Computers & Security 116 (2022) 102686

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

A novel deep framework for dynamic malware detection based on API sequence intrinsic features

Ce Li[a,b], Qiujian Lv[a], Ning Li[a], Yan Wang[a,b,*], Degang Sun[a,b], Yuanyuan Qiao[c]

[a] *Institute of Information Engineering, Chinese Academy of Science, Beijing, China*
[b] *School of Cyber Security, University of Chinese Academy of Science, Beijing, China*
[c] *School of Artificial Intelligence, Beijing University of Posts and Telecommunications, Beijing, China*

A R T I C L E I N F O

Article history:
Received 31 October 2021
Revised 24 January 2022
Accepted 7 March 2022
Available online 8 March 2022

Keywords:
Malware detection
API sequence
Deep learning
Intrinsic features
Feature fusion

A B S T R A C T

Dynamic malware detection executes the software in a secured virtual environment and monitors its runtime behavior. This technique widely uses API sequence analysis to identify whether the running software is malicious or not. However, existing solutions typically only consider the API name or frequency of API usage, and the feature mining of API sequence is not sufficient, which leads some malware to escape from being detected. In this paper, we propose a novel malware detection framework using deep learning models to capture and combine more meaningful features which are called intrinsic features of the API sequence. Specifically, we first apply embedding and convolutional layers to conduct a joint representation of multiple APIs to represent the software behavior. Secondly, we use the category, action, and operation object of the API to represent the semantic information of each API call. Finally, we use the Bi-LSTM module to mine the relationship information between APIs. Our proposed model achieves an accuracy of 0.9731 and an F1-score of 0.9724 on a large real dataset, which outperforms baselines significantly. We also conduct ablation studies to prove the effectiveness of our intrinsic features.

1. Introduction

Malicious software (Malware) has seen rapid expansion in recent years due to the swift growth in computer and Internet technology. Large-scale malware attacks (Li et al., 2017), in which criminals use scripts to generate large amounts of malware automatically, are currently the main means of malware intrusion. Different kinds of malware such as worm, virus, Trojan, and backdoor change very quickly and their variants have become the biggest threat to cyber security. According to an AV-TEST report from 2019 to 2020,[1] more than 114 million new malware were developed and over 78 percent of them have been applied to Windows systems. Furthermore, over 43 million 0-day malware were recorded in the first quarter of 2020. Therefore, it is necessary to devise an effective automatic detection method to counter the malware attacks.

Malware detection approaches can be divided into static and dynamic malware analysis. Static analysis methods (Ahmadi et al., 2016; Christodorescu and Jha, 2003; Ding and Zhu, 2019; Nair et al., 2010; Vasan et al., 2020) try to analyze the source code but malware authors can obfuscate the contents of a binary and make static data extraction meaningless (Han et al., 2019; You and Yim, 2010). On the contrary, dynamic malware analysis extracts data (including system calls, network traffic packets, file or registry changes, memory dumps, etc.) while running binary programs (Galal et al., 2016; Salehi et al., 2017). Malware must unpack itself while executing. Its original code will be loaded into the main memory and it will perform the real behaviors (Raff and Nicholas, 2020). Therefore, dynamic malware analysis is effective against various code obfuscation techniques (Amer and Zelinka, 2020; Damodaran et al., 2017). In this paper, we focus on dynamic malware analysis.

The Application Programming Interface (API) calls are widely used in dynamic analysis (Alazab et al., 2011; Elhadi et al., 2014; Qiao et al., 2013; Zhang et al., 2020). A running software calls many APIs which characterize all the software operations including network access, file creation and modification, etc. These APIs form an API sequence which is often used to analyze the *software behavior*. For example, on Windows platform, the API sequence of *RegCreateKey*, *RegSetValue*, and *RegCloseKey* always represents the behav-

* Corresponding author at: Institute of Information Engineering, Chinese Academy of Science, Beijing, China.
E-mail addresses: lice@iie.ac.cn (C. Li), lvqiujian@iie.ac.cn (Q. Lv), lining01@iie.ac.cn (N. Li), wangyan@iie.ac.cn (Y. Wang), sundegang@iie.ac.cn (D. Sun), yyqiao@bupt.edu.cn (Y. Qiao).

[1] https://www.av-test.org/de/news/fakten-analysen-zur-bedrohungslage-der-av-test-sicherheitsreport-2019-2020/.

https://doi.org/10.1016/j.cose.2022.102686
0167-4048/© 2022 Elsevier Ltd. All rights reserved.

ior of operating a registry key, and these three APIs are often used together. Thus, it is important to find a type of joint representation of multiple APIs because a single API does not mean whether the software behavior is malicious or not. Moreover, APIs are not independent of each other in an API sequence. In other words, each API in the sequence is usually associated with some past or future APIs. Such relationship may contains some contextual patterns that perform malicious activities (Amer and Zelinka, 2020). Thus, it is meaningful to capture the *relationship information* between API calls.

There are many approaches based on artificial intelligence that have been applied to analyze API sequences. Machine learning (ML) algorithms such as K-Nearest Neighbor (KNN), Naive Bayes (NB), Decision Tree (DT), and Support Vector Machine (SVM) are widely used in API sequence analysis (Fan et al., 2018; Kim, 2018; Lin et al., 2018). Researchers are also exploring deep learning (DL) models for feature mining to improve detection accuracy (Agrawal et al., 2018; Çatak et al., 2020; Kolosnjaji et al., 2016). Unfortunately, most of these studies often only consider the API name or frequency of API usage while ignoring some *semantic information* (including category, action, operation object, etc) of the API calls. For instance, the API *RegSetValue* indicates setting a value for a registry key, and its action is *set*, the operation object is the value of the registry key. These semantic information can helps the detector understand the meaning of the API. Therefore, traditional detection methods without API semantic information are still inadequate.

The software behavior, semantic information and relationship information mentioned above can not be obtained directly from the API sequence, but they can be represented after the representation learning on a large number of API sequence corpus. We define the features obtained by representation learning as the *intrinsic features* of the API sequence. In this paper, we propose a deep framework for dynamic malware detection based on multiple API sequence intrinsic features. Through mining the meaningful intrinsic features, our proposed framework is able to detect whether the software is malicious or not. To justify the idea, we collect a large number of software samples and use our framework to distinguish between malware and goodware. The results show that our proposed work outperforms the previous works in using API sequence to detect the malware.

Our contributions in this paper are as follows:

- We devise a deep framework for dynamic malware detection based on API sequence intrinsic features. We also create a dataset which contains different kinds of malware and benign software (Goodware) samples to evaluate our model. We open source our dataset and code on Github.[2]
- We design an encoder to represent the API sequence intrinsic features including the software behavior, the semantic information of APIs, and the relationship between API calls. To the best of our knowledge, this research is the first to apply multi-feature based deep learning to represent and combine these 3 kinds of intrinsic features of the API sequence. Extensive experiments verify the effectiveness of our proposed method.
- We conduct ablation studies and analyze the contribution of each intrinsic feature to the model. It provides valuable insights in the API sequence feature learning.

