ER-Fuzz：Conditional Code Removed Fuzzing

Xiaobin Song, Zehui Wu, Yan Cao and Qiang Wei*
China National Digital Switching System Engineering and Technological Research Center
Zhengzhou, Henan 450002 – China
[e-mail:xdxiaobin@gmail.com]
[e-mail:prof_weiqiang@163.com]
*Corresponding author: Qiang Wei

Received September 2, 2018; revised December 2, 2018; accepted January 8, 2019; published July 31, 2019

Abstract

Coverage-guided fuzzing is an efficient solution that has been widely used in software testing. By guiding fuzzers through the coverage information, seeds that generate new paths will be retained to continually increase the coverage. However, we observed that most samples follow the same few high-frequency paths. The seeds that exercise a high-frequency path are saved for the subsequent mutation process until the user terminates the test process, which directly affects the efficiency with which the low-frequency paths are tested. In this paper, we propose a fuzzing solution, ER-Fuzz, that truncates the recording of a high-frequency path to influence coverage. It utilizes a deep learning-based classifier to locate the high and low-frequency path transfer points; then, it instruments at the transfer position to promote the probability low-frequency transfer paths while eliminating subsequent variations of the high-frequency path seeds. We implemented a prototype of ER-Fuzz based on the popular fuzzer AFL and evaluated it on several applications. The experimental results show that ER-Fuzz improves the coverage of the original AFL method to different degrees. In terms of the number of crash discoveries, in the best case, ER-Fuzz found 115% more unique crashes than did AFL. In total, seven new bugs were found and new CVEs were assigned.

Keywords: Fuzzing, Deep learning, instrumentation, conditional code

This work was supported by the Ministry of Science and Technology of China under Grant 2017YFB0802901.

http://doi.org/10.3837/tiis.2019.07.010 **ISSN : 1976-7277**

1. Introduction and Motivation

Fuzzing [1] is an automated software testing technique that inputs randomly generated information to a program and then monitors the program to catch exceptions during execution. Due to the simplicity and efficiency of fuzzing, it has been widely used for testing by software manufacturers and in open source software development and has led to the discovery of large numbers of vulnerabilities in various software programs. However, the wide application of software security testing tools and improvements in code security from developer awareness, vulnerabilities now usually appear in deep code structures. The existing fuzzers are effective in exploiting shallow vulnerabilities, but have difficulty catching exceptions when facing complex code. This problem occurs because most inputs execute along the same high-frequency paths, while exploring low-frequency paths is more difficult. Along this line, researchers have combined other relevant techniques with fuzzing, such as symbol execution [2,3], dynamic analysis [4,5] and others. Driller [6] combined symbol execution to achieve a balanced approach using fuzzing and selective concolic execution to find deep errors. Driller uses selective concolic [7] execution to test fuzzers considered as more "valuable" but that have blocked paths. By combining the advantages of lightweight fuzzing and concolic execution, it avoids the inherent defects of path explosion in symbolic execution and incomplete fuzzing. Sanjay et al. proposed an application-aware evolutionary fuzzing method, Vuzzer [8], that used lightweight static analysis and dynamic analysis of control flow, data flow and target attribute characteristics. The input is optimized by calculating the weight of the code block and result feedback; then, better input is generated to detect deep code. AFLFast [9] proposed a technology based on a Markov chain [10] model to identify low-frequency paths and optimize seed-sorting and selection strategies with code coverage [11] to improve the probability of low-frequency path testing.

Although the above methods adopted different technologies to improve the probability of low-frequency path tests, high-frequency path sample testing still occurs, which not only limits the probability of low-frequency path tests but also fails to improve the overall test efficiency substantially. Instead, in this paper, we propose a new idea: identifying low-frequency path transfer conditional code before execution and using a path record truncation strategy to cancel the high-frequency inputs, promoting the probability of deep code testing. The methods acquired during preprocessing in this procedure do not affect the test efficiency. Moreover, it avoids the expensive overhead incurred by symbolic execution or dynamic program analysis.

This paper also presents a low-frequency transfer point recognition method based on deep learning. This method is used to implement a prototype of ER-Fuzz, which is based on American Fuzzy Lop (AFL) [12]. The code classification model is generated by Word2vec [13] and LSTM [14]. Ten open source projects developed based on C/C++ were selected as the basic dataset used to train, verify and evaluate the classify model. In the experiments, the proposed method reached a recognition accuracy of 97%. Meanwhile, to verify the fuzzer's practicability, it was used to compare and test a dataset of popular applications, and multiple 0 day vulnerabilities were found in the experiment.

A low-frequency path transfer condition refers to conditional statements in the program that result in a sample being unable to explore the deep code. As shown in **Fig. 1**, at the #3 basic block, the left subtree is the low-frequency path, and the right subtree is the high-frequency path. Therefore, the #3 basic block is a low-frequency conditional-code path

transfer block. Common transfer conditions include file-format validation checks, magic byte [15] checks, and so on.

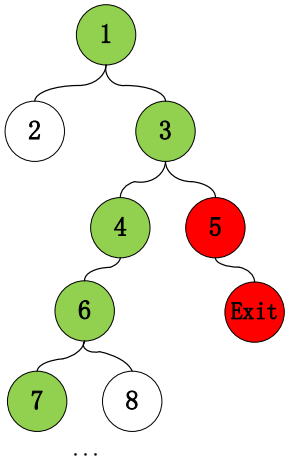

Fig. 1. High and low-frequency paths

In general, most input parsers include a large number of format checks and corresponding error-handling structures for check failures that often result in coverage that is difficult to increase. Therefore, error-handling code is the most representative type of low-frequency transfer point. Error-handling code refers to code segments that are executed when the program exits abnormally for various reasons. While most languages (e.g., C and C++) include their own error-handling mechanisms, they also support user-defined error-handling methods. Analysis reveals that the error-handling mechanism generally has the following three characteristics: 1) the error condition is written in *if - else* form; 2) the error-handling code snippets contain specific keywords, such as internal function names or output strings containing the word "*error*", or similar terms; 3) the program's exit status may include *return*, *goto* or contain error macro definitions. Although vulnerabilities may exist in error-handling code, analysis has found that the proportion of vulnerabilities contained in such code is low. The statistical data in reference [16] summarized the number of bugs in the error-handling code and the total number bugs of 6 open source projects, indicating that most of the bugs occur in normal code.

