World Wide Web (2020) 23:991–1010
https://doi.org/10.1007/s11280-019-00675-z

MALDC: a depth detection method for malware based on behavior chains

Hao Zhang[1,2] • Wenjun Zhang[1,2] • Zhihan Lv[3] • Arun Kumar Sangaiah[4] • Tao Huang[1,2] • Naveen Chilamkurti[5]

Received: 8 September 2018 / Revised: 18 February 2019 / Accepted: 25 February 2019 / Published online: 22 March 2019

Abstract

Malicious behavior detection is a key topic that has been a focus in the field of intrusion detection. Current intrusion detection systems are primarily based on single-point monitoring and detection and cannot detect attack modes with a hidden attack frequency. The idea presented in this paper is the incorporation of API call sequence software into the analysis and the construction of behavior chains to express the behavior patterns in software. This paper introduces related definitions of behavioral points and behaviors and proposes a depth-detection method for malware based on behavior chains (MALDC). The method monitors behavior points based on API calls and then uses the calling sequence of those behavior points at runtime to construct a behavior chain. Finally, we use depth detection method based on long short-term memory(LSTM) to detect malicious behavior from the behavior chains. To verify the performance of the proposed model, we conducted a large experiment on 54,324 malware and 53,361 benign samples collected from Windows systems and used those samples to train and test the model. Comparative verification by using various classifiers showed that the behavior points extracted based on the above method and the constructed behavior chains can be used to recognize malicious behavior at a high recognition rate. The method achieved an accuracy of 98.64% with a false positive rate of less than 2% in the best case, which is a satisfactory recognition rate for detecting malicious software behavior.

Keywords Malicious behavior · API call sequence · Behavior chain · LSTM

This article belongs to the Topical Collection: *Special Issue on Security and Privacy in Network Computing*
Guest Editors: Xiaohong Jiang, Yongzhi Wang, Tarik Taleb, and Hua Wang

✉ Zhihan Lv
lvzhihan@gmail.com

✉ Arun Kumar Sangaiah
arunkumarsangaiah@gmail.com

Extended author information available on the last page of the article

1 Introduction

Current mainstream and practical methods for antivirus and malicious behavior detection determine whether behavior is malicious based on virus signature databases. However, this approach has limitations, and the employed virus signature databases are derived from manual analysis and extraction, leaving open the possibility for the development of future malicious behaviors. Existing solutions to this problem rely on human experts to define features and often miss many vulnerabilities (i.e., incurring a high false negative rate). Moreover, because network technology is updated continuously, malicious behavior variants are also increasing rapidly; consequently, feature libraries must be continuously updated, resulting in low detection accuracy and high labor costs. Furthermore, the only malicious behavior these approaches can detect is behavior that exists in the feature database, whereas they are unable to detect new malicious behavior variants. In this study, we describe an approach that uses deep learning-based malware detection to relieve human experts from the tedious and subjective task of manually defining features and we specifically aimed to lower the false negative rate(FNR) and the false positive rate(FPR). By mining relationships in the data, the error rate of human experts is reduced, and the accuracy of the detection models is improved.

The behavior-based malicious behavior detection methods proposed by researchers can mainly be divided into static behavior detection and dynamic analysis detection. Among these, early malware detection efforts employed static analyses [10, 16], and static behavior detection has been the main technique used in code analysis to acquire information concerning software behavior; however these approaches are unable to detect files that adopt techniques such as packing or reverse decompression [15]. Dynamic behavior analysis refers to analysis that occurs while the software is actually running, capturing its behavior for analysis. This approach can effectively address problems that cannot be solved by static detection. Many experts in the field currently use dynamic behavior techniques to detect malicious behavior [3, 22]. Classification techniques are generally used to classify unknown malware into known types [25]. By considering the extracted malicious code behavior as a detection feature, this method avoids problems resulting from code obfuscation because it focuses on the actual behavior of malicious code. However, this behavior-based feature is still limited to grammatical features and is easily confused by equivalent behavior substitution. Interference such as that caused by the technique proposed by Sekar et al. [23], which involves a confusing attack that injects garbage behavior while allowing the attack to simulate a normal behavioral sequence, can bypass detection systems. Other researchers have used various APIs and other dynamic fields to detect certain types of malware [11] or use API function calls, API function parameters, and a collection of paired features to calculate detection sets based on the concept of information entropy. Identifying malicious information by distinguishing the differences in information gain between benign and malicious behavior was proposed in [27]. Although the above methods can detect most malicious behaviors, some of these methods are complex, or their analysis processes are extremely complex; others rely on a specific feature library for analysis, or the applied algorithm considers only problems with large probabilities but does not consider the occurrence of hidden events, making them impractical in real-world situations.

The rapid development of neural networks and deep learning has led many researchers to apply these models to malware detection because they are adept at recognizing complex and abstract patterns from large numbers of malware samples. There is always a certain probability of detecting malicious behavior—even in instances with frequent mutations [21, 28, 34, 35]; considering this aspect, deep learning offers some advantages. Nevertheless, deep learning itself is vulnerable to what is known as "adversarial samples" [2, 26], which means that these systems can be easily deceived by dangerous manipulation [8]. Recent research has already demonstrated that a malware author can leverage feature amplitude inequilibrium to bypass malware detectors powered by deep neural networks [1, 9]. To address this problem, this paper monitors API call sequences at program runtime and proposes a malware depth detection method based on behavior chains. Current operating systems provide a large number of APIs; essentially all programs must call the APIs that correspond to specific tasks. In this paper, we analyze the API call sequence and study how to perform feature extraction and implement detection methods for malicious behavior. Next, a method for describing behavior that uses the behavioral associations based on the program's API call sequences is established to construct corresponding behavior chains. Then, the behavior chains are used to train a long short-term memory (LSTM) model. Finally, the trained model can be used to detect malicious behavior. The presented experiments demonstrate that this method corresponds to improved detection ability.

The remainder of this paper is organized as follows. In Section 2, we discuss the relevant background information concerning malicious behavior detection and the LSTM model. Section 3 describes the proposed work and the methodology in detail. Section 4 presents the experiments and an analysis of the results, and Section 5 concludes the paper and proposes directions for future work.

2 Background

The main contribution of this paper is a method for constructing behavior chains based on malicious behavior and their use in a detection method. To construct a behavior chain, the first step is to analyze the malicious behavior in a running process. Based on this analysis, the behavior and its descriptive characteristics are extracted, and a corresponding behavior chain is constructed that can be used for malicious behavior detection based on the LSTM model.

2.1 Literature survey

Behavior-based malicious behavior detection can be generally divided into two types of approaches: static analysis detection and dynamic analysis detection [31]. Static analysis involves disassembling malicious code using disassembly tools such as IDA Pro or W32Dasm. This approach does not require executing the program; instead, it obtains malicious behavior information is obtained solely through code analysis. For example, Wang [32] proposed a method for comparing code in a malicious behavior file with related data. First, this approach determines the code block; then, it compares the data block. However, this method is only applicable to malicious behavior code that has not undergone large changes. The authors of [7] proposed a system for detecting malicious Android applications by statically analyzing application behavior. This system extracts static features from secure applications to detect malicious behavior; however, this approach cannot protect devices from transient attacks or modified malware. Vida Ghanaei [12] presented a static block analysis of the basic block frequencies of malware samples to classify malware families; however, this study used only static analysis and did not consider the actual operation of the malware. Dullien [4] analyzed the execution semantics of malicious behavior programs based on the control flow of the program. This method improved the accuracy of malicious behavior judgment, but could not deal with obfuscated code because it only compared the basic program blocks. Because static analysis relies on disassembly techniques, some malicious behavior code protects itself through techniques such as compression, encryption [29], and so on. Moreover, because statically analyzed code does not accurately represent real code that implements real functionality, it is extremely difficult to judge the true results. Therefore, dynamic analysis methods have emerged.