The rest of this paper is organized as follows: related work and background of previous research are discussed in Section 2. Section 3 introduces some necessary preliminaries. Section 4 introduces our proposed method of malware detection. Section 5 provides the detailed experimental setup. Section 6 shows the experiment results and model evaluation. Section 7 presents the limitations and our future work. Finally, Section 8 concludes this paper.

[2] https://github.com/friendllcc/Malware-Detection-API-Sequence-Intrinsic-Features.

2. Related work

In this section, we firstly review the research of API based malware detection. Then, we present the intrinsic feature representation of API sequence. Finally, we introduce deep learning-based malware detection approaches.

2.1. API based malware detection methods

Among many API sequence based malware detection research works, the methods based on frequency statistics and sequence encoding are widely used.

2.1.1. Frequency statistics based methods

Detecting malware by counting the frequency of API usage is a kind of traditional method. Tian et al. (2010) use a hash table to store all the API strings with their global frequency. Sami et al. (2010) calculate the frequency of API calls in each file and analyze statistical results. However, this kind of method assumes that the APIs are independent of each other and ignores the relationship between API calls.

2.1.2. Sequence encoding based methods

Sequence encoding tends to represent the API sequence as standard data formats such as vector, matrix, etc. The encoding scheme has a direct impact on the final detection performance. Tran and Sato (2017) use paragraph vector to divide API sequences and give weights to each API using term frequency-inverse document frequency (TF-IDF). The objective of TF-IDF is to convert n-grams into numerical input features where machine learning algorithms can work. However, TF-IDF methods do not preserve any contextual relationship that exists between APIs. Amer and Zelinka (2020) use word embedding method to represent every API name as a vector which contains contextual information. However, this method only consider the API name and the extraction of intrinsic features is not comprehensive enough.

Compared to frequency statistics based methods, sequence encoding based methods can process API sequences better. Inspired from sequence encoding based methods, researchers can extract more meaningful intrinsic features to improve the performance of malware detection.

2.2. Intrinsic feature representation of API sequence

Many studies try to express API sequence intrinsic features to enhance the performance of detection models. The intrinsic features mainly contains the software behavior, the semantic information of APIs, and the relationship between API calls.

2.2.1. Software behavior

An API represents a software operation, and multiple operations represent a software behavior. In order to learn the behavior information, Lin et al. (2015) define malware behavior vectors to represent multiple APIs and calculate the cosine similarity to classify the malware. Similarly, Kim et al. (2016) use multiple sequence alignment to generate malware behavioral feature chain patterns. Ki et al. (2015) extract common API sequence patterns of malicious behavior from malware in different categories. However, these approaches ignore the semantic information of each API in the sequence.

2.2.2. Semantic information

The semantic information represents the meaning of each API, such as category, action, and operation object. These information can help the detection model understand the API sequence more accurately. Zhang et al. (2020) propose a pre-defined semantic vector of API calls. They extract name, category, and API arguments of the API calls using hash approach. However, this approach ignores the actions and operation objects of the API calls. Liu et al. (2011) define a behavior operation set which includes file actions, process actions, network actions and registry actions. They use the action and operation object of API call to represent the semantic information. Unfortunately, this method extract only 18 kinds of actions and the information captured is not enough.

2.2.3. Relationship between API calls

The relationship between API calls is a common intrinsic feature and many studies confirm its effectiveness. Çatak et al. (2020) use embedding layer and Long Short-Term Memory (LSTM) model to capture relationship between API calls in the API sequence. However, this approach uses unidirectional LSTM which only consider the relationship information with the past APIs while ignoring the future APIs. Amer and Zelinka (2020) group related APIs based on their contextual similarity and generate a simple call graph that characterizes the running process of malware. However, this ”contextual similarity” is gradually weakened in the process of model building due to the model simplification.

Malware usually tries to escape from being tracked or detected. Using only one type of intrinsic features is not enough to correctly detect malware in a real-world environment (Amer and Zelinka, 2020). Thus, it is important to design algorithms that can handle multiple categories of intrinsic features extracted from API sequences.

2.3. Deep learning-based approaches

The attention to deep learning in the malware detection community is increasing. David and Netanyahu (2015) use a deep belief network (DBN) to process the software running reports. Pascanu et al. (2015) use recurrent neural networks (RNNs) to predict the next API call based on the previous APIs. They also feed the outputs of RNNs into a max-pooling layer to transform features for malware classification. Agrawal et al. (2018) propose a feature representation based on one-hot vectors from API name and top N frequent n-gram of API arguments. They combine several LSTMs to build a classifier.

Deep learning-based methods are able to achieve more flexible feature representation. In this paper, we use deep learning models to represent and combine multiple API sequence intrinsic features.

3. Preliminaries

In this section, we mainly briefly describe two feature representation models adopted in our method: one-dimensional convolutional neural network (1D CNN) and LSTM.

3.1. 1D CNN

Convolutional neural network (CNN) models are developed for learning an internal representation of a two-dimensional input such as image data. This same process can be harnessed on one-dimensional sequences of data, such as in the case of API sequence data for malware detection.

1D CNN model (Kim, 2014) learns to extract features from sequences of observations and represent the features as feature maps. It applies a convolution layer to construct a joint representation of multiple items in the input sequence. For example, for a input sequence $\{\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}, \boldsymbol{x_4}, \ldots,\}$ ($\boldsymbol{x_i}$ is the feature vector of the item i), when the size of the convolution filter $\boldsymbol{W}$ is 3, the sequence will be express as $\{y_{123}, y_{234}, \ldots\}$, where y_{ijk} is the feature map of $\boldsymbol{x_i}, \boldsymbol{x_j}, \boldsymbol{x_k}$:

$$y_{ijk} = f(\boldsymbol{W} \cdot [\boldsymbol{x_i} \oplus \boldsymbol{x_j} \oplus \boldsymbol{x_k}] + b). \tag{1}$$

Here $\oplus$ is the feature concatenation operator, b is a bias term and f is a non-linear activation function.

The benefit of using the 1D CNN for sequence classification is that it can learn from the raw time series data directly, and in turn does not require domain expertise to manually engineer input features.

