API Call-Based Malware Classification Using Recurrent Neural Networks

Chen Li$^{1,*}$ and Junjun Zheng$^2$

$^1$Department of Bioscience and Bioinformatics, Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology, Iizuka, Japan
$^2$Department of Information Science and Engineering, Ritsumeikan University, Kusatsu, 525-8577, Japan
E-mail: lichen7283@gmail.com; jzheng@asl.cs.ritsumei.ac.jp
*Corresponding Author

Received 09 January 2021; Accepted 25 February 2021; Publication 25 May 2021

Abstract

Malicious software, called malware, can perform harmful actions on computer systems, which may cause economic damage and information leakage. Therefore, malware classification is meaningful and required to prevent malware attacks. Application programming interface (API) call sequences are easily observed and are good choices as features for malware classification. However, one of the main issues is how to generate a suitable feature for the algorithms of classification to achieve a high classification accuracy. Different malware sample brings API call sequence with different lengths, and these lengths may reach millions, which may cause computation cost and time complexities. Recurrent neural networks (RNNs) is one of the most versatile approaches to process time series data, which can be used to API call-based Malware classification. In this paper, we propose a malware classification model with RNN, especially the long short-term memory (LSTM) and the gated recurrent unit (GRU), to classify variants of malware by using long-sequences of API calls. In numerical experiments, a benchmark dataset is used to illustrate the proposed approach and validate its accuracy. The
numerical results show that the proposed RNN model works well on the malware classification.

**Keywords:** Malware classification, API call sequence, Recurrent neural network, Long short-term memory, Gated recurrent unit (GRU).

### 1 Introduction

With the rapid development of the Internet, the daily life of hundreds of millions of users is closely related to the Internet, such as online shopping, ordering meals, banking, etc. For example, Pandalabs has reported that there will be more than 50 billion devices connected to the Internet by 2020 [1]. Despite making our lives more convenient, the Internet makes us face the risk of being attacked, such as viruses, worms, and trojan horses.

Malicious software, or malware, can perform harmful actions on servers, computer systems, and mobile devices, to gain unauthorized access to destroy the system and steal data. Malware can not only maliciously expose users’ private information, but also may pose security threats to the economy, resulting in undesired losses such as information leakage and economic damage. Therefore, great efforts are required to prevent malware attacks.

In general, there are two main approaches for malware classification, static-based approach [2, 3] and dynamic-based approach [4, 5]. For the static-based malware classification approach, the executable file is examined on the structure without viewing the actual instructions. For example, the opcode sequence is one of the malware features, which is usually used for static analysis of malware [6]. However, the information contained in opcode sequences is quite limited, and the cost of obtaining and executing the opcode is quite high. Moreover, the static-based approach is difficult to distinguish the type of malware by only observing the fixed features, since there are a large number of variants made every moment. For the dynamic-based malware classification approach, malware is executed and identified without the need for reverse engineering of malware.

A well-known approach for dynamic-based malware classification is extracting features from binary files, which can be transformed as the input of the Convolutional Neural Network (CNN). However, the information contained in the binary data is mainly spatial, which may lack temporal information of malware. Application Programming Interfaces (APIs) contain temporal features of malware, which can be also used for dynamic malware classification [7]. Since the malware calls the APIs provided by the operating
system (OS) to execute their malicious tasks, and API call sequences can be observed according to the dynamic analysis using some automated malware analysis systems such as the Cuckoo Sandbox and Alkanet tracer system. The observed API call sequences seem to be of a good choice as features for malware classification. However, one of the main issues is how to generate a suitable feature for the algorithms of classification to achieve a high classification accuracy, since different malware sample brings API call sequence with different lengths, and these lengths may reach millions resulting in difficult computation and increased time complexity.