Dynamic analysis methods generally monitor malicious behavior through system monitoring or debugging tools. The analysis is performed while the program is running to judge whether its actions represent malicious behavior. This method circumvents interference by techniques such as packing malicious behavior code or confusion; therefore, it is more suitable for environments in which malicious behaviors occur. Most recent studies have focused on dynamic behavior analysis. For example, Wang Rui [33] proposed a semantically-based approach to extract malware behavioral signatures and perform detection. This approach extracts critical malware behaviors and the dependencies among these behaviors and then acquires anti-interference malware behavior signatures using an anti-obfuscation engine to identify semantically irrelevant and semantically equivalent behaviors, which improves the ability to recognize malicious code. Compared with any pattern-based approach for detection, the code similarity-based approach is advantageous in that a single code instance can detect the same malicious behavior in the target program. However, it can only detect malicious behavior that identical or almostidentical code clones [24]. To achieve a higher effectiveness of malicious behavior detection, experts need to define features to automatically select the correct code similarity algorithms for different kinds of malicious behavior [19]. Zhen Li [20] studied the used of deep learning-based vulnerability detection to relieve human experts from the tedious and subjective task of manually defining features. Yujie Fan [6] proposed an effective sequence-mining algorithm to discover malicious sequence patterns. They performed malicious detection using an ANN classifier, achieving good results. Yanfang Ye [14] proposed a HinDriod system architecture that first generates smali code through decompilation and then analyzes the resulting smali code. A complete Android API call list, representing two entity types and four types of relationship characteristics can be extracted in this manner. Then, the relationships among the extracted API calls can be further analyzed. The heterogeneous information network HIN is used to solve the complex relationships, and relationships between applications are found through the meta-path method. This approach capitalizes on the idea of using multicore learning to build a classification network that makes binary “safe” or “malicious” judgments concerning applications. In [13] a new dynamic analysis method called component traversal was proposed. This method also automatically decompiles a given Android application as much as possible and then analyzes its code to determine whether it is a malicious program.

However, the above studies did not consider the call sequences of the application, nor was the overall behavior trajectory of the software operation analyzed, thus, the accuracy of the test results is not high. Software vulnerabilities are detected in [20], although the method does not detect whether software with vulnerabilities exhibits malicious behavior, resulting in some hidden attacks that can escape detection. This paper proposes a new method and using a deep learning model to detect malicious behavior with the aim of achieving a lower false negative rate, and ultimately obtaining a higher malware detection efficiency.

2.2 Long short-term memory (LSTM)

On one hand, because malicious behavior code mutations occur quickly and the variants have multiple styles, many malicious behavior detection software implementations cannot be altered in time, resulting in huge losses. On the other hand, because many current malicious attacks have good latent characteristics, it is imperative to find a way to quickly and accurately detect malicious attacks.

Traditional neural networks do not consider chronological factors and cannot remember previous content. To address this problem, the recurrent neural network (RNN) was developed. The logical architecture of an RNN is depicted in Figure 1. The hidden state h_t is obtained from the input X_t at time t and the from the output h_{t-1} at the previous moment. The latter is used to calculate the model loss of the current layer and to calculate the h_{t+1} of the next layer. However, because an RNN suffers from gradient decay, the hidden structure of its sequence index position t was improved to avoid the gradient disappearance problem. Thus, a special RNN model called an LSTM can learn long-term dependency information. An LSTM is somewhat different from the typical neural network module A of RNNs. In an RNN, the repeated neural network module A has a very simple structure, such as a tanh layer:

$$\mathrm{h}_t = \tanh\left(W_h\left[h_{(t-1)}, X_t\right] + b_h\right).$$

In contrast, an LSTM has four neural network layers that interact in a special way, as shown in Figure 2. An LSTM has the ability to delete or add information to memory through a specially designed structure called a "gate." The gate is actually the place to select the operational data information; it contains a sigmoid neural network layer and a multiplication operation. The sigmoid layer changes the input through the sigmoid function and outputs a value between 0 and 1, describing how much input can pass through that network part. A "0" indicates that no data are allowed to pass, while a "1" indicates that all data are allowed to pass. The gate structure of an LSTM at each sequence index position t generally includes a forgetting gate, an input gate and an output gate. The output f_t of the sigmoid is a value in the range [0, 1].

The forget gate decides what information to discard or retain from the memory of the previous moment:

$$f_t = \sigma\left(W_f\left[h_{(t-1)}, X_t\right] + b_f\right).$$

The input gate determines the information that should be saved:

$$i_t = \sigma\left(W_i\left[h_{(t-1)}, X_t\right] + b_i\right).$$

A tanh layer creates a new candidate value vector:

$$C_t = \tanh\left(W_c\left[h_{(t-1)}, X_t\right] + b_c\right),$$

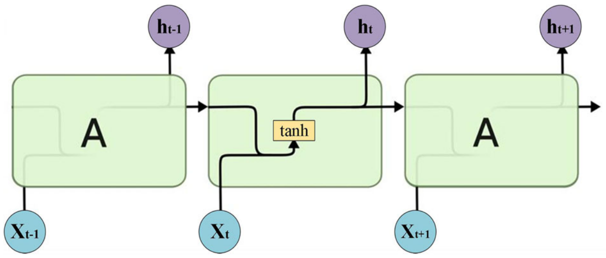


Figure 1 Recurrent neural networks structure

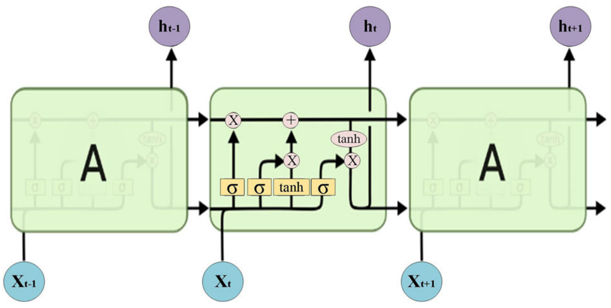


Figure 2 Neural network LSTM structure

which will be added to the status. The input gate determines the updating of the candidate value vector, and the forget gate determines whether the information should be retained or discarded to construct the final memory:

$$C_t = f_t{*}C_{t-1} + i_t{*}C_t.$$

Finally, the output gate determines which part of the memory is ultimately output:

$$o_t = \sigma\left(W_o\left[h_{(t-1)}, X_t\right] + b_o\right).$$

Then, the passed data flows into the tanh layer for processing. The output is a value between [−1, 1], which is multiplied by the output gate. Finally, the output is determined by

$$\mathrm{h}_t = o_t{*}\tanh(C_t).$$

All the above methods from the literature survey achieved quite good experimental results, However, some used only static analysis, which has certain limitations, and some adopted both static and dynamic analysis methods, but focused on the process of dynamic analysis. Most existing research pertained to analysis and malware detection for Android applications; these methods first perform decompilation, then analyze the API calls in the decompiled code, and then analyze the employed APIs to determine whether the application is malicious. In contrast, this paper analyzes and detects malware applications under a Windows environment. The analysis process adopted here is simpler and more convenient than the processes in the research described above. Here, we avoid the decompilation process; instead, we extract the corresponding API behavior points from the running process of the monitored program and build a behavior chain to generate the dataset. Then, using the deep learning LSTM model to train detection models, higher accuracy can be obtained.