3.2. LSTM

LSTM (Staudemeyer and Morris, 2019) is a recurrent neural network architecture. It is able to capture the long-term context information through several gates designed to control the information transmission status (Pichotta and Mooney, 2016). The input of LSTM model is a sequence $\{\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}, \ldots\}$, where $\boldsymbol{x_i}$ is the feature vector of the item i. Let $\boldsymbol{W_i}$, $\boldsymbol{W_f}$, $\boldsymbol{W_C}$, $\boldsymbol{W_o}$ be the weight matrices and $\boldsymbol{b_i}$, $\boldsymbol{b_f}$, $\boldsymbol{b_C}$, $\boldsymbol{b_o}$ be the offset vectors. The calculation of LSTM at moment t has two steps. Firstly, calculate the input gate as

$$\boldsymbol{i_t} = \sigma(\boldsymbol{W_i} \cdot [\boldsymbol{h_{t-1}}, \boldsymbol{x_t}] + \boldsymbol{b_i}), \tag{2}$$

forget gate as

$$\boldsymbol{f_t} = \sigma(\boldsymbol{W_f} \cdot [\boldsymbol{h_{t-1}}, \boldsymbol{x_t}] + \boldsymbol{b_f}), \tag{3}$$

and memory gate of the middle state value as

$$\tilde{\boldsymbol{C}}_t = tanh(\boldsymbol{W_C} \cdot [\boldsymbol{h_{t-1}}, \boldsymbol{x_t}] + \boldsymbol{b_C}). \tag{4}$$

Secondly, calculate the memory gate status values as

$$\boldsymbol{C_t} = \boldsymbol{i_t} * \tilde{\boldsymbol{C}}_t + \boldsymbol{f_t} * \boldsymbol{C_{t-1}}, \tag{5}$$

output gate as

$$\boldsymbol{o_t} = \sigma(\boldsymbol{W_o} \cdot [\boldsymbol{h_{t-1}}, \boldsymbol{x_t}] + \boldsymbol{b_o}), \tag{6}$$

and hidden layer output as

$$\boldsymbol{h_t} = \boldsymbol{o_t} * tanh(\boldsymbol{C_t}). \tag{7}$$

The hidden layer output $\boldsymbol{h_t}$ contains the information from the past.

4. Proposed method

In this section, we introduce our proposed method for malware detection. First, Section 4.1 introduces our detection framework which mainly contains an intrinsic feature encoder and a Multi-Layer Perceptron (MLP) classifier. Second, in Sections 4.2–4.4, we elaborate our intrinsic feature encoding methods including *API Phrase, Semantic Chain*, and *Bi-LSTM*. Finally, the specific model architecture will be given in Section 4.5.

4.1. System overview

To construct an effective detector, we implement the system framework shown in Fig. 1. This framework aims to characterize the multiple intrinsic features from the API sequence, and combine them to identify whether the running software is malicious or not. In this paper, we mainly focus on the malware on Windows platform.

The Windows portable executable (PE) files are used as input samples. Then, a sandbox is applied to record the API sequence while running each sample. An intrinsic feature encoder is designed to represent and combine 3 kinds of intrinsic features of

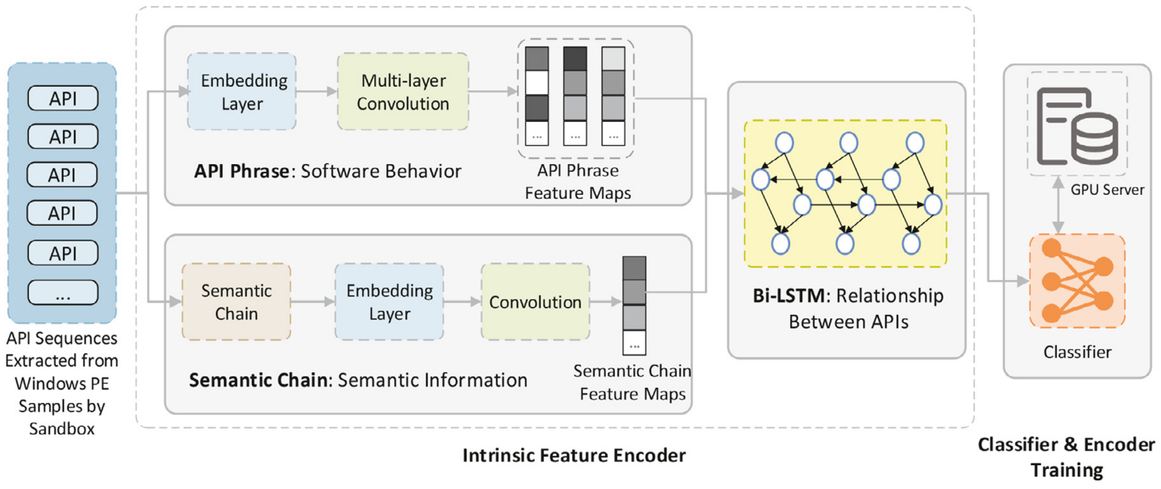
API
API
API
API
API
...
API Sequences Extracted from Windows PE Samples by Sandbox
Embedding Layer
Multi-layer Convolution
API Phrase Feature Maps
API Phrase: Software Behavior
Semantic Chain
Embedding Layer
Convolution
Semantic Chain Feature Maps
Semantic Chain: Semantic Information
Bi-LSTM: Relationship Between APIs
Intrinsic Feature Encoder
GPU Server
Classifier
Classifier & Encoder Training

Fig. 1. System Framework. The system mainly consists of two parts. (1) Intrinsic Feature Encoder: to represent and combine 3 kinds of intrinsic features. (2) Classifier & Encoder Training: to build a model which consists of the intrinsic feature encoder and a classifier, then train this model.

the API sequence including software behavior, semantic information, and relationship between APIs. The encoder has three modules:

- *API Phrase:* By using embedding and convolutional layers, this module represents the software behavior as API phrase feature maps.
- *Semantic Chain:* This module first constructs a semantic chain to represent semantic information of the API sequence. Then, the semantic chain is represented as feature maps by embedding and convolutional layers.
- *Bi-LSTM:* Bidirectional LSTM (Bi-LSTM) module is applied to capture the relationship between API calls.

An MLP classifier is connected after the intrinsic feature encoder, and outputs the final classification result of each input sequence. The complete model includes the intrinsic feature encoder and a binary classifier. After model training, this framework can be used to detect whether a new software sample is malware.

In the following Sections 4.2–4.4, we will introduce our intrinsic feature encoder including API Phrase, Semantic Chain, and Bi-LSTM.

4.2. API Phrase for software behavior representation

In this subsection, we introduce API Phrase module which depicts how to represent the software behavior.

API Phrase which represents software behavior is a type of joint representation of multiple API calls. This module has two steps: API embedding and multi-layer convolution. First, an embedding layer is used to turn every API name into a vector and turn the API sequence into a sequence matrix. Then, three convolutional layers with filter sizes of 3, 4, and 5 are applied to convert the sequence matrix into feature maps.

4.2.1. API embedding

Embedding layer (Ketkar and Santana, 2017) is a feature representation method that is widely used in the field of natural language processing. In our framework, the embedding layer is applied in order to turn each API name into a k-dimensional dense feature vector. As shown in Fig. 2(a) Step-1, let $\boldsymbol{a_i} \in \mathbb{R}^k$ be the k-dimensional feature vector corresponding to the ith API in the API sequence. Then, an API sequence which consists of n APIs is represented as

$$\boldsymbol{A_{1:n}} = \boldsymbol{a_1} \oplus \boldsymbol{a_2} \oplus \cdots \oplus \boldsymbol{a_n}, \tag{8}$$

where $\oplus$ is the concatenation operator and $\boldsymbol{A_{1:n}} \in \mathbb{R}^{n\times k}$ is a $n \times k$ sequence matrix.