Table. 1. Number of error-handling bugs in different projects

Project	LOC	Studied period	Total commits	Total bug-fixes	Error-handling Bugs
OpenSSL	469,525	2016-01-01 2017-01-01	3925	924	126
GnuTLS	168,777	2016-01-01 2017-01-01	7035	760	29
WolfSSL	166,667	2016-01-01 2017-01-01	1240	297	31
Curl	153,732	2016-01-01 2017-01-01	11654	2853	190
Httpd	1,832,007	2016-01-01 2017-01-01	6781	1049	70
Linux	10,462,319	2016-01-01 2017-01-01	3234	1377	263
Total	13,253,027		33,869	7260	709

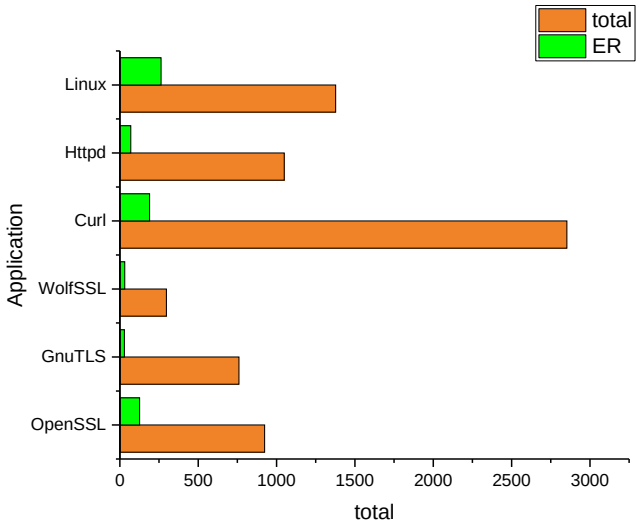

Fig. 2. Comparison of the number of bug fixes with the total number of bugs

Error-handling code typically causes a program to output information about the abnormal condition and then exit, as shown in **Fig. 3**. Such code is relatively small, has a simple structure, and most error handling does not involve complex memory operations, that is, situations that can be triggered when error handling is performed, such as missing return values, incorrect resource releases [17], and so on. The method in this paper can also catch exception occurrences and reduce false negatives.

Coverage-guided fuzzer utilize coverage as an important metric because high code coverage is more likely to trigger unknown vulnerabilities in the program. AFL is a widely used fuzzer that is based on coverage feedback. The coverage statistics are collected by instrumentation at program branches and the statistical results are stored in a shared memory location. After each sample is executed, its execution path is recorded. A genetic algorithm (GA) [18] is utilized to improve sample quality and increase the coverage rate. However, in some cases, this approach produces inefficient results. **Listing 1** shows a simplified version of Open vSwitch's [19] packet-parsing code function, which contains a check code fragment for two different fields. When AFL tests the following function, because its sample mutation is random, it is likely that at least one check in the following format is not met, but AFL adds instrumentation to all the branches. Because new blocks were found, the samples that triggered subsequent branches will be retained, and samples that triggered the error-handling branches will be mutated during the next round of testing even if subsequent samples are found that can meet all the checks.

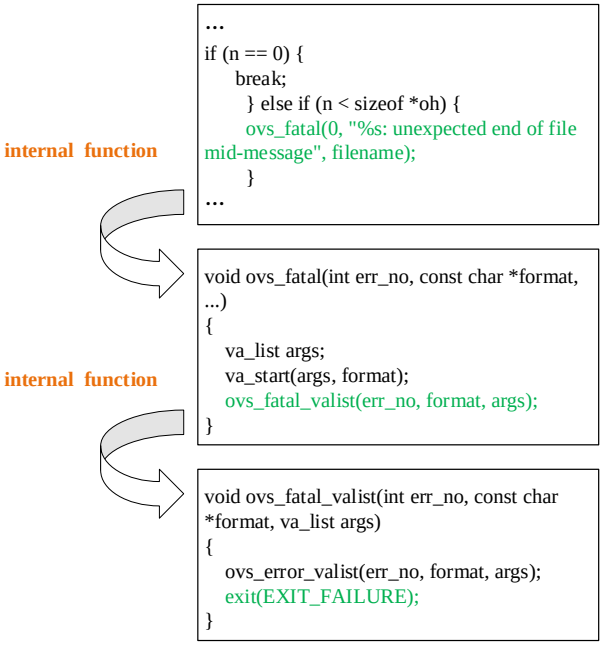

Fig. 3. Error-handling code execution flow

For the error-handling branches, normal samples cannot be triggered; that is, the optimal sample of the path is an exception sample. However, AFL does not have the ability to differentiate between normal and abnormal samples, it depends entirely on the path coverage information to determine whether the samples should be retained. Consequently, samples of this type will be retained and further mutated.

Listing. 1. Packet parse code in Open vSwitch

```
1  static void ofctl_ofp_parse(struct ovs_cmdl_context *ctx)
2  {
3      …
4      length = ntohs(oh->length);
5      if (length < sizeof *oh) {
6              ovs_fatal(0, "%s: %"PRIuSIZE"-byte message is too short for 7
   OpenFlow",filename, length);
7       }
8      tail_len = length - sizeof *oh;
9      tail = ofpbuf_put_uninit(&b, tail_len);
10      n = fread(tail, 1, tail_len, file);
11      if (n < tail_len) {
12      ovs_fatal(0, "%s: unexpected end of file mid-message", filename);
13      }
14      ofp_print(stdout, b.data, b.size, NULL, verbosity + 2);
15  }
```

In **Fig. 1**, the code in light green is a normal block, and 5 is an abnormal code block. Samples that are exercised in AFL as 1→3→4→6→7 and 1→3→5→*Exit* will be retained. However, vulnerabilities often exist in deep code. In this case, it is easier to find such vulnerabilities by increasing the number of samples similar to the execution path of the former. We set the original sample set as $S=\{S_1,S_2...,S_x\}$. After the samples have mutated, there are two situations. The first type of sample S_e executes the error-handling path after mutation. This type of sample is defined as a high-frequency sample, while the opposite is defined as a low-frequency sample. The second type of sample S_r performs the normal exit after mutation.

$$S_i \in S \rightarrow S_r \mid S_e \tag{1}$$

The mutated sample set is $S_{m1}=\{S_{state}\in\{S_r^1,S_e^1\}\}$. And $S_r^1=\{S_{r0}^1,S_{r1}^1...,S_{ri}^1\}$, $S_e^1=\{S_{e0}^1,S_{e1}^1...,S_{ev}^1\}$. The total time cost of the modified sample set is T_{total}, which includes the time cost of the high-frequency sample set T_e and the low-frequency sample set T_r.

$$T_{total}=T_r+T_e \tag{2}$$

The average time cost of a single sample is T_{avg}.

$$T_{avg}=p_r*t_r+p_e*t_e \tag{3}$$

where p_r and p_e are the probabilities that the samples will mutate to low-frequency or high-frequency samples, respectively, and

$$p_r+p_e=1 \tag{4}$$

In (3), t_r and t_e are the average time cost of the low-frequency and high-frequency samples, respectively. N_r^1 and N_e^1 are the number of low-frequency and high-frequency samples, respectively, in S_{m1}, and

$$t_r=\frac{\sum_{\varepsilon=0}^{i}c_{S_{r\varepsilon}^1}}{N_r^1} \quad t_e=\frac{\sum_{\varepsilon=0}^{v}c_{S_{e\varepsilon}^1}}{N_e^1} \tag{5}$$

where $c_{S_{r\varepsilon}^1}$ and $c_{S_{e\varepsilon}^1}$ are the time cost of the sample ε from different sample set of S_{m1}. Therefore, it can be concluded that

$$T_r=N_r^1*t_r \quad T_e=N_e^1*t_e. \tag{6}$$

Under the premise of reducing the number of high-frequency samples, it is assumed that the new mutant sample set is $S_{m2}=\{S_{state}\in\{S_r^2,S_e^2\}\}$. N_r^2 and N_e^2 are the number of low-frequency and high-frequency samples, respectively, in S_{m2}. Although $N_e^1>N_e^2$, the average time cost of the different samples remains unchanged.

