and manual protocol analysis, which suffer from blind variation with low efficiency and complex construction with high cost, respectively. To address these issues, this paper proposes leveraging deep learning technology to assist test case generation. By harnessing the strength of recurrent neural networks in processing character text sequences, the model learns structural features from sample data and predicts new data that conforms to these features. Combined with random mutation algorithms, this approach constructs an automatic generation model. Through experiments on PDF file input test case generation and evaluation using LSTM and GRU algorithm models, the generated test cases demonstrate superior performance compared to conventional methods, achieving better pass rates and coverage. This method leverages recurrent neural networks to realize the advantages of fast, efficient generation and low construction difficulty, striking a balance between generation effectiveness and cost.

Keywords: deep learning; recurrent neural networks; fuzzing; LSTM; GRU

0 Introduction

Over more than two decades of evolution, fuzzing technology and methodologies have continuously improved, with automation levels and applicable scenarios progressively advancing and expanding. Researchers have developed increasingly intelligent and automated tools such as PROTOS, SPIKE, FileFuzz, COMRaider, Sulley, and Peach3, tailored to different application scenarios and workflows. These tools have significantly enhanced vulnerability discovery capabilities and efficiency. However, constrained by the vast diversity of information systems and the complexity of protocols and structures, test case generation technology—a crucial component—still exhibits notable deficiencies. First, random test case construction is fast but suffers from blind mutation, resulting in low pass rates and code coverage. It is heavily influenced by initial samples and yields low overall efficiency. Second, manual format or protocol analysis demands high comprehensive skills from personnel, involves cumbersome script writing, and exhibits complexity that increases with business logic, incurring high time and effort costs while being difficult to reuse, scale, or automate.

Benefiting from the rapid development and outstanding performance of deep learning technology, this paper proposes a test case mutation generation method based on recurrent neural networks (RNNs). By capitalizing on the fact that test cases consist of serialized data and the inherent architectural advantages of RNNs for modeling sequential data, we integrate RNNs to learn from test case sequence data, automatically identify data features and structures, and predict newly generated data. This enables the automatic generation of test cases with excellent pass rates and coverage. This approach combines the strengths of both

methods: it largely adheres to the basic specifications of the target system while simultaneously producing data that violates specifications to trigger anomalies.

1 Fuzzing Test and Test Case Generation Overview

1.1 Fuzzing Principles and Process

Fuzzing is a method for discovering software vulnerabilities by providing unexpected inputs to a target system and monitoring for anomalous results. It automatically uses large volumes of semi-valid data as program input and identifies potential security vulnerabilities based on whether the program exhibits abnormal behavior. Although information systems vary widely in type, scale, and application environment, the overall testing phase structure is similar, as illustrated in Figure 1 [Figure 1: see original paper].

As shown in Figure 1, the critical phase in the fuzzing process for vulnerability discovery is constructing test cases that violate the target system' s normal processing, thereby triggering anomalies and revealing system vulnerabilities. Consequently, research on test case generation methods constitutes an important aspect of improving fuzzing effectiveness.

1.2 Test Case Generation Methods: Characteristics and Overview

Current test case generation methods essentially fall into two categories: mutation-based data generation and generation-based data generation.

a) Mutation-based data generation involves starting from a normal sample and modifying certain fields according to specific rules and strategies to produce new malformed test cases. This approach does not require understanding the structure and format of the current sample file, thus offering broad applicability. However, it is highly dependent on initial samples, with different samples yielding varying code coverage, testing depth, and effectiveness, resulting in low overall efficiency.

b) Generation-based data generation requires a thorough understanding of the target software' s expected input format. Based on in-depth research into the target software' s file format or protocol specifications, it generates test data that violates certain rules. This method achieves higher legitimacy rates and code coverage, improving pass rates during validation and covering potentially high-risk code segments to discover hidden security vulnerabilities. However, it demands substantial time and effort to understand file formats or protocol specifications and to write corresponding rules. Different target software exhibits significant variation, making this approach difficult to reuse and limited in scope.