4.2.2. Multi-layer convolution

The 1D CNN (Kim, 2014; Zhang and Wallace, 2017) described in Section 3.1 is applied to transform the sequence matrix into the API phrase feature maps which can characterize the software behavior (as shown in Fig. 2(a) Step-2). Generally, let $\boldsymbol{A_{i:i+j}} \in \mathbb{R}^{j\times k}$ refer to the concatenation of vectors from $\boldsymbol{a_i}$ to $\boldsymbol{a_{i+j}}$. A convolutional layer involves a filter $\boldsymbol{W_a} \in \mathbb{R}^{h\times k}$, where h is the number of vectors this filter can process each time, and is also the length of the API phrase. The filter $\boldsymbol{W_a} \in \mathbb{R}^{h\times k}$ is applied to h vectors from $\boldsymbol{a_i}$ to $\boldsymbol{a_{i+h-1}}$ and a feature m_i is generated from $\boldsymbol{A_{i:i+h-1}}$ by

$$m_i = f(\boldsymbol{W_a} \cdot \boldsymbol{A_{i:i+h-1}} + b_a). \tag{9}$$

Here $b_a \in \mathbb{R}$ is a bias term and f is a non-linear activation function such as the hyperbolic tangent. This filter is applied to each possible window of APIs in the sequence $\{\boldsymbol{A_{1:h}}, \boldsymbol{A_{2:h+1}}, \ldots, \boldsymbol{A_{n-h+1:n}}\}$ to produce a feature map

$$\boldsymbol{m} = [m_1, m_2, \ldots, m_{n-h+1}]. \tag{10}$$

API phrases of different lengths represent behavioral characteristics of different granularities. Multi-layer convolution applies several convolutional layers with different filter sizes to extract API phrases of different lengths. Fig. 2(b) shows the example of using three convolution layers with different filter sizes. In this example, three convolution layers with filter size h of 3, 4, and 5 are applied to extract API phrases of different lengths in order to represent more behavioral features.

4.3. Semantic chain for semantic information representation

Semantic Chain module is designed to represent the semantic information of each API. As shown in Fig. 3, the Semantic Chain module extracts API semantic information in two steps: (1) semantic chain construction and (2) feature map representation of semantic chain. Step-1 aims to extract the action, operation object, and category of each API call and construct a semantic chain for the API sequence. Step-2 attempts to represent the semantic chain as feature maps by using an embedding layer and a convolutional layer.

4.3.1. Semantic chain construction

When naming APIs, operating system developers adopt strict naming conventions, and they usually take the semantic-based

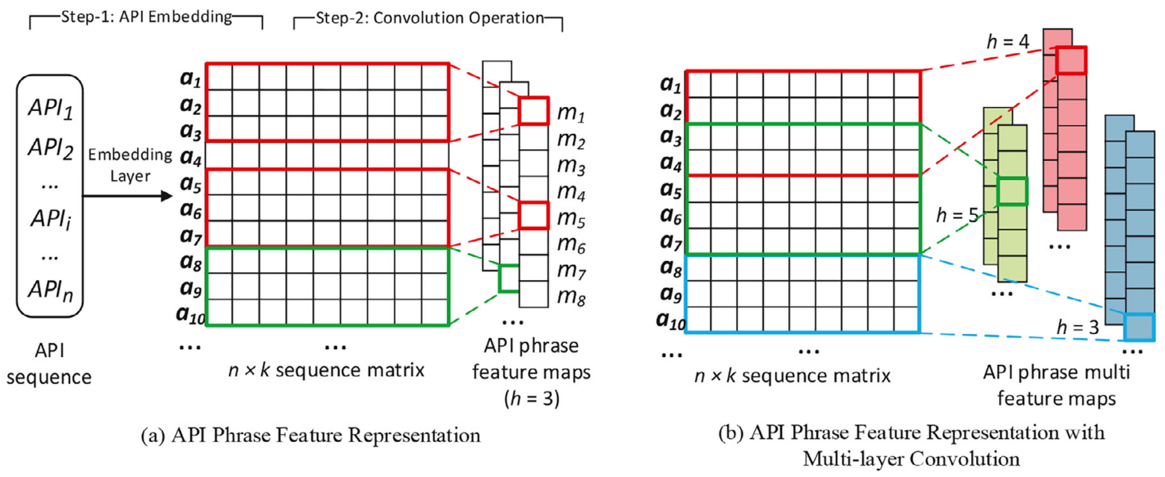
Step-1: API Embedding
Step-2: Convolution Operation
API1
API2
APIi
APIn
Embedding Layer
API sequence
n × k sequence matrix
API phrase feature maps (h = 3)
(a) API Phrase Feature Representation
h = 4
h = 5
h = 3
API phrase multi feature maps
(b) API Phrase Feature Representation with Multi-layer Convolution

Fig. 2. (a) API Phrase Feature Representation: An embedding layer turns API_i to a vector $\boldsymbol{a_i}$. A convolutional layer represents multiple APIs ($\boldsymbol{a_i}, \boldsymbol{a_{i+1}}, \ldots, \boldsymbol{a_{i-h+1}}$) as a feature m_i. (b) API Phrase Feature Representation with Multi-layer Convolution: In the convolution operation stage, three convolution layers with filter size $h = 3, 4, 5$ are applied to extract API phrases of different lengths.

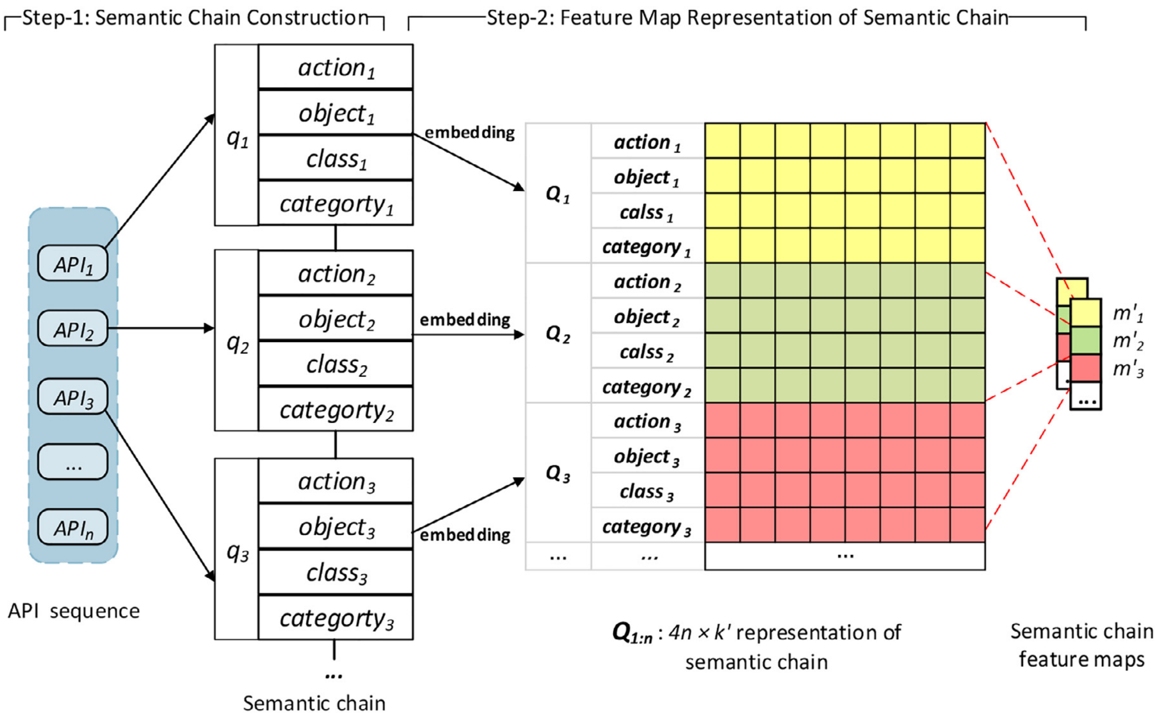
Step-1: Semantic Chain Construction
Step-2: Feature Map Representation of Semantic Chain
API sequence
q1 action1 object1 class1 categorty1
q2 action2 object2 class2 categorty2
q3 action3 object3 class3 categorty3
Semantic chain
embedding
Q1 action 1 object 1 calss 1 category 1
Q2 action 2 object 2 calss 2 category 2
Q3 action 3 object 3 class 3 category 3
Q1:n : 4n × k' representation of semantic chain
m'1 m'2 m'3
Semantic chain feature maps

