IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received January 14, 2020, accepted April 5, 2020, date of publication April 8, 2020, date of current version April 23, 2020.

*Digital Object Identifier 10.1109/ACCESS.2020.2986578*

# Recognizing the Data Type of Firmware Data Segments With Deep Learning

**RUIQING XIAO, YUEFEI ZHU, BIN LU, XIAOYA ZHU, AND SHENGLI LIU**
State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China

Corresponding author: Shengli Liu (dr_liushengli@163.com)

This work was supported by the National Key R&D Plan under Grant 2019QY1300, Grant 2016YFB0801505, and Grant 2016YFB0801601.

**ABSTRACT** Data segment analysis is of great value for firmware analysis. The data segment contains abundant information such as pointers and strings which is helpful for accelerating the process of code segment analysis. In this paper, we propose a novel approach of applying deep learning to solve the problem of data type identification in data segments, that is a fundamental problem in data segment analysis. We define 3 data types of data segment, then design several data segment byte feature extraction methods to construct feature sequences, and finally present a deep learning-based approach with feature sequences as input to recognize the data type byte by byte. Then, the recognized type can be further corrected efficiently by prior knowledge. Based on the data segment of a firmware, we built a dataset that included 18,032,352 samples (in bytes of data segment). We implement a prototype system and evaluate it with our dataset, then determine reasonable models and hyperparameters through several experiments, and eventually confirm that deep learning techniques are suitable for identifying the data type in data segment. Kappa coefficient of our data type recognition reached 0.96 and the models can be retained quickly. Using 131,072 samples in our dataset for 32 seconds of training, the accuracy can reach 90%; the accuracy can reach 97% with 273 seconds of training and 950,272 samples. Furthermore, our approach has higher accuracy than IDA in string recognition. In experiments, the recall and precision of our approach reached 96.5% and 90% respectively, whereas corresponding results of IDA is 92.9% and 85.7%. In addition, we selected 8 open source software to compile and test, and compared the detection results with TypeMiner. Experiments show that our method has certain cross-platform and operating system capabilities, and performs better than TypeMiner on some software.

**INDEX TERMS** Reverse engineering, binary analysis, data segment, data type recognition, deep learning.

## I. INTRODUCTION

Binary program analysis is an important task in the security field, especially for firmware analysis. Based on the analysis results of a binary program, it is possible to carry out research from multiple security perspectives such as binary translation [1], patch comparison, and vulnerability detection [2]. Binary program analysis includes data segment analysis and code segment analysis. Most of the research in binary program analysis has focused on code segments (e.g., function recognition [3] and software genes). However, data segment analysis still has great value.

Firmware analysis is a relatively difficult scenario in binary analysis. In the firmware analysis phase, it is difficult to analyze the code segment because of numerous functions in firmware and the absence of API, and function recognition failure or function boundary identification failure frequently occur. In the binary comparison process, mismatches are prone to occur as for a large number of similar functions in the firmware. For the above reason, we perform the data segment analysis before code segment analysis so as to improve the effectiveness of the code segment analysis.

An efficient and accurate data segment analysis method is useful for binary program analysis. The data segment includes valuable information, such as strings and pointers, that is not included in the code segment. The information contained in the data segment can often be used as an important basis in the reverse analysis process such as a string containing the source file name and a pointer, which is helpful to find function boundaries, to the start address of the function.

The associate editor coordinating the review of this manuscript and approving it for publication was Kim-Kwang Raymond Choo.

Take the binary comparison problem as an example. If the string of the data segment can be correctly identified, then, according to the function's reference to the string, it can help one find the trusted base point in the comparison process [4] and can also be used to detect whether the binary comparison result is correct.

The type recognition problem of data segment is the basis of data segment analysis. The purpose of the data type recognition is to determine the data type (e.g., string or pointer) to which each byte of the data segment belongs. Some researchers have conducted research on type recognition [26]–[30]. Current type recognition tools usually depend on code segment heuristics or dynamic analysis whereas firmware analysis often lacks the dynamic analysis environment. We expect to start with the data segment analysis, which can support the analysis of the code segment. We pay more attention to types represented by pointers and strings, while related research has different goals for type recognition. Only TypeMiner [27] has similarities with our research, but not identical.

In this paper, we propose a method to determine the category of data segment content based on deep learning. In recent years, the application of deep learning has increased dramatically. Deep learning techniques, represented by convolutional neural networks (CNN, including LeNet5 [5], AlexNet [6], VGG [7], GoogleNet [8], and ResNet [9]) and recurrent neural networks (RNN, including LSTM [10] and GRU [11]) are gradually being applied in the field of binary analysis [12]–[17]. In terms of classification, deep learning has excellent performance such as in image recognition [18].

The byte-by-byte determination of the data type category to which the data segment data belongs is substantially a classification problem. The difficulty in this problem is how to design the correct classification rules. The advantage of deep learning is that one can find rules that are difficult to obtain manually. Therefore, we wonder that whether deep learning techniques can be used to solve the type recognition problem of the data segment data in binary files. Through the investigation, we have not found the research of applying deep learning technology to solve the data segment byte type recognition problem so far. However, our experiments show that this idea is feasible.

In response to our questions, we propose the following solutions. First, we select some features for representing bytes to form a feature vector, and three data types (S, D, and U) for analysis are defined. Second, we train a neural network with a sequence of byte features consisting of several byte feature vectors as input, representing a byte, and predicting the type of data to which the byte belongs. Third, the results obtained from the neural network are corrected by using prior knowledge. We manually parse the data segment of a binary file (18,032,352 bytes in total, Architecture: Powerpc big endian) and label it byte by byte to build a dataset. We use this dataset to train and test our models, evaluate our approach and compare the detection results of our approach directly with the detection results of IDA [19] (a binary analysis tool with string recognition). We found that we can train the transformed dataset within 8778 seconds to obtain a string recognition accuracy better than IDA. When recognizing the S-type byes, the F1 score of our method is 99.3%, yet IDA is 97.7%. When considering two string isomorphisms as positive results, the F1 score of our approach reached 93.1%, yet IDA is 89.1%. When recognizing start byte and end byte of string, the F1 scores of our method are 95.8% and 93.4%, yet the IDA's are 89.2% and 91.1% respectively. By detecting the entire data segment, for the three types of data we defined (S, D, and U), the final precision of our method reaches 99.3% (S), 96.8% (D), 95.4% (U), and the kappa coefficient is 0.96. We conduct a comparative analysis of the neural networks to find a better architecture and parameters. The experiments show that our models have the ability to quickly retrain. Using 131,072 samples in our dataset for 32 seconds of training, the accuracy rate can reach 90%; the accuracy can reach 97% when using 950,272 samples in our dataset for 273 seconds of training. In addition, we selected 8 open source software which were compiled to x86_x64 binary programs for test, and compared the detection results with TypeMiner. Experiments show that our method has certain cross-platform and operating system capabilities, and performs better than TypeMiner on some software.

In the remainder of the paper, we first define the problem at hand. We describe the particular architecture that we choose for our approach and the rules to correct the result. We give the results of our empirical evaluation, describe some related works in neural networks, and conclude with a discussion.

We summarize our contributions as follows:

1) We propose a method for the data type recognition in binary file data segments based on deep learning. It can be used before code segment analysis.

2) We propose several feature extraction methods for data segment bytes.

3) We propose several rules for correcting the data types of data segments.

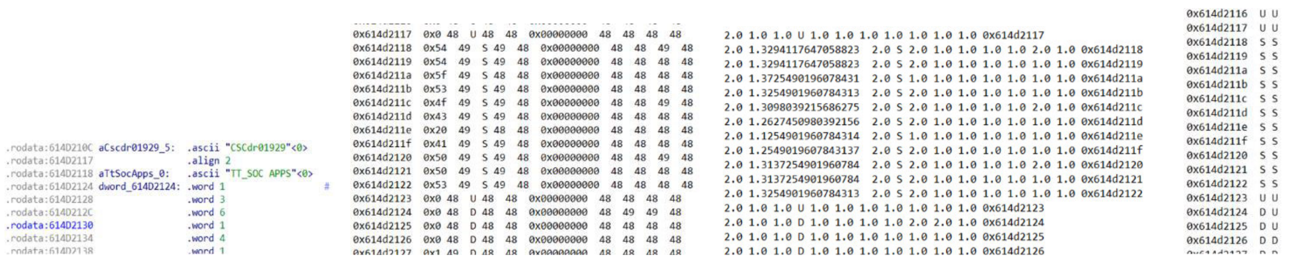
4) We implement a prototype system, evaluate our method and determine the appropriate models and parameters through comparative experiments. Experiments show that our approach has certain universality under different operating systems and architectures.

## II. OVERVIEW AND PROBLEM DEFINITION

This section will abstract the problem of binary data type (hereafter referred to as the data type or byte type) recognition for binary file data segments. Then, the definition and process of the entire task are described, and the evaluation indicators are explained.

### A. NOTATION

A binary file often contains several data segments (such as.data and.rodata). We treat the binary data $B$ of all data segmentis as a contiguous sequence of bytes $B[0]$, $B[1], \ldots, B[l]$, where each byte $B[i]$ satisfies $B[i] \in \mathbb{Z}_{256}$



(a)

```
.rodata:614D210C aCscdr01929_5:   .ascii "CSCdr01929"<0>
.rodata:614D2117                  .align 2
.rodata:614D2118 aTtSocApps_0:    .ascii "TT_SOC APPS"<0>
.rodata:614D2124 dword_614D2124:  .word 1                    #
.rodata:614D2128                  .word 3
.rodata:614D212C                  .word 6
.rodata:614D2130                  .word 1
.rodata:614D2134                  .word 4
.rodata:614D2138                  .word 1
```

(b)

```
0x614d2117  0x0 48  U 48  48  0x00000000  48  48  48  48
0x614d2118  0x54  49  S 49  48  0x00000000  48  48  49  48
0x614d2119  0x54  49  S 49  48  0x00000000  48  48  48  48
0x614d211a  0x5f  49  S 48  48  0x00000000  48  48  48  48
0x614d211b  0x53  49  S 49  48  0x00000000  48  48  48  48
0x614d211c  0x4f  49  S 49  48  0x00000000  48  48  49  48
0x614d211d  0x43  49  S 49  48  0x00000000  48  48  48  48
0x614d211e  0x20  49  S 48  48  0x00000000  48  48  48  48
0x614d211f  0x41  49  S 49  48  0x00000000  48  48  48  48
0x614d2120  0x50  49  S 49  48  0x00000000  48  48  49  48
0x614d2121  0x50  49  S 49  48  0x00000000  48  48  48  48
0x614d2122  0x53  49  S 49  48  0x00000000  48  48  48  48
0x614d2123  0x0 48  U 48  48  0x00000000  48  48  48  48
0x614d2124  0x0 48  D 48  48  0x00000000  48  49  49  48
0x614d2125  0x0 48  D 48  48  0x00000000  48  48  48  48
0x614d2126  0x0 48  D 48  48  0x00000000  48  48  48  48
0x614d2127  0x1 49  D 48  48  0x00000000  48  48  48  48
```

(c)

```
2.0 1.0 1.0 U 1.0 1.0 1.0 1.0 1.0 1.0 1.0 0x614d2117
2.0 1.3294117647058823  2.0 S 2.0 1.0 1.0 1.0 1.0 2.0 1.0 0x614d2118
2.0 1.3294117647058823  2.0 S 2.0 1.0 1.0 1.0 1.0 1.0 1.0 0x614d2119
2.0 1.3725490196078431  2.0 S 1.0 1.0 1.0 1.0 1.0 1.0 1.0 0x614d211a
2.0 1.3254901960784313  2.0 S 2.0 1.0 1.0 1.0 1.0 1.0 1.0 0x614d211b
2.0 1.3098039215686275  2.0 S 2.0 1.0 1.0 1.0 1.0 2.0 1.0 0x614d211c
2.0 1.2627450980392156  2.0 S 2.0 1.0 1.0 1.0 1.0 1.0 1.0 0x614d211d
2.0 1.1254901960784314  2.0 S 1.0 1.0 1.0 1.0 1.0 1.0 1.0 0x614d211e
2.0 1.2549019607843137  2.0 S 2.0 1.0 1.0 1.0 1.0 1.0 1.0 0x614d211f
2.0 1.3137254901960784  2.0 S 2.0 1.0 1.0 1.0 1.0 2.0 1.0 0x614d2120
2.0 1.3137254901960784  2.0 S 2.0 1.0 1.0 1.0 1.0 1.0 1.0 0x614d2121
2.0 1.3254901960784313  2.0 S 2.0 1.0 1.0 1.0 1.0 1.0 1.0 0x614d2122
2.0 1.0 1.0 U 1.0 1.0 1.0 1.0 1.0 1.0 1.0 0x614d2123
2.0 1.0 1.0 D 1.0 1.0 1.0 1.0 2.0 2.0 1.0 0x614d2124
2.0 1.0 1.0 D 1.0 1.0 1.0 1.0 1.0 1.0 1.0 0x614d2125
2.0 1.0 1.0 D 1.0 1.0 1.0 1.0 1.0 1.0 1.0 0x614d2126
```

(d)

```
0x614d2116  U U
0x614d2117  U U
0x614d2118  S S
0x614d2119  S S
0x614d211a  S S
0x614d211b  S S
0x614d211c  S S
0x614d211d  S S
0x614d211e  S S
0x614d211f  S S
0x614d2120  S S
0x614d2121  S S
0x614d2122  S S
0x614d2123  U U
0x614d2124  D U
0x614d2125  D U
0x614d2126  D D
```

**FIGURE 1. Input and output of the data type recognition. (a) Raw data (IDA analysis result); (b) Byte feature sequence (partial feature, including the label); (c) Regularized feature sequence (including the label); (d) The recognition result. Each line from left to right is the address, actual byte type, and predictive byte type.**

and $i$ represents the index value of the sequence (i.e., $B[i]$ is the $i^{th}$ byte relative to the first byte of the data segment). The address value of $B[i]$ is $A[i]$, $0 \leq A[i], A[i] \in \mathbb{Z}$, and the address sequence is $A[0], A[1], \ldots, A[l]$. According to the extractable features in the data segment, we use the vector $F[i]$ to represent the feature of the $i^{th}$ byte $B[i]$, which has $F[i] = \{f_1, f_2, \ldots, f_n\}$, where $n$ is the number of features. Therefore, the data segment byte sequence is based on the representation of the feature as $F[0], F[1], \ldots, F[l]$.