Test case generation methods are increasingly integrated with new technologies and approaches, such as genetic algorithms to enhance effectiveness. Given deep learning's excellent performance in image recognition and language processing, it naturally follows to combine it with test case generation principles and methods.

2 Generation Model Based on Recurrent Neural Networks

2.1 Overview of Recurrent Neural Networks

Recurrent neural networks (RNNs) are neural networks designed for processing sequential data. Traditional neural network models progress from input layers to hidden layers to output layers, with full connections between layers but no connections between nodes within the same layer. Such conventional networks struggle with context-dependent problems. For instance, predicting the next word in a sentence requires knowledge of preceding words, as words in a sentence are not independent. RNNs can handle sequentially changing data and have achieved tremendous success and widespread application in natural language processing (NLP).

The most distinctive feature of RNNs is the weighted connections between hidden layers, establishing relationships between current and previous or subsequent inputs. Figure 2(a) [Figure 2: see original paper] illustrates a typical RNN connection pattern, showing intra-layer node connections. RNNs contain input units marked as {x , x , ···, x , x , ···}, output units marked as {y , y , ···, y , y , ···}, and hidden units marked as {s , s , ···, s , s , ···}, with the hidden units performing the primary computational work. Figure 2(b) [Figure 2: see original paper] shows the RNN unfolded into a full neural network for easier comprehension.

Fuzzing targets receive input data composed of various sequential data types with contextual relationships, such as structured documents and various formatted data arrangements. Systems process these according to predefined parsing workflows, which motivates the use of RNNs for such problems.

2.2 Generation Model Principles and Description

Based on the previous comparison of test case generation methods, generation-based approaches offer superior effectiveness but incur high time and resource costs, while mutation-based methods are fast but less effective. This paper proposes a compromise: a generation model based on RNNs. The fundamental principle combines deep learning' s advantages to construct feature extraction algorithms and models that enable machines to learn from valid text data, automatically adjust and optimize network parameters to form an optimal processing model, and thereby automatically generate new data conforming to data structures and specifications. Finally, mutation rules and strategies are incorporated to form a test case set. This approach aligns well with fuzzing' s original intent: it largely follows the target system' s basic specifications while producing specification-violating data to trigger anomalies.

This paper selects character text-type input data as the research object for several reasons. First, considering model implementation complexity, common visible characters in computing number approximately 98, making network construction and training relatively manageable. Second, character-type data re-

quires fewer mappings, demanding relatively less computational and storage resources, making it suitable for conventional computing capabilities with moderate resource and time requirements. Third, with the rapid development of web applications, character text types like XML and JSON are widely used, offering high cost-effectiveness in practical applications. Fourth, while the model currently handles characters, words, or easily vectorized data, advances in deep learning algorithms and dimensionality reduction methods will enable processing of more data types.

The generation model architecture consists of three layers, as shown in Figure 3 [Figure 3: see original paper]:

2.2.1 Input Processing Layer This layer preprocesses training data through cleaning and conversion, transforming it into suitable data types and structures for network model processing. It creates the RNN input space and provides a data input source for training, completing the following steps:

a) Generate mapping vector tables. To convert character data sets into vector data suitable for neural network computation, the training dataset is first traversed to generate a unique character set. An appropriate algorithm then maps each character in the set one-to-one to a data vector, creating character-to-vector and vector-to-character conversion mapping tables. Common methods include one-hot encoding and sorted generation. Drawing from Word2Vec principles, more complex data sets require additional vector mapping methods for correlation computation, enabling mapping processing for non-character or character-combination data types to accommodate more training data varieties.

b) Vectorize training datasets. After generating the mapping vector table, the input training dataset undergoes vectorized mapping conversion, transforming the character set into a digital vector set.

c) Mini-batch segmentation processing. Due to the scale of training datasets and computational capacity limitations, full batch learning is impractical. Appropriate batch-size and sequence_length parameters are selected based on computational resources and network architecture to segment the data for batch training. Defining N sequences per batch (batch_size) and M as single sequence length (sequence_length), each batch forms an N×M array.

d) Create data input space. According to the mini-batch segmentation size, input and target tensors with Shape=N×M are established, with keep_prob space defined to control dropout retention nodes.