Fig. 3. Semantic Chain Feature Representation: Step-1 re-describes API_i as q_i and constructs a semantic chain for the API sequence. Step-2 represents the semantic chain as feature maps by using an embedding layer and a convolutional layer.

naming methods (Hart, 1997). Thus, the API name string contains some semantic information that can reflect the API's action and operation object. To capture the semantic information contained in the API name, we introduce a template to re-describe each API as a 4-tuple. Let q_i be the 4-tuple corresponding to the i-th API in the API sequence. The q_i is represented as

$$q_i =< action_i, object_i, class_i, category_i > . \tag{11}$$

The value of $action_i$ and $object_i$ are extracted from the API name. We firstly analyze the most commonly used 312 APIs embodied in the sandbox and 70 words (mostly verbs) are extracted to form an action dictionary (AD). The content of AD is shown in Table 1 and these words describe the core actions of API calls. Then, for any API, its $action_i$ and $object_i$ can be obtained through Algorithm 1. For example, for the API 'RegCreateKeyExW', its $action_i$ is 'Create' and $object_i$ is 'RegKeyEx'. The reason why we remove the 'W' or 'A' at the end of API name string is that this character represents different code styles and has no effect on the semantic information.

In the process of invading a computer system, malware needs to query or modify the system resources to achieve intrusion (Mylonas and Gritzalis, 2012). Depending on the impact degree of the API on the system, we define the value of $class_i$ as one of 'Select', 'Update' and 'Other' (Eq. (12)). The meaning of each value is defined as follows:

- *Select:* The impact of this API is to query the system resources.
- *Update:* The impact of this API is to modify the system resources.
- *Other:* This API has no impact on the system.

Table 1
Action dictionary (AD).

Content of Action Dictionary (AD)	#Instance
'Acquire', 'Add', 'Allocate', 'Assign', 'Close', 'Compress', 'Connect', 'Control', 'Copy', 'Crack', 'Create', 'Decode', 'Decompress', 'Decrypt', 'Delete', 'Download', 'Draw', 'Duplicate', 'Encode', 'Encryp', 'Exec', 'Exit', 'Export', 'Enum', 'Free', 'Find', 'First', 'Get', 'Gen', 'Hash', 'Initialize','Is', 'Load', 'Lookup', 'Make', 'Map', 'Move', 'Next', 'Obtain', 'Open', 'Put', 'Protect', 'Queue', 'Query', 'Read', 'Recv', 'Register', 'Remove', 'Resume', 'Save', 'Send', 'Set', 'Socket', 'Start', 'Suspend', 'Search', 'Select', 'Sizeof', 'Status', 'listen', 'Terminate', 'Unhook', 'Uninitialize', 'Unload', 'Unmap', 'Unprotect', 'Write', 'accept', 'bind', 'shutdown'	70

Table 2
An example of conversion from API_i to q_i.

API_i	q_i			
	$action_i$	$object_i$	$class_i$	$category_i$
...	...	...	...	...
GetProcessHeap	Get	ProcessHeap	Select	process
RtlAllocateHeap	Allocate	RtlHeap	Update	system
SetLastError	Set	LastError	Update	system
RegCreateKeyExW	Create	RegKeyEx	Update	registry
RegQueryValueExW	Query	RegValueEx	Select	registry
RegSetValueExW	Set	RegValueEx	Update	registry
RegCloseKey	Close	RegKey	Update	registry
...	...	...	...	...

$$class_i \in \{'Select', 'Update', 'Other'\} \tag{12}$$

In addition, CuckooSandbox provides a classification standard that divides APIs into 18 categories which are depicted in Eq. (13). We quote it as the value of $category_i$.

$$category_i \in \{'system', 'network', 'process', 'misc', 'file', 'registry', 'service', 'crypto', 'resource', 'ole', 'exception', 'None', 'netapi', 'synchronisation', 'ui', 'certificate', 'iexplore', 'notification'\} \tag{13}$$

In summary, each API_i in the API sequence is re-described as the 4-tuple q_i (see an example in Table 2). Similar to the API sequence, these 4-tuples are connected into a tuple sequence which is called "semantic chain".

4.3.2. Feature map representation of semantic chain

After step-1, we convert each API name to a 4-tuple and form a semantic chain. Then we use an embedding layer to transform the semantic chain into a matrix. Finally, we apply a convolutional layer to obtain the feature maps of a semantic chain (as shown in Fig. 3 Step-2). Through the embedding layer, each element ($action_i$, $object_i$, $class_i$, and $category_i$) in q_i is turned into a k'-dimensional vector and q_i is turned into $\boldsymbol{Q_i} \in \mathbb{R}^{4\times k'}$. A semantic chain of length n is represented as

$$\boldsymbol{Q_{1:n}} = \boldsymbol{Q_1} \oplus \boldsymbol{Q_2} \oplus \cdots \oplus \boldsymbol{Q_n} \tag{14}$$

with $\boldsymbol{Q_{1:n}} \in \mathbb{R}^{4n\times k'}$. Let $\boldsymbol{W_q} \in \mathbb{R}^{4\times k'}$ refer to the convolution filter, then a feature m'_i is generated from $\boldsymbol{Q_i}$ by

$$m'_i = f(\boldsymbol{W_q} \cdot \boldsymbol{Q_i} + b_q), \tag{15}$$

where $b_q \in \mathbb{R}$ is a bias term. This filter is applied to each q_i in the semantic chain to produce a feature map

$$\boldsymbol{m'} = \left[m'_1, m'_2, \ldots, m'_n\right]. \tag{16}$$

4.4. Bi-LSTM for relationship information representation

In this subsection, the Bi-LSTM model is applied to capture the relationship between API calls.