3 Construction of the MALDC model

Malicious behavior detection is actually a binary classification problem. We extract the behavior analysis from the collected data and divide it into two groups: malicious behaviors and benign behaviors. We denote the sequence of behaviors that we collect by $\mathrm{X} = \{X_1, X_2, \cdots, X_n\}$, where X_i represents one behavior in multiple behavior sets and n represents the number of behavior sequences. $\mathrm{Y} = \{Y_1, Y_2\}$ represents the behavior categories, where Y_1 and Y_2 represent malicious behavior and benign behavior, respectively. Therefore, our goal is to find a suitable mapping relationship $\mathrm{f}(X_i) \rightarrow Y_j$, where $\mathrm{i} \in (1, n)$,$\mathrm{j} = \{1, 2\}$, and f is the mapping function of the classification model. We collected the sample data and trained it using an appropriate LSTM model and finally judged whether a sample was malicious based on the trained model results. Figure 3 shows the processing flow of this paper.

3.1 Related definition

3.1.1 Behavior point

All programs are executed to achieve a certain goal. Each operational step constitutes a "behavior point" in the process of reaching that goal. From the perspective of the operating system, the behavior points are calls by the program to certain API functions, which can be represented by the triplet, $B = (R, P, Pro)$, where R is the return value of the behavior point call, P denotes the input and output of the behavior point, and Pro is an attribute that represents the purpose of the behavior point, which can be a file behavior point, a registry behavior point, a network behavior point, and so on.

The elements $P\{PI(P_1 : V_1; P_2 : V_2); PO(P_3 : V_3; P_4 : V_4)\}$in the triplet contain multiple parameters, where PI represents the input of the behavior point, PO represents the output of the behavior point, and $\{P_1, P_2\}$ and $\{V_1, V_2\}$ represent the input parameters and parameter values of the behavior point, respectively, while $\{P_3, P_4\}$ and $\{V_3, V_4\}$ respectively represent the output parameters and parameter values of the behavior point.

3.1.2 Behavior

During a running process, sequences occur between each behavior point. The time and position of each behavior point can be different, the meaning can be different, and the

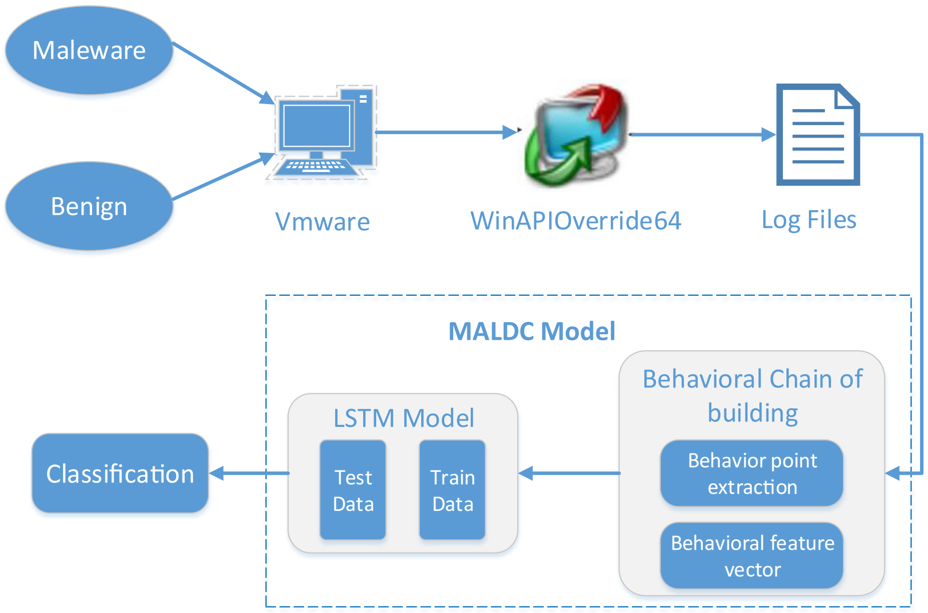


Figure 3 Overview of the system

final goal can be different. Therefore, we call a collection of one or more behavior points a "behavior". A behavior is represented by an N-tuple A = (B_1, B_2, ⋯, B_n), where B_1 indicates the behavior point of the program running at time t, B_2 indicates the behavior point of the program running at time t + 1 and B_n indicates the behavior point of the program running at time t + n. Overall, from time t to time t + n, the API call constitutes a behavior.

3.1.3 Description of association behavior

A behavioral relationship involves the relationship between behavior points and acts and refers primarily to the timing relationship. For example, the behavior point B_1 indicates the first API call when the program is running, and the behavior point B_2 indicates the second API call. The parameters and values of B_2 are inherited from the previous behavior point B_1, that is, the output of B_1 forms the input for behavior point B_2. Therefore, at the output of behavior point B_1, B_2 depends on the behavior point B_1. Consequently, the relationship between the two behavior points can be expressed as: $B_1 \rightarrow B_2$.

3.2 Construction of association behavior

Before a behavior chain can be constructed, it is first necessary to determine the corresponding behavior point features from the collected data, a process called "behavior feature extraction". The collected data will include much useless information that constitutes interference. To remove the interference, the data must be preprocessed and the required behavior points extracted. Then, these behavior points are combined into behaviors according to the call sequence. The behavior chain is composed of these behavior combinations.

3.2.1 Feature extraction

During the course of the experiment, the API call format is relatively standardized; the API names, parameters and parameter values appear in the regular monitoring log and can be expressed as follows:

$$\text{File}\{B_1(\text{P}_1 : V_1; \text{P}_2 : V_2; \text{P}_3 : V_3; \cdots), \text{B}_2(\text{P}_3 : V_3; \text{P}_4 : V_4; \cdots), \cdots\}$$

We first need to extract the API, the behavior point, the corresponding parameters and their parameter values from the collected log files. During the extraction process, because the irrelevant interference data appear in each log file, it is difficult to extract only the desired parameters and parameter values. Accomplishing this task requires the use of string processing techniques from text analysis. The raw data holds three different types of data, API names, API names and parameters, and API names and parameters along with parameter values. This article mainly addresses the behavior points and considers only API functions themselves; it ignores their parameters and parameter values. A behavior point may appear multiple times in a log file, but number of occurrences is not considered, only the order in which the APIs are called during program execution and the behavior points with which it has a sequential relationship.

The purpose of behavior extraction is to combine the behavior points obtained through monitoring into behaviors, such as the monitored behavior points B_1, B_2, and B_3, into obtain behaviors, such as $A = (B_1, B_2, B_3)$. When integrating behavior points, only the same types can be combined; the type of a behavior point depends on behavior point classification. Therefore, the extracted behavior points must first be mapped to their corresponding behavior categories . Then, we can address the problem of merging the behavior points in the same category. In this paper, the corresponding behavior points B and A are extracted by Algorithm 1 (shown in Figure 4). Based on this process, we can see that behavior extraction actually involves data extraction. For example, during read-file behavior, the data in the file must be read into memory; therefore, the application will call APIs such as CreateFile, ReadFile, OpenFile, WriteFile, and so on.

3.2.2 Construction of the behavior chain

Program execution mainly calls various APIs to achieve a certain purpose, that is, the relationship between the behaviors is represented by the transfer relationship between the behavior points; therefore, the relationships can be established through the transmission relationships between the behavior points.