2.2.2 Recurrent Neural Network Layer This core layer receives data from the input layer and predicts new data, serving as the interface between layers. Data processing occurs in two phases: network training and network prediction. In the training phase, the RNN is constructed according to selected network architecture and training parameters. Input layer data is fed into the network for

iterative training and optimization to produce network parameters while evaluating training effectiveness. In the prediction phase, initial data and trained network parameters are loaded, and the RNN predicts new probability matrix outputs sent to the generation/mutation layer.

The most critical factor affecting this layer' s performance is neural network model selection and construction. Since the introduction of RNNs, researchers have continuously optimized and proposed various complex variants. Here we focus on the most widely used and successful models: LSTM and GRU.

a) LSTM (Long Short-Term Memory) is a special RNN first proposed by Hochreiter and Schmidhuber in 1997 to address gradient vanishing and explosion problems in long sequence training. Compared to standard RNNs, LSTM performs better on longer sequences. As shown in Figure 4(a) [Figure 4: see original paper], gate states control information transmission, remembering what needs long-term retention and forgetting unimportant information—unlike standard RNNs that only have a single memory superposition method. This makes LSTM particularly effective for tasks requiring "long-term memory." However, this introduces more parameters and increases training difficulty.

b) GRU (Gated Recurrent Unit) also addresses long-term memory and gradient problems in backpropagation. GRU's input-output structure resembles standard RNNs with internal concepts similar to LSTM. Compared to LSTM, GRU has one fewer "gate," as shown in Figure 4(b) [Figure 4: see original paper], resulting in fewer parameters while achieving comparable performance.

2.2.3 Generation and Mutation Layer This layer first generates new data from the RNN-predicted sequence matrix, then applies mutation strategies and algorithms, and finally assembles test cases according to the target' s data processing rules.

a) Data generation process. After loading network parameters and initial data, the network continuously predicts new sequence vector probability matrices based on learned parameters and weights. Using appropriate strategies, these probability matrices are mapped to new target data, achieving new data generation. Adjusting softmax sampling temperature and introducing random selection are common methods affecting new data generation. Temperature changes impact generation confidence—lower temperature yields higher confidence and more conservative data, while higher temperature produces more diverse, more malformed data with greater potential to trigger anomalies. Similarly, introducing random selection beyond the highest probability prediction increases data diversity. These optimization methods ultimately enable the network to generate more diverse yet high-pass-rate test data.

b) Data mutation process. While RNN-predicted data possesses structural similarity and some variability, mapping matrix set limitations prevent generating mutations beyond the mapping collection. Therefore, introducing random

or targeted mutation algorithms increases variation, expanding the test case mutation space to trigger more anomalies and cover broader execution paths.

c) Assembly generation process. Considering feasibility, computational resources, and time costs, RNN-learned and generated data are often data fragments or unsegmented blocks that cannot be directly executed. They must be segmented and assembled according to the test target' s data processing rules and strategies to produce executable test cases, ultimately forming a test case set.

3 Experiments and Results Analysis

To validate the effectiveness of the RNN generation model in test case generation, this paper selected CAJViewer7.2 reader and PDF files for experimental evaluation. The model learns character text data from PDF files and generates novel data distinct from training samples to verify its capability to produce new data, forming the basis for test case generation. This section first outlines the basic situation of CAJViewer7.2 and PDF file formats, then proposes an experimental scheme combining PDF format characteristics and model features, briefly describes implementation, and finally analyzes results using pass rate and coverage—two critical metrics.