Additionally, we categorize the byte type into three categories:

- S. If a byte $B[i]$ belongs to a string, the type of byte $B[i]$ is S. This type does not contain the string end identifier \0.
- D. If $B[i]$ is a byte that constitutes a variable of a basic data type (C language, such as *int float*), the type of byte $B[i]$ is D. The basic data types include pointers. The variable length is less than or equal to 8 bytes. For example, the integer $0 \times 00000001$ contains 4 bytes, each of which belongs to type D.
- U: If $B[i]$ does not belong to type D or type S, it belongs to type U. This type includes the character \0 (ascii code: $0 \times 00$, which can be used to align strings). In addition, Type U contains situations that cannot be directly analyzed, such as compression packages, virtual machine protection codes, encrypted data, special encoded strings.

Based on the above three types (S, D, and U), we define $L[0], L[1], \ldots, L[l]$ to represent the type sequence of bytes, that is, $L[i]$ is the type label of byte $B[i]$, where $L[i]$ is one-hot encoded. $L[i] = \{\varphi_0, \varphi_1, \varphi_2\}$, and $\varphi_a = \{0 \mid 1\}$, $0 \leq a \leq 2, a \in \mathbb{Z}$.

The input and output of the data type recognition are shown in Fig. 1.

### B. TASK FLOW

The purpose of this paper is to find a method $\Phi$ that recognizes the binary data segment data type (byte by byte). $L = \Phi(B)$, where $L$ represents the result of predicting the type of data $B$ based on method $\Phi$. For the type $L[i]$ predicted by the byte $B[i]$, if $L[i]$ is equal to the label $L[i]$, the type recognition of the byte $B[i]$ is considered to be correct.

The tasks considered in this paper are listed as follows:

#### 1) FEATURE SELECTION AND EXTRACTION

For all bytes, the appropriate feature $f$ is selected and extracted to form the feature vector $F[i]$, thereby representing byte $B[i]$.

#### 2) DATA TYPE RECOGNITION BASED ON DEEP LEATNING

Based on the byte feature sequence, we can construct a model $\Phi_1$ which is suitable for recognizing byte types. We need to determine the input and output form based on the feature sequence $F$ to form the input $\sigma 1[i]$ applicable to the byte $B[i]$ and obtain the preliminary recognition result $\tau 1[i]$, $\tau 1[i] = \Phi_1(\sigma 1[i])$. $\tau 1[i]$, being similar to $L[i]$, is a three-dimensional vector composed of one-hot codes. If $\tau 1[i]$ is equal to $L[i]$, the type of data $B[i]$ is considered to be recognized.

#### 3) CORRECTION OF RESULTS BASED ON PRIOR KNOWLEDGE

For the result $\tau 1[i]$ obtained by the model proposed in B), combined with the feature sequence $F$, the partial features are added to form a new input $\sigma 2[i]$. Based on prior knowledge, the recognition method $\Phi_2$ can be formed for correcting $\tau 1[i]$. The type result $L$ can be obtained by the method $\Phi_2$, $\Phi_2{:}\sigma 2 \rightarrow L$. Because the method is relatively simple, this article only introduces some of the complementary features and rules for correcting the results and will not be elaborating from the perspective of the algorithm. The data type recognition process is shown in Fig. 2.

### C. METRICS

The questions presented in this paper are multi-classification issues, while indicators such as F1 and AUC only apply to the two-category problem. To evaluate the performance of our approach, we choose the kappa coefficient[20] as an indicator. The equations are as follows, where $K$ represents the kappa coefficient:

$$K = \frac{p_0 - p_e}{1 - p_e} \tag{1}$$

$$p_0 = \frac{\sum_{i=1}^{m} a'_i}{n} \tag{2}$$

$$p_e = \frac{\sum_{i=1}^{m} a_i * b_i}{n^2} \tag{3}$$

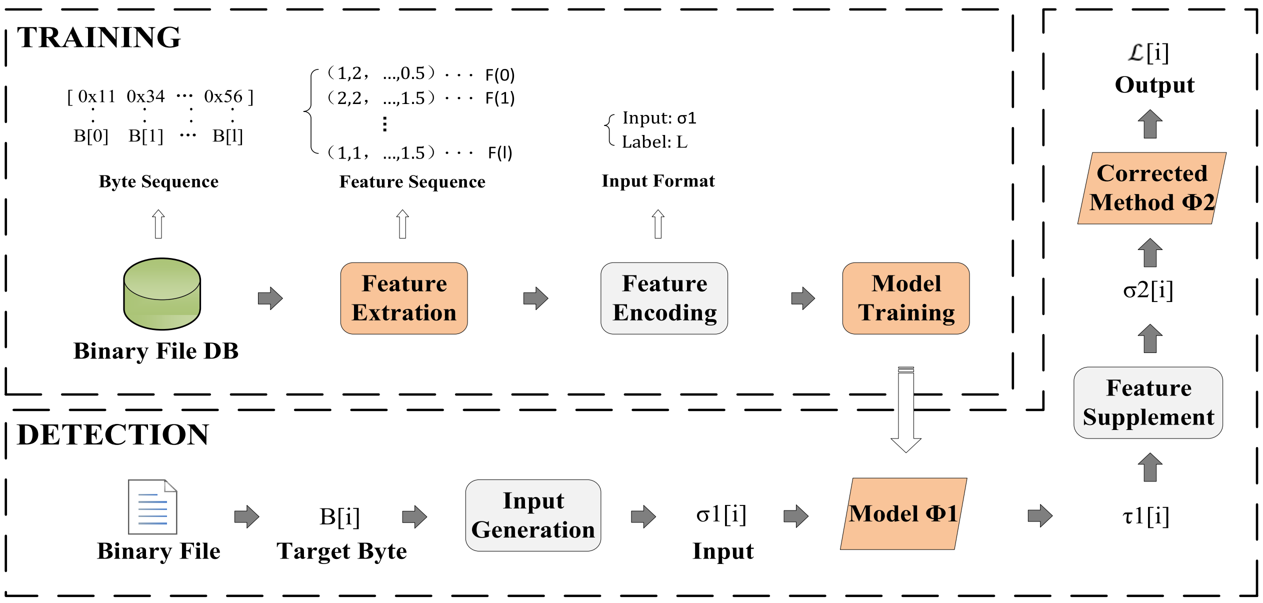


FIGURE 2. The approach overview.

For the case in which the number of classifications is $m$ and the total number of samples is $n$, $p_0$ represents the sum of the number of samples that are correctly classified in each classification and divided by the total number of samples, that is, the overall classification accuracy; (In this paper, $p_0$ is usually used to represent the accuracy of multi-classification problems, called accuracy.) $a_i$ represents the actual number of samples of the $i^{th}$ classification; and $b_i$ indicates the number of samples that are predicted to belong to the $i^{th}$ classification. $a'_i$ indicates the number of samples that are predicted to be the $i^{th}$ classification and actually belong to the $i^{th}$ classification.

The $K$ value is generally divided into five intervals to indicate consistency: $K \in [0\ ,\ 0.2)$ indicates slight consistency, $K \in [0.2\ ,\ 0.4)$ indicates fair consistency, $K \in [0.4\ ,\ 0.6)$ means moderate consistency, $K \in [0.6\ ,\ 0.8)$ indicates substantial consistency, and $K \in [0.8\ ,\ 1])$ means almost perfect consistency. K can be less than 0, which indicates an inconsistency. In general, $K < 0.4$ is considered to be unsatisfactory, and $K > 0.75$ is considered to be quite satisfactory.

In addition, we will use the precision, recall and F1 indicators to evaluate the results of string recognition, where $TP$ is the number of true positive predictions, $FP$ is the number of false positive predictions, and $FN$ is the number of false negative predictions. The F1 score is the harmonic mean of precision and recall and allows us to conveniently compare different results using one number. They have the following definitions:

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

$$Recall = \frac{TP}{TP + FN} \tag{5}$$

$$F1 = \frac{2 * Precision * Recall}{Recall + Precision} \tag{6}$$

## III. FEATURE EXTRACTION

Before the code segment analysis and data segment analysis, only two pieces of information, address and byte value (such as [0 × 614.211b, 0 × 53]), are available. It's hardly to be directly used for recognizing byte data type. Therefore, we will introduce the method of feature extraction from the two perspectives of byte and address value.

The next 8 features constitute a feature vector $F[i]$ after normalization. The sequence of features in Fig. 1 (c) is shown in Table 1.

TABLE 1. The sequence of features in Fig. 1 (c).

| Name | Section | Column Number in Fig.1 (c) |
|---|---|---|
| Distance | B.3) | 0 |
| Byte_Value | A.1) | 1 |
| Zero_Flag | A.2) | 2 |
| Letter_Flag | A.3) | 4 |
| Pointer_Flag | B.2) | 5 |
| Pointer_Target_Value | B.4) | 6 |
| Aligned_Flag | B.1) | 9 |
| Referenced_Flag | B.5) | 10 |

### A. FEATURE EXTRACTION FROM BYTE VALUES

We can extract features from byte values directly as follows (in the case of byte $B[i]$).

#### 1) BYTE VALUE

Regardless of the perspective, for the $i^{th}$ byte $B[i]$ of the data segment, the byte value itself has an important meaning; it is a feature that must be adopted. Since the value range is an integer in the interval [0, 255], it needs to be normalized before being used to train the model.

### 2) WHETHER THE BYTE VALUE IS ZERO

Without a doubt, $0 \times 00$ is a special byte value. It can be used to mark the end of a string and fill the gap between structures to ensure that the base address of the structure is aligned to the specified position, and it can also appear in the data of the base type (C language). For example, for the integer variable 1 (*int*), its binary form is $0 \times 00000001$, including three $0 \times 00$ bytes. Therefore, we place a flag item in the vector $F\,[i]$ to indicate whether the current byte is zero.

### 3) WHETHER THE BYTE VALUE IS THE ASCII ENCODING OF A NUMBER OR LETTER

If the current byte value is an ASCII code for letters or numbers, it may be S type data. Of course, there are also S-type characters such as tabs (ASCII code: $0 \times 9$) and spaces (ASCII code: $0 \times 20$) that do not conform to the previous rules. However, letters and numbers are indeed important criteria for identifying strings. Therefore, an item in the feature vector $F\,[i]$ is used as a flag to indicate whether the current byte value $B\,[i]$ is an ASCII form of letters or numbers. For some binary files, the string is stored in Unicode code. In that case, it is considered to extract features based on Unicode codes.

## B. ADDRESS-BASED FEATURE EXTRACTION

Due to the large difference in the base addresses of different binary files, as far as versatility is concerned, it is not ideal to directly use the address of a byte as a feature. We need to extract some more general features to meet the needs of analysis in different scenarios and more comprehensively reflect the role of the current byte on type recognition.

It is stated here that for byte $B\,[i]$, four bytes are read consecutively to form a double word $Dword\,[i]$, $Dword\,[i] = \sum_{k=0}^{3} B\,[i+k] * 0 \times 100^{3-k}$, and $\forall i+k > l, B\,[i+k] = 0$, where $l$ is the length of the byte sequence B.

Following the idea, the following features were selected to describe the data segment data (in the case of byte $B\,[i]$, its address value is $A\,[i]$, and its double word is $Dword\,[i]$):

### 1) WHETHER THE ADDRESS OF THE CURRENT BYTE IS ALIGNED

If $A\,[i]$ modulo 4 is 0, we consider the address of byte $B\,[i]$ is address-aligned. The alignment flag is set to 1; otherwise, it is set to 0. This feature is intended to find the first byte of strings and basic data type variable. It is always easier based on the first byte of $Dword\,[i]$, to determine if $Dword\,[i]$ is a pointer and to determine if a cross-reference exists.

We choose 4 as the modulo based on the following considerations. For RISC (reduced instruction set computing), such as PowerPC, the instruction and variable lengths are often multiples of 4. Even for CISC (complex instruction set computing), the length of the variable are often multiples of 4, which is related to whether the target system is a 32-bit system. In the CISC environment, it may be necessary to consider whether assembly code exists in the data segment. However, we will not discuss it in this paper. We mainly study the reduced instruction set; thus, we use the above conditions to determine address alignment.