Assume that the following sequences are extracted in three log files:

$$\begin{aligned}
&\text{File1: } \{A_1(B_1, B_2), A_2(B_1, B_3, B_5), A_5(B_3, B_4, B_6), \cdots\}\\
&\text{File2: } \{A_1(B_1, B_3, B_4), A_3(B_2, B_4, B_5), \cdots\}\\
&\text{File3: } \{A_2(B_2, B_3), A_3(B_1, B_4, B_6), A_4(B_3, B_4, B_5), \cdots\}
\end{aligned}$$

In these three files, based on the feature extraction operations mentioned above, we can extract behavior points and behaviors from the original files to construct a behavior chain that includes the following three behavior chains:

$$\begin{aligned}
&\{A_1(B_1, B_2) \to A_2(B_1, B_3, B_5) \to A_5(B_3, B_4, B_6) \to \cdots\}\\
&\{A_1(B_1, B_3, B_4) \to A_3(B_2, B_4, B_5) \to \cdots\}\\
&\{A_2(B_2, B_3) \to A_3(B_1, B_4, B_6) \to A_4(B_3, B_4, B_5) \to \cdots\}
\end{aligned}$$

Note that the three behavior chains constructed above do not consider the parameters and parameter values used when calling the API during program execution; only the behavior point

Algorithm 1: API extraction algorithm

```
Open Systemlog_file
Declare v
Input all log_files to v
n=size of v
close file
file=Open(API_file.txt).readlines()
for i in file:
   begin
     if API belong to ".dll"
        if the API is equal to the previous API
             continue
        else
             Save the API to API_file.txt
   end
Close file
Output API_file.txt
```

Figure 4 Feature extraction

APIs are considered: the parameters and their parameter values are ignored. The final construction result is shown in Figure 5. Each behavior chain consists of different behaviors, and each behavior contains different behavior points.

Each time the program runs, there will be a series of calls to system APIs until the program reaches its desired purpose. Therefore, we construct the corresponding behavior chain by extracting the API function calls.. We construct the behavior chain with temporal characteristics to express the intrusive malicious behavior process. When a certain program is known to call an API, a certain behavior is triggered; then the probability of the next behavior being malicious or benign is determined through the behavior chain, which can be used to prepare for interception in advance. In this study, we use the dataset collected above and extract the behavior chain constructed by the features. Then, we combine the behavior chain with the deep learning LSTM model to predict whether a program exhibits malicious behavior.

3.3 MALDC model construction based on behavior chains

The previous section described extracting the required feature data from the log file and constructing a behavior chain temporal characteristics. Here, we construct trained models based on these behavior chains and the deep learning LSTM. Because the anomalous behavior of latent unknown attacks is quite subtle, attackers often try to obscure their attack behaviors. Usually, a single behavior appears normal, but when behaviors are related, it is possible to combine them into abnormal behavior. Therefore, this paper analyzes abnormal behaviors from the perspective of system API calls. Normal or malicious programs will both make API calls. Therefore, we establish the API behavior chain and then use the LSTM recursive neural network model as an effective recognition method. The overall architecture of the algorithm is shown in Figure 6. The processed behavior chains are input into the LSTM individually for detection and recognition, and the hidden states obtained at each moment are aggregated. Averaged pooling is then used to reduce the dimensions to obtain a converted data expression and finally, the model makes a classification from the converted data using classification algorithms.

As a simple behavior example, we introduce the cooccurrence feature of behavior points in behavior actions into the LSTM network design and use it as a parameter learning constraint for the network to optimize the recognition performance. The purpose of malicious behavior is often related to some specific set of behavior points, and the interactions of behavior points in this set are closely related. To judge whether a program is a malicious Trojan virus, behavior points such as “open port”, “receive remote host connection”, “receive remote host

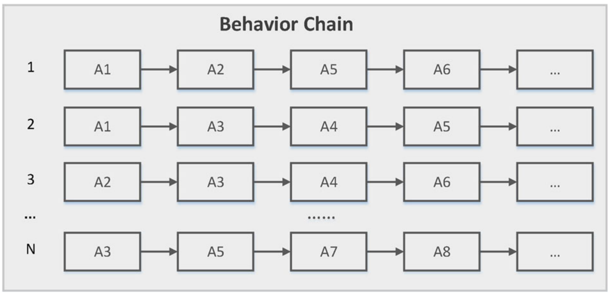


Figure 5 Behavior chain construction

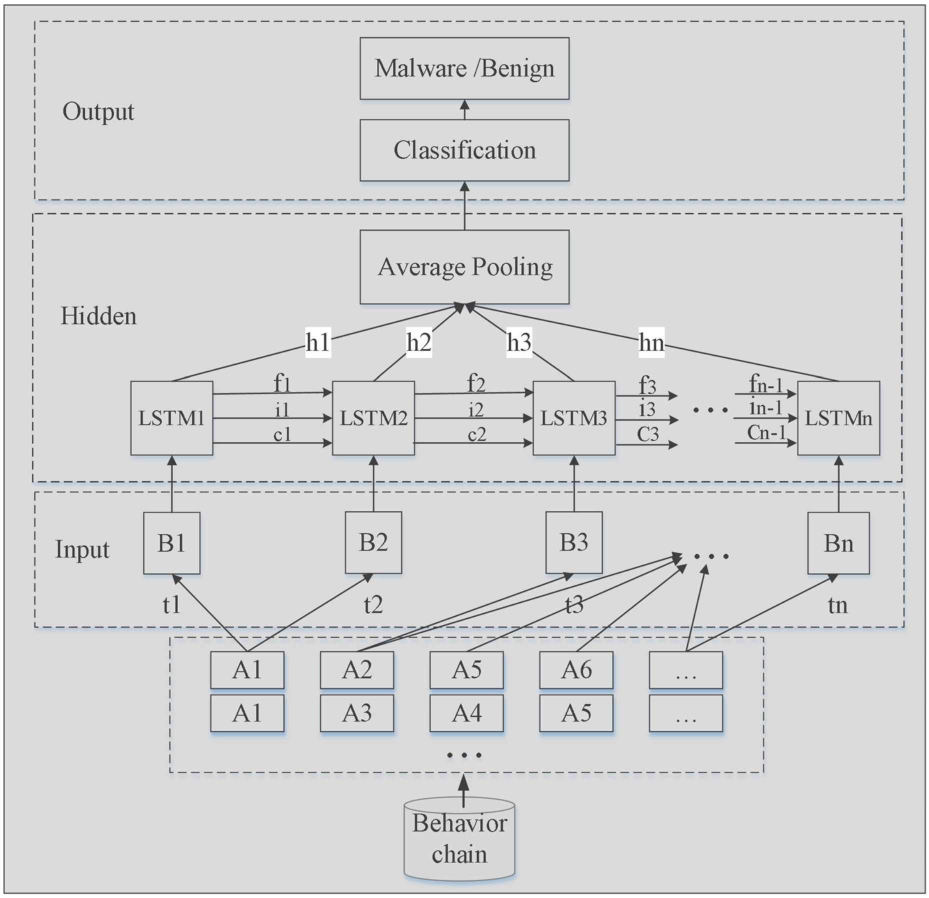


Figure 6 MALDC model based on behavior chains

information", "send information to remote host", "start program", and "end process", and behavior series such as "reading files" and "screening" are very important. Different malicious programs feature different combinations of such closely related behavior points, but the order in which they call APIs during execution varies. Generally, malicious programs will call APIs such as "CreateFile", "ReadFile", and "WriteFile", forming a set of nodes with discriminative properties. We have designated these behavioral characteristics that can be discriminated into cooccurrences.

In the model training phase, we introduce a constraint on the weight of the behavior point and the neuron in the objective function; thus, the same group of neurons has a greater weight connection to a subset containing certain behavior points along with other behavior points. There are smaller weight connections that reflect the cooccurrences of behavior points. As shown in Figure 7, an LSTM layer is composed of multiple LSTM neurons. These neurons are divided into M groups. Each neuron in the same group has a greater connection weight with certain behavior points (behavior points that are associated with a certain class or with certain types of malicious behavior constitute a subset of behavior points) but have smaller connection weights with other behavior points. Different groups of neurons have different sensitivities to different behaviors, and the subsets of behavior points in different groups of neurons corresponding to larger connection weights are also different.