3.1 Experimental Objects and Design Analysis

3.1.1 Experimental Objects Overview a) CAJViewer is a dedicated full-text format reader for China Academic Journals Network, supporting TEB, CAJ, NH, KDH, and PDF formats across Windows, MAC, iPad, iPhone, and Android platforms. The Windows version CAJViewer7.2 was selected for these experiments.

b) PDF (Portable Document Format) is a file format developed by Adobe Systems for application, operating system, and hardware-independent document exchange. The PDF file format is a key experimental object. As shown in Figure 5(a) [Figure 5: see original paper], a PDF file consists of four main parts: header, body, cross-reference table, and trailer.

PDF file structure components: - **Header:** The file beginning in %PDF-1.7 format, where the last digit indicates the PDF version specification (e.g., "1.7" ; the latest version is 1.7). - **Body:** Composed of multiple obj objects. As shown in Figure 5(b) [Figure 5: see original paper], the first number is the object ID (unique identifier), and the second is the generation number (indicating modification count, with 0 for newly created objects). Objects start with the "obj" keyword, contain content between « and », and end with "endobj." Objects with binary data embed "stream" and "endstream" keywords. - **Cross-reference table:** Indexes obj object locations for random access, as shown in Figure 5(c) [Figure 5: see original paper]. The "xref" marker indicates the table start. The first line "0 8" shows objects start from ID 0 with 8 total objects. "0000000000

65535 f" typically begins the table, with object 0 at address 0000000000, generation number 65535 (maximum, unchangeable), and "f" indicating a free object (file header). "0000000009 00000 n"represents object 1 with generation number 0 (unmodified) and "n" indicating in-use. - **Trailer:** The logical file content start, located at the file end. As shown in Figure 5(d) [Figure 5: see original paper], "/Size 8" indicates total object count; "/Root 1 0 R" specifies the root object ID; "Startxref 553" gives the cross-reference table offset to locate all objects; "%%EOF" marks the file end.

3.1.2 Analysis and Implementation Approach a) Understanding key document elements and relationships. The xref table marks object addresses, functioning as an object "directory." The trailer is the actual parsing header, passing control to obj objects via the Root pointer to the Catalog object. The entire document structure forms a tree marked by ID pointers. As shown in Figure 6 [Figure 6: see original paper], the trailer' s root points to the Catalog ID, which branches to Pages and Outline, then further to Page and Resources, extending sequentially by ID. Thus, CAJViewer' s core processing logic involves continuously discovering and parsing obj objects by following pointers. Therefore, obj object analysis is the focus.

b) Analyzing obj object characteristics. As analyzed, PDF document structures primarily consist of obj objects. Adobe' s PDF 1.7 specification document (published November 2006) contains approximately 1,200 pages describing this portion. Manual analysis for fuzzing template generation faces two major challenges: enormous workload due to numerous obj types and rules, and high requirements for template developers' computer knowledge. However, obj analysis reveals that except for stream sections with binary encoding, most content uses visible character encoding, making it suitable for generation models.

c) Design considerations and implementation. Stream sections are complex, mostly handled by external libraries, but limited by computational capacity and time costs. Therefore, we decompose PDF documents into obj object blocks as processing units, designing experimental steps: PDF sample collection, obj data extraction, RNN training on obj objects, RNN prediction of new obj objects, obj mutation, obj assembly into test case sets, and experimental evaluation.

3.2 Experimental Process

Following the above approach, experiments were conducted in four parts: PDF sample collection and preprocessing, model training and analysis, test case set generation, and test execution results evaluation.