### 2) WHETHER THE CURRENT DOUBLE WORD HAS THE CHARACTERISTICS OF A POINTER

In the case where the binary code segment is not parsed, it is difficult to determine if a double word in the data segment is a pointer. However, we do need to make corresponding guesses. Therefore, we design an algorithm whereby if the current double word $Dword\,[i]$ satisfies the corresponding condition, and we assume that it is a pointer if it has a high probability of being a pointer.

For byte $B\,[i]$, according to the pointer determination algorithm in Table 1, it is guessed whether the corresponding double word $Dword\,[i]$ is a pointer.

The pseudocode of the pointer determination algorithm is presented in Table 2.

**TABLE 2. Pseudocode of pointer determination algorithm.**

```
1.  Segments = Get_All_Segment (),  $IsPointer$ = **Flase**
2:  $Dword[i]$ = Get_Dword ($A, B, i$)
3:  **Foreach** seg **in** Segments:
4:      **If** $Dword[i]$>= seg.StartAddr
5:          **AND** $Dword[i]$<seg.ENDAddr:
6:          IsPointer = **TRUE**
7:          **Break**
8:  ParsedResult = Get_Address_Context_Value ($Dword[i]$)
9:  **If** ParsedResult == **Flase**:
10      $IsPointer$ = **Flase**
11  **Return** $IsPointer$
```

**Input:** the byte sequence $B$, the address sequence $A$, and the current byte index $i$.

**Output:** The flag *IsPointer*, which indicates whether the current double word is pointer.

The algorithm is divided into 3 stages:

1) According to the byte sequence $B$, the address sequence $A$ and the current index $i$, the double word $Dword[i]$ corresponding to the current byte is obtained.

2) The start and end addresses of all segments are obtained, and we determine if $Dword[i]$ is in the segment space. If not, the pointer guess result is set to False; otherwise, go to 3).

3) Use $Dword[i]$ as the address to determine whether the context value of the corresponding position can be resolved (Get_Address_Context_Value in Talbe 1 Line 8). If it is False, the result is False; otherwise, the result is True.

### 3) DISTANCE

For byte $B\,[i]$, the distance is the difference between $A\,[i]$ and $Dword\,[i]$. We use distance as features based on the following considerations.

Since the memory storage space of the binary program can be arbitrarily selected within the interval [0 × 0, 0 × FFFFFFFF] (for example, a 32-bit system), some pointers to a specific address space are also parsed into a string. For example, the pointer 0 × 0.000000 may be parsed into a string consisting of a newline character. The feature in 2) is not sufficient to solve this problem; thus, new features need to be added.

Address $A[i]$ is an important reference for determining whether $Dword[i]$ is a pointer. If the test set has a similar base address and similar storage space as the training set, the address $A[i]$ can be used as a feature. However, since the base address range of the binary file is too large, the file to be tested usually does not have the same base address as the training sample. If $A[i]$ is used directly as a feature, there is a risk of overfitting. For pointers, the difference between $A[i]$ and $Dword[i]$ should generally fluctuate within a certain range. The use of distance as a feature can alleviate above problems.

#### 4) THE VALUE POINTED TO BY THE POINTER

Extracting this feature is also intended to eliminate the case where byte $B[i]$ can be identified as an S type and as a D type at the same time. Many pointers point to functions instead of data. The start instructions of functions are often similar. Therefore, this feature can alleviate the above problems to some extent.

#### 5) WHETHER THE CURRENT BYTE IS REFERENCED

For byte $B[i]$, whether it is referenced is an important feature. If the current byte is referenced, the probability that the byte and the following bytes belong to type U will be low.

However, as in Section 2), it is difficult to determine whether the data in the data segment are referenced when the code segment is not analyzed. We can only propose an approximate scheme by scanning the data segment, based on the double word of the suspect pointer, to determine whether the current byte is referenced or not.

The pseudocode of the byte reference determination algorithm is presented in Table 3.

**Input:** The byte sequence $B$, the address sequence $A$, and the current byte index $i$.

**Output:** The flag $BeReferenced$, i.e., whether the byte is referenced.

1) We traverse the entire program and obtain information about all the data segments.

2) For byte $B[i]$,we iterate through all the data segments and check the double word $Dword[j]$, index by index, to determine whether $A[i]$ is the same as the address $Dword[j]$. If they are consistent, we set $BeReferenced$ to 1 and end program.

## IV. DATA TYPE RECOGNITION BASED ON DEEP LEARNING

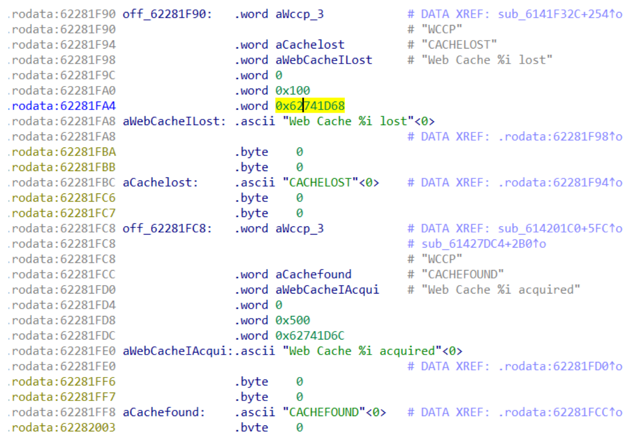
This section first analyzes an example in A) to explain the reason we select neural network, and then determines the format of the input information for data type recognition in B). Thereafter, we explain our classification model in C). Finally, we describe the training set construction in D).

**TABLE 3.** Pseudocode of referenced byte determination algorithm.

```
1.   Segments = Get_All_Segment (),
2:   $BeReferenced$ = 0
3:   **Foreach** seg **in** Segments:
4:       **For** curAddr  **in** Range(seg.StartAddr, seg.ENDAddr):
5:           CurIndex = Get_Index_of_Addr(curAddr )
6:           $Dword[CurIndex]$ = Get_Dword ($A$, $B$, $CurIndex$)
7:           **If** $BeReferenced$ == 1
8:               **OR** $Dword[CurIndex]$ == $A[i]$:
9:                   $BeReferenced$ = 1
10                   **Break**
11   **Return** $BeReferenced$
```

### A. EXAMPLES ANALYSIS

This section will first analyze an example. Take Fig. 3 as an example. In this figure, the corresponding byte value at address 0 × 62281 fa4 is 0 × 62. This value can be resolved to the letter b (ASCII code). Its previous byte is 0 × 00, and the next byte is 0 × 74 (which can be resolved to the letter t). From this perspective alone, this should be a string starting with the letter 'b', and the byte value corresponding to address 0 × 62281 fa4 should be recognized as type S. However, that is obviously incorrect.



```
.rodata:62281F90 off_62281F90:   .word aWccp_3           # DATA XREF: sub_6141F32C+254↑o
.rodata:62281F90                                         # "WCCP"
.rodata:62281F94                 .word aCachelost        # "CACHELOST"
.rodata:62281F98                 .word aWebCacheILost    # "Web Cache %i lost"
.rodata:62281F9C                 .word 0
.rodata:62281FA0                 .word 0x100
.rodata:62281FA4                 .word 0x62741D68
.rodata:62281FA8 aWebCacheILost: .ascii "Web Cache %i lost"<0>
.rodata:62281FA8                                         # DATA XREF: .rodata:62281F98↑o
.rodata:62281FBA                 .byte    0
.rodata:62281FBB                 .byte    0
.rodata:62281FBC aCachelost:     .ascii "CACHELOST"<0>   # DATA XREF: .rodata:62281F94↑o
.rodata:62281FC6                 .byte    0
.rodata:62281FC7                 .byte    0
.rodata:62281FC8 off_62281FC8:   .word aWccp_3           # DATA XREF: sub_614201C0+5FC↑o
.rodata:62281FC8                                         # sub_61427DC4+2B0↑o
.rodata:62281FC8                                         # "WCCP"
.rodata:62281FCC                 .word aCachefound       # "CACHEFOUND"
.rodata:62281FD0                 .word aWebCacheIAcqui   # "Web Cache %i acquired"
.rodata:62281FD4                 .word 0
.rodata:62281FD8                 .word 0x500
.rodata:62281FDC                 .word 0x62741D6C
.rodata:62281FE0 aWebCacheIAcqui:.ascii "Web Cache %i acquired"<0>
.rodata:62281FE0                                         # DATA XREF: .rodata:62281FD0↑o
.rodata:62281FF6                 .byte    0
.rodata:62281FF7                 .byte    0
.rodata:62281FF8 aCachefound:    .ascii "CACHEFOUND"<0>  # DATA XREF: .rodata:62281FCC↑o
.rodata:62282003                 .bvte    0
```

**FIGURE 3.** Partial data type recognition result of a binary file data segment (based on manual recognition).

When we look at a larger range (from address 0 × 62281.90 to 0 × 62282004), this seems to include two structures (C language). Among them, there are pointers, strings and others. At address 0 × 62281 fa4, a pointer should be stored, where the byte should resolve to type D. Although 0 × 62 can be parsed into the letter b, the first byte of the address in this example is also 0 × 62.

Similarly, for byte 0 × 00, if it appears after a string, it should be recognized as type U, such as 0 × 62281 fba in Fig. 3. If it appears between two pointers, it may be recognized as type D such as 0 × 62281 fd4 in Fig. 3.

Obviously, if the type of the bytes which around current byte have been analyzed, the results can better guide the type recognition of the current byte. However, in the early stage of reverse analysis, neither the code segment nor the data segment was analyzed. Type recognition based on manual rules will generate many iterations, because the change of the type recognition result of each byte may bring the change of the type recognition result of nearby bytes. In view of this situation, it is more reasonable to use a neural network to build a model and directly take a certain range of byte information as input.

## B. FEATURE ENCODING

Through above two cases, we find that the feature vector $F[i]$ based on a single byte $B[i]$ is not sufficient for byte type recognition. We need to aggregate the information of the surrounding bytes of byte $B[i]$ to aid in the analysis. Feature encoding is to combine the feature vectors near $F[i]$ into a higher-level numeric vector. The format of the new vector is the input format of the neural network.

For byte $B[i]$, on the one hand, we need to pay attention to the information from a sequence of bytes long enough before and after byte $B[i]$ to find information that appears periodically such as the pointer that appears periodically in the structure (C language).

On the other hand, we need to pay attention to the small range information before and after $B[i]$. For the problem that $0 \times 00$ belongs to type U or type D, the small-scale information extraction effect is better than the former.

Based on this consideration, we have positioned the input format as follows: $\sigma 1[i] = \{\lambda_1, \lambda_2\}$. The input consists of two parts, a long-range feature sequence $\lambda_1$ and a local feature sequence $\lambda_2$.

### 1) LONG-RANGE FEATURE SEQUENCE

Based on byte $B[i]$, first input part $\lambda_1$ is a sequence of long-range byte features, consisting of two parts, $\lambda_1 = \{\eta_1, \eta_2\}$. $\lambda_1$ summarizes the feature information of each $m1 + 1$ byte before and after byte $B[i]$ by the feature sequence $F$. To highlight byte $B[i]$, we construct $\eta_1$ based on the features of the previous $m1$ bytes of $B[i]$ and itself, and make $\eta_2$ based on the features of the next $m1$ bytes and itself. The expression for $\lambda_1$ is listed as follows:

$$\eta_1 = \{I1_0, I1_1, \ldots, I1_{m1}\} \tag{7}$$

$$\eta_2 = \{I2_0, I2_1, \ldots, I2_{m1}\} \tag{8}$$

$$I1_j = \begin{cases} F[i - m1 + j], & i - m1 + j \geq 0 \\ F_\phi, & i - m1 + j < 0 \end{cases} \tag{9}$$

$$I2_j = \begin{cases} F[i + m1\text{-j}], & i + m1 - j < l \\ F_\phi, & i + m1 - j \geq l \end{cases} \tag{10}$$

$l$ is the length of the byte sequence $B$, $m1$ is the number of bytes involved in $\eta_1$ (except for byte $B[i]$). $\eta_1, \eta_2$ have the same size, $j$ is an integer in the interval $[0, m1]$, and $F_\phi$ is a zero vector with the same shape as $F[i]$. When $B[i]$ is close to the beginning or end of the byte sequence $B$, $F_\phi$ is used to fill the portion beyond the boundary to ensure that the $\eta$ format is the same.

### 2) LOCAL FEATURE SEQUENCE

Based on byte $B[i]$, $\lambda_2$ is the second part of input $\sigma 1[i]$, and $\lambda_2$ includes the feature information of $m2$ bytes near $B[i]$. Its form is listed as follows:

$$\lambda_2 = \{I2_0, I2_1, \ldots, I2_{m2}\} \tag{11}$$

$$I2_j = \begin{cases} F[\theta(j)], & 0 \leq \theta(j) < l \\ F_\phi, & \theta(j) < 0 \\ F_\phi, & \theta(j) \geq l \end{cases} \tag{12}$$

$$\theta(j) = j + 4 * \left\lfloor \frac{i - \left\lfloor \frac{m2}{2} \right\rfloor}{4} \right\rfloor, 0 \leq j < m2 \tag{13}$$

$l$ is the length of the byte sequence $B$, $m2$ is the number of bytes involved in $\lambda_2$, including byte $B[i]$, and $j$ is an integer in the interval $[0, m2)$. Through the above operation, it can be ensured that the byte information around byte $B[i]$ is relatively uniformly extracted. In addition, the extracted information can ensure the address alignment of the first byte, which is more conducive to capture pointer-related information.