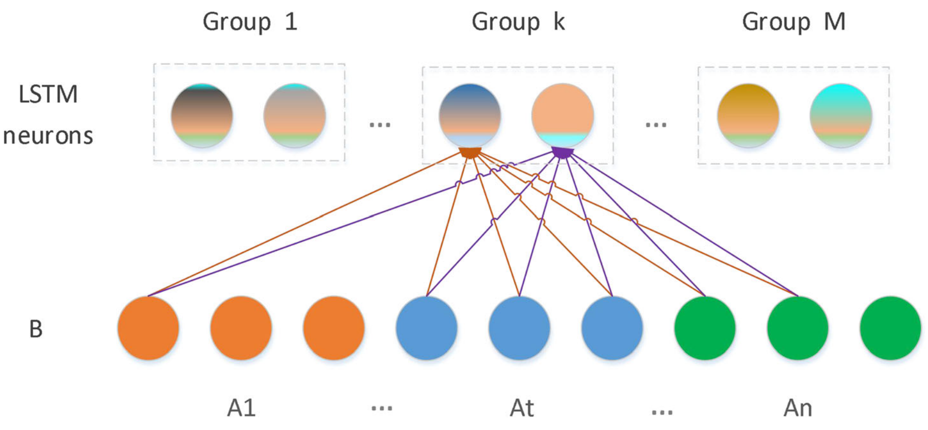


Figure 7 Cooccurrences of behavior points

Because various latent unknown attacks exist, the LSTM model is used for analysis and processing. Using the preprocessed behavior chain data with temporal characteristics as the training set, a mapping relationship between the input and output is found using the LSTM. The MALDC model training algorithm is shown in Figure 8. In the behavior chain, the calling order of the APIs and the context of the API calls before and after an API call are highly important factors.

4 Experiments and analysis

This paper selects a certain number of benign and malware samples to generate sample data sets. The benign software is selected from the Windows system and consists of widely used and well-known software. The malware samples contain viruses, Trojans, worms, etc.; the main source for the malicious Windows system programs was acquired from https://virusshare.com/. A selection of 578 benign and 950 malware samples is used as the dataset in this study,

Algorithm 2

Input*: *Behavior chain data
Output*: MALDC *model

```
1:   Initialization: the weight of each network layer
2:   While more input data exists do:
3:     Forward propagation, calculate output y
4:     Calculate the current total error value
5:     Calculate the current hidden layer error gradient
6:     Calculate the current error gradient of hidden cell states
7:     Calculate the weight updates for the forget gate、input gate and output gate
8:     Perform weight updates
9:     if maximum number of iterations do
10:      Save weights
11:      end
12:    end
13:    Save weights
14:    end
```

Figure 8 MALDC model-training algorithm

Table 1 VM configuration

Operating system	Windows 7 professional
CPU	Intel(R) Xeon(R) CPU E5–2620 v4 @ 2.10 GHz
Hard disk	60 GB
Memory	4 GB
VMware	VMware Workstation 12 PRO
Development language and tools	Python2, Pycharm

and Table 2 shows the different categories of malware samples and the number and percentage of samples in each category(raw data). Because the dataset provided by the site is relatively small, we expand the dataset according to each malicious type in the malware dataset to ensure that all types of malicious features remain intact. Then, we randomly sample 60% of the data for each type from the malicious sample set and combine these same types of data to expand the dataset. Similarly, all the other malicious types and 40% of the benign sample data are expanded by this method, and a total of 54,324 malware and 53,361 benign samples are acquired. The experiment was executed in a virtual machine, whose configuration is shown in Table 1:

The log data were obtained by running the sample program in the virtual machine and monitoring it using the WinAPIOverride64 tool [27, 30]. Each sample is executed until all its processes terminate or a 90-s timeout period elapses. This timeout value was selected because 90 s is generally to enough time for most malware programs to execute their immediate payloads. Generally, the monitoring tool ended the process after the monitoring timeout was reached, generated a corresponding monitoring log, and saved the log. The system was then reverted to its initial state, and the next program was run and monitored in a clean environment until all the data were collected (Table 2).

The Windows operating system contains a large number of API calls in the form of dynamic link libraries (DLLs). If these libraries are monitored, the amount of data collected would be very large. Moreover, some APIs have no relevance to this research and would simply be noise during the analysis. By referring to DLLs selected by researchers worldwide [5, 17, 27], this paper identified six important dynamic link libraries to be monitored; these DLLs include advapi32.dll, rasapi32.dll, kernel32.dll, ntdll.dll, shell32.dll, and user32.dll. Each DLL's functions are demonstrated in Table 3. The APIs in these libraries can create, delete or open registry keys and set or save values to them; create a process; directory or file; search, delete or move a file; create a network connection; etc.

An API call situation of the malicious program from the log of the detection tool WinAPIOverride64 is shown in Figure 9. The Call column specifies the API call, including the name, parameters, and parameter values. Other columns specify the process ID, thread ID, address, registration value, and so on.

Table 2 The different type of malware samples(raw data)

Type	Number of files	Sample ratio	Type	Number of files	Sample ratio
virus	50	3.3	hacktool	180	11.7
trojan	100	6.5	P2Pwarm	140	9.2
backdoor	200	13.1	exploit	80	5.2
constructor	200	13.1	Benign	578	37.8

Table 3 A summary of the selected DLLS

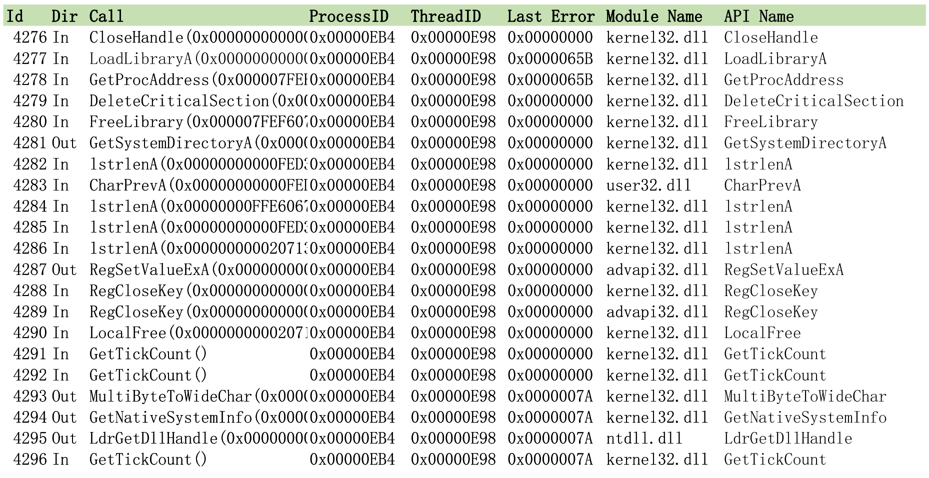
DLL name	Description
advapi32.dll	Advanced API services library that supports numerous APIs, including many security and registry calls
rasapi32.dll	The Remote Access API (RAS) used by Windows applications to control modem connections
kernel32.dll	Contains hundreds of functions for managing memory and various processes
ntdll.dll	NT-Layer DLL that controls NT system functions.
shell32.dll	A library that contains Windows Shell API functions used when opening web pages and files
user32.dll	Contains numerous user interface functions and is involved in the creation of application windows and their interactions.

The collected log files are preprocessed by Algorithm 1, which extracts the features from the logs. Then, the behavior chain is constructed according to the behavior chain construction method described earlier. The behavior chain is coded, and finally, the LSTM network is applied to train the model to determine whether the analysis is malicious.