3.2.1 PDF Sample Collection and Preprocessing a) Crawling PDF sample files. The Internet hosts numerous PDF files, providing an excellent sample source. To collect diverse PDF samples, we used the Baidu search engine with "topic keyword + filetype:pdf"queries. To bypass anti-crawling mechanisms,

we developed a browser-based crawler using Python and Selenium2. Based on keyword rankings and categories, we selected 300 topic keywords and ran the crawler for one day, collecting 12,006 PDF files.

b) PDF sample set organization. Downloaded PDFs varied widely in type and contained duplicates. We performed version identification, classification, and deduplication, obtaining 11,096 valid PDF files covering versions 1.0–1.7, with sizes ranging from 8 KB to 43 MB. Using CAJViewer as the target system, we wrote execution verification scripts and used PeachMinset to minimize the dataset, yielding a refined sample set of 1,149 PDF files.

c) Training dataset preprocessing. To simplify experiments and adapt to model requirements, we filtered out binary "stream" data and extracted character-type obj objects from the refined sample set—content from "obj" to "endobj" as shown in Figure 5(b) [Figure 5: see original paper]. PDF samples had multiple encoding and character set issues; processing them individually was time-consuming, so we extracted only displayable ASCII characters. After calculating lengths and MD5 hashes and deduplicating, we extracted 316,991 obj objects ranging from 11 to 435,016 bytes. The length distribution is shown in Figure 7(a) [Figure 7: see original paper]. After analysis, we sampled 78,406 data items based on length distribution, with the final training dataset length distribution shown in Figure 7(b) [Figure 7: see original paper], forming a 14.6 MB training dataset.

3.2.2 Model Training and Analysis a) Deep learning platform selection. TensorFlow is Google' s second-generation AI learning system open-sourced in 2015, based on DistBelief, for transmitting complex data structures through AI neural networks for analysis and processing. Compared to Caffe, Theano, Torch, and MXNet, TensorFlow has the most GitHub forks and stars, with rich applications in image classification, audio processing, recommendation systems, and NLP. For these reasons, we selected TensorFlow 1.3 for building the neural network layer.

b) Training parameters and hardware configuration. To compare training effectiveness, we selected LSTM and GRU models with batch sizes of 20 and 50, forming four parameter groups (LSTM-20, LSTM-50, GRU-20, GRU-50). Other parameters were uniformly set: sequence length 10, 2 layers, 128 hidden nodes, 0.002 learning rate, and 100 training epochs. Hardware: two Intel(R) Xeon(R) CPU E5-26xx v2 servers with 2 GB RAM. Each 100-epoch training took approximately 30 hours. The character mapping table contained 98 characters; LSTM model parameters were 3,125,436 bytes, GRU parameters were 2,382,012 bytes. Training parameters and results are shown in Table 1 .

c) Training process analysis. Figure 8 [Figure 8: see original paper] shows the Perplexity comparison during training. Perplexity measures language model prediction quality—lower values indicate better predictive performance. LSTM-50 achieved the best validation Perplexity, while GRU-50 trained fastest. Dif-

ferent batch sizes significantly impacted training effectiveness, consistent with neural network layer theory.

3.2.3 Test Case Set Generation Test case generation involves three processes: obj object generation, data mutation, and data assembly.

a) Obj object generation. To compare and analyze, we loaded the four best-trained model parameters into RNN networks, generating 10,000 obj objects per network (40,000 total), with lengths from 17 to 38,630 bytes. As shown in Figure 9 [Figure 9: see original paper], the network-generated data demonstrates excellent quality, correctly producing obj object headers, footers, common keywords, and formats similar to the training set, differing only in data values. By importing generated obj objects into a database and comparing MD5 hashes with the total obj dataset, no identical data was found, confirming the network' s ability to generate novel, distinct data.

b) Obj object mutation. Based on obj structural characteristics and to achieve better pass rates, we mutated only data between "obj" and "endobj" using character set and binary mutations. Figure 10 [Figure 10: see original paper] shows a Python implementation of a threshold-controlled random mutation algorithm, with mut_frac and p parameters adjusting mutation quantity and method probabilities. Figure 11 [Figure 11: see original paper] shows mutation examples with mut_frac=0.05 and p=6.

c) Data assembly. Network-generated obj objects cannot be directly opened by CAJViewer. For evaluation, we created a 7-object-node PDF template based on PDF structure analysis, replacing Catalog, Outlines, Pages, Page, and Contents sections with mutated obj objects to test effects at different nodes. For comparison, we created three test datasets: - **PEACH dataset:** 5,000 test cases generated using Peach3' s random mutation on the PDF template - **NORMAL dataset:** 5,000 test cases assembled from 1,000 randomly extracted obj objects from the total dataset - **RNNS dataset:** 5,000 test cases assembled from 1,000 RNN-generated obj objects with random mutation

3.2.4 Results Analysis To validate model feasibility and test case effectiveness, we performed two analyses: generation model performance/feasibility and comparative test case effectiveness evaluation.