After experiments, we got the best experimental results when $m1 = 128$ and $m2 = 16$, and the experiment specifics will be provided in section V.

## C. BASIC ARCHITECTURE

In this section, we explain our design of the byte type recognition model based on deep learning. The model architecture is shown in Fig. 4.

First, the long-range feature sequence $\lambda_1$ is used as an input, therein using the RNN and CNN modules to obtain two output results and the activation function to process the results; then, we obtain two vectors $\gamma 1_1$ and $\gamma 1_2$, each of them has $2*k$ items. We connect the vectors $\gamma 1_1$ and $\gamma 1_2$, and we use the fully connected layer and the activation function to process the result to obtain the vector $\gamma 2_1$ with $k$ items. The specific construction of the CNN and RNN will be described later.

Second, the local feature sequence $\lambda_2$ is converted into the vector $\gamma 2_2$ with $k$ items by using a layer of fully connected layers and an activation function.

Third, we connect the vectors $\gamma 2_1$ and $\gamma 2_2$ and use fully connected layer and the activation function to convert them to vector $\gamma 3$ with $k$ items, finally adopt fully connected layer to convert $\gamma 3$ to a vector $\tau 1$ with 3 items.

The experiments show that when $k$= 128, the best performance is obtained. See section V for the specifics of the experiments. The formulas are as follows:

$$\gamma 1_1 = Sigmoid\,(\mathcal{R}(\lambda_1)) \tag{14}$$

$$\gamma 1_2 = Sigmoid\,(\mathcal{C}(\lambda_1)) \tag{15}$$

$$\gamma 2_1 = Sigmoid\left(\mathcal{F}2_1\left(\gamma 1_1, \gamma 1_2\right)\right) \tag{16}$$

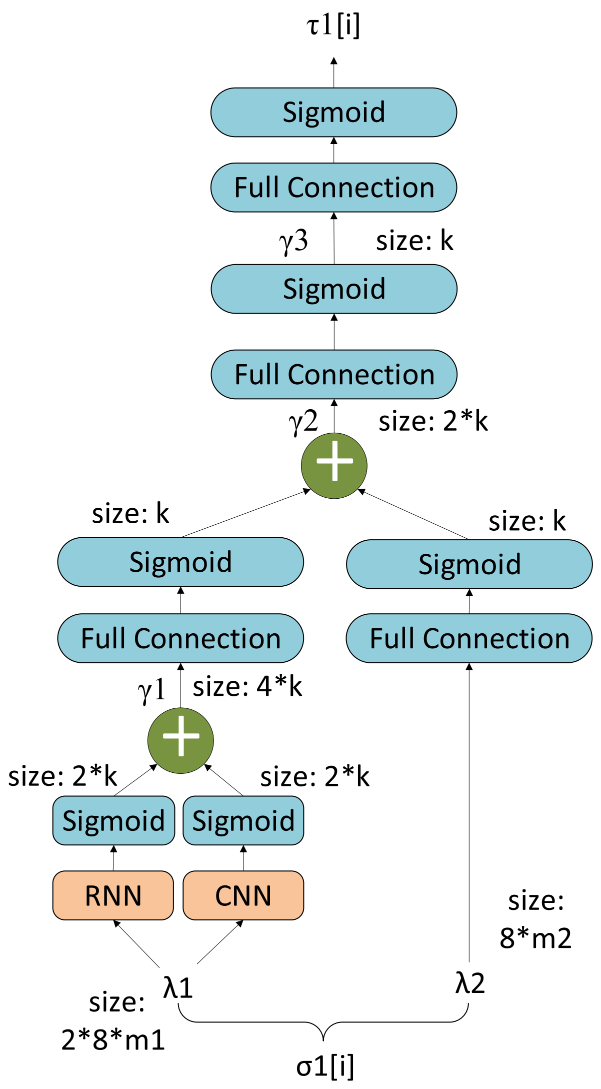


**FIGURE 4.** Byte type identification model.

$$\gamma 2_2 = Sigmoid\left(\mathcal{F}2_2\left(\lambda_2\right)\right) \tag{17}$$

$$\gamma 3 = Sigmoid\left(\mathcal{F}3\left(\gamma 2_1, \gamma 2_2\right)\right) \tag{18}$$

$$\tau 1 = F4\left(\gamma 3\right) \tag{19}$$

$\mathcal{R}$ is the RNN module, $\mathcal{C}$ is the CNN module, and $\mathcal{F}$ is the fully connected layer.

### 1) RNN MODULE

The long-range feature sequence $\lambda_1$ includes two parts: $\eta_1$ and $\eta_2$. Both $\eta_1$ and $\eta_2$ are sequences of byte features consisting of $m1 + 1$ bytes. For sequences, we use the bidirectional LSTM model for processing, and connect the outputs to form a vector $\gamma 1_1$ with $2 * k$ items. It is shown in Fig 5.

### 2) CNN MODULE

First, we connect the vectors $\eta_1$ and $\eta_2$ together as input. This is then converted into an 8-channel matrix with h bytes per line. The number of matrices is determined by the features' number of the byte feature vector $F[i]$. Then, a $g * g$ matrix is used as a convolution kernel to extract information in steps of $d1$. Next, with $d2$ as the step size, we pool the matrix with a range of $s * s$. Finally, we use the fully connected layer to control the size of output vector $\gamma 1_2$ to be $k$.

In general, we take $g = 4$, $d1 = 1$, and $h = 16$ to align the information of byte and hope to obtain some basic type data information. We let $s = 2$ and $d = 2$ to reduce unnecessary duplicate information.Through experiments, we found that except for $h$, which should be guaranteed to be a multiple of 4, changes in other parameters have little effect on this module.

The purpose of integrating CNN to our model is mainly to avoid the problem of over-fitting caused by the separate RNN module. In the experimental chapter, we will not discuss the hyperparameter selection of CNN modules, but analyze the value of the existence of CNN modules.

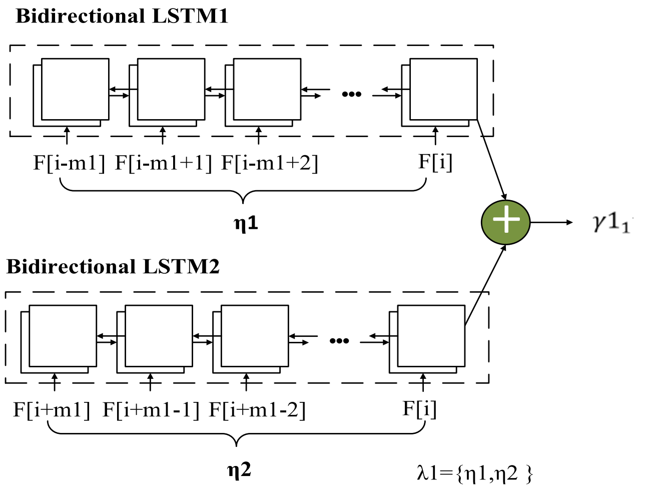


**FIGURE 5.** RNN module with bidirectional LSTM.

## D. CONSTRUCT TRAINING SET

First, in the construction process of the training set, three types of data, S, D, and U, should be randomly extracted in the same proportion. Particularly, the S type of data should be extracted according to the first byte of the string, the last byte of the string, and the other bytes of the string.

Since a string is generally longer than the basic type data, if the characters in the string are randomly extracted, the probability that the boundary character (the first and last characters of the string) is extracted is low. This will cause the inaccuracy of trained model in string boundary character recognition and affect string detection. Since the string non-boundary character is in the middle of the string, it is relatively easy to extract string features from surrounding characters, and the boundary character is relatively difficult to do this.

Second, since the pointer has a strong relationship with the base address, to ensure the versatility of the model, the sample should be transformed before being used as a training set. That need to add the same offset value to the address of the

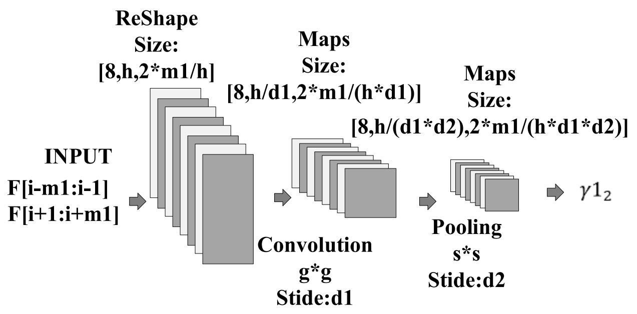
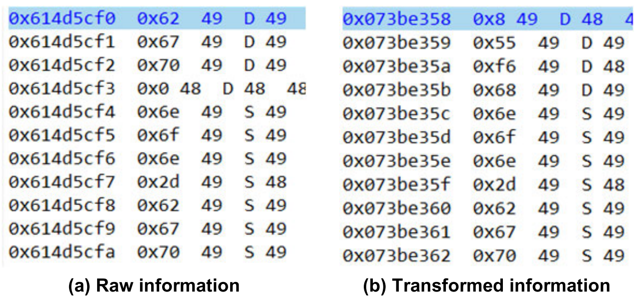


**FIGURE 6.** CNN module.

data segment dataĄCbyte by byte, and add the same offset to the data in the data segment which is a pointer. Then, information, such as the byte sequence $B$, the address sequence $A$, and the feature sequence $F$, are re-extracted to form a new data set. The new sample generation by transformation is shown in Fig. 7.



```
0x614d5cf0  0x62   49   D 49
0x614d5cf1  0x67   49   D 49
0x614d5cf2  0x70   49   D 49
0x614d5cf3  0x0 48   D 48   48
0x614d5cf4  0x6e   49   S 49
0x614d5cf5  0x6f   49   S 49
0x614d5cf6  0x6e   49   S 49
0x614d5cf7  0x2d   49   S 48
0x614d5cf8  0x62   49   S 49
0x614d5cf9  0x67   49   S 49
0x614d5cfa  0x70   49   S 49
```

(a) Raw information

```
0x073be358  0x8 49   D 48   4
0x073be359  0x55   49   D 49
0x073be35a  0xf6   49   D 48
0x073be35b  0x68   49   D 49
0x073be35c  0x6e   49   S 49
0x073be35d  0x6f   49   S 49
0x073be35e  0x6e   49   S 49
0x073be35f  0x2d   49   S 48
0x073be360  0x62   49   S 49
0x073be361  0x67   49   S 49
0x073be362  0x70   49   S 49
```

(b) Transformed information

**FIGURE 7.** A sample transform example.

It is true that the previous work of this paper has avoided the association of pointer type data with the sample base address, such as using the distance as features instead of the address. But this is not enough. The sample source is relatively single when the byte value is selected as the feature, so it is necessary to transform the sample.

## V. CORRECTION RESULT BASED ON PRIOR KNOWLEDGE

This section focuses on methods to correct the results in Section III based on prior knowledge. We first add some features and then introduce the byte-type correction rules based on address alignment and local features for bytes.

### A. FEATURE SUPPLEMENT

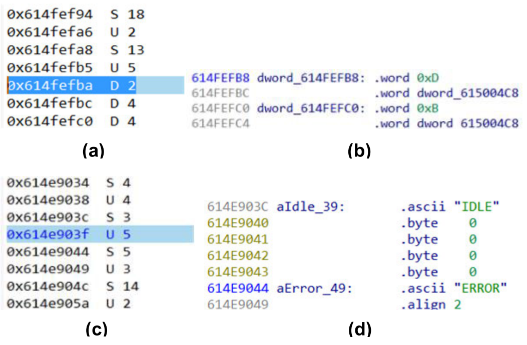
As a supplementary feature, whether adjacent bytes are same or not is a good selection. If current byte value is equal to previous byte value, the flag is set to 1. This feature is mainly used to address the problem of consecutive unrecognizable characters in a string. It is not used in deep learning since its unsubstantial help in other scenarios (byte type is U or D).

### B. CORRECTION METHOD

The correction method includes information integration, and result correction rules.

#### 1) LOCAL INFORMATION INTEGRATION

Based on the deep learning model, the byte $B[i]$ is recognized as the result $\tau 1[i]$. With the address sequence $A$ and result sequence $\tau 1$ as input, we combine the same type of bytes that appear consecutively several times into one line. The output format of one line is $[\tau 1[i], A[i], lenth]$, as shown in Fig. 8 (a) and Fig. 8(c). The algorithm is as follows:



```
0x614fef94   S 18
0x614fefa6   U 2
0x614fefa8   S 13
0x614fefb5   U 5
0x614fefba   D 2
0x614fefbc   D 4
0x614fefc0   D 4
```

(a)

```
614FEFB8 dword_614FEFB8: .word 0xD
614FEFBC                 .word dword_615004C8
614FEFC0 dword_614FEFC0: .word 0xB
614FEFC4                 .word dword 615004C8
```

(b)

```
0x614e9034   S 4
0x614e9038   U 4
0x614e903c   S 3
0x614e903f   U 5
0x614e9044   S 5
0x614e9049   U 3
0x614e904c   S 14
0x614e905a   U 2
```

(c)

```
614E903C aIdle_39:       .ascii "IDLE"
614E9040                 .byte    0
614E9041                 .byte    0
614E9042                 .byte    0
614E9043                 .byte    0
614E9044 aError_49:      .ascii "ERROR"
614E9049                 .align 2
```

(d)

**FIGURE 8.** Misrecognition examples.

**Input:** the result sequence $\tau 1$, the address sequence $A$, and the sequence length $l$.

**Output:** the result of the information integration *Type_Line*.

If the current byte is as same as the previous byte type, it is merged into the same row. In addition, the bytes belonging to type D shall not be longer than 4 bytes after merging. If the length is greater than 4, a cut will be performed according to the aligned address.