A Bi-LSTM is composed of two LSTMs stacked together but with different directional inputs. The unidirectional LSTM is able to capture the relationship between the current API with the past APIs through several gates designed to control the information transmission status. Compared to unidirectional LSTM, Bi-LSTM can integrate the relationship information from the past and future states at the same time (Agrawal et al., 2018). Thus, the Bi-LSTM is applied to capture the complex relationship between APIs.

The sequence $\boldsymbol{X} = \boldsymbol{m} \oplus \boldsymbol{m'} = \{\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_t}, \ldots\}$ is input to Bi-LSTM, where $\boldsymbol{x_t} = m_t \oplus m'_t$ is the features in the feature maps obtained by the API Phrase and Semantic Chain modules. As described in Section 3.2, Bi-LSTM model calculates the hidden layer output $\boldsymbol{H} = \{\boldsymbol{h_1}, \boldsymbol{h_2}, \boldsymbol{h_t}, \ldots\}$ of two different directional LSTMs. The $\boldsymbol{H}$ of Bi-LSTM is used as the input of the classification module for malware detection.

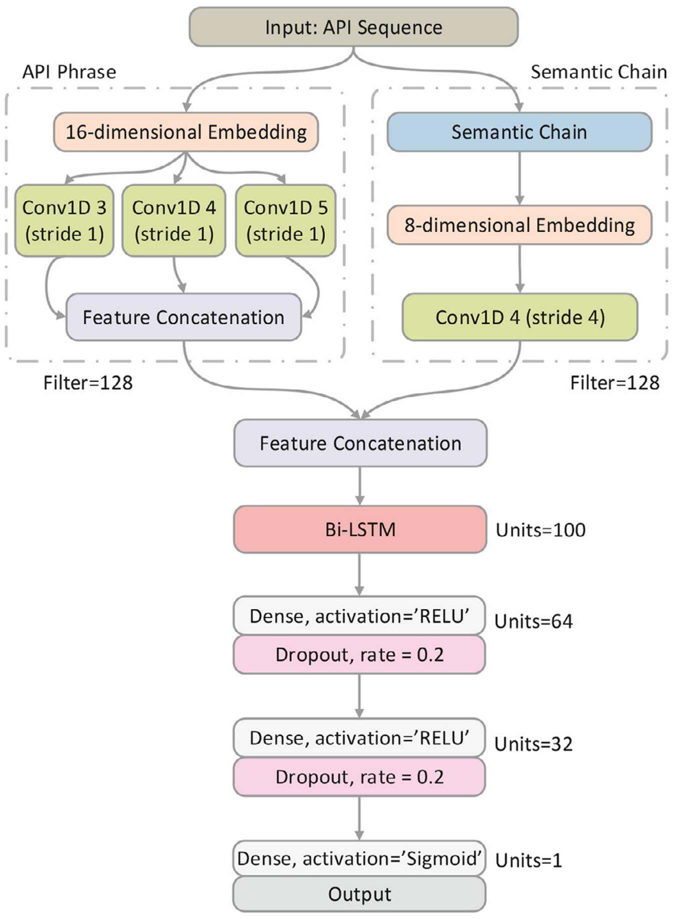

Fig. 4. Model architecture.

4.5. Model architecture

Our final model architecture is shown in Fig. 4. This deep neural network is designed to judge whether the input API sequence is malicious or not. The proposed model contains the intrinsic feature encoder (including API Phrase, Semantic Chain, and Bi-LSTM) and an MLP classifier.

Table 3
Dataset overview.

Dataset	Malware (Positive)	Goodware (Negative)	Total
Jan.–Jun. 2019 (Training Set)	10,433	10,875	21,308
Jul.–Dec. 2019 (Test Set)	10,454	11,245	21,699
ToTal	20,887	22,120	43,007

To express rich software behavior, API Phrase module sets an embedding layer and three parallel convolutional layers whose filter sizes are 3, 4, 5, respectively.

After semantic chain construction, the Semantic Chain module applies an embedding layer and a convolutional layer with filter size 4 and stride length 4. In this module, each API is turned into a matrix with the first dimension of 4 ($\boldsymbol{q}'_i \in \mathbb{R}^{4\times k'}$), so the convolutional layer can process every specific API.

All outputs from convolution layers are concatenated together. Then, the Bi-LSTM module is applied to capture the relationship between APIs.

After the Bi-LSTM module, an MLP classification module with dense layers is used to make a decision. After every dense layer, a dropout layer is used to reduce overfitting. In this paper, we apply 2 dense and 2 dropout layers as the hidden layer of the MLP. Then, the last dense layer with 1 unit applies a Sigmoid activation to output the probability of malware estimation. Our model applies Adam as the optimizer and supervises each input sequence with the label. To measure the loss of the training stage, we use the binary cross-entropy function in Eq. (17).

$$L = -\left(\sum_{i=1}^{n} y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log (1 - \hat{y}^{(i)})\right) \tag{17}$$

The detailed model structural configurations will be discussed in Section 6.2.

5. Experimental setup

In this section, we introduce the dataset and hyperparameter settings used for experiments. We also introduce the evaluation metrics and baseline models used for the evaluation.

5.1. Data collection

5.1.1. PE files collection

We collect Windows PE files in 2019 to create a software dataset. The label used for malware samples is “positive”, while the one used for goodware is “negative”.

The malicious PE files is obtained from the VirusShare website using a daily downloading script. The benign software is obtained from popular free software sources including Softonic.com, Sourceforge.net and Portableapps.com.

However, the samples from VirusShare or the free software sources are not guaranteed to be malicious or benign. To clean the fake malware and goodware, each sample is validated with VirusTotal, where more than 60 antivirus engines are applied to vote whether the test sample is malicious or benign. In our dataset, the samples with more than 10 malicious votes are chosen as the final malicious samples, and samples with 0 vote are chosen as the final benign samples. The samples with votes between 1 and 9 are omitted to prevent contaminating the dataset with false positives.

The final dataset which contains 20,887 malware and 22,120 goodware is shown in Table 3. The malware in dataset consists of several types including Spyware, Backdoor, Virus, Downloader, Ransom, Adware, Worm, and Trojan (as shown in Table 4). ”Disputed” class represents those malware whose type contains multiple categories and the specific type could not be determined.

Table 4
The malware instances of different types.