As the number of high-frequency samples decreases, a decrease in p_e results in a decrease in T_{avg}. If T_{total} remains unchanged, then $N_r^2+N_e^2$ is greater than $N_r^1+N_e^1$. Because N_e^1 is greater than N_e^2 and N_r^2 is greater than N_r^1 due to the decrease in p_e, more low-frequency samples can be tested in the same amount of time, and the test probability of deep code can increase, making it easier to find vulnerabilities.

The main contributions of this paper are as follows: 1) We propose a low-frequency transfer point recognition method based on deep learning for the C/C++ languages and evaluate the proposed method. 2) We propose a fuzzing method under optimized instrumentation. 3) We implemented a prototype ER-Fuzz based on the above methods and conducted performance testing and comparison with state-of-art fuzzers on four applications. 4) The proposed method found several unreported vulnerabilities during the experiment.

2. Background

In this section, we first introduce previous research achievements in the field of fuzzing. Then, we provide background information concerning the two main tasks in ER-Fuzz, the original instrumentation method in the fuzzer and error code identification.

2.1 Related Work

A. Coverage-based fuzzing

Coverage is an important metric for assessing a fuzzer. The size of code coverage directly affects the probability of finding vulnerabilities. The fuzzer adjusts the selection strategy of seeds based on coverage. Microsoft's Patrice Godefroid et al. proposed a seed file generation method called learn&fuzz [20], which used a large number of PDF samples to train a sequence-to-sequence deep neural network model. The trained model could generate new PDF files and then test programs for reading PDF documents. Experiments showed that the generated PDF files achieved high code coverage. Peng Chen et al. proposed Angora [21], which does not rely on symbol execution technology to improve coverage. First, the input-related byte offset in a conditional branch is found through byte-level taint data tracking; then, input to trigger the new branch is calculated through the gradient descent algorithm commonly used in machine learning to infer the input bytes for variables and types. The method was tested on eight common open source projects and found multiple vulnerabilities. The advantage of ER-Fuzz is that it does requires neither complex program analysis, such as static analysis, dynamic taint analysis, etc., nor complex strategies, and it has no specific requirements for test targets, consequently, it has more extensive applicability.

B. Directed fuzzing

Directed fuzzing is not intended to cover paths as comprehensively as possible but to achieve coverage testing for a particular code target type (instruction, basic block, etc.). Fuzzing based on taint tracking can be used to determine which bytes of input can be mutated, and fuzzing based on symbols can be used to determine the accessibility of the target path. Therefore, these two research approaches have been used in many tools. M Mouzarani et al. [22] proposed a new intelligent fuzzing method to detect stack overflows in binary code. In the proposed method, concolic execution is used to calculate the path and vulnerability constraint of each execution path in the program. The vulnerability constraint determines the parts of the input data and to what lengths they should be extended to cause buffer overflows in the execution path. Based on the calculated constraint, test data that can cause a buffer overflow in the detection program path are generated. Marcel Bohme et al. implemented AFLgo [23], a guided greybox fuzzer that modified the seed energy allocation strategy of AFL. After identifying a sensitive point in the program, AFLgo always selected seed files close to the target point for testing and completed the distance calculation according to the simulated annealing algorithm on the basis of the call graph (CG) and control flow graph (CFG). AFLgo can be used for patch testing, crash reconstruction, static report verification and other scenarios.

2.2 Standard Instrumentation

AFL is able to determine whether the sample covers a new basic block by adding instrumentation to obtain information about program execution flow. Although full visibility of the basic block coverage information can be achieved, test efficiency is reduced by excessive instrumentation. On one hand, executing large amounts of instrumentation code requires a certain time overhead. On the other hand, because vulnerabilities are usually hidden in deep code, AFL has difficulty finding because it cannot judge the importance of branches. For example, some branch transfers belong to high-frequency paths, leading to wasted resources for samples performing these high-frequency paths, which reduces test efficiency. Although AFL can selectively instrument parts of blocks, doing so may omit important coverage information. Therefore, it is a challenge to minimize the coverage impact. At present, most existing technologies adopt heuristic methods, but they are not fully applicable to fuzzing. Hsu et al. proposed INSTRIM [24], a lightweight instrumentation method suitable for fuzzing, which described the problem as a path differentiation problem on the control flow graph and proposed two algorithms to solve the accuracy and path differentiation problems, respectively. Although this method reduces the cost of instrumentation and improves test efficiency, it still ignores some relatively important coverage information, such as the number of branch executions. Thus, it can fail to discover some overflow vulnerabilities and has difficulty making accurate evaluations.

2.3 Error-Handling Code Identification

Jana et al. proposed a method to identify error-handling codes based on error paths and three heuristics. Liu et al. [25] proposed a method combined with machine learning to identify error-handling code segments in large-scale software [26]. By analyzing and summarizing seven features of error-handling code segments, a decision tree model was used for classification. However, the above methods all rely on human experience, and to a certain extent, they lack accuracy because they ignore the contextual syntactic relations in the code segments, resulting in high false alarm rates. Therefore, their universality needs to be further improved. ER-Fuzz proposes using deep learning for classification to eliminate the limitations of complex operations and heuristic methods that require extracting features manually. Word2vec and LSTM have been widely used and have achieved good results in text classification.

Word2vec is a two-layer neural network that is very efficient at processing text. It takes text as input and outputs a feature vector of the words in the input text, which is useful in making computers understand natural language. However, Word2vec is not just useful for parsing natural-language statements, it is also useful for performing pattern recognition in code. The output of Word2vec is a vocabulary that contains the vectors for all the words within the limited frequency of the text corpus. These vectors can be fed into a deep learning network.

The Long Short Term Memory network (LSTM), is a recurrent neural network (RNN) branch that solves the bottleneck of RNN in dealing with long-term dependence. The LSTM architectural unit includes memory units C_t with three gates, namely, an input gate i_t, an output gate o_t and forgetting gate f_t. The state of the LSTM unit depends not only on the current state of x_t but also on the previous state. The LSTM cell calculation steps are as follows:

$$i_t = \sigma(Wi \bullet [h_{t-1}, x_t]) + b_i) \tag{7}$$

$$f_t = \sigma(W_f \bullet [h_{t-1}, x_t] + b_f) \quad (8)$$

$$o_t = \sigma(W_o \bullet [h_{t-1}, x_t] + b_o) \quad (9)$$

$$\widetilde{C}_t = \tanh(W_c \bullet [h_{t-1}, x_t] + b_c) \quad (10)$$

$$C_t = f_t \bullet C_t - 1 + i_t \bullet \widetilde{C}_t \quad (11)$$

$$h_t = o_t \bullet \tanh(C_t) \quad (12)$$

The three gates determine when new input is allowed, when the current cell state is cleared, and when the cell state affects the current network input. LSTM implements long-term memory by storing and modifying state information.