**TABLE 4.** Local information integration algorithm.

```
1.   Type_line = []
2:   i = 0, one_Line_buf=[0,0,0]
3:   While i < $l$:
4:       If $\tau 1[i] == \tau 1[i-1]$
5:       AND (type_of($\tau 1[i]$))=='D' AND $A[i]$%4 ==0) is False:
6:           one_Line_buf [2] += 1
7:       Else:
8:           Type_line.append(one_Line_buf)
9:           one_Line_buf=[ $\tau 1[i]$, $A[i]$,1]
10       i += 1
11   RETURN Type_line
```

#### 2) RESULT CORRECTION

Based on the *Type_Line* obtained after the local information is merged, we find the following cases.

1) A long byte sequence of continuous U-type bytes contains some bytes of other type that are less than 4 bytes in length.

2) Being within a large range of consecutive D-type bytes containing individual U-type bytes.

3) S- or D-type data starting with a non-aligned address; see Fig. 8 (a) (b).

4) In the case where the S and U types alternately appear, the length of U type byte sequence is greater than 4; see Fig. 8 (c) (d).

For 1) and 2) in the above cases, it is recommended to directly adjust the relevant byte type to be consistent with the surrounding bytes.

For case 3), we usually read the current double word $Dword[i]$ based on the aligned address. If there is such a double word $Dword[j]$ in its vicinity, the difference between $Dword[a]$ and $Dword[i]$ is less than a specific value; then, all four bytes $Dword[j]$ will be recognized as of D type. For example, we get the double word 0xD at address $0 \times 614$ fefb8 and the double word 0xB at 0xg14fefbc. The difference between the two double words is only 4, so all four bytes starting at $0 \times 614$ fefb8 are corrected to the D type.

Case 4) mainly occurs when the strings have characters other than alphanumeric. The byte type is corrected by determining whether the U-type byte is $0 \times 00$ and by the feature suppled in IV.A.

## VI. EVALUATION

In this section, we evaluate our approach with respect to recognition accuracy and efficiency. First, we briefly describe the experiment setup and the data sets used in our evaluation. Second, we evaluate the effectiveness of the model by comparing it with other models and evaluate the effectiveness of the hyperparameters in the model. Third, we evaluate the prototype system based on accuracy. Then, we conduct a comparison against the string recognition plugin of IDA in terms of the accuracy. Finally, we verify the universality of our method on different operating systems and architectures by comparing with Typeminer[27]. We use 8 binary programs which are compiled from popular open source projects.

### A. IMPLEMENTATION AND SETUP

We implement a prototype system for our approach with Python. The system consists of three parts: feature extraction, deep learning model, and result correction. We implement the feature extractor, a plugin to IDA, with Python 2.7. In addition to extracting features, the plugin can also construct inputs for deep learning models. We implement the deep learning model with TensorFlow [21] in Python 3.6. We implement the result correction method with Python 3.6.

Our experiments are conducted on a desktop machine equipped with an Intel Core i7-8700K CPU (6 cores in total) running at 3.7 GHz, 64 GB memory, 2 TB SSD, and an NVIDIA GeForce GTX 1080 GPU (8GB) card. During feature extraction, training, and evaluation, only one GPU card is used, and the program runs in a single thread.

The version of IDA Pro used in the experiment is version 7.0.170914 on Windows $\times 64$ (32-bit address size), and the version of TensorFlow is 1.14 (stable).

### B. DATASET

We randomly select *c3725-ipbase-mz.124-5.bin*, an image of Cisco IOS, among the total firmware possibilities of the Cisco router, with the architecture of PowerPC (big endian). After analysis with IDA, it took 3 weeks to manually recognize the data types in the *.data* and *.rodata* data segments of firmware and we labeled them byte by byte according to the three types (S, D, U) of this article, for a total of 18,032,352 bytes. Based on these labeled data, we built the training set and the test set. Admittedly, our approach to building datasets is different from other researchers, but our approach has its own necessities which we have analyzed in discussion section. In addition, in order to verify the generality of our approach, we chose the same test set as other researchers for entity verification.

#### 1) TRAINING DATASET

As mentioned in Section 3, based on the tagged data in the firmware, we randomly extracted three types of data: S, D, and U, with the same probability. A total of 360,000 bytes were taken as samples. Then, we divided the interval $[0 \times 00000000, 0 \times \mathrm{FFFFFFFF}]$ into 16 equal continuous intervals, randomly selected an address in each interval as the base address, and performed address translation on the previously extracted data according to these base addresses. The transformed data formed 16 datasets. The base address of these 16 datasets was different from that of the original firmware. We randomly extracted *dataset_num* datasets from these 16 datasets to construct a training set. The training set contains a total of $dataset\_num * 360,000$ samples. Experiments show that the effect is optimized when *dataset_num*= 14.

#### 2) TEST DATASET I

All firmware data manually parsed is directly used as a test set (a total of 18,032,352 byte were taken as samples, the firmware c3725-ipbase-mz.124-5.bin). Therefore, every time we use this test set for testing, it is equivalent to testing our approach through the firmware itself.

#### 3) TEST DATASET II

We changed the label of the D type data to U in Test Dataset I to form a new data set used to test the string recognition capabilities of our methods. The Test Dataset II has a total of 8873420 S-type bytes and 9158932 U-type bytes.

#### 4) TEST DATASET III

We collected 8 of the 14 open source codes mentioned in the literature [27] (the remaining 6 codes were not found or had compilation problems), and compiled them into programs in $\times 86$-64 (little endian), and operation system adopted Ubuntu18.04. Then we chose.rodata segment as test dataset, as shown in Table 5. This is the Test Dataset III, C which is used to test the performance of the method in different architectures and operating systems.

### C. MODEL AND HYPERPARAMETER

In this section, we evaluate the effectiveness of models and hyperparameters in our approach. As mentioned above,

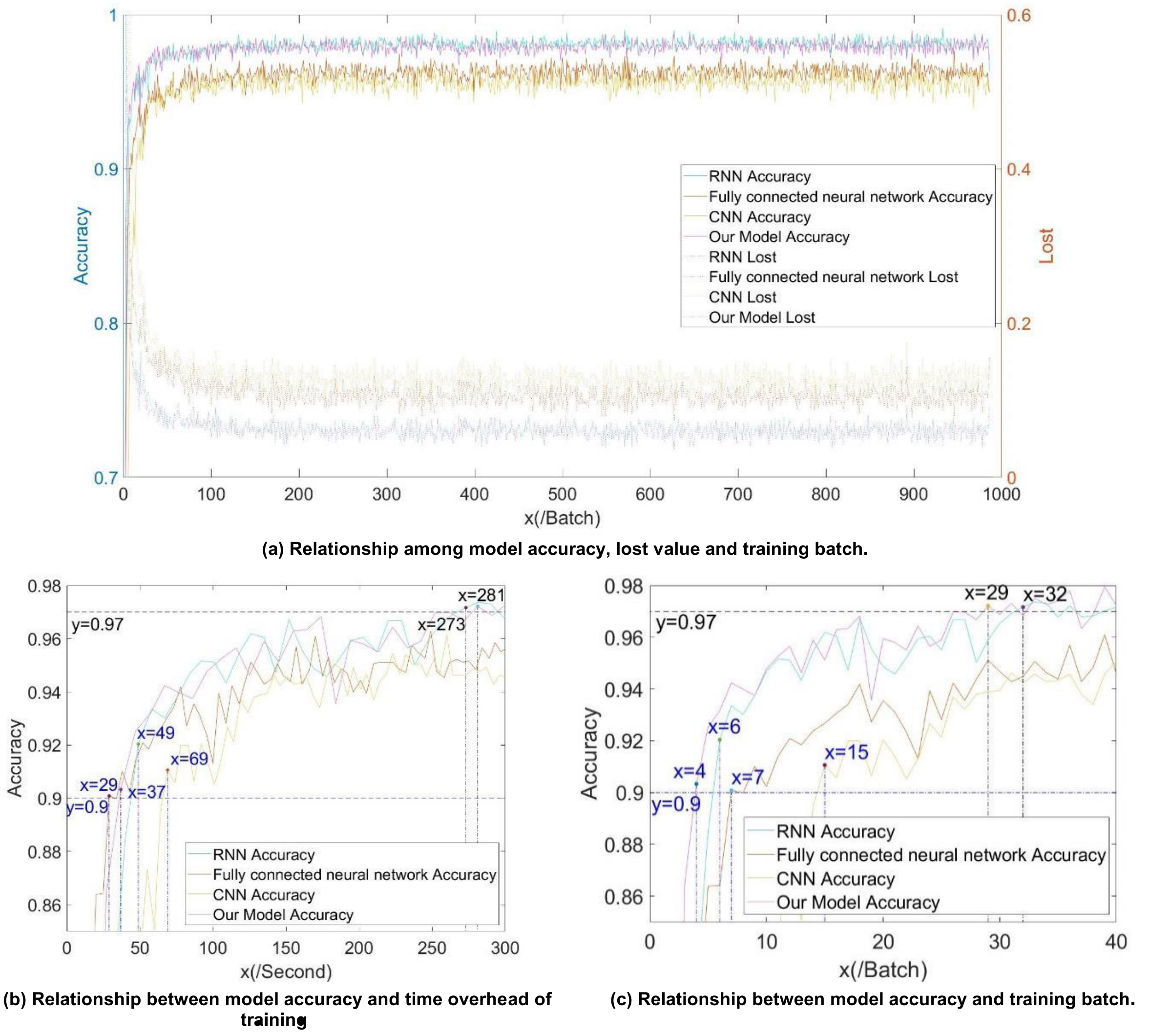


**FIGURE 9.** Training results of 4 Model.

**TABLE 5.** Overview of the binary programs in Test Dataset III.

| Program | Size | Number of data segment bytes |
|---|---|---|
| bash | 5841KB | 220808 |
| bc | 278KB | 6664 |
| cflow | 584KB | 2484 |
| gawk | 2771KB | 63984 |
| grep | 980KB | 12762 |
| gzip | 332KB | 8924 |
| indent | 286KB | 7784 |
| sed | 807KB | 13088 |

the training set is transformed from the test set, but its base address is different from the test set. Therefore, we adopt an appropriate test strategy. Whenever training a batch of training set data, a random batch of data from the test set is combined to form a batch for testing. Based on this strategy, we can quickly compare different models and parameters. In this section, we use Test Dataset I for testing.

### 1) MODEL

We compare the training test results of this model with other models (including the fully connected model, RNN, and CNN). This experiment is mainly to determine the importance of the RNN and the CNN module in our model. We removed the CNN module in our model to form an RNN-based model, referred to as RNN. We removed the RNN module in our model to form an CNN-based model, referred to as CNN. We removed the CNN module and the RNN module in our model to form a fully connected model, referred to as fully connected neural network. The accuracy of each model is shown in Fig. 9, where $m1 = 128$, $m2 = 16$, $k= 128$, $n= 14$, and the batch size is 32768 (16 * 2048).

Under current conditions, we need about 900 batches of training to traverse the training dataset, as shown in Fig. 9 (a). The training effects of four models are close, but our model works best.

We calculated the variance about the accuracy of the four models to assess the fluctuations in the accuracy of each model throughout the training process, as shown in Table 6. The variance of our model is smaller than RNN model. From this we can know that although RNN model is similar with our model in terms of experimental results, but with the increase in training batches, the accuracy of the simple RNN model fluctuates more. The model of this paper alleviates this problem because it incorporates the CNN model.

**TABLE 6. Model training time overhead and variance of accuracy.**

| | Training Time Overhead | Variance of Accuracy |
|---|---|---|
| RNN | 7649s | 0.026208 |
| Fully connected neural network | 3965s | 0.025344 |
| CNN | 4097s | 0.038176 |
| Our Model | 8778s | 0.023880 |

The time overhead is shown in the Table 6. For our model, 32768 samples per batch took about 9.25 second. In fact, the time overhead for each batch is relatively stable in the same model. The time overhead of a batch is independent from the number of samples in a batch, but is related to the model architecture. The storage space of the graphics card affects the maximum sample number of a batch. Therefore, the prime factors that determine the time consumption of this model for detecting firmware are the hardware and model architecture.

Fig. 9 (b) is a graph of the relationship between model accuracy and training time. Fig. 9 (c) reflects the relationship between model accuracy and training batches. The experimental results show that our model can achieve more than 90% accuracy with only 4 training batches (using 37 seconds), and more than 97% accuracy with 29 training batches (using 273 seconds).