1) Generation model performance and feasibility analysis. The experiment used LSTM and GRU models, with optimal parameters generating data as shown in Figure 9. The network successfully produced data structurally consistent with PDF samples, correctly generating obj headers, footers, keywords, and formats. Testing 200,000 generated data items revealed no duplicates, with lengths similar to the training set, demonstrating the network' s ability to produce similar yet novel data with better anomaly-triggering potential than random mutation. Using conventional dual-core 26xx series CPUs with 2 GB RAM Linux servers, 100 training epochs required approximately 30 hours. Table 1 and

Figure 8 show Perplexity comparisons, confirming LSTM-50's optimal validation Perplexity and GRU-50' s fastest training speed, with batch size significantly impacting results.

2) Comparative test case effectiveness evaluation. We evaluated using code coverage and pass rate—the most representative fuzzing metrics.

Coverage was assessed using dynamic link library address coverage (equivalent to code coverage). Using trace files generated by peachminset during CAJViewer execution, we statistically analyzed covered libraries and addresses. More coverage indicates better test case effectiveness.

Pass rate was determined by monitoring CAJViewer' s file processing: executing CAJViewer with test cases and checking for errors or exception dialogs. Cases without anomalies were marked as passed.

Coverage and pass rate comparisons are shown in Table 2 . The PEACH dataset (random mutation) showed low coverage and pass rates due to initial sample dependency, resulting in low vulnerability discovery efficiency. The RNNS dataset achieved the highest coverage with moderate pass rates, balancing both metrics. RNN-generated data was rejected by CAJViewer only due to structural specification violations, but its inherent variability and high coverage provide more opportunities to trigger anomalies. The NORMAL dataset had high pass rates but medium coverage with insufficient variation, making anomaly generation less likely compared to RNNS.

These results demonstrate that the generation model intelligently learns rules for character text-type data, reasonably generating test cases that balance pass rate and coverage. The approach is suitable for file formats and communication protocols based on character text representation. However, limitations exist: effectiveness depends on sample data quality, offering less advantage for simple specifications or insufficient learnable data.

Beyond validating feasibility and effectiveness, this process inspires further deep learning and fuzzing integration research: (1) Expanding model applicability requires reducing vectorization difficulty, potentially using CNNs or feature extraction for dimensionality reduction; (2) Improving generation effectiveness could incorporate feedback mechanisms from advanced fuzzers like AFL and VUzzer, using neural networks to learn trace information and form feedback loops for deeper path exploration; (3) Researching appropriate deep learning networks to learn vulnerability features and anomaly-oriented distributions could make test case generation more targeted and anomaly-prone.

4 Conclusion

This paper described the fundamental principles and architecture of a test case generation model based on recurrent neural networks. Using CAJViewer' s PDF parsing as an experimental object, we conducted training data collection, processing, network training, data generation and assembly, and evaluation.

Results validated the model' s feasibility and practicality. Through the journey from fuzzing and deep learning theory to model design and practice, we deeply appreciated deep learning' s advantages in solving complex problems and understood how data-driven feature extraction and analysis can replace tedious manual work to improve efficiency. Fuzzing technology has numerous demands for feature discovery and application, creating natural synergy with deep learning. As AI technology advances and neural network models evolve, we anticipate more and better algorithms and models will emerge to enhance fuzzing capabilities.

References

[1] Zhang Xiong, Li Zhoujun. Overview of fuzzy testing technology [J]. Computer Science, 2016, 43 (5): 1-8, 26.