This paper uses the word2vec continuous bag-of-words (CBOW) to encode and train the collected sample data, finally generating a 50-dimensional feature vector representation for each API. Each behavior point in the behavior chain has a strong contextual relationship. If the traditional word bag model (BOW) were used, the relationship between each behavior point and other behavior points could not be considered, and the context of the behavior point would be ignored; therefore, the document encoding version of the word vector is used to encode the behavior chain, and the vector representation of each API can be obtained by training word2vec. During the process of training the word vector, we set the word vector to 50 dimensions. The number of iterations was set to 5 by default, and the size of the training window was 3.

The preconstructed behavior chain is input into the LSTM network for training. The LSTM consists of a three-layer network: an input layer with 256 input units, a hidden layer with 128 LSTM units, and an output layer. The sigmoid activation function is used in the output layer to normalize the value as the output of the neural network.

Under the same conditions as the experimental sample, we use the control variable method to adjust the optimal parameters. First, we fixed the number of units in each LSTM layer and



```
Id   Dir Call                          ProcessID   ThreadID   Last Error Module Name  API Name
4276 In  CloseHandle(0x00000000000(0x00000EB4  0x00000E98 0x00000000 kernel32.dll CloseHandle
4277 In  LoadLibraryA(0x0000000000(0x00000EB4  0x00000E98 0x0000065B kernel32.dll LoadLibraryA
4278 In  GetProcAddress(0x000007FEI0x00000EB4  0x00000E98 0x0000065B kernel32.dll GetProcAddress
4279 In  DeleteCriticalSection(0x0(0x00000EB4  0x00000E98 0x00000000 kernel32.dll DeleteCriticalSection
4280 In  FreeLibrary(0x000007FEF6070x00000EB4  0x00000E98 0x00000000 kernel32.dll FreeLibrary
4281 Out GetSystemDirectoryA(0x000(0x00000EB4  0x00000E98 0x00000000 kernel32.dll GetSystemDirectoryA
4282 In  lstrlenA(0x00000000000FED:0x00000EB4  0x00000E98 0x00000000 kernel32.dll lstrlenA
4283 In  CharPrevA(0x00000000000FEI0x00000EB4  0x00000E98 0x00000000 user32.dll   CharPrevA
4284 In  lstrlenA(0x00000000FFE6060x00000EB4  0x00000E98 0x00000000 kernel32.dll lstrlenA
4285 In  lstrlenA(0x00000000000FED:0x00000EB4  0x00000E98 0x00000000 kernel32.dll lstrlenA
4286 In  lstrlenA(0x00000000002071:0x00000EB4  0x00000E98 0x00000000 kernel32.dll lstrlenA
4287 Out RegSetValueExA(0x00000000(0x00000EB4  0x00000E98 0x00000000 advapi32.dll RegSetValueExA
4288 In  RegCloseKey(0x00000000000(0x00000EB4  0x00000E98 0x00000000 kernel32.dll RegCloseKey
4289 In  RegCloseKey(0x00000000000(0x00000EB4  0x00000E98 0x00000000 advapi32.dll RegCloseKey
4290 In  LocalFree(0x0000000000207]0x00000EB4  0x00000E98 0x00000000 kernel32.dll LocalFree
4291 In  GetTickCount()               0x00000EB4  0x00000E98 0x00000000 kernel32.dll GetTickCount
4292 In  GetTickCount()               0x00000EB4  0x00000E98 0x00000000 kernel32.dll GetTickCount
4293 Out MultiByteToWideChar(0x000(0x00000EB4  0x00000E98 0x0000007A kernel32.dll MultiByteToWideChar
4294 Out GetNativeSystemInfo(0x000(0x00000EB4  0x00000E98 0x0000007A kernel32.dll GetNativeSystemInfo
4295 Out LdrGetDllHandle(0x0000000(0x00000EB4  0x00000E98 0x0000007A ntdll.dll    LdrGetDllHandle
4296 In  GetTickCount()               0x00000EB4  0x00000E98 0x0000007A kernel32.dll GetTickCount
```

Figure 9 APIOVERRIDE64 monitoring data

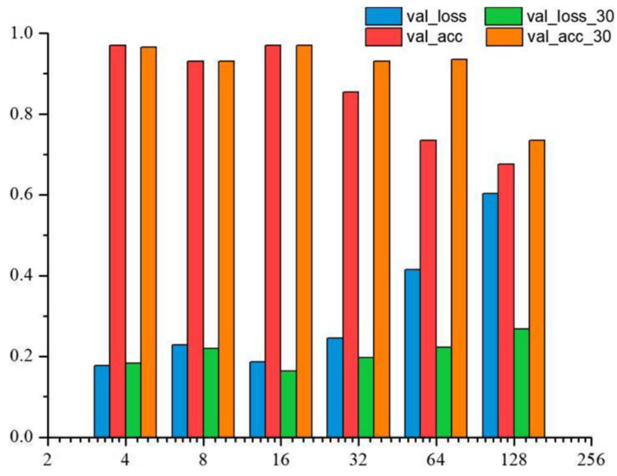


Figure 10 Comparison of different batch-sizes

then changed the number of samples used for training and for the batch_size and analyzed the results. The number of training iterations was set to 30, and the batch_size was set to 4, 8, 16, 32, 64, and 128. The final accuracies levels are shown in Figure 10 and Table 4.

Where val_loss and val_acc represent the loss value and accuracy on the test set with 10 iterations, and val_loss_30 and val_acc_30 represent the loss value and accuracy on the test data with 30 iterations. As Figure 10 shows, the loss value and accuracy obtained by the two different iteration values are different. In general, when the batch_size has a smaller value, the loss value and accuracy obtained are relatively stable, but when the batch_size is relatively large, the loss value and accuracy are unstable, which is highly related to the amount of data. Because the data set used in this paper is relatively small, considering the overall performance, a batch_size of 32, at which the loss value and accuracy are relatively stable, was selected as the experimental condition.

Next, with the batch_size fixed at 32, the number of iterations set to 30, and other conditions left unchanged, the activation function is changed, and the results are analyzed. In Figure 11, val_loss represents the loss value on the test set, and val_acc is the accuracy rate on the test set. In this experiment, a total of six different activation functions were tested. According to the results, the sigmoid function had the highest accuracy and the smallest loss value under the given conditions. Therefore, this study used the sigmoid function was used as the activation function.

The above experiments determined the parameters that need to be adjusted, and subsequent model training was based on these established conditions. Among the sample data collected, 80% were selected as a training set, and 20% were used as the test set. These are input into the

Table 4 Comparison of different batch-sizes

	val_loss	val_acc	val_loss_30	val_acc_30
4	0.0819	0.9346	0.0843	0.9412
8	0.0699	0.9542	0.0729	0.9423
16	0.0999	0.9412	0.1094	0.9019
32	0.0608	0.9739	0.0586	0.9842
64	0.0856	0.9368	0.0721	0.9477
128	0.2445	0.8542	0.0800	0.8745

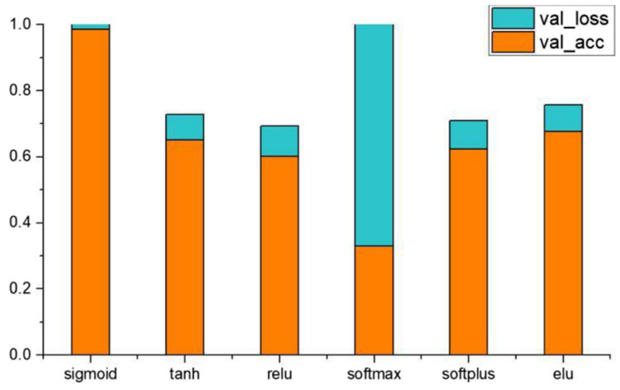


Figure 11 Comparison of different activation functions

LSTM model. The final experimental results are shown in Figure 12, where loss represents the loss value on the training data, acc represents the accuracy on the training data, val_loss is the loss value on the test data, and val_acc is the accuracy on the test data. The abscissa shows the number of iterations, and the ordinate shows percentages.