3. Design and Implementation

This section introduces the specific implementation of ER-Fuzz, mainly from the following two aspects. First, the entire system is introduced, including its components and functions. Second, the design and implementation of the two modules are introduced and analyzed in detail.

3.1 Overview

The system is mainly composed of two parts. The first part addresses error-handling code fragment recognition in the test program. This part extracts all conditional construction code fragments in the program; then, it applies the classification model obtained by Word2vec and LSTM to predict whether each fragment constitutes is an error-handling code segment. The second part addresses path record truncation. All the error-handling code segments obtained through the classification are input to the lightweight instrumentation module, which inserts instrumentation at the corresponding positions in the source file based on defined instrumentation rules.

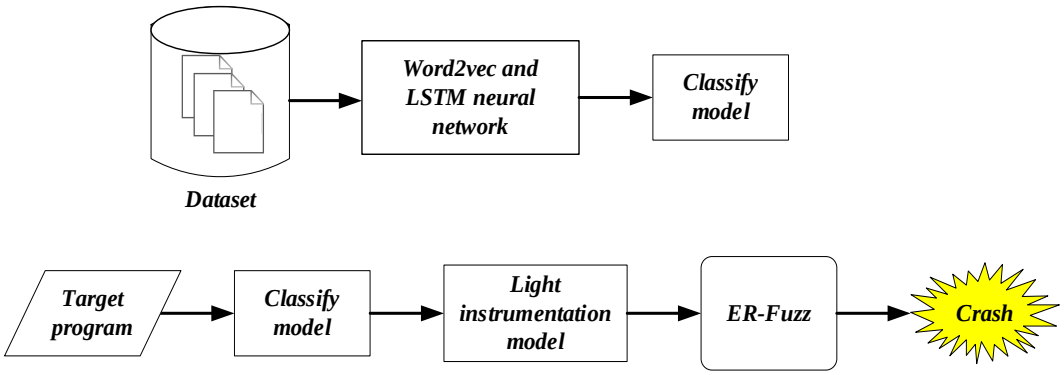

Fig. 4. ER-Fuzz workflow

3.2 Error-handling code identification

There are two phases in identifying error-handling code segments: a training phase and a detection phase. In the training phase, a large number of source code files from open source

projects are selected and preprocessed to obtain code fragments, which are vectorized and used as the input to the neural network model. Finally, a trained classification model is obtained.

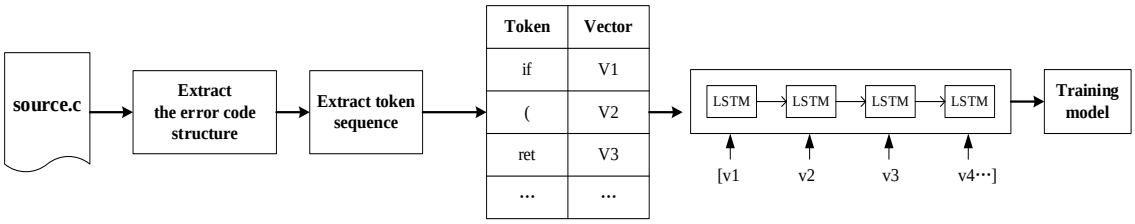

Fig. 5. Training stage

3.2.1 Construction of the LSTM error-handling code classifier

Training models usually include several steps due to the particularities of error-handling code, such as differences in their structural characteristics compared to normal codes as well as the complexity of the entire code structure, such as whether they contain nested structures, and so on. Therefore, we need to solve a specific case of data preprocessing, labeling, and the problem of vectorization. The following is a detailed description of these problems.

Error-handling structure extraction method

The symbols required in the method description are defined first, as shown in **Table 2**.

Table. 2. Symbols to describe

Symbols	Describe
S_o	Source files before preprocessed
S_n	Source files after preprocessed
R	Regular expression
I_e	*if-else* structure collection
I_t	Current if-else structure
I_n	Nested *if-else* structure
I_r	Non-nested *if-else* structure
B_l	Left parenthesis
B_r	Right parenthesis
C_s	The number of brackets in the stack
C_l	The number of brackets in the stack at the end of the current structure
E	Error-handling code structures

First, S_o needs to be preprocessed to remove unnecessary information, such as code comments, line feeds, and so on. R is used to effectively extract code and obtain S_n. After processing, S_n has been extracted from I_e. We propose a balance based on stack padding

with brackets, which is $\forall I \in I_e$, $C_l \equiv 0$. Therefore, we build a similar stack structure when B_l is recognized, then $C_s + 1$, when B_r is recognized, then $C_s - 1$, when $C_s = 0$, I_t is added to I_e. At the same time, the corresponding relationship between the code segment and the source file location is recorded for subsequent instrumentation. However, nested structures I_n will appear after extraction, causing misjudgments. For example, $I_n^1 \in E \,\&\, I_n^2 \notin E$ and $I_n^1 + I_n^2 = I_n$, if the model determines that $I_n \in E$, then $I_n^2 \in E$, which results in a conflict. Therefore, I_n needs to be iterated to ensure that the extracted code fragment is minimized: I_e needs to be extracted from the first extraction following the same approach until each structure is an I_r. **Listing 2** shows an extracted code snippet.

Listing. 2. Extraction code snippets

```
1: if (!frame→buf[i]) {
2:     av_frame_unref(frame);
3:     return AVERROR(ENOMEM);
4: }
```

Code parsing based abstract syntax tree (AST)

The extracted code fragment is parsed into a word sequence, and all the fragments are parsed into an equal-length sequence in this step for easy input into the LSTM network. An abstract syntax tree (AST) [27] is used to extract the code sequences in the parsing phase. Simultaneously, symbolization is carried out. For example, an integer is represented as *num* and a string is represented as *str*. However, in this classification, the string contents will have some impact, as shown in **Listing 3**.

Listing. 3. A code segment containing a string

```
1: if (!f) {
2:         fprintf(stderr, "%s: I/O error\n", filename);
3:     exit(1);
4: }
```

We have studied that most error-handling code fragments contain one feature. That is, if the code fragment contained a string, the string usually contains the words *error*, *fail* or other words with similar meanings. Therefore, we use two methods for string symbolization, including whether the string contain special keywords. These symbols are specifically expressed as *errstr* and *str*. In **Listing 3**, for example, the resolved form would be similar to ['*if*', '(','. ', '*f*', ') ', '{',' *fprintf* ', '(',' *stderr* ', ', ', '*errstr*', '*filename*'..]. Although some fault-indicating expressions are extracted by the previous analysis, their number is limited. WordNet [28] is used to extend this process. WordNet is an English dictionary, established and maintained by Princeton University. The words grouped by definitions, and each entry with the same meaning constitutes a collection of synonyms. We can use this group of entries to expand the

vocabulary of fault-indicating expressions.