This indicates that the model can be quickly retrained when the conditions of a scene change. And we only need to manually mark several batches of samples, which reduces the burden of building a training set.

All in all, our model has the best results. Our model and RNN have better accuracy which are visible to the naked eye than the other two models. Compared with the simple RNN model, our model has smaller fluctuations. More importantly, our model can complete the retraining work faster that is fit for multiple scenarios than others.

The factors that restrict retraining not only include the time spent on training, but also the time it takes to build a training set. For the model in this article, we only need 37 seconds of training with 4 batches (131,072 samples, accounting for 2.3% of the total Training Dataset sample, accounting for approximately 0.73% of the total Test Dataset I sample) to achieve 90% accuracy. RNN need 49 seconds of training with 6 batches to achieve 90% accuracy. As for 97% accuracy, our model requires 273s to train 29 batches, and RNN requires 281s to train 32 batches. The difference in training time cost between the two models consume to achieve the specified accuracy is less than 50s, but RNN model requires about 65536 more samples than our model. Obviously, it takes much more than 100 seconds to manually add 60,000 suitable samples to the training set. Regardless of this aspect, our model performs better than RNN model in retraining.

### 2) HYPERPARAMETER

We examine the impact of the transformed data set count, input size, and hidden layer output vector size. The experimental results are shown in Fig. 10.

#### *a: NUMBER OF TRANSFORMED DATA SETS*

For the count of data sets, *dataset_num* is used to build the training set. We performed 6 experiments, let *dataset_num* is equal to 6, 8, 10, 12, 14, 16 respectively, where $m1 = 64$, $m2 = 16$, $k = 64$. The experimental results are shown in Fig. 10 (a) and Fig. 10 (b). After training 100 batches, the accuracy of the model is relatively stable. We have calculated the mean and variance of the accuracy from the 100 batches to the end of the training, as shown in Table 7.

**TABLE 7. Mean and variance of the accuracy about dataset_num (after 100 batches).**

| *dataset_num* | Mean of Accuracy | Variance of Accuracy |
|---|---|---|
| 16 | 0.979125 | 0.003193 |
| 14 | 0.981298 | 0.003150 |
| 12 | 0.979246 | 0.003410 |
| 10 | 0.977579 | 0.003449 |
| 8 | 0.973253 | 0.003587 |
| 6 | 0.970708 | 0.003863 |

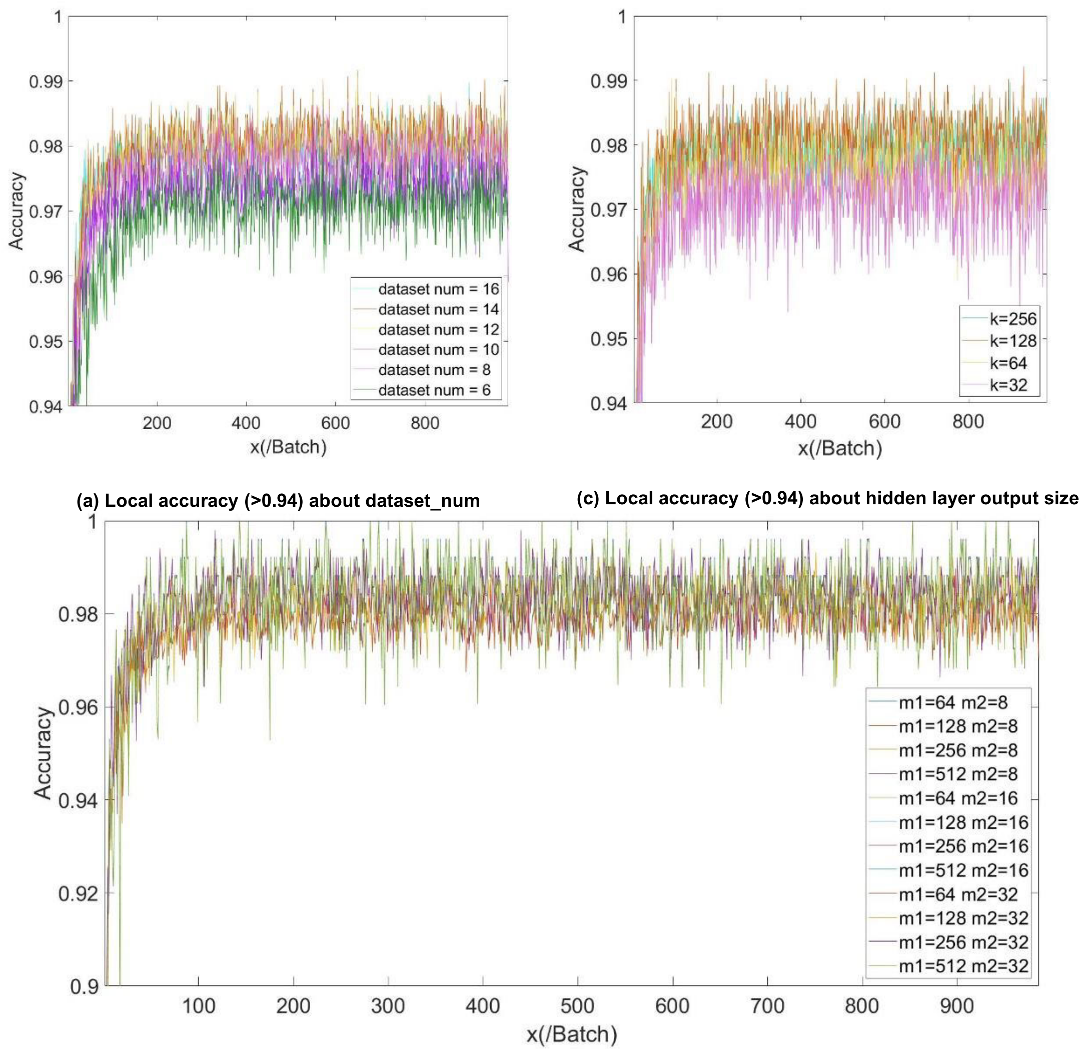
In order to clearly observe the fluctuation of accuracy, we selected the part with an accuracy of more than 94% for display, as shown in Fig. 10 (a).

Experiments show that the best effect is when *dataset_num* = 14, the mean and variance of the accuracy are better than other cases.

#### *b: INPUT SIZE (BYTE COUNTS)*

For the size of the input $\lambda_1$, we performed 12 experiments, let $m1$ is equal to 64, 128, 256, 512 and let $m2$ is equal to 8, 16, 32, where *dataset_num* = 14. After training 100 batches, the accuracy of the model is relatively stable. The mean and variance of the accuracyare as shown in Table 8.

In order to clearly observe the fluctuation of accuracy, we selected the part with an accuracy higher than 90% for display, as shown in Fig. 10 (b).



**(a) Local accuracy (>0.94) about dataset_num**

**(c) Local accuracy (>0.94) about hidden layer output size**

**(b) Local accuracy (>0.9) about input size**

**FIGURE 10. Training results with different dataset count.**

Experiments show that the best effect comes with $m1 = 64$, $m2 = 16$, the mean and variance of the accuracy are better than other cases.

As far as the experimental results are concerned, when $m1 = 64$, the average accuracy is less than 98%. When $m2 = 8$, the accuracy rate fluctuates more than others. When $m1$ is fixed and $m2 > 8$, the change of $m2$ has little effect on the experimental results. It means that the size of $m1$, $m2$ should not be too small.

When $m1$ is increased, it will basically bring about an increase in the mean value of the accuracy, and will also increase the variance. The increase in input size will result in a significant increase in training time. Considering comprehensively, the experimental results of $m1 = 128$, $m2 = 16$ and $m1 = 128$, $m2 = 32$ are ideal. For the experimental results in this paper, we think that $m1 = 128$ and $m2 = 16$ are better than other hyperparameters.

### *c: HIDDEN LAYER OUTPUT VECTOR SIZE*

For the hidden layer output vector size $k$, we performed 4 experiments, let $k$ equal 32, 64, 128, 256, where $m1 = 128$, $dataset\_num = 14$, $m2 = 16$. After training 100 batches, the accuracy of the model is relatively stable. We have calculated the mean and variance of the accuracy from the 100 batches to the end of the training, as shown in Table 9.

In order to clearly observe the fluctuation of accuracy, we selected the part with an accuracy of more than 94% for display, as shown in Fig. 10 (c). Experiments show that when $k = 128$, the mean of the accuracy are better than other cases; when $k = 256$, the variance of the accuracy are better than other cases. $k = 128$ is a better option because it has a higher accuracy mean and the variance is close to $k = 256$.

## *D. EMPIRICAL EVALUATION*

In this section, we adopt Test Dataset I to evaluate our approach (include corrected method). The kappa coefficient

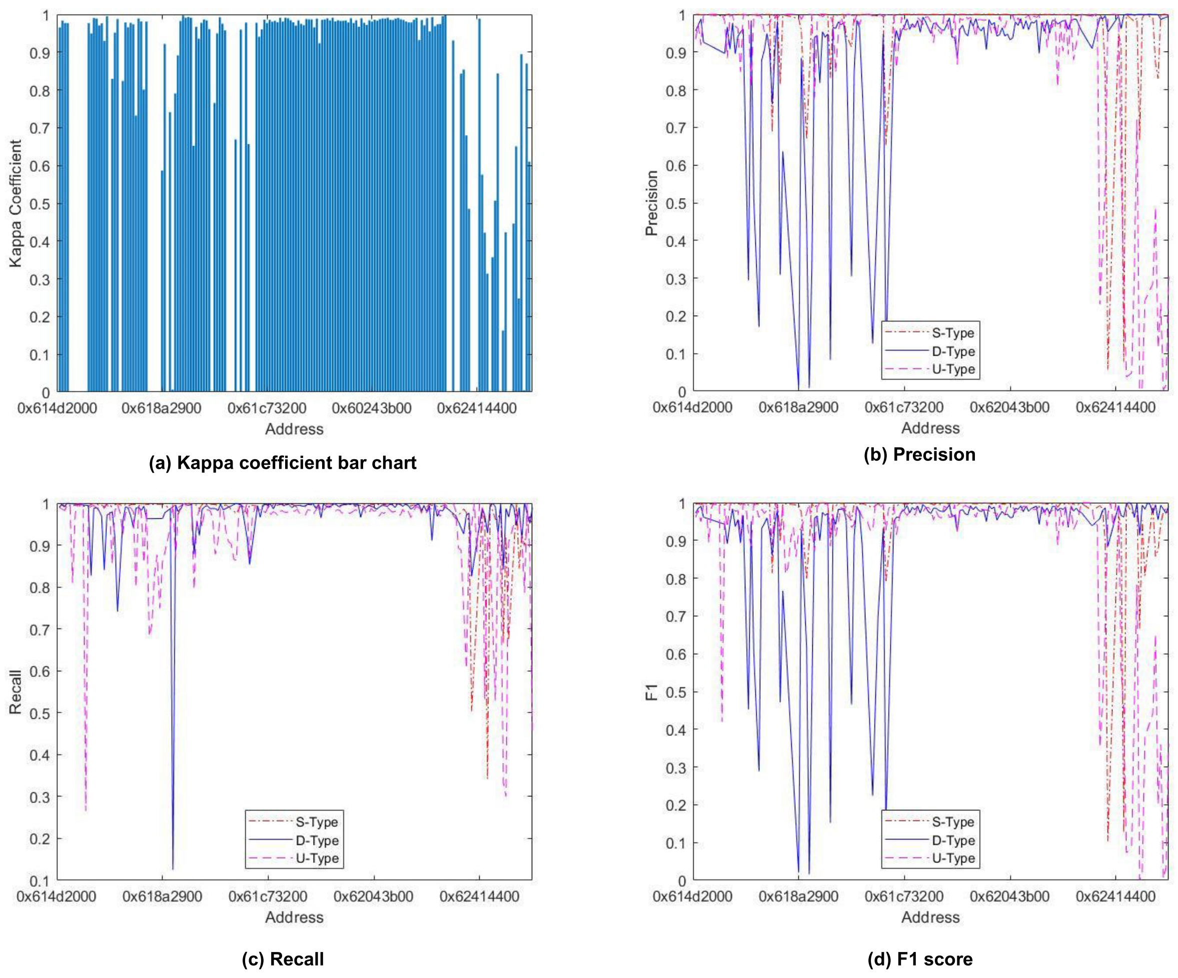


**FIGURE 11.** Evaluation result on Test Dataset I with 100,000 per item.

**TABLE 8.** Mean and variance of the accuracy about input size (after 100 batches).

| $m1$ | $m2$ | Mean of Accuracy | Variance of Accuracy |
|---|---|---|---|
| 64 | 8 | 0.978569 | 0.005373 |
| 128 | 8 | 0.981677 | 0.006053 |
| 256 | 8 | 0.985013 | 0.005449 |
| 512 | 8 | 0.984259 | 0.008441 |
| 64 | 16 | 0.978743 | 0.003292 |
| 128 | 16 | 0.983569 | 0.004111 |
| 256 | 16 | 0.984375 | 0.004781 |
| 512 | 16 | 0.984762 | 0.006889 |
| 64 | 32 | 0.979545 | 0.002964 |
| 128 | 32 | 0.984104 | 0.004333 |
| 256 | 32 | 0.984829 | 0.005676 |
| 512 | 32 | 0.986419 | 0.006579 |

of the entire test set ran up to 0.96. From Table 10, we can see that the model works well across the entire test set. The detection effect of S type byte is best among three types.