A neural network is a popular mathematical approach, which can automatically find an approximate function with the given input. Recently, a number of tasks have been done using RNN, such as image caption generation [8], speech recognition [9] and language modeling [10]. Recurrent Neural Network (RNN) is one of the most versatile approaches to sequence-based classification [11]. Since there exists a hidden state in RNN, the prior information in the sequence can be memorized and updated [12]. Unfortunately, conventional RNN may cause the gradient vanishing problem when processing long sequences. To resolve the gradient vanishing problem, two popular RNN variations are proposed, which are called Long Short-term Memory (LSTM) [13, 14] and Gated Recurrent Unit (GRU) [15]. For the LSTM and GRU models, the long-term dependencies in sequences can be effectively learned with a memory cell and gated units. The GRU controls the flow of information like an LSTM unit, but without having to use the memory cell. In other words, the GRU exposes the full hidden content without any control. Besides, the performance of GRU is comparable to LSTM, but the computational efficiency is higher since the gated units are simpler. In this paper, we propose a novel malware classification model with RNN, especially LSTM and GRU, to clarify which one is better for malware classification with long-sequences of API calls.

The main contributions of this paper are as follows:

1. We use an RNN model, especially LSTM and GRU models, to classify malware families considering API call information.
2. A preprocessing algorithm is proposed to reduce the noise of the long sequences of API calls and to improve the classification accuracy.
3. Numerical experiments demonstrate several evaluation indicators to compare the classification accuracy.

The remainder of the paper is organized as follows. Section 2 gives an overview of related works. Section 3 briefly introduces the conventional RNN
model, and its two variants, named LSTM and GRU, respectively. Section 4 demonstrates the proposed model. In Section 5, the experimental results are exhibited. Finally, the paper is concluded in Section 6.

2 Related Works

In this section, the past literature on malware classification is overviewed. Specifically, the static-based and dynamic-based malware classification approaches are surveyed in Sections 2.1 and 2.2 respectively.

2.1 Static-based Malware Classification

For the static analysis techniques for malware classification, there is no need to execute the malware in a controlled environment. In general, the static-based approach usually integrates machine learning algorithms in the classifier. For example, the executable file is first executed by the reverse engineer. Then, the signature, which is extracted from the malware source code, is compared with a regularly updated database.

Schultz et al. [16] designed a framework and used naive Bayes to classify malicious code statically. In the work, they extracted the features of malware by using binary profiling, string sequences, and hex dumps, and they analyzed the entire set of malicious executables instead of only boot-sector viruses, or only Win32 binaries. Kolter et al. [17] used overlapping four-byte sequences as features to improve the accuracy of classification. They presented empirical results from an extensive study of inductive methods for detecting malicious executables in the wild. Shankarapani et al. [18] proposed two general malware classification methods; static analyzer for vicious executables and malware examiner using disassembled code, which used static API calls and assembly calls for analysis, respectively. The experiment showed that the assembly call-based method was superior to the API call-based method. Raff et al. [19] proposed a feedforward neural network to classify malicious executables from raw byte sequences. The solution avoided a number of the issues with the more common byte n-gram approach, such as brittle features and over-focusing on the PE-Header as important information.

However, almost all the above approaches are difficult to distinguish the malware families by only observing the fixed features, since there are a large number of variants made every moment. Moreover, the computation cost of the static-based analysis techniques is very high, such as the byte n-gram method, which is difficult to be extended in a large domain.
2.2 Dynamic-based Malware Classification

The dynamic-based malware classification approach executes the executable file and identifies features without the need for reverse engineering of malware, such as memory writes, system calls, and API calls.