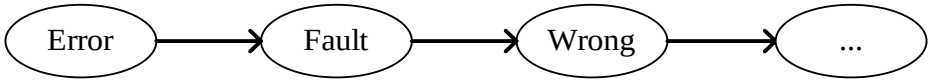

Fig. 6. Vocabulary expansion

Heuristics-based labeling

Because our approach adopts supervised learning, each sample needs to be labeled. A code segment that belongs to an error-handling code segment will be marked as 1. Otherwise, it will be marked as 0. Code snippets are labeled heuristically. The following five heuristics are summarized from the analysis of a large number of source files.

1) An *if* structure usually includes one and more comparisons as shown in **Listing 4**
2) If the segment contains a string, the string contains an error expression.
3) The segment may contain return or jump keywords such as *return*, *goto*, etc.
4) If the segment contains functions, the function name usually includes an error expression, as shown in **Listing 4**.
5) The segment may contains system error-macro definitions, such as *'EPERM'*, *'ENOENT'*, etc.

Listing. 4. A code segment containing a comparison

```
1: if (ret != length)
2:   png_error(png_ptr, "PNG Write Error");
```

Listing. 5. A code segment containing an error macro definition

```
1: if (strncmp(dev_name, prefix, strlen(prefix)))
2:       return ENODEV;
```

Input vectorization and LSTM network training

The obtained token sequences are used as input for the vectorization process. We use Word2vec, a tool widely used for text vectorization. The word vector model is obtained by setting the feature vector dimension and the word frequency parameter. The vocabulary index and word vector dictionary are established based on the model and used as input to the subsequent LSTM model. Different sections of code contain different numbers of tokens; however, the LSTM can accept only input vectors of a given length. Therefore, the vectors need to be padded or pruned. After obtaining the code segment vectorization results and the code segment labels, the LSTM network can be trained. In addition to the necessary embedded layers [29,30], LSTM units such as basic and dropout layers are added to avoid overfitting to some degree.

3.2.2 Error code detection based on the trained model

The detection phase is used to detect the type of a given code fragment. If a block of code consists of error-handling code, the trained model will output the file to which it belongs and the fragment's location in the source file. Given an unknown project, the specific detection process is as follows.

1. Extract the error-handling code snippet structures from each project file and record the file name and locations to which they belong
2. Analyze the code fragment to obtain a corresponding token sequence
3. Use the previously trained Word2vec model to vectorize the token sequence obtained in step 2 according to the preceding rules
4. Input the obtained vector into the trained LSTM network for classification

3.3 Record truncation based on lightweight instrumentation

Lightweight instrumentation mainly involves instrumenting source code and optimizing the fuzzer instrumentation. The goal of the first part is to find the proper location in the source code to instrument. The second part determines the path record truncation based on the instrumented source code.

Instrumentation position analysis and code structure repair

Based on the result of the classification model and combined with the index of the *if-else* structure location in the source file, the error-handling code segment is instrumented using both internal and external instrumentation. **Listing 6** shows an example. Because an *if* statement is compiled as a conditional jump instruction, the fuzzer instrumentation is determined by the conditional jump instruction. Therefore, instrumentation added before an *if* structure can influence whether subsequent basic blocks have been instrumented. Instrumentation added before the first statement of the *if* structure can determine the recording mode of the subsequent basic blocks. Because the first statement is the beginning of a basic block after a jump instruction, a conditional jump instruction may still exist in subsequent execution; therefore, the first statement determines the recorded results of traversing all the subsequent basic blocks on this block path containing *if* structures. Specific reasons will be explained later in this paper. There are three main situations.

1) Within an *if* structure

Listing. 6. *if* structure code fragment

```
1: if (ret < 0) {
2:   response = xasprintf("Device '%s' can not be detached", argv[1]);
3:   goto error;
4: }
```

In the first case, the error-handling code is inside the *if* structure. In this case, the code only needs to be instrumented before the *if* statement and the first statement in the *if* structure.

2) Within an *else* structure

Listing. 7. *else* structure code fragment

```
1: else {
2:     VLOG_WARN("tc: Invalid policy '%s'", policy);
3:     return;
4: }
```

In this case, error-handling code also appears in the *else* structure; therefore, it is necessary to instrument the first statement in the *else* structure as well as the adjacent *if* statement before the *else* structure. If the preorder structure is *else if* structure, the *else if*

structure needs to be transformed into an *if* structure (as shown in **Fig. 11**) and then instrumented before the *if*.

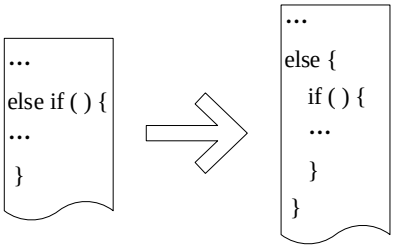

Fig. 11. *else if* structure transformation

3) Within an *else if* structure

Listing. 8. *else if* structure code segment

```
1: else if (*total_len < pin->packet_len) {
2:     VLOG_WARN_RL(&bad_ofmsg_rl, "NXT_PACKET_IN2 claimed full_len < len");
3:     return OFPERR_OFPBRC_BAD_LEN;
4: }
```

In the third case, the error-handling code appears inside an *else if* structure. After performing the process shown in **Fig. 11**, this code is instrumented before the *if* statement and before the first statement in the *else if* structure. When performing *else if* instrumentation, the original code structure is broken and must be fixed to ensure that the code compiles and runs correctly. Algorithm 1 is used to repair the source code structure.

Algorithm 1 Integrity Repair

```
1:procedure Repair(current struct)
2:   neighbor struct = current struct→next
3:   while neighbor struct do
4:      switch neighbor structure do
5:         case if struct then
6:            if neighbor struct is nest then
7:               goto next
8:            else then
9:               INTEGRITY (end of latest else-if struct)
10:           end if
11:        case else struct then
12:           if neighbor struct is nest then
13:              goto next
14:           else then
15:              INTEGRITY(end of neighbor struct)
16:           end if
17:        case else-if struct then
18:           goto next
19:   end while
20: next:
21:      neighbor struct = neighbor struct→next
22:      continue
23: end procedure
```

After executing the above algorithm, the program can be compiled and run accurately.