As Figure 12 shows, the accuracy on the training set stabilizes when the number of iterations reaches 10. When the number of iterations reaches 15 times, the rate of loss reduction is also very slow. The accuracy and loss values in Figure 12 and Table 4 show that the accuracy rate is 98.64%. In the following experiments, we compare the detection results of using traditional processing algorithms and various deep learning models. In addition, the experimental results of each model highlight the advantages of the deep learning model and prove the effectiveness of the behavior chains. Table 5 summarizes the evaluation measures such as: False Positive Rate(FPR),False Negative Rate(FNR), Precision(P), area under ROC(AUC), Recall(R) and F1-measure(F1) for each experiment. The experimental results are presented and discussed in the next paragraphs.

The following observations can be made from Table 5 and Figure 13. First, the deep learning model substantially outperforms the other traditional algorithms overall, as the CNN (3.86% and 5.99%), the DNN(4.73% and 6.90%), the GRU(3.17% and 1.71%) and the

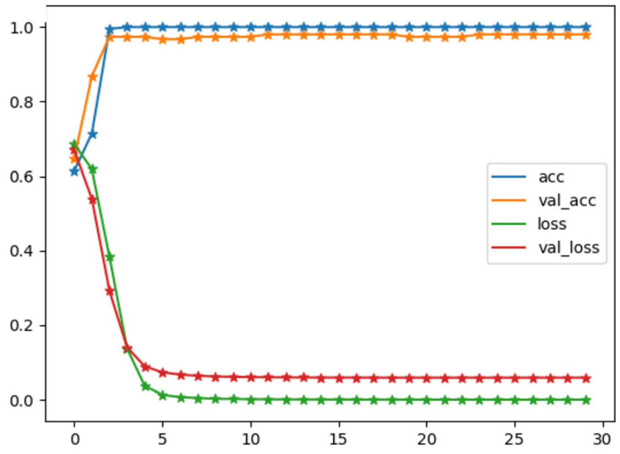


Figure 12 Accuracy and loss values

Table 5 Comparison of experimental results of each model

	FPR(%)	FNR(%)	P(%)	R(%)	F1(%)	AUC(%)
NB [27]	4.76	58.12	79.4	70.36	71.98	78.1
LR	1.59	88.03	81.37	59.51	58.03	61.08
KNN	3.11	5.13	95.61	96.31	95.94	96.42
SVM [18]	3.17	14.53	93.8	91.81	92.06	91.15
MLP	2.65	53.85	83.75	72.07	73.16	71.75
CNN [18]	3.86	5.99	96.13	97.38	96.96	97.01
DNN [18]	4.73	6.90	97.98	96.55	97.18	96.47
GRU	3.17	1.71	97.32	98.01	97.46	97.56
LSTM	1.05	3.42	98.64	97.98	98.01	97.76

LSTM(1.05% and 3.42%) have much smaller FPR and FNR values than the other traditional algorithms. Considering the other models, we find that the FRP and the FNR are also relatively small, for the KNN because the distributions of malware and benign data are not uniform, which may be due to sample imbalance. Second, for the traditional algorithms, i.e.NB and LR, low FPRs(4.76% and 1.59%, respectively) are achieved only with high FNRs(58.12% and 88.03%, respectively), which lead to lower F1-measures. Similarly, the *P* values of the SVM and MLP algorithms are not high for the following reasons: Traditional algorithms usually calculate frequency by statistics, do not consider the context of the data, and cannot process time series data. Therefore, the accuracy will be lowe for data with context and hidden attributes, which explains why traditional algorithms incur high FNRs.

In contrast, deep learning algorithms can effectively extract feature values based on powerful nonlinear feature representation. Such algorithms can process time series data using RNNs with feedback and time parameters. Therefore, the results of the deep learning model are better than those of the traditional algorithm model overall. Among them, LSTM led to better results than CNN, DNN and GRU, and LSTM corresponded to a much higher F1-measure(i.e.,98.01% vs .97.46% for GRU, 97.18% for DNN and 96.96% for CNN) because it had much lower FNR, we also note that the LSTM model had an FPR of 1.05% and a Precision of 98.64%.

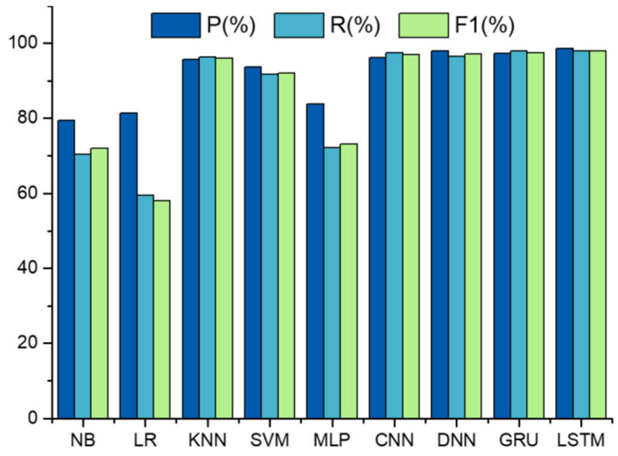


Figure 13 Comparison of the experimental results

Considering previous studies that reported traditional machine learning NB algorithm [27] and deep learning RNN and CNN algorithm [18] for malware detection, Table 5 and Figure 13 show that the NB accuracy rate is 79.4%, and that the DNN and CNN accuracy rates are similar. However, these algorithms have accuracy that are lower than that of the LSTM algorithm and FPR and FNR values that are higher than those of the LSTM algorithm The LSTM method used in this paper achieved an accuracy of 98.6%, the FPR of 1.05% and the FNR of 3.42%. However, the results of the entire experiment, indicate that our research still has much room for improvement. Because the amount of raw data used in this study was relatively small, the behavior relationships obtained from the mining process may be limited. Our goal is to find combinations of abnormal behaviors from multiple associations; consequently, only with a larger dataset can we better explore these relationships. Nonetheless, the experiment demonstrates the effectiveness of our approach.

5 Conclusions and future work

In this paper, we propose a depth-detection method for malware based on behavior chains. The behavior points required for the experiment are extracted from application monitoring log files by monitoring API call sequences. Based on these API call sequences, behavior chains with temporal characteristics are constructed and then input into the LSTM network to train the MALDC model, which is finally used to make malware or benign software classifications. In the final experimental results, the model's accuracy on the test data reached 98.64%. Because the construction of the MALDC model requires a large amount of training data, the accuracy of the experiment will likely improve substantially with more train data.

Moreover, this study analyzed only individual APIs; it did not attempt to consider the impact of the parameters or parameter values that were input to or output by these APIs on detecting malicious behavior. Therefore, in future work, we plan to continuously collect malicious data to expand the data set and to consider the API parameters and parameter values. Through future experiments, we will be able to analyze whether the parameters and parameter values have a significant impact on malicious behavior judgments and improve malware identification.

Acknowledgments We are grateful to the volunteers for capturing the data. This research is supported by the National Key Research and Development Program of China (No. 2017YFB1401300, 2017YFB1401304), the National Natural Science Foundation of China (No. 61702211,No. L1724007), the Hubei Provincial Science and Technology Program of China (No. 2017AKA191) and the Self-Determined Research Funds of CCNU from the Colleges' Basic Research (Nos. CCNU17QN0004 and CCNU17GF0002), and Natural Science Foundation of Shandong Province(ZR2017QF015).