[2] Li Tong, Huang Xuan, Huang Rui. Test case generation method in fuzzy testing [J]. Computer System Application, 2015, 24 (4): 139-143.

[3] Li Xiaopeng. Research and application of text representation algorithm [D]. Beijing: Beijing University of Posts and Telecommunications, 2016.

[4] Lipton Z C. A critical review of recurrent neural networks for sequence learning [C]// Proc of Computer Science. 2015: 1-35.

[5] Hochreiter S, Schmidhuber J. Long short-term memory [J]. Neural Computation, 1997, 9 (8): 1735-1780.

[6] Michael Eddington. Peach [EB/OL]. http://peachfuzzer.com.

[7] Godefroid P, Levin M Y, Molnar D. Sage: whitebox fuzzing for security testing [J]. Queue, 2012, 10 (1): 20.

[8] Huang Lei, Du Changshun. Text classification based on recurrent neural network [J]. Journal of Beijing University of Chemical Technology: Natural Science Edition, 2017, 44 (1): 98-104.

[9] Li Xuelian, Duan Hong, Xu Mu. Chinese word segmentation method based on GRU neural network [J//OL]. Journal of Xiamen University: Natural Science Edition: 1-9. [2018-05-16].

[10] Graves A, Schmidhuber J. Framewise phoneme classification with bidirectional LSTM and other neural network architectures [J]. Neural Networks, 2005, 18 (5): 602-610.

[11] Chung J, Gulcehre C, Cho K, et al. Gated feedback recurrent neural networks [C]//Proc of International Conference on Machine Learning.

[12] Pennington J, Socher R, Manning C D. GloVe: global vectors for word representation [C]//Proc of Conference on Empirical Methods in Natural Language Processing. 2014: 1532-1543.

[13] Zhang Qian, Gao Zhangmin, Liu Jiayong. Research on micro-blog short text classification based on Word2vec [J]. Information Network Security, 2017 (1): 57-62.

[14] Qi Jian, Chen Xiaoming, You Weiqing. Research on network protocol security assessment method based on Fuzzing test [J]. Information Network Security, 2017 (3): 59-65.

[15] Chen Lei. Text representation model and feature selection algorithm [D]. Hefei: University of Science & Technology China, 2017.

[16] Zhou Lian. The working principle and application of Word2vec [J]. Science and Technology Information Development and Economy, 2015, 25 (2): 145-148.

[17] Guo Lili, Ding Shifei. Research progress in deep learning [J]. Computer Science, 2015, 42 (5): 28-33.

[18] Ouyang Yongji, Wei Qiang, Wang Jiajie, et al. Software safety testing based on vulnerability feature oriented [J]. Journal of Tsinghua University: Natural Science Edition, 2017, 57 (9): 903-908.

[19] Ouyang Yongji, Wei Qiang, Wang Qingxian, et al. An intelligent Fuzzing method based on anomalous distribution oriented [J]. Journal of Electronics and Information, 2015, 37 (1): 143-149.

[20] Zhang Yao, Zhang Chaorong, Lin Teng, et al. Design and implementation of binary code test coverage evaluation system [J]. Command Information System and Technology, 2015, 6 (6): 13-17.

[21] Wu Haibo. Research and application of detector generation algorithm based on neural network [J]. Information Network Security, 2015 (9): 249-252.

[22] Shi Ji, Zeng Zhaolong, Yang Congbao, et al. Test technology overview [J]. Information Network Security, 2014 (3): 87-91.

[23] Zhang Minmin, Xu Heping, Wang Xiaojie, et al. Google tensor flow machine learning framework and application [J]. Microcomputers and Applications, 2017, 36 (10): 58-60.

[24] Adobe Systems Incorporated. PDF reference, 6th edition [EB/OL] (2006). Available at http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/pdf_reference_1-7.pdf.

Note: Figure translations are in progress. See original paper for figures.