Path recording truncation based on lightweight instrumenting

The fuzzer instrumentation process is performed after the source file has been compiled into assembly code. Thus, the source code is instrumented by assembly code in this part. The mapping of shared memory, records of branch information and so on are implemented in the original instrumentation to collect the coverage information statistics. ER-Fuzz achieves the function of subsequent record cancellation by inserting the *continue_log* flag in the source code. Informally, we call *continue_log* named as record flag. This flag is defined in the BSS segment. Because the BSS segment contains uninitialized, this segment's memory is cleared before each round. Therefore, this flag can be set to 1 to indicate that subsequent basic blocks should no longer be recorded. By searching the *continue_log* at the entrance point of the original instrumenting code, when the flag is 1, the code can jump to the return statement to cancel the record of the basic block. The flag is subsequently always marked as 1 ; therefore, subsequent basic blocks are no longer recorded. That is, if the original path is 1→2→3→… and 3 contains the flag, the record becomes 1→2. At the beginning of the next round of testing, the flag is cleared to allow a normal execution path recorded. However, such instrumentation can occur in two situations, as shown in **Fig. 7** and **Fig. 8**.

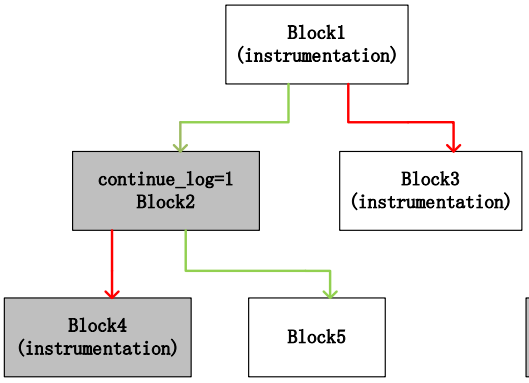

Fig. 7. Flag in uninstrumented basic block

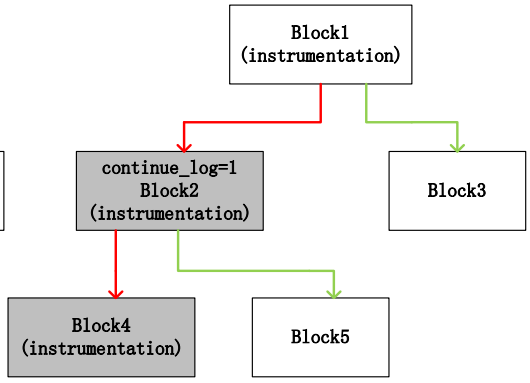

Fig. 8. Flag in instrumented basic block

```
…
mov     ds:continue_log, 1
…
jmp     loc_40091D
```

Fig. 9. Uninstrumented basic block

```
…
call    __afl_maybe_log
…
mov     ds:continue_log, 1
…
jmp     loc_40091D
```

Fig. 10. Instrumented basic block

If a record flag is inserted after the instrumentation code (**Fig. 10**), the basic block is still logged. Although subsequent blocks will not continue recording because the flag has been set to 1, this record will still be considered to have generated a new path so that the test sample is retained, which fails to accomplish the goal. Therefore, as shown in **Fig. 10**, the original instrumentation for the current conditional jump needs to be cancelled.

Because it should not affect the normal execution flow of the program, a *nop* instruction is used for instrumentation and another flag is added at the instruction annotation. Because the

assembly code annotation is not cleared after the source code is compiled as assembly code, the annotation can determine to cancel original instrumentation. When an annotation flag is encountered, an instrumentation flag is assigned. When the conditional jump instruction is encountered, the instrumentation is judged according to this flag; then, the flag is cleared. The process will be repeated until all the code has been instrumented. In this way, the subsequent code block can skip the instrumentation process. The statement *if (i! =0)* assembles instructions in different source files because of compile optimization, which may include JZ or JNZ opcodes, etc. The original instrumentation is performed only at the point of a negative jump. ER-Fuzz responds to the above situation by eliminating instrumentation at branches with flags. Although a basic block record is cancelled, the information concerning the effective path is not affected.

The advantage of this method is that when a sample exists that can meet the condition check during the continuous mutation process, the previously retained wrong sample will no longer be used to conduct mutation. Instead, ER-Fuzz will invest resources in samples that are more likely to generate new paths.

4. Experiments and Results

We implemented ER-Fuzz using both Python and C. We evaluated and compared the classification model used for error-handling code identification. In addition, we selected real applications to evaluate the performance of ER-Fuzz in terms of both code coverage and crash findings.

4.1 Error-handling code identification

A classification model is usually evaluated using accuracy and F-score metrics, which are calculated as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$F1 - score = 2 * \frac{precision * recall}{precision + recall}$$

$$Precision = \frac{TP}{TP + FP} \qquad Recall = \frac{TP}{TP + FN}$$

True Positive (TP) and True Negative (TN) belong to the cases in which samples of type 1 and type 2 are correctly classified, and False Positive (FP) and False Negative (FN) belong to the cases in which samples of type 1 and type 2 are incorrectly classified. The accuracy rate reflects the model's prediction accuracy over the entire dataset and is usually used to evaluate classifier performance. However, in the case of unbalanced datasets, the accuracy score may be misleading. As a harmonic average from P and R, the comprehensive results (F1-score) can better evaluate the effectiveness of the model. For a dataset, we selected ten widely used open source projects developed in C/C++, including *mpg123-1.25.10, libpng-1.6.35, Open vSwitch-2.9.0, libtiff-4.0.8, ImageMagick-7.0.8, Exiv2-0.26, libbpg-0.9.8, libming-0.4.8, libraw-0.19, libpcap-1.7.4*. We calculated the size of each project, the number of C/C++ source files it contained, and the number of error-handling code structures. In total, the number

of error code sections collected was 16,968, and the number of normal code sections was 20,000. We randomly chose 10,200 error code sections and 12,000 normal code sections as the training set. A total of 3,400 error code sections and 4,000 normal code sections were randomly chosen as the test set and the same numbers of sections were used for the verification set.

Table. 3. Data set information statistics

Applications	Size（MB）	Files(c/cpp)	EH
mpg123-1.25.10	4.15	103	623
libpng-1.6.35	5.67	115	1031
Open vSwitch-2.9.0	50.3	384	3959
libtiff-4.0.8	1.33	114	1287
ImageMagick-7.0.8	50.8	275	1442
Exiv2-0.26	51.9	138	75
libbpg-0.9.8	12	158	212
libming-0.4.8	21	188	1281
libraw-0.19	2.45	28	198
libpcap-1.7.4	3.11	63	933

Several rounds of training were conducted for different times on the training set. The results of each training round were statistically analyzed. The experimental results showed that after 10 training sessions, the accuracy rate and callback rate were stable at approximately 0.97, and the effect was significant.