Type	#Instance
Spyware	2205
Backdoor	3213
Virus	1443
Downloader	1863
Ransom	1449
Adware	1159
Worm	1244
Trojan	4965
Disputed	3346

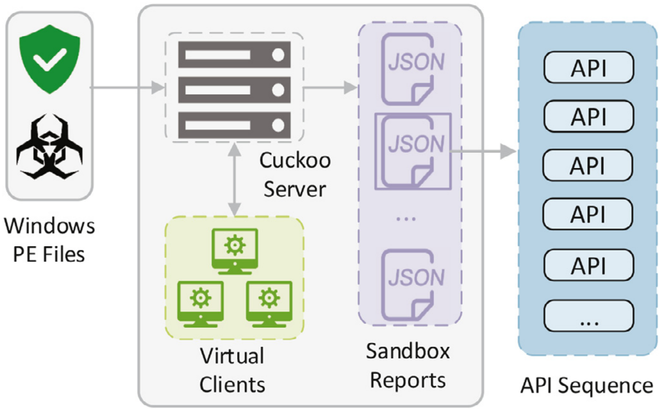

Fig. 5. Data collection.

Goodware samples consist of almost all application categories including System, Internet, Games, Bussiness, Media, Software Development Kit, Education, Social, Travel and Tools. The age of our samples is between January 2019 and December 2019, based on VirusTotal first seen time. In order to ensure the temporal training consistency (Pendlebury et al., 2019), we use the samples of the first half of 2019 as the training set, including 10,443 malware and 10,875 goodware. The 10,454 malware and 11,245 goodware in the latter half of 2019 are used as the test set.

Considering that new malware is being generated over time, there could be many PE files for new malware in the test dataset. Therefore, the experimental results indicate the model’s capability for detecting unknown malware to a certain degree.

5.1.2. API sequence collection

After PE files collection, the CuckooSandbox (2020) is used to run the PE files and gather execution logs. It executes each PE file inside virtual machines and uses API hooks to monitor the API call sequence.

CuckooSandbox (2020) is an open source tool for automating analysis of files. User can upload files to the Cuckoo server. Then, in a matter of minutes, Cuckoo executes an uploaded file inside a realistic but isolated environment and employs API hooks to trace API calls of the file.

Fig. 5 shows the process of extracting the API sequences from the uploaded files. We upload the Windows PE samples in our dataset to the Cuckoo server that is installed with Ubuntu 16.04 LTS. We apply several virtual machines which are maintained as

Table 5
Hyperparameter search space and the best value to tune the model.

HyperParameter	Search Space	Best Value
Units of embedding layer in API Phrase	{8, 16, 32}	16
Units of embedding layer in Semantic Chain	{8, 16, 32}	8
Number of filters of convolution layers	{32, 64, 128}	128
Activation function of convolution layers	{relu, sigmoid, tanh}	relu
Units of Bi-LSTM layer	{100, 200}	100
Activation function of Bi-LSTM layers	{relu, sigmoid, tanh}	sigmoid
Units of the first dense layer	{64, 128}	64
Units of the second dense layer	{32, 64}	32
Activation function of dense layers	{relu, sigmoid, tanh}	relu
Rate of Dropout layers	{0.2, 0.5}	0.2
Learning rate	$\{10^{-4}, 10^{-3}, 10^{-2}\}$	10^{-3}

clients on Cuckoo server. All the virtual clients are installed with a Windows 7 64-bit system and several daily-use software to ensure the successful execution of the PE samples in the dataset. The snapshot feature of the virtual machine is leveraged to roll it back after execution to ensure the uniformity of software running environment. Besides, Cuckoo simulates some user actions (such as clicking a button, typing some texts, etc.) to trigger malicious behavior of malware. We also adopt some "age" mechanisms (Miramirkhani et al., 2017) to prevent sandbox evasion.

After a PE file is uploaded, Cuckoo server begins to call a free client to execute the file and record the API calls automatically. In this paper, we set the maximum running time of each sample to 5 min. That is to say, the sandbox process completes when the uploaded sample ends itself or runs to 5 min. When the process completes, Cuckoo server will generate a sandbox report about this uploaded file and an API sequence can be extracted from this report. Since the length of API sequence formed by distinct software is different, we use the first 1000 APIs that are not repeated continuously as the input sequence to the model. Specifically, when the same API is called several times continuously in a row it appears as it was only invoked once. For example, for an original API sequence $\{\underline{a, a, a}, b, \underline{c, c}, d, a, \ldots\}$, it will be processed as $\{a, b, c, d, a, \ldots\}$, whose the first 1000 items will be input to the model.

5.2. Hyperparameter settings

There are many hyperparameters that need to be set in the model. Table 5 shows our hyperparameter search space and the hyperparameter values for the best classification performance. We use a grid search and conduct the 4-fold cross-validation on the training set to select the best hyperparameters. During the model training, we use the *EarlyStopping* which is ensured that the education is terminated without reaching the extreme fit of the model.

To conduct the 4-fold cross-validation, the malware and goodware training datasets are split into 4 equal-sized subsets. In each fold of the cross-validation, 3 subsets (75%) of the data are used for training a brand new model, and the rest subset (25% of the data), different in each fold, is used to evaluate the resultant model. For a particular set of model parameters, when all 4 training-validation folds are finished, we compute the final validation loss by averaging the 4 validation losses over the 4 folds. The final validation loss is treated as the score of this model and is further used as the criterion for comparing with different hyperparameter settings. Finally, we choose the best model with the minimum average validation loss.

5.3. Evaluation metrics

The confusion matrix is applied to evaluate the performance of our proposed model. We calculate accuracy, precision, recall, and F1-score (Eqs. (18)–(21)) to assess our detection performance and make comparisons. The receiver operating characteristic (ROC) curve and area under curve (AUC) score are also used to compare with baseline models.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{18}$$

$$Precision = \frac{TP}{TP + FP} \tag{19}$$

$$Recall = \frac{TP}{TP + FN} \tag{20}$$

$$F1 - score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{21}$$

5.4. Baselines

To investigate the performance improvement, the proposed model is compared with several baseline models. All the baselines focus on API-based malware analysis. The baseline models can be divided into two groups: frequency statistics based ML models and sequence encoding based models.

5.4.1. Frequency statistics based ML models

Some machine learning models including Naive Bayes (NB), K-Nearest Neighbor (KNN), Decision Tree (DT), Random Forest (RF), Support Vector Machine (SVM) and eXtreme Gradient Boosting (XGBoost) are widely used in frequency statistics based methods (Alazab et al., 2011; Fan et al., 2018; Lin et al., 2018; Sami et al., 2010). In this paper, we apply these ML models to verify the experimental performance of the frequency statistics based methods. The frequency of each API calls in each software sample are input to these ML models.

5.4.2. Sequence encoding based models

Several sequence encoding based studies are applied as the second group of our baselines. All these works focus on API sequence based malware detection and try to represent some intrinsic features. The baselines are as follows:

- Kolosnjaji et al. (2016): This approach combines convolutional neural networks with recurrent neural networks to represent software behavior and relationship information. The input is one-hot vectors for each API name.
- Tran and Sato (2017): They apply paragraph vector to represent software behavior and give weights to each API using TF-IDF. Then the multiple ML models including SVM, KNN, MLP and RF are applied to analyze the processed API sequence.
- Kim (2018): They group every API sequence into various n-grams to characterize software behavior and weight them with TF-IDF. They apply multinomial NB and linear SVM to do malware detection.