For example, Gregio et al. [20] proposed a dynamic model capture malware behavior with memory writes, which is a certain subset of instructions writes to memory during program execution. In 2015, Canzanese et al. [21] analyzed system calls to monitor executing processes to classify malware that evade traditional defense. The system monitored executing processes to identify compromised hosts in production environments. Also, Canzanese et al. [22] presented a malware analysis system, which is used to classify malicious processes at run-time on production hosts. Different to the other works, the analysis system does not require the use of specialized analysis environments. Eskandari et al. [23] proposed a binary classification approach by applying n-gram to distinguish malware binaries from API calls. In general, API call sequences contain more dynamical information of malware than the static information, such as signatures. Ahmed et al. [24] extracted spatial and temporal information from API calls and used the extracted features to classify malware. They applied statistical analysis to extract the arguments and return values from API calls as the spatial features. Moreover, they applied a Markov chain to extract the temporal information from the transition matrix by using API calls. Hansen et al. [25] and Qiao et al. [26] also proposed a malware classification system with API call-related features. Hansen et al. used API call sequences as the features for malware family classification, which are transformed from the dynamic analysis results of the Cuckoo sandbox analysis. Different from Hansen’s work, Qiao et al. conducted the malware classification by using the frequency of the API call sequences. The API call sequences were obtained from CWSandbox and Cuckoo Sandbox.

In general, the API calls often appear in the form of sequences, and the lengths of the sequences can vary from one to millions, which made the feature extraction difficult. To improve the accuracy of the malware classification from API call sequences, many researchers considered using the models of deep learning, such as the RNN model. RNN has demonstrated a powerful ability in processing time series sequences, especially in natural language processing. For example, Yazi et al. [27] presented a binary classification for malware families with a single-layer LSTM model. The LSTM model can distinguish whether an API call sequence is a malware or not, and the API
call dataset was described in [28]. Also, Catak et al. [29] detected API calls from the Windows Operating system, which is a new dataset and does not exist in this domain before. In the work, they proposed a single layer LSTM and a two-layer LSTM to classify malware families, respectively. Tobiyama et al. [30] proposed a malware process detection method based on process behavior in possible infected terminals. Specifically, they first proposed an LSTM to extract the sequence features from API calls, and then the extracted features are feed into a convolutional neural network (CNN). The CNN continuously extracted and classified the features for a binary classifier.

Most of the above LSTM models considered the malware classification problem as a binary classification, which can only classify the executable files or the identified features as malware. In this paper, we propose an RNN model, especially LSTM and GRU, to classify malware into different families, such as viruses, worms, and trojans. Also, we present to use API calls as the raw data for the RNN model. To improve the malware classification ability and reduce the noise of the RNN model, we propose a simple but effective algorithm to extract the effective information from the API call sequences.

3 Preliminaries

In this section, we briefly introduce the conventional RNN model and two well-known variants, called LSTM cell and GRU cell, respectively.

3.1 Conventional RNN Model

RNN is a kind of artificial neural network to model sequential data. In general, RNN maintains a state, which can be updated at each time step, so that the previous information can be memorized and utilized for current prediction [12]. Figure 1 shows an example of the basic RNN model. Blank circle represents a state and gray circle represents a input. In this work, the input is the API call sequences, and the output is the malware family labels, such as viruses and worms. If we unfolded the RNN according to time steps, we can represent RNN as the network like the one shown in right part of Figure 1. \(x_t, y_t\) and \(h_t\) be the input, output and hidden state of the RNN model at t-th time step, and \(U, W\) and \(V\) represent the weight matrices. Then, the hidden state \(h_t\) is updated based on the previous \(h_{t-1}\) and the current input \(x_t\):

\[
h_t = f(Ux_t + Wh_{t-1})
\]
where $f$ is the activation function, such as sigmoid, tanh, and ReLU function. The output $y_T$ is computed by:

$$y_T = \text{softmax}(Vh_T)$$  \hfill (2)

### 3.2 Long Short-term Memory

LSTM [13] and GRU are the two variants of the basic RNN model, which are better at extracting long term dependencies of time series data. Figure 2 demonstrates the LSTM cell. In general, there exist three main gates in a general LSTM cell, which are called the input gate, forget gate, and output gate, respectively.
The LSTM function is implemented by the following composite functions:

\[ i_t = \sigma (W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \]  
\[ f_t = \sigma (W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \]  
\[ c_t = f_t c_{t-1} + i_t \tanh (W_{xc}x_t + W_{hc}h_{t-1} + b_c) \]  
\[ o_t = \sigma (W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_{t-1} + b_o) \]  
\[ h_t = o_t \tanh (c_t) \]