References

1. Anderson, H.S., Woodbridge, J., Filar, B.: DeepDGA: adversarially-tuned domain generation and detection. In: Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security (AISec), pp. 13–21. ACM (2016)
2. Barreno, M., Nelson, B., Joseph, A.D., Tygar, J.D.: The security of machine learning. Mach. Learn. **81**(2), 121–148 (2010)

3. Berlin, K., Slater, D., Saxe, J.: Malicious behavior detection using windows audit logs. In: Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security(AISec), pp. 35–44. ACM (2015)
4. Dullien, T., Rolles, R.: Graph-Based comparison of executable objects (English version). In: Proceedings of the Symposium sur la sécurité des technologies de l'information et des communications(SSTIC). http://actes.sstic.org/SSTIC05/Analyse_differentielle_de_binaires/ (2005). Accessed Jan 2019
5. Fan, C., Hsiao, H.W., Chou, C.H., Tseng, Y.F.: Malware detection systems based on API log data mining. In: Proceedings of the IEEE 39th Annual Computer Software and Applications Conference(COMPSAC), pp. 255–260. IEEE (2015)
6. Fan, Y., Ye, Y., Chen, L.: Malicious sequential pattern mining for automatic malware detection. Expert Syst. Appl. **52**(C), 16–25 (2016)
7. Fereidooni, H., Conti, M., Yao, D., Sperduti, A.: ANASTASIA: android malware detection using static analysis of applications. In: Proceedings of the 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS), pp. 1–5 (2016)
8. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572. (2014)
9. Grosse, K., Papernot, N., Manoharan, P., Backes, M., Mcdaniel, P.: Adversarial perturbations against deep neural networks for malware classification. arXiv preprint arXiv:1606.04435. (2016)
10. Han, K.S., Kim, I.K., Im, E.G.: Malware classification methods using API sequence characteristics. Lecture Notes in Electrical Engineering(LNEE). **120**, 613–626 (2012)
11. Han, L., Fu, C., Zou, D., Lee, C.H., Jia, W.: Task-based behavior detection of illegal codes. Math. Comput. Model. **55**(1–2), 80–86 (2012)
12. Hansen, S.S., Larsen, T.M.T., Stevanovic, M., Pedersen, J.M.: An approach for detection and family classification of malware based on behavioral analysis. In: Proceedings of the International Conference on Computing, Networking and Communications (ICNC), pp.1–5 (2016)
13. Hou, S., Saas, A., Chen, L., Ye, Y.: Deep4MalDroid: a deep learning framework for android malware detection based on Linux kernel system call graphs. In: Proceedings of the 2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW), pp. 104–111. IEEE (2016)
14. Hou, S., Ye, Y., Song, Y.: HinDroid: an intelligent android malware detection system based on structured heterogeneous information network. In: Proceedings of the 23rd ACM SIGKDD International Conference, pp. 13–17. ACM (2017)
15. Huang, J., Swindlehurst, A.L: Secure communications via cooperative jamming in two-hop relay systems. In: IEEE Globecom, pp. 1–5 (2010)
16. Idika, N., Mathur, A.P.: A Survey of Malware Detection Techniques. Purdue University (2007)
17. Karbalaie, F., Sami, A., Ahmadi, M.: Semantic malware detection by deploying graph mining. International Journal of Computer Science Issues (IJCSI). **9**(1), 373–379 (2012)
18. Kolosnjaji, B., Zarras, A., Webster, G., Eckert, C.: Deep learning for classification of malware system call sequences. In: Proceedings of the Australasian Joint Conference on Artificial Intelligence, pp. 137–149. Springer (2016)
19. Li, Z., Zou, D., Xu, S., Jin, H., Hu, J.: VulPecker: an automated vulnerability detection system based on code similarity analysis. In: Proceedings of the 32nd Annual Conference on Computer Security Applications, pp. 201–213. ACM (2016)
20. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: Vuldeepecker: a deep learning-based system for vulnerability detection. In: Proceedings of the 25th Annual Network and Distributed Systems Security Symposium (NDSS'2018) (2018)
21. MIT Technology Review. Machine-Learning Algorithm Combs the Darknet for Zero Day Exploits, and Finds Them. MIT Technology Review. https://www.technologyreview.com/s/602115/machine-learning-algorithm-combs-the-darknet-for-zero-day-exploits-and-finds-them/ (2016). Accessed Jan 2019
22. Mosli, R., Li, R., Yuan, B., Pan, Y.: Automated malware detection using artifacts in forensic memory images. In: Technologies for Homeland Security (HST), pp. 1–6. IEEE (2016)
23. Parampalli, C., Sekar, R., Johnson, R.: A practical mimicry attack against powerful system-call monitors. In: Proceedings of the 2008 ACM symposium on Information, Computer and Communications Security, pp. 156–167. ACM (2008)
24. Rattan, D., Bhatia, R., Singh, M.: Software clone detection: a systematic review. Inf. Softw. Technol. **55**(7), 1165–1199 (2013)
25. Rieck, K., Laskov, P.: Linear-time computation of similarity measures for sequential data. J. Mach. Learn. Res. **9**(9), 23–48 (2008)
26. Rndic, N., Laskov, P.: Practical evasion of a learning-based classifier: a case study. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy, pp. 197–211. IEEE (2014)

27. Salehi, Z., Ghiasi, M., Sami, A.: A miner for malware detection based on API function calls and their arguments. In: Proceedings of the 16th CSI International Symposium on Artificial Intelligence and Signal Processing (AISP 2012), pp. 563–568. IEEE (2012)
28. Saxe, J., Berlin, K.: Deep neural network based malware detection using two dimensional binary program features. In: Proceedings of the 10th International Conference on Malicious and Unwanted Software, pp. 11–20. IEEE (2015)
29. Sun, M., Li, X., Lui, J.C.S., Ma, R.T.B., Liang, Z.: Monet: a user-oriented behaviour-based malware variants detection system for android. IEEE Transactions on Information Forensics and Security. **12**(5), 1103–1112 (2017)
30. Tian, R., Islam, R., Batten, L., Versteeg, S.: Differentiating malware from cleanware using behavioural analysis. In: Proceedings of the 5th International Conference on Malicious and Unwanted Software(MALWARE), pp. 23–30. IEEE (2010)
31. Uppal, D., Sinha, R., Mehra, V., Jain V.: Malware detection and classification bases on extraction of API sequences. In: Proceedings of the International Conference on Advances in Computing, Communications and Informatics(ICACCI), pp. 2337–2342. IEEE (2014)
32. Wang, Z., Pierce, K., McFarling, S.: BMAT—a binary matching tool for stale profile propagation. The Journal of Instruction-Level Parallelism(JILP). **10**(2), 23–25 (2000)
33. Wang, R., Feng, D.G., Yang, Y., Su, P.R.: Semantics-based malware behavior signature extraction and detection method. Journal of Software. **23**(2), 378–393 (2012)
34. Matt Wolff Andrew Davis: Deep learning on disassembly data. https://www.blackhat.com/docs/us-15/materials/us-15-Davis-Deep-Learning-On-Disassembly.pdf (2015). Accessed Jan 2019
35. Yuan, Z., Lu, Y., Wang, Z., Xue, Y.: Droid-sec: deep learning in android malware detection. Acm Sigcomm Computer Communication Review. **44**(4), 371–372 (2014)

Affiliations

Hao Zhang[1,2] • Wenjun Zhang[1,2] • Zhihan Lv[3] • Arun Kumar Sangaiah[4] • Tao Huang[1,2] • Naveen Chilamkurti[5]

[1] National Engineering Laboratory for Educational Big Data, Central China Normal University, Wuhan, China

[2] National Engineering Research Center for E-Learning, Central China Normal University, Wuhan, China

[3] School of Data Science and Software Engineering, Qingdao University, Qingdao 266071, China

[4] School of Computing Science and Engineering, Vellore Institute of Technology, Vellore 632014, India

[5] Department of Computer Science and Computer Engineering, La Trobe University, Melbourne, Australia