Table. 4. classification model experimental results

Epoch	Validation accuracy	Validation loss	Testing accuracy	Testing loss	Precision	Recall	F-score
5	0.946	0.156	0.942	0.151	0.993	0.889	0.938
5	0.962	0.139	0.957	0.124	0.971	0.947	0.959
5	0.946	0.143	0.951	0.117	0.992	0.890	0.938
10	0.968	0.116	0.968	0.095	0.969	0.960	0.965
10	0.968	0.115	0.957	0.116	0.963	0.966	0.965
10	0.971	0.123	0.969	0.085	0.975	0.962	0.968
20	0.972	0.106	0.979	0.060	0.986	0.953	0.969
20	0.973	0.101	0.971	0.085	0.970	0.972	0.971
20	0.971	0.109	0.967	0.095	0.971	0.966	0.968
30	0.973	0.095	0.962	0.106	0.977	0.966	0.971
30	0.975	0.109	0.956	0.121	0.984	0.961	0.972
30	0.973	0.110	0.970	0.099	0.975	0.967	0.971

As a comparison, we refer to the data reported for IdenEH. The comparison results are shown in **Table 5**. The model using LSTM is superior to the decision tree model.

Table. 5. LSTM and Decision tree model comparison

Model	Precision	Recall	F-score
Decision tree	0.860	0.840	0.849
LSTM	0.970	0.972	0.971

4.2 Various Applications

We used a set of applications as the test data set (*mpg123-1.25.10, exiv2-0.26, tcptrace+libpcap-1.7.4, swftotcl+libming-0.4.8*) to evaluate the performance of ER-Fuzz on different volumes of error-handling code. AFLFast was selected for comparative testing in consideration of its representativeness and its open source status among coverage-based fuzzing methods (while other representative tools exist, most are not open source). In this performance comparison, we ran AFL and AFLFast in the same environment and evaluated them on 4 applications under the same configuration, i.e., a virtual machine configured with a 4 core 2GHz Intel CPU and 4 GB of RAM running Ubuntu 15.10. Each program was evaluated for 12 hours. The test metrics include code coverage and the number of crash triggers.

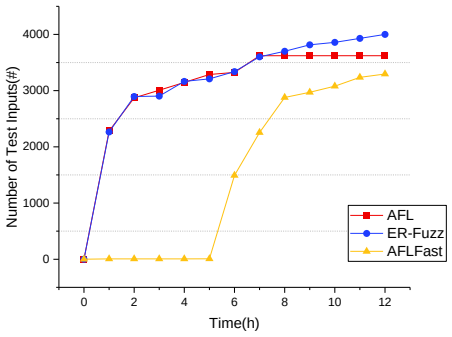

Fig. 11. swftotcl

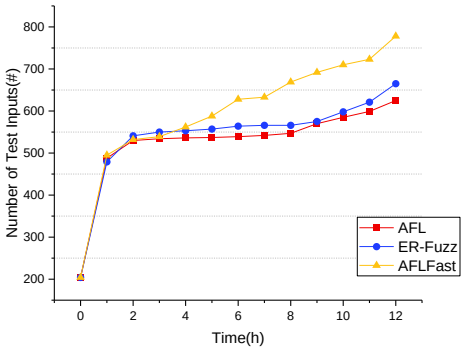

Fig. 12. Exiv2

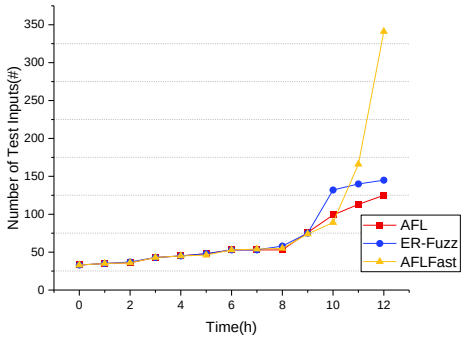

Fig. 13. mpg123

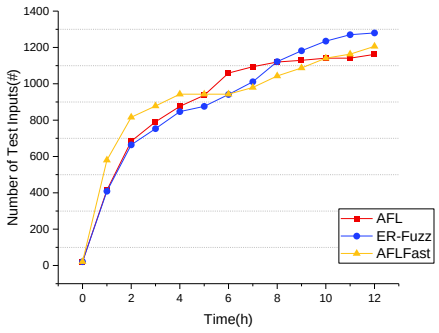

Fig. 14. Tcptrace

Figs. 11-14 show coverage comparison diagrams of AFLFast, AFL, and ER-Fuzz, where the X-axis represents the test time, and the Y-axis represents the number of new samples generated. No obvious differences appear at the beginning of the test, but after a period of time, AFL finds new paths at a slower rate than does ER-Fuzz. This result occurs because ER-Fuzz pays more attention to the low-frequency samples in the sample set, which substantially increases the coverage rate. As shown, on *swftotcl* and *tcptrace*, ER-Fuzz achieved improved coverage compared with AFL and AFLFast. However, on the other two projects, ER-Fuzz was not as effective as AFLFast. This result is mainly related to the number of error-handling code structures in the project. In other words, when the error-handling code structures constitute a larger proportion of the project, the effect of ER-Fuzz becomes more significant. In contrast,

when the proportion of error-handling code is relatively low, the improvement may not be obvious, such as on the *Exiv2* project. In all cases, no obvious change occurs at the early stage because during the early stage of testing, few paths are covered, and new paths grow relatively quickly.

In addition, we compared the number of crash findings. In two of these projects, ER-Fuzz found more crashes than did AFL and AFLFast. However, for *Exiv2*, where the number of error-handling structures is small, ER-Fuzz found fewer crash triggers than did AFLFast. During this process, ER-Fuzz found seven 0 day vulnerabilities, and CVEs were assigned.

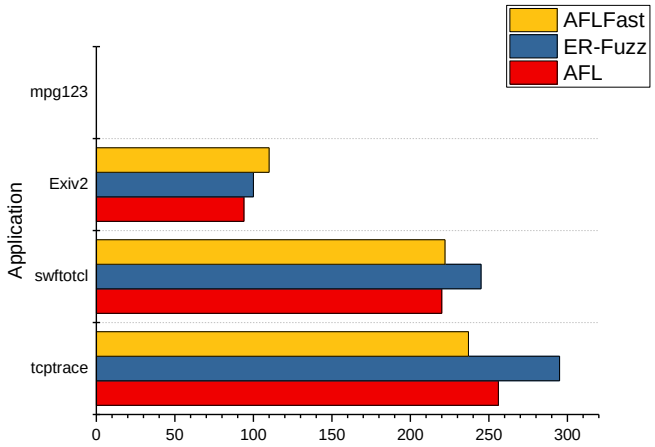

Fig. 12. Comparison of crash amount

Table 6. Vulnerability List

exiv2-0.26	CVE-2018-10958
	CVE-2018-10998
	CVE-2018-10999
	CVE-2018-11037
libming-0.4.8	CVE-2018-13066
	CVE-2018-13250
	CVE-2018-13251

5. Conclusion

In this paper, we studied the influence of high-frequency paths on fuzzing efficiency and proposed a fuzzing solution, ER-Fuzz, which resolves the issue of high-frequency path samples affecting the efficiency of the state-of-art fuzzer AFL. We proposed an instrumentation method that influences path record truncation to steer the fuzzer toward unexplored paths. In the experiments, four applications were tested and compared. The code coverage and the number of crashes found by ER-Fuzz were significantly higher than those of AFL; moreover, ER-Fuzz found seven-day vulnerabilities during the experiment. These experimental results verify the influence of high-frequency paths in overall testing. In future research, we plan to investigate whether the advantages of directed fuzzing technology can be integrated, such as concolic execution, to further improve the breakthrough time of complex condition detection or magic byte detection. In addition, we plan to address situations involving multicondition judgments and functions in judgments.