where \( i_t, f_t, o_t, \) and \( c_t \) are the input gate, forget gate, output gate and cell content, respectively. Also, \( \sigma \) and \( \tanh \) are the sigmoid and tanh function. \( W_{hi}, W_{xo}, W_{xi}, W_{ci}, W_{xf}, W_{hf}, \) and \( W_{xc} \) are the hidden-input gate matrix, input-output gate matrix, input gate matrix, cell-input matrix, input-forget gate matrix, hidden-forget gate matrix and input-cell gate matrix, respectively. \( b_i, b_f, b_c \) and \( b_o \) are the basis of input gate, forget gate, cell and output gate, respectively.

### 3.3 Gated Recurrent Unit

Different from the LSTM cell, the structure of GRU is relatively simple. Figure 3 shows the basic structure of a GRU model. In general, there are two main gates in a GRU [15] model, which are called the reset gate and update gate, respectively.

Define \( r_t \) and \( z_t \) represent the reset gate and update gate at \( t \)-th time step, respectively, which are computed as follows:

\[ r_t = \sigma (W_r x_t + U_r h_{t-1}) \]  
\[ z_t = \sigma (W_z x_t + U_z h_{t-1}) \]
where $W_r, U_r, W_z$ and $U_z$ are weight matrices of the reset gate and update gate, respectively. Then, the hidden state $h_t$ can be calculated as follows:

$$h_t = z_t h_{t-1} + (1 - z_t) \tilde{h}_t$$  \hspace{1cm} (10)

$$\tilde{h}_t = \tanh (W_r x_t + U_r (r_t \odot h_{t-1}))$$  \hspace{1cm} (11)

where $\odot$ represents the Hadamard product.

4 API Call Based Recurrent Neural Networks

The RNN-based malware classification model is introduced in this section. In general, we first describe the creation of the malware dataset, and then, we propose a novel preprocessing algorithm for the dataset to reduce the noises of the API call sequences. Finally, we generate the RNN model to classify the malware families.

4.1 Extraction of the API Call Sequences

The malware family samples are used to generate the malware executable repository (MER). Then, the generated API call sequences are sent to Cuckoo Sandbox for the dynamic analysis. Cuckoo Sandbox is a well-known framework for malware dynamic analysis, which can generate analysis reports for each malware sample. The generated analysis reports contain static and dynamic analysis results, such as hash values. In this paper, we only use the API call sequences as the input of the RNN model. On the other hand, the VirusTotal service API is used to generate malware family labels. The malware family labels and API call sequences together form the malware dataset.

For a better understanding, Figure 4 demonstrates the process of dataset creation. The created dataset contains a series of (label, sequence) pairs, where the label represents the type of malware, while the sequence contains a series of names of the executed API calls.

Table 1 shows some examples of the (label, sequence) pairs. $L_i$ represents malware family label. For example, $L_i$ is a kind of worms, viruses, trojan, downloader, backdoor, dropper, spyware, and adware. $S_i$ represents the name of API calls, such as “\texttt{exception}” and “\texttt{regclosekey}”. In the example, it is observable that the sequence of $L_1$ type calls the API file $S_1$ repeatedly. Also, the $L_2$ and $L_3$ repeatedly call the sub-sequence $\{S_1, S_2\}$ and $\{S_1, S_2, S_3\}$, respectively. However, the repeatedly called API sub-sequences
have little effect on the classification results. On the contrary, the repeated API calls or the sub-sequences may increase the noise of the RNN model.