**TABLE 9.** Mean and variance of the accuracy about hidden layer output vector size k (after 100 batches).

| $k$ | Mean of Accuracy | Variance of Accuracy |
|---|---|---|
| 256 | 0.981216 | 0.004093 |
| 128 | 0.983569 | 0.004111 |
| 64 | 0.976490 | 0.004704 |
| 32 | 0.971745 | 0.005125 |

**TABLE 10.** Data type recognition on Test Dataset I.

| Type | Precision | Recall | F1 |
|---|---|---|---|
| S | 0.99321 | 0.99306 | 0.993135 |
| D | 0.96784 | 0.92347 | 0.945131 |
| U | 0.95352 | 0.97730 | 0.965261 |

The final precision of our approach reached respectively 99.3%(S), 96.8%(D), 95.4%(U).

When taking 100,000 consecutive bytes as a group, the analysis results are shown in Fig 11. The charts describe the Kappa coefficients, precision, recall, F1 score, respectively. It can be seen that the results of S-type identification are superior to D-type and U-type identification in our method. For U-type data and D-type data, the recognition effect is poor only in scenes where several groups' addresses are close. In general, our approach works well except for individual scenarios. In subsequent studies, an analysis should be conducted for such situations.

### *E. STRING RECOGNITION*

In this section, we adopt Test Dataset II for experiments. For the firmware c3725-ipbase-mz.124-5.bin, we compare the accuracy of our model with the IDA string recognition module. The IDA string recognition module refers to the function in IDA that is specifically used to discover strings and attempts to recognize string of all data in the data segment.

The experimental results are shown in Table 11 when analyzing the experimental results in bytes. F1 score of our method is 99.3%, yet IDA is 97.7%.

**TABLE 11. S-type bytes recognition: comparison of our results with IDA.**

| | Our Approach | IDA String recognition |
|---|---|---|
| TP | 8811855 | 8821074 |
| TN | 9098669 | 8851364 |
| FP | 60263 | 307568 |
| FN | 61565 | 52346 |
| Precision | 0.99321 | 0.96631 |
| Recall | 0.99306 | 0.9941 |
| F1 | 0.99314 | 0.97736 |

When analyzing the experimental results in units of strings, we need to merge consecutive S-type bytes into strings and U-bytes into one line (called U-Line). There are 341110 strings in the dataset, and 341109 U-Lines in the dataset. At this point, the evaluation can be performed from three angles, the string is isomorphic or not, the string start byte is consistent or not, and the string end byte is consistent or not.

String isomorphism means that two strings have the same starting address, length, and the same content. When regarding two string isomorphisms as positive results, the experimental results are shown in Table 12. The precision of our approach reaches 90% and the recall is 96%, F1 score is 93.1%. The precision of IDA is 85.7%, the recall is 92.9%, and F1 score is 89.1%.

The same string start byte means that the first byte of the two strings has the same address. And the concept of same string end byte is similar to the former. The experimental results are shown in Table 13.

In terms of same string start byte, the precision of our approach is 92.6% and the recall is 99.2%, F1 score is 95.8%.

**TABLE 12. String recognition with string isomorphisms: comparison of our results with IDA.**

| | Our Approach | IDA String recognition |
|---|---|---|
| TP | 329098 | 316933 |
| FP | 36246 | 52746 |
| FN | 12012 | 24177 |
| Precision | 0.90079 | 0.85732 |
| Recall | 0.96479 | 0.92912 |
| F1 | 0.93169 | 0.89178 |

**TABLE 13. String recognition with string parts are the same: comparison of our results with IDA.**

| | String Start Byte Same (Our Approach / IDA String recognition) | String End Byte Same (Our Approach / IDA String recognition) |
|---|---|---|
| TP | 338455/317042 | 329991/323613 |
| FP | 26889/52637 | 35353/46066 |
| FN | 2655/24068 | 11119/17497 |
| Precision | 0.9264/0.85761 | 0.90323/0.87539 |
| Recall | 0.99222/0.92944 | 0.9674/0.94871 |
| F1 | 0.95818/0.89208 | 0.93421/0.91057 |

The corresponding results of IDA is 85.8%, 92.9%, 89.2% respectively.

In terms of same string end byte, our precision, recall, F1 score reaches 90.3%, 96.7%, 93.4% respectively while the corresponding results of IDA is 87.5%, 94.9%, 91.1% respectively.

Some researchers believe that the analysis results of IDA are more conservative, but experiments show that, in either case, the precision, recall and F1 score of our approach are better than IDA.

### *F. CROSS-ARCHITECTURE EXPERIMENT*

In this experiment, we use Training Dataset for training (Arch: PPC big endian, OS: Cisco IOS), and use TEST DATASET III for test (Arch: ×86-×64 little endian, OS: Ubuntu 18.04).

Fig 12 shows the accuracy (Eq.2), precision, recall and F1 scores achieved by our approach for all 8 binary programs as a bar plot. And we summarize the accuracy of 8 programs in bytes, as shown in Fig 13. The first item is the total recognition accuracy of the 8 programs.

93% of all data objects are correctly classified as pointer or arithmetic types by TypeMiner [27]. This work is roughly similar to our D-type recognition. However, TypeMiner does not have accurate data to form tables for each program. Therefore, we quote the Figure 7 in that reference below, in order to compare the difference between our approach and TypeMiner, as shown in Fig 14.

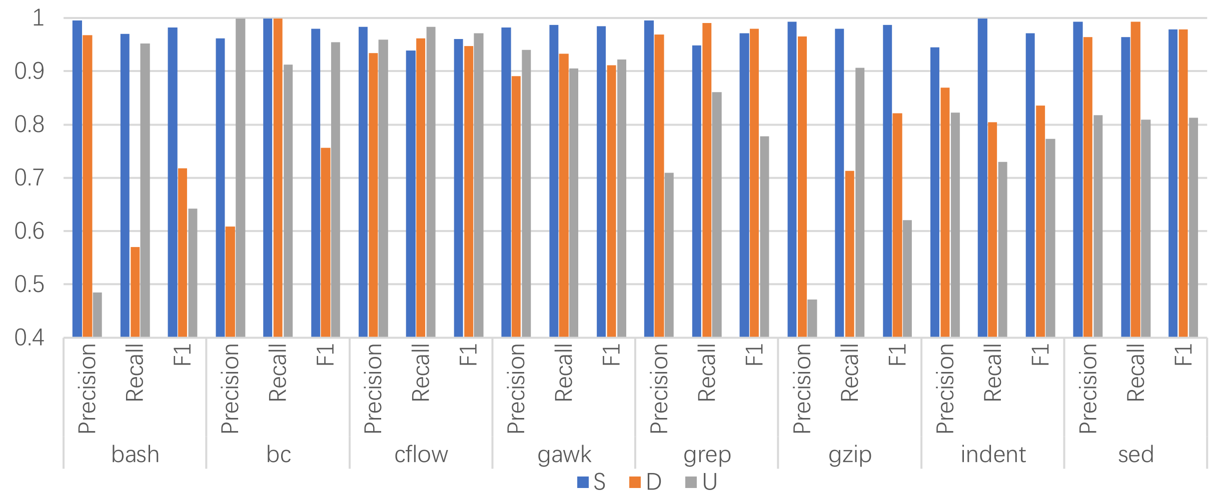


**FIGURE 12.** Evaluation result on Test Dataset III (Precision, recall and F1 scores).

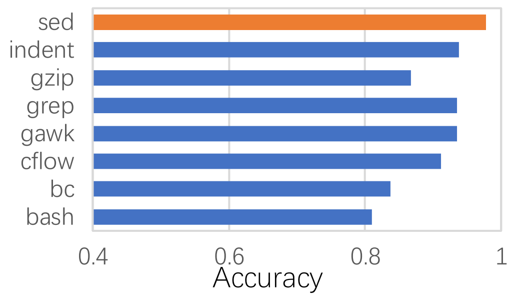


**FIGURE 13.** Evaluation result on Test Dataset III (Accuracy).

Experiments show that the method in this paper has a recognition effect close to TypeMiner, and performs better than TypeMiner on some software, but our approach only rely on data segments. And the model trained by this method has a certain cross-architecture recognition ability. Among them, the S type and D type data used to assist code segment analysis are relatively conservative in the recognition process. The precision of D-type is about 90%, and the precision of S-type is generally higher than 95%.

It is true that the performance of this method on TEST DATASET III is not ideal when compared with TEST DATASET I. But there are some reasons for this. First, the training set is little-endian and the test set is big-endian. This difference directly affects the storage form of D-type data, so the recognition F1 score of D-type data is reduced; for strings, the storage form is not affected, so the recognition F1 score is still high. Second, there are fewer absolute addresses and more relative addresses in the D-type data. This problem combines with little-endian mode can reduce the F1 score of D-type data.

## VII. DISCUSSION

### 1) DATASET

In recent years, researchers have adopted certain data set construction strategies when studying binary analysis techniques. One can obtain the source code of the open source program and then compile it. Through the symbol table, one can then obtain the information of the binary program accurately, such as the function start address, and further build a dataset. This strategy can be found in [14]–[17].

Such datasets have their limitations, and their data types often lack U-type data. In these programs, the problem becomes a bi-classification problem. However, the data segment of the firmware often contains unknown types of compressed file, and the data segment of the malicious program may contain code. Such type of data does not belong to types S or D. These programs need to solve a multiclassification problem. Therefore, the data set that we built is a rational one.

### 2) CNN MODEL

We compared the pure CNN model with our model in Section V. The CNN is good at image recognition. In our considerations, the structures are a relatively regular arrangement of data that can be easily transformed into images for processing. We predict that the composite model is superior to the single model in experimental results, but the difference in effect is so significant that we did not foresee it. We suspect that the following reasons have caused the CNN model to be less than ideal:

(1) The input is in bytes, which does not reflect structural information very well.

(2) The input size is not large enough to obtain the structure information.

(3) Data in the data set that conform to structural features represent only a small fraction. For other situations, the CNN does not apply.

In future work, we will try to improve the performance of the CNN model to improve our model.

### 3) RNN MODEL

The RNN model is able to use the information of the surrounding bytes of the target byte for analysis.

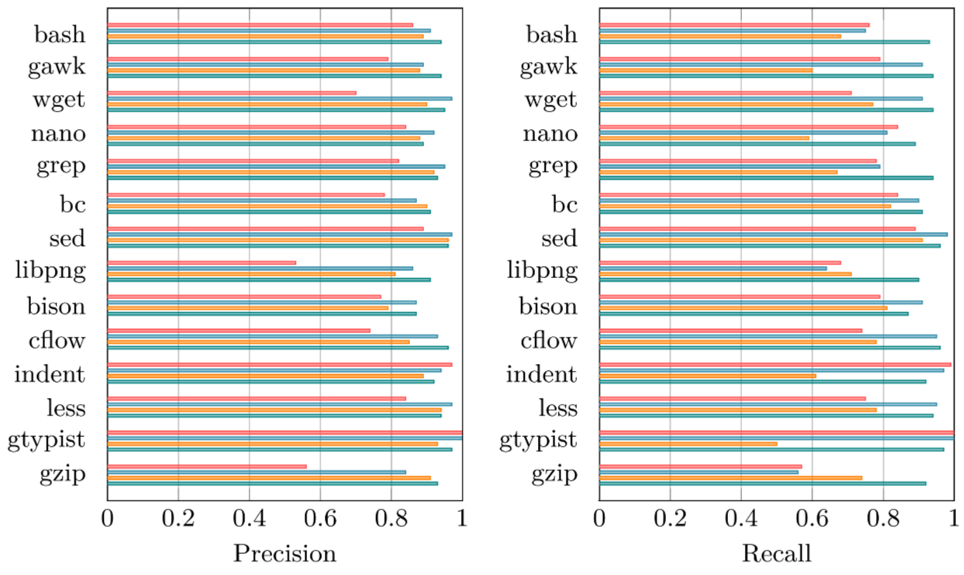


**FIGURE 14. Precision and recall of each binary program by TypeMiner. The results for different classification stages are separated by different bars (bottom to top): pointer vs. arithmetic types (🟦), pointer types( 🟧), arithmetic types (🟦), and signed vs. unsigned types (🟥). Only first item pointer vs. arithmetic is needed, it can be used to compare with type D in our approach.**

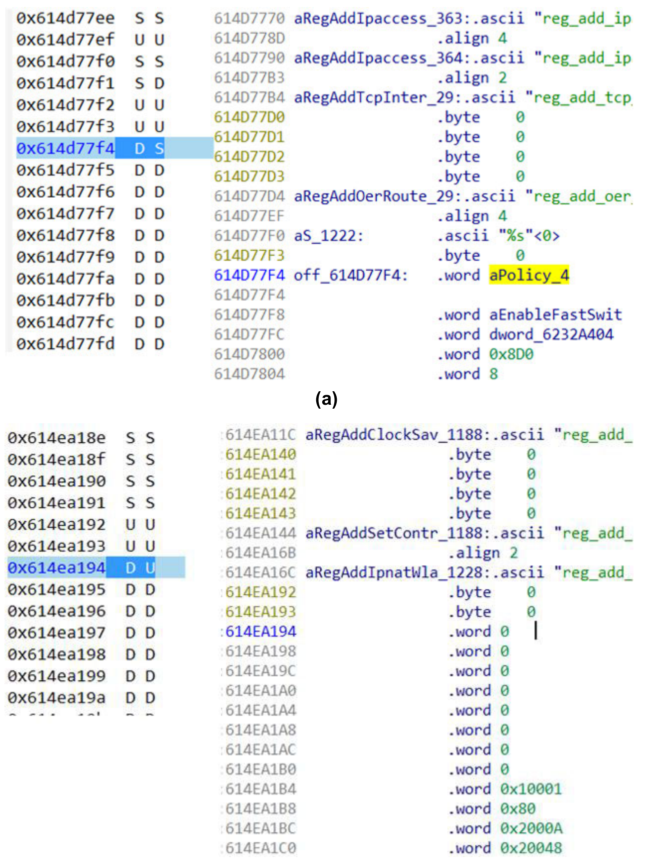
Therefore, the experimental results are also relatively ideal. However, for bytes within a certain range from the target byte, their features are not necessarily reflected in the RNN model. In this case, to ensure relatively stable capture of byte features near the target byte, we added some local input and fully connected layers to the model.