6. Acknowledgments

This work was supported by the Ministry of Science and Technology of China under Grant 2017YFB0802901.

References

[1] Sutton M, Greene A, Amini P, “Fuzzing: brute force vulnerability discovery,” *Pearson Education*, 2007.
[2] CAO Yan, “Research on Software Vulnerability Analysis Oriented Parallel Symbolic Execution,” 2013.
[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler, “KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs,” in *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI’08)*, vol. 8, pp. 209-224, December, 2008.
[4] James Newsome, Dawn Song, James Newsome, and Dawn Song, “Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software,” in *Proc. of the 12th Network and Distributed Systems Security Symposium (NDSS)*, 2005
[5] Andrew Henderson, Lok Kwong Yan, Xunchao Hu, Aravind Prakash, Heng Yin, Stephen McCamant, “DECAF: A Platform-Neutral Whole-System Dynamic Binary Analysis Platform,” *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 164–184, 2017. Article (CrossRef Link).
[6] Stephens, Nick, et al., “Driller: Augmenting Fuzzing Through Selective Symbolic Execution,” *NDSS*, vol. 16, pp. 1-16, February, 2016. Article (CrossRef Link).
[7] R. Majumdar and K. Sen, “Hybrid concolic testing,” in *Porc. of ICSE’07. Washington, DC, USA: IEEE Computer Society*, pp. 416–426, 2007. Article (CrossRef Link).
[8] Rawat, Sanjay, et al., “VUzzer: Application-aware evolutionary fuzzing,” in *Porc. of the Network and Distributed System Security Symposium (NDSS)*, February, 2017. Article (CrossRef Link).
[9] Böhme, Marcel, Van-Thuan Pham, and Abhik Roychoudhury, “Coverage-based greybox fuzzing as markov chain,” in *Porc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM*, pp. 1032-1043, 2016. Article (CrossRef Link).
[10] J. R. Norris, “Markov Chains (Cambridge Series in Statistical and Probabilistic Mathematics),” *Cambridge University Press*, July 1998.
[11] Jääskelä E, “Genetic Algorithm in Code Coverage Guided Fuzz Testing,” *University of Oulu*, 2016.
[12] M. Zalewski, “American fuzzy lop,”. Article (CrossRef Link).
[13] Mikolov, Tomas, et al., “Efficient estimation of word representations in vector space,” *arXiv preprint arXiv:1301.3781*, 2013.
[14] Hochreiter, Sepp, and Jürgen Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no.8, pp. 1735-1780, 1997. Article (CrossRef Link).
[15] Li, Yuekang, et al., “Steelix: program-state based binary fuzzing,” in *Porc. of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM*, pp. 627-637, August, 2017. Article (CrossRef Link).
[16] Tian, Yuchi, and Baishakhi Ray, “Automatically diagnosing and repairing error handling bugs in c,” in *Porc. of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM*, pp. 752-762, August, 2017. Article (CrossRef Link).
[17] Jana, Suman, et al., “Automatically Detecting Error Handling Bugs Using Error Specifications,” *USENIX Security Symposium*, pp. 345-362, August, 2016.
[18] Mitchell M, “An introduction to genetic algorithms,” *MIT press*, 1998.
[19] Shastry B, Maggi F, Yamaguchi F, et al., “Static exploration of taint-style vulnerabilities found by fuzzing,”*arXiv preprint arXiv:1706.00206*, 2017.

[20] Godefroid, Patrice, Hila Peleg, and Rishabh Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Porc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering. IEEE Press*, pp. 50-59, October, 2017. Article (CrossRef Link).
[21] Chen, Peng, and Hao Chen, "Angora: Efficient Fuzzing by Principled Search," *arXiv preprint arXiv:1803.01307*, 2018.
[22] Mouzarani, Maryam, Babak Sadeghiyan, and Mohammad Zolfaghari, "Smart fuzzing method for detecting stack-based buffer overflow in binary codes," *IET Software*, vol. 10, no. 4, pp. 96-107, 2016. Article (CrossRef Link).
[23] Böhme, Marcel, et al., "Directed greybox fuzzing," in *Porc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM*, pp. 2329-2344, October, 2017. Article (CrossRef Link).
[24] Hsu, Chin-Chia, et al., "INSTRIM: Lightweight Instrumentation for Coverage-guided Fuzzing,". Article (CrossRef Link).
[25] Liu, Jinyu, et al., "IdenEH: Identify error-handling code snippets in large-scale software," in *Porc. of Computational Science and Its Applications (ICCSA), 2017 17th International Conference on. IEEE*, pp. 1-8, July, 2017. Article (CrossRef Link).
[26] Bottou, Léon, Frank E. Curtis, and Jorge Nocedal, "Optimization methods for large-scale machine learning," *SIAM Review*, vol. 60, no. 2, pp. 223-311, 2018. Article (CrossRef Link).
[27] Rabinovich, Maxim, Mitchell Stern, and Dan Klein, "Abstract syntax networks for code generation and semantic parsing," *arXiv preprint arXiv:1704.07535*, 2017.
[28] Miller, George A, "WordNet: a lexical database for English," *Communications of the ACM*, vol. 38, no. 11, pp. 39-41, 1995. Article (CrossRef Link).
[29] Liao, Xin, Zheng Qin, and Liping Ding, "Data embedding in digital images using critical functions," *Signal Processing: Image Communication*, vol. 58, pp. 146-156, 2017. Article (CrossRef Link).
[30] Liao, Xin, Qiaoyan Wen, and Jie Zhang, "Improving the Adaptive Steganographic Methods Based on Modulus Function," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 96, no. 12, pp. 2731-2734, 2013. Article (CrossRef Link).

Xiaobin Song, M.A. degree candidate of The China National Digital Switching System Engineering and Technological Research Center. His research interests include Software Vulnerability and Cyberspace Security.

Zehui Wu, received Ph.D degree from The China National Digital Switching System Engineering and Technological Research Center. He is currently a lecturer at China National Digital Switching System Engineering and Technological Research Center. His research interests include Software Vulnerability and Software-Defined Networking(SDN).

Yan Cao, received Ph.D degree from The China National Digital Switching System Engineering and Technological Research Center. He is currently a lecturer at China National Digital Switching System Engineering and Technological Research Center. His research interests include Software Vulnerability and Program analysis.

Qiang Wei is a professor and doctoral tutor at China National Digital Switching System Engineering and Technological Research Center, Zhengzhou, China. His research interests include Software Vulnerability and Cyberspace Security.