4.2 Data Preprocessing

To reduce the noise of the RNN model, we propose a simple but effective algorithm to preprocess the original API call sequences. For example, since there exists the repeated API $S_1$ in the API sequence of $\{S_1, S_1, S_1, S_1, \ldots\}$, we remove the continuously same API $S_1$. Besides, for the API sequences $\{S_1, S_2, S_1, S_2, \ldots\}$ and $\{S_1, S_2, S_3, S_1, S_2, S_3, \ldots\}$, there also exist the same API sub-sequences $\{S_1, S_2\}$ and $\{S_1, S_2, S_3\}$, we delete these continuously same sub-sequences, respectively. The algorithm of sequence preprocessing is illustrated in Algorithm 1. After data preprocessing, data noise and the lengths of

---

**Table 1** An example of the (label, sequence) pairs in the dataset

<table>
<thead>
<tr>
<th>label</th>
<th>sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1$</td>
<td>$S_1, S_1, S_1, \ldots$</td>
</tr>
<tr>
<td>$L_2$</td>
<td>$S_1, S_2, S_1, S_2, \ldots$</td>
</tr>
<tr>
<td>$L_3$</td>
<td>$S_1, S_2, S_3, S_1, S_2, S_3, \ldots$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
API call sequences are reduced. The algorithm result str as the input is feed into the RNN model.

**Algorithm 1** Removal of continuously same API call.

**Input:** The original API call sequence seq.

**Output:** Sequence str after removing the duplicate pattern.

1: \( s_1 = [i[0] \text{ for } \{i \text{ in groupby(seq)}\}] \)
2: \( str = \text{''}.join(s_1) \)
3: \( \text{index} = [] \)
4: \( \text{for } i \text{ in range (len(str))}: \)
5: \( \text{if } str[i] == str[i + 2] \text{ and } str[i + 1] == str[i + 3]: \)
6: \( \text{index.extend([i + 2, i + 3])} \)
7: \( i += 2 \)
8: \( s_2 = \text{np.delete(str, index)} \)
9: \( str = \text{''}.join(s_2) \)
10: \( \text{index} = [] \)
11: \( \text{for } i \text{ in range (len(str))}: \)
12: \( \text{if } str[i] == str[i + 3] \text{ and } str[i + 1] == str[i + 4] \text{ and } str[i + 2] == str[i + 5]: \)
13: \( \text{index.extend([i + 3, i + 4, i + 5])} \)
14: \( i += 3 \)
15: \( s_3 = \text{np.delete(str, index)} \)
16: \( str = \text{''}.join(s_3) \)
17: \( \text{return str} \)

### 4.3 RNN Model Description

Formally, given an API call sequence \( X = \{x_1, x_2, \ldots, x_m\} \), where \( x_i \) (1 ≤ \( i \) ≤ \( m \)) is the index of an API call from the API dictionary with size \( v \). Notice that \( m \) is very different since the length of the API call sequence can reach one to millions as we mentioned above. However, there exists a large number of redundant subsequences in the API call sequence, which have little effect on malware classification. To reduce the length of the sequence to fit the proposed LSTM model, we remove the continuously same API from the sequence. Although the removal operation is a simple method, it can effectively reduce the length of sequences and improve the accuracy of classification. Let \( T \) be the length of the LSTM model, then the sequence...
is padded or truncated to \( X = \{ x_1, x_2, \ldots, x_T \} \). In our model, we use \( v \)-dimensional one-hot vector to represent \( x_i \), which means \( x_i \in \mathbb{R}^v \).

Figure 5 demonstrates the architecture of our proposed RNN model. The one-hot vector is high-dimensional and sparse, which means only one element is 1, and the rest are 0 in the one-hot vector. Then, the one-hot vector \( x_i \) as input to be feed into the embedding layer. The embedding layer uses a dense vector to represent the one-hot vector, which can reduce the dimension of the one-hot vector.

The output is defined by \( y \), where \( y = [y_1, y_2, \ldots, y_K] \) and \( y \in \mathbb{R}^v \). \( y \) is generated by the softmax function, which is a probability distribution over all \( K \) malware families. In other words, \( y_i \) represents the probability of malware family \( i \), where \( i = (1, 2, \ldots, K) \), and \( \sum_{i=1}^{K} y_i = 1 \).