### 4) ERROR ANALYSIS

Type misrecognition by the deep learning model generally occurs when the types of bytes before and after the target byte are aggregated in different modes. For example, the byte is preceded by a number of strings, and the byte is followed by a number of pointers, as shown in Fig. 15. Regarding this situation, the deep learning model has its limitations, and further processing based on prior knowledge is a more reasonable strategy.

## VIII. RELATED WORK

Although there are some papers discuss the problem of type recognition, its application scenarios are quite differient from this article. Caballero *et al* [26] carried out type recognition based on binary code segments combined with dynamic and static analysis, and they can identify relatively many types. Maier et al [27] focused on the recognition of pointers and arithmetic types, used machine learning, carried out static analysis and research in the code segment, and implemented the TypeMiner system. Lin et al [28] proposed a method (called REWARDS) to automatically analyze the data structure from the binary file, using the timestamp type carried by each memory location accessed by the program. After that, Lee et al [29] proposed a new method on the basis of REWARDS, they transformed the assembly



```
0x614d77ee   S S     614D7770 aRegAddIpaccess_363:.ascii "reg_add_ip
0x614d77ef   U U     614D778D                     .align 4
0x614d77f0   S S     614D7790 aRegAddIpaccess_364:.ascii "reg_add_ip
0x614d77f1   S D     614D77B3                     .align 2
0x614d77f2   U U     614D77B4 aRegAddTcpInter_29:.ascii "reg_add_tcp
0x614d77f3   U U     614D77D0                     .byte    0
0x614d77f4   D S     614D77D1                     .byte    0
0x614d77f5   D D     614D77D2                     .byte    0
0x614d77f6   D D     614D77D3                     .byte    0
0x614d77f7   D D     614D77D4 aRegAddOerRoute_29:.ascii "reg_add_oer
0x614d77f8   D D     614D77EF                     .align 4
0x614d77f9   D D     614D77F0 aS_1222:            .ascii "%s"<0>
0x614d77fa   D D     614D77F3                     .byte    0
0x614d77fb   D D     614D77F4 off_614D77F4:       .word aPolicy_4
0x614d77fc   D D     614D77F4
0x614d77fd   D D     614D77F8                     .word aEnableFastSwit
                     614D77FC                     .word dword_6232A404
                     614D7800                     .word 0x8D0
                     614D7804                     .word 8
```

(a)

```
0x614ea18e   S S     614EA11C aRegAddClockSav_1188:.ascii "reg_add_
0x614ea18f   S S     614EA140                     .byte    0
0x614ea190   S S     614EA141                     .byte    0
0x614ea191   S S     614EA142                     .byte    0
0x614ea192   U U     614EA143                     .byte    0
0x614ea193   U U     614EA144 aRegAddSetContr_1188:.ascii "reg_add_
0x614ea194   D U     614EA16B                     .align 2
0x614ea195   D D     614EA16C aRegAddIpnatWla_1228:.ascii "reg_add_
0x614ea196   D D     614EA192                     .byte    0
0x614ea197   D D     614EA193                     .byte    0
0x614ea198   D D     614EA194                     .word 0
0x614ea199   D D     614EA198                     .word 0
0x614ea19a   D D     614EA19C                     .word 0
                     614EA1A0                     .word 0
                     614EA1A4                     .word 0
                     614EA1A8                     .word 0
                     614EA1AC                     .word 0
                     614EA1B0                     .word 0
                     614EA1B4                     .word 0x10001
                     614EA1B8                     .word 0x80
                     614EA1BC                     .word 0x2000A
                     614EA1C0                     .word 0x20048
```

(b)

**FIGURE 15. Error example.**

language into the intermediate language BIL and completed the type inference by combining dynamic and static analysis. Srinivasan *et al.* [30] presented a reverse-engineering tool (called Lego), which utilized information obtained from dynamic analysis and relationship about inheritance, to recover class hierarchies and composition relationships from stripped binaries.

In addition, the study of binary analysis has been ongoing for several years.

In the field of function identification, Rosenblum *et al.* [22] formulated the function identification problem as structured classification using conditional random fields, which incorporate both idiom features and control flow structure features. Karampatziakis *et al.* [23] adopted an SVM to identify basic blocks of code in a binary executable program while learning a mapping from a byte sequence to a segmentation of the sequence. Shin *et al.* [3] found that RNNs can identify functions in binaries. Yin *et al.* [24] proposed a function recognition algorithm based on the structure information. With the help of the function call resolve process, they identified functions by determining the termination position of the function based on branch instruction and termination signatures of the function.

In the field of binary comparison, research on relative techniques has continued for several years. Flake *et al.* [25] presented a method based on a control flow graph (CFG) to heuristically construct an isomorphism between the sets of functions in two similar but differing versions of the same executable file. Feng *et al.* [16] converted the CFGs into high-level numeric feature vectors. Then, they compared binary files in IoT devices across different architectures based on these features. Feng *et al.* [16] and Xu *et al.* [17] proposed a neural network-based approach to compute the embedding based on the CFG of each binary function for cross-platform binary code similarity detection. Zuo *et al.* [12] proposed a new approach based on neural machine translation (NMT), with regard to instructions as words and basic blocks as sentences, for cross-platform binary code similarity detection.

In addition, deep learning techniques are also used in other areas, such as vulnerability mining. Lyu *et al.* [13] presented a novel mutation scheduling scheme MOPT, which utilizes customized particle swarm optimization (PSO) algorithm to find the optimal selection probability distribution of operators with respect to fuzzing effectiveness and provides a pacemaker fuzzing mode to accelerate the convergence speed of PSO.

## IX. CONCLUSION

In this paper, we proposed a deep-learning-based approach for data type identification of data segment in binary file. By consulting the literature, we found there are some previous works which apply deep learning to solve problems about data type recognition of binary file. But these methods can't be used for firmware analysis, because they usually work after code segment analysis and depend on dynamic analysis. We proposed our approach to firmware data type identification problems, which based on features extraction, constructing deep learning models, and determining result correction rules. Experiments confirm that this method has higher accuracy than IDA, and has cross-architecture capabilities, the overall result is similar to TypeMiner. Our approach only rely on data segments to complete the analysis. We hope that this work will attract researchers' attention to data segment analysis and support the development of binary analysis technology.

## ACKNOWLEDGMENT

The authors would like to thank Huijie Li, Xiaokang Yin and Kang He for their helpful comments and suggestions to improve this paper. They would also like to thank the anonymous reviewers of this work for their helpful feedback, and AJE (secure.aje.com) for providing linguistic assistance during the preparation of this manuscript.

## REFERENCES

[1] M. P. Khatri and R. Radhakrishnan, "System and method for field programmable gate array-assisted binary translation," U.S. Patent Appl. 15 909 936, Sep. 5, 2019.

[2] Q. Feng, R. Zhou, Y. Zhao, J. Ma, Y. Wang, N. Yu, X. Jin, J. Wang, A. Azab, and P. Ning, "Learning binary representation for automatic patch detection," in *Proc. 16th IEEE Annu. Consum. Commun. Netw. Conf. (CCNC)*, Jan. 2019, pp. 1–6.

[3] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Proc. 24th USENIX Secur. Symp. (USENIX Secur.)*, 2015, pp. 611–626.

[4] W. Qiang, J. Ran, and W. Qingxian, "Structural signature comparison algorithm based on credible nodes," (in Chinese), *Mini-Micro Comput. Syst.*, vol. 29, no. 11, 2008.

[5] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*. [Online]. Available: http://arxiv.org/abs/1409.1556

[8] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[10] M. Sundermeyer, R. Schlüter, and H. Ney, "LSTM neural networks for language modeling," in *Proc. 13th Annu. Conf. Int. Speech Commun. Assoc.*, 2012, pp. 1–4.

[11] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," 2014, *arXiv:1412.3555*. [Online]. Available: http://arxiv.org/abs/1412.3555

[12] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," 2018, *arXiv:1808.04706*. [Online]. Available: http://arxiv.org/abs/1808.04706

[13] C. Lyu, S. Ji, and C. Zhang, "MOPT: Optimized mutation scheduling for fuzzers," in *Proc. 28th USENIX Secur. Symp. (USENIX Secur.)*, 2019, pp. 1949–1966.

[14] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Cross-architecture binary semantics understanding via similar code comparison," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, vol. 1, Mar. 2016, pp. 57–67.

[15] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient cross-architecture identification of bugs in binary code," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–15.

[16] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*. New York, NY, USA: ACM, 2016, pp. 480–491.

[17] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*. New York, NY, USA: ACM, 2017, pp. 363–376.

[18] Y. Sun, D. Liang, X. Wang, and X. Tang, "DeepID3: Face recognition with very deep neural networks," 2015, *arXiv:1502.00873*. [Online]. Available: http://arxiv.org/abs/1502.00873

[19] *IDA Pro. F.L.I.R.T.* Accessed: Apr. 16, 2018. [Online]. Available: https://www.hex-rays.com/products/ida/tech/_irt/in_depth.shtml

[20] S. Vanbelle, "Asymptotic variability of (multilevel) multirater kappa coefficients," *Stat. Methods Med. Res.*, vol. 28, nos. 10–11, pp. 3012–3026, Nov. 2019.

[21] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2016, pp. 265–283.

[22] N. E. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt, "Learning to analyze binary computer code," in *Proc. AAAI*, 2008, pp. 798–804.

[23] N. Karampatziakis, "Static analysis of binary executables using structural SVMs," in *Proc. Adv. Neural Inf. Process. Syst.*, 2010, pp. 1063–1071.

[24] X. Yin, S. Liu, L. Liu, and D. Xiao, "Function recognition in stripped binary of embedded devices," *IEEE Access*, vol. 6, pp. 75682–75694, 2018.

[25] H. Flake, "Structural comparison of executable objects," in *Proc. DIMVA*, vol. 46, 2004, pp. 161–173.

[26] J. Caballero and Z. Lin, "Type inference on executables," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 1–35, May 2016.

[27] A. Maier, H. Gascon, C. Wressnegger, and K. Rieck, "TypeMiner: Recovering types in binary programs using machine learning," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Cham, Switzerland: Springer, 2019, pp. 288–308.

[28] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proc. 11th Annu. Inf. Secur. Symp.* West Lafayette, IN, USA: CERIAS-Purdue Univ., Mar. 2010, Art. no. 5.

[29] J. H. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," in *Proc. NDSS*, 2011.

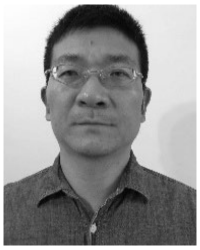
[30] V. Srinivasan and T. Reps, "Recovery of class hierarchies and composition relationships from machine code," in *Proc. Int. Conf. Compiler Construct.* Berlin, Germany: Springer, 2014, pp. 61–84.

**RUIQING XIAO** was born in Heilongjiang, China, in 1992. He received the M.S. degree in computer science and engineering from the State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China, in 2017, where he is currently pursuing the Ph.D. degree in computer science and engineering. His research interests include network and infrastructure security, deep learning, and binary reverse engineering.

**YUEFEI ZHU** was born in 1962. He is currently a Professor and a Doctoral Supervisor with the State Key Laboratory of Mathematical Engineering and Advanced Computing. His research areas are intrusion detection, cryptography, and information security.

**BIN LU** was born in 1982. He is currently an Associate Professor with the State Key Laboratory of Mathematical Engineering and Advanced Computing. His research areas are information security, machine learning, and cryptanalysis.

**XIAOYA ZHU** was born in 1996. She is currently pursuing the M.S. degree in computer science and engineering with the State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China. Her research areas are machine learning and binary reverse engineering.



**SHENGLI LIU** received the Ph.D. degree in computer science and engineering from the State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China. He is currently a Professor with the State Key Laboratory of Mathematical Engineering and Advanced Computing. His research interests include network attack behavior detection, malicious code analysis, and network infrastructure security.