In the training phase, backpropagation-through-time (BPTT) is often used for propagating gradients of errors. BPTT is a variant of the backpropagation approach for training feedforward neural networks. Error is computed using cross-entropy loss function:

\[
\text{Loss}(y', y) = \sum_{i=0}^{K} y'_i \log(y_i)
\]  

where \( y' \) is the true probability distribution, and \( y \) is the predicted probability distribution.

Besides, we apply Adagrad as the optimization method, which is a modified stochastic gradient descent method with a per-parameter learning rate. Also, the dropout technique is used as a regularization to alleviate overfitting.
5 Numerical Experiments

In this section, several extensive experiments are conducted to evaluate the effectiveness of our proposed model. We evaluate some indications by using a benchmark dataset. Finally, we evaluate the performance of the proposed LSTM model and GRU model. All experiments are run in Keras 2.4.3 with Python 3.5.6.

5.1 Dataset

A benchmark dataset for malware classification is used in the experiment. The dataset contains 7,107 malware records for 8 different families of malware, which are represented by 0 to 7. The 8 families are worms, viruses, trojan, downloader, backdoor, dropper, spyware, and adware. Figure 6 shows the number of malware sequences for the 8 families, where the x-axis represents the eight malware families, and the y-axis is the number of corresponding sequences of each family.

Figure 7 shows the original lengths of API call sequences, where the x-axis and y-axis represent the lengths of API call sequences and the number of corresponding sequences, respectively.

Figure 8 demonstrates the lengths of API call sequences after the preprocessing operation.

It is observable that the lengths of most of the API call sequences are less than 400 after removing redundant API calls from the original sequences.
5.2 Performance Metrics

We calculate the accuracy, precision, recall, F1-score, Macro-average and weighted-average. Formally, define TP, TN, FP and FN as the true positives,
true negatives, false positives and false negatives, respectively. Figure 9 demonstrates the confusion matrix for each of the eight malware families.

Then, the accuracy, precision, recall and F1-score can be defined as

\[
\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (13)
\]

\[
\text{precision} = \frac{TP}{TP + FP} \quad (14)
\]

\[
\text{recall} = \frac{TP + TN}{TP + FN} \quad (15)
\]

\[
F1 = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \quad (16)
\]

Besides, the Macro-average can be computed by averaging the precision, recall, and F1-score, respectively. Also, similar to the Macro-average, the weighted-average metric assigns weights to every malware family according to the number of samples of each family, hence the name.

5.3 Parameter Setting

We select a single hidden layer of the RNN model to classify the eight different types of malware. More specifically, we use the LSTM model and GRU model for the classification, respectively. Tables 2, 3 and Figures 10, 11 show the single hidden layer RNN model and the optimal hyperparameters of the RNN model, respectively.

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Optimal hyperparameters for the proposed LSTM model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hyperparameters</td>
<td>Value</td>
</tr>
<tr>
<td>LSTM length</td>
<td>200</td>
</tr>
<tr>
<td>Embedding units</td>
<td>32</td>
</tr>
<tr>
<td>Length of hidden layer</td>
<td>32</td>
</tr>
<tr>
<td>Activation function</td>
<td>Sigmoid function</td>
</tr>
<tr>
<td>Kernel initializer</td>
<td>Random uniform</td>
</tr>
<tr>
<td>Dropout</td>
<td>0.5</td>
</tr>
<tr>
<td>Optimizer</td>
<td>Adam</td>
</tr>
</tbody>
</table>
Table 3  Optimal hyperparameters for the proposed GRU model

<table>
<thead>
<tr>
<th>Hyperparameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRU length</td>
<td>200</td>
</tr>
<tr>
<td>Embedding units</td>
<td>64</td>
</tr>
<tr>
<td>Length of hidden layer</td>
<td>64</td>
</tr>
<tr>
<td>Activation function</td>
<td>Sigmoid function</td>
</tr>
<tr>
<td>Kernel initializer</td>
<td>Random uniform</td>
</tr>
<tr>
<td>Dropout</td>
<td>0.5</td>
</tr>
<tr>
<td>Optimizer</td>
<td>Adam</td>
</tr>
</tbody>
</table>

Figure 10  Single hidden layer LSTM classification model structure.

Figure 11  Single hidden layer GRU classification model structure.

5.4 Experimental Results

Figures 12–15 show the training loss, accuracy of the LSTM model and GRU model, respectively. Notice that the EarlyStopping approach is used to solve the overfitting issue. The analysis results of the testing phase are shown in
Figure 12  Training loss of the proposed LSTM model.

Figure 13  Training accuracy of the proposed LSTM model.
Figure 14  Training loss of the proposed GRU model.

Figure 15  Training accuracy of the proposed GRU model.
### Table 4  Classification results of the proposed LSTM model

<table>
<thead>
<tr>
<th>Malware Families</th>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 1</th>
<th>Case 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adware</td>
<td>0.58</td>
<td>0.77</td>
<td>0.95</td>
<td>0.91</td>
<td>0.72</td>
<td>0.83</td>
</tr>
<tr>
<td>Backdoor</td>
<td>0.60</td>
<td>0.55</td>
<td>0.28</td>
<td>0.51</td>
<td>0.38</td>
<td>0.53</td>
</tr>
<tr>
<td>Downloader</td>
<td>0.65</td>
<td>0.71</td>
<td>0.52</td>
<td>0.72</td>
<td>0.58</td>
<td>0.71</td>
</tr>
<tr>
<td>Dropper</td>
<td>0.26</td>
<td>0.62</td>
<td>0.37</td>
<td>0.51</td>
<td>0.30</td>
<td>0.56</td>
</tr>
<tr>
<td>Spyware</td>
<td>0.14</td>
<td>0.38</td>
<td>0.24</td>
<td>0.32</td>
<td>0.17</td>
<td>0.35</td>
</tr>
<tr>
<td>Trojan</td>
<td>0.05</td>
<td>0.30</td>
<td>0.23</td>
<td>0.38</td>
<td>0.09</td>
<td>0.33</td>
</tr>
<tr>
<td>Virus</td>
<td>0.53</td>
<td>0.72</td>
<td>0.76</td>
<td>0.76</td>
<td>0.63</td>
<td>0.74</td>
</tr>
<tr>
<td>Worms</td>
<td>0.47</td>
<td>0.45</td>
<td>0.35</td>
<td>0.52</td>
<td>0.40</td>
<td>0.48</td>
</tr>
<tr>
<td>Accuracy</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0.41</td>
<td>0.55</td>
</tr>
<tr>
<td>Macro-average</td>
<td>0.41</td>
<td>0.56</td>
<td>0.46</td>
<td>0.58</td>
<td>0.41</td>
<td>0.57</td>
</tr>
<tr>
<td>Weighted-average</td>
<td>0.50</td>
<td>0.55</td>
<td>0.41</td>
<td>0.55</td>
<td>0.43</td>
<td>0.55</td>
</tr>
</tbody>
</table>

### Table 5  Classification results of the proposed GRU model

<table>
<thead>
<tr>
<th>Malware Families</th>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 1</th>
<th>Case 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adware</td>
<td>0.64</td>
<td>0.74</td>
<td>0.88</td>
<td>0.90</td>
<td>0.74</td>
<td>0.81</td>
</tr>
<tr>
<td>Backdoor</td>
<td>0.68</td>
<td>0.40</td>
<td>0.39</td>
<td>0.65</td>
<td>0.49</td>
<td>0.50</td>
</tr>
<tr>
<td>Downloader</td>
<td>0.59</td>
<td>0.64</td>
<td>0.71</td>
<td>0.74</td>
<td>0.64</td>
<td>0.69</td>
</tr>
<tr>
<td>Dropper</td>
<td>0.40</td>
<td>0.61</td>
<td>0.47</td>
<td>0.46</td>
<td>0.44</td>
<td>0.53</td>
</tr>
<tr>
<td>Spyware</td>
<td>0.08</td>
<td>0.36</td>
<td>0.24</td>
<td>0.38</td>
<td>0.12</td>
<td>0.37</td>
</tr>
<tr>
<td>Trojan</td>
<td>0.36</td>
<td>0.40</td>
<td>0.25</td>
<td>0.32</td>
<td>0.29</td>
<td>0.36</td>
</tr>
<tr>
<td>Virus</td>
<td>0.63</td>
<td>0.68</td>
<td>0.79</td>
<td>0.76</td>
<td>0.70</td>
<td>0.72</td>
</tr>
<tr>
<td>Worms</td>
<td>0.43</td>
<td>0.60</td>
<td>0.45</td>
<td>0.52</td>
<td>0.44</td>
<td>0.56</td>
</tr>
<tr>
<td>Accuracy</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0.48</td>
<td>0.55</td>
</tr>
<tr>
<td>Macro-average</td>
<td>0.48</td>
<td>0.56</td>
<td>0.52</td>
<td>0.59</td>
<td>0.48</td>
<td>0.57</td>
</tr>
<tr>
<td>Weighted-average</td>
<td>0.52</td>
<td>0.55</td>
<td>0.48</td>
<td>0.55</td>
<td>0.48</td>
<td>0.54</td>
</tr>
</tbody>
</table>

Tables 4 and 5. Notice that case 1 and case 2 represent the evaluation values of the original sequences and after preprocessing, respectively.

It is observable that the precision, recall and F1-score of the Spyware and Trojan families are quite lower than other malware families. One reason may be that the noise influence of these two families may cause low performance.
Also, the performance of the proposed LSTM and GRU model in case 2 are almost the same, and they are better than the performance of the model in case 1. Moreover, the performance of the proposed GRU model in case 1 is better than the proposed LSTM model in case 1. In other words, the API call-based LSTM model and GRU model work well on the malware classification.

6 Conclusions

Malware classification with time-series data, such as the API call sequences, is important to finance and the economy. RNN has achieved great success in prediction and classification with time-series data. In this paper, we proposed an RNN-based architecture to classify 8 malware families with API call sequences. More specifically, the LSTM cell and GRU cells were considered as the model for malware classification. In the experiment, a benchmark dataset was used to evaluate the classification accuracy, precision, recall, F1-score, Macro-average, and weighted-average. The analysis results illustrated that the proposed LSTM model and GRU model are effective in the API call-based malware classification.

In the future, we would like to compare the considered approaches in this paper with other classification methods. Another direction is to consider a hybrid neural network, such as the LSTM-CNN model, for the malware classification with API call sequences. Moreover, the attention mechanism is also needed to be considered to improve the accuracy of the classification.

References


Biographies

Chen Li received the B.S. degree in computer engineering from Qingdao College, Ocean University of China, in 2012, and the M.S. and D.Eng. degrees in engineering from Hiroshima University, Higashihiroshima, Japan, in 2016 and 2019, respectively. In 2019 and 2020, he was a Visiting Researcher with the Graduate School of Advanced Science and Engineering, Hiroshima University, Japan. Since 2021, he has been a post-doctoral researcher with the Department of Bioscience and Bioinformatics, Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology, Japan. His research interests include performance evaluation, data mining and deep learning.
Junjun Zheng received the B.S.E. degree in engineering from Fujian Normal University, Fuzhou, China, in 2010, and the M.S. and D.Eng. degrees in engineering from Hiroshima University, Higashihiroshima, Japan, in 2013 and 2016, respectively. In 2016 and 2017, he was a Visiting Researcher with the Department of Information Engineering, Graduate School of Engineering, Hiroshima University. Since 2018, he has been an Assistant Professor with the Department of Information Science and Engineering, Ritsumeikan University, Japan. His research interests include performance evaluation and dependable computing.

Zheng is a member of the Operations Research Society of Japan, the Reliability Engineering Association of Japan, the Institute of Electrical, Information and Communication Engineers, and the Institute of Electrical and Electronics Engineers.