Table 6
Comparisons with the frequency statistics based ML models.

Method	Accuracy	Precision	Recall	F1-score
Naive Bayes (NB)	0.6428	0.7307	0.4096	0.5249
K-Nearest Neighbor (KNN)	0.8846	0.8985	0.8576	0.8775
Decision Tree (DT)	0.9078	0.8921	0.9198	0.9058
Random Forest (RF)	0.9284	0.8957	0.9636	0.9284
Support Vector Machine (SVM)	0.9045	0.8895	0.9154	0.9023
eXtreme Gradient Boosting (XGBoost)	0.9302	0.9118	0.9467	0.9289
Proposed Model	**0.9731**	**0.9617**	**0.9835**	**0.9724**

- Agrawal et al. (2018): They extract one-hot vectors from the API call sequence and frequent n-gram vectors from the API arguments. The detection model consists of several stacked LSTMs.
- Çatak et al. (2020): This deep learning method applies an embedding layer and several LSTM layers to capture the relationship between APIs.
- Zhang et al. (2020): They utilize a feature hashing trick to encode the API name and arguments to represent the semantic information of APIs. They also use gated convolutional neural networks to transform the features and a Bi-LSTM layer to capture the relationship between APIs.

All the baselines are reimplemented against our dataset with the same detection model hyperparameters and settings from the original papers. Specifically, all the experiments are implemented on the API sequences extracted from the Cuckoo reports in our dataset. Note that the maximum time we set for the program to run is 5 min, which is greater than most of the baselines and ensures the more adequate monitoring of software behavior. Thus, the comparison experiment is fair relative to the baseline settings.

6. Evaluation

In this section, we show our experiment results and make a comparison between our proposed model and baseline models. We also conduct ablation studies to analyze the contribution of each intrinsic feature to our model. We conduct some other experiments to demonstrate the effectiveness of our method to malware detection.

6.1. Comparisons with baselines

The proposed model has an accuracy of 0.9731, a precision of 0.9617, a recall of 0.9835, and an F1-score of 0.9724 on the test dataset. Table 6 shows that our proposed method outperforms ML baselines significantly, which indicates the feature mining of API sequence is not sufficient in frequency statistics based ML models. Fig. 6 shows ROC curves of different ML baseline models and the proposed model. It should be noted that the model RF as well as XGBoost achieve a quite good AUC by using ensemble learning methods. This indicates the importance of multiple features combination. Thus, we adopt a fusion strategy in multiple API sequence intrinsic features and achieve the AUC score of 0.9929.

The proposed method is compared with sequence encoding based models in Table 7. These methods try to use machine learning (ML) or deep learning (DL) to represent some intrinsic features including software behavior (SB), semantic information (SI), and relationship information (RI). By applying deep learning to represent and combine more intrinsic features of API sequence, our proposed model achieves the best performance. It should be notice that Agrawal et al. (2018) and Zhang et al. (2020) design a complex feature engineering about the API arguments to represent the semantic information of APIs. In this paper, we use the category, action, and operation object to represent the semantic information of

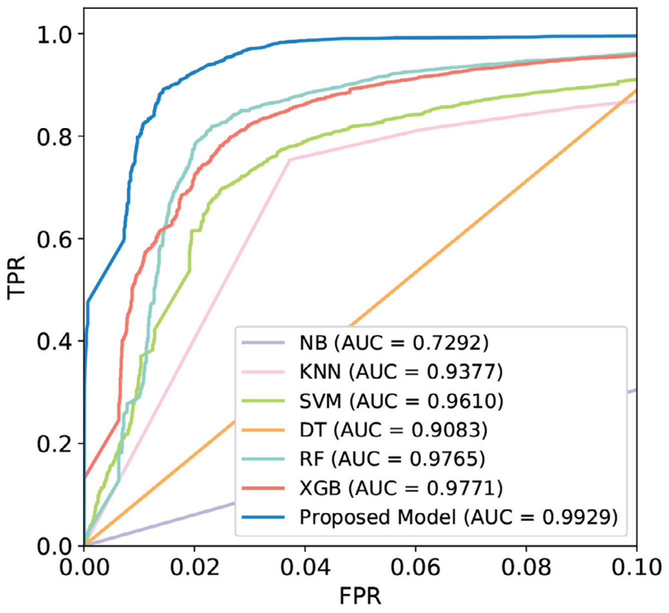

Fig. 6. ROC curve comparisons with the frequency statistics based ML models.

APIs instead of their arguments. Our proposed method doesn't require complex feature engineering operations about API arguments but achieve a good performance.

For time overhead, the proposed model takes about 25s per epoch in training stage under environment of one NVIDIA RTX 3090. In addition, the average inference time (including the time for API sequence processing and model prediction) of each sample is less than 100ms. Since the data collection using Cuckoo sandbox is time-consuming, and costs 2–5 min for most samples (over 90%), the time overhead of our analysis process is relatively small and acceptable.

6.2. Ablation studies

Our framework mainly handles 3 kinds of intrinsic features including software behavior (by API Phrase module), semantic information of each API (by Semantic Chain module), and the relationship between APIs (by Bi-LSTM module). In order to assess the contribution of each intrinsic feature, we remove each feature module from the model architecture and then evaluate the performance of the remaining model. Additionally, the structure of API Phrase module and Bi-LSTM module can be flexibly adjusted. We change the lengths of API phrases or the direction of LSTM to explore the performance with different model configurations.

As shown in Table 8, the lack of any part of the model will cause performance degradation. Fig. 7 shows ROC curves of different parts of the proposed model, and the AUC score will decrease after the removal of any module. That is to say, each intrinsic feature has a good effect on malware detection, which enables the model adapt to complex detection environments. It is worth not-

Table 7
Comparisons with sequence encoding based models.

Study	Method	Accuracy	Precision	Recall	F1-score	Intrinsic Features		
						SB	SI	RI
Kolosnjaji et al. (2016)	DL	0.9137	0.8895	0.9543	0.9208	√	-	√
Tran and Sato (2017)	ML	0.9531	0.9495	0.9620	0.9557	√	-	-
Kim (2018)	ML	0.9373	0.9523	0.9272	0.9400	√	-	-
Agrawal et al. (2018)	DL	0.9581	0.9459	0.9683	0.9570	-	√	√
Çatak et al. (2020)	DL	0.9552	0.9476	0.9682	0.9578	-	-	√
Zhang et al. (2020)	DL	0.9674	0.9591	0.9799	0.9693	-	√	√
Proposed Model	DL	**0.9731**	**0.9617**	**0.9835**	**0.9724**	√	√	√

Table 8
The ablation experimental results.

Method	Accuracy	Precision	Recall	F1-score
Without API Phrase module	0.9399	0.9247	0.9527	0.9385
Without Semantic Chain module	0.9503	0.9312	0.9683	0.9494
Without Bi-LSTM module	0.9581	0.9394	0.9760	0.9574
Proposed Model	**0.9731**	**0.9617**	**0.9835**	**0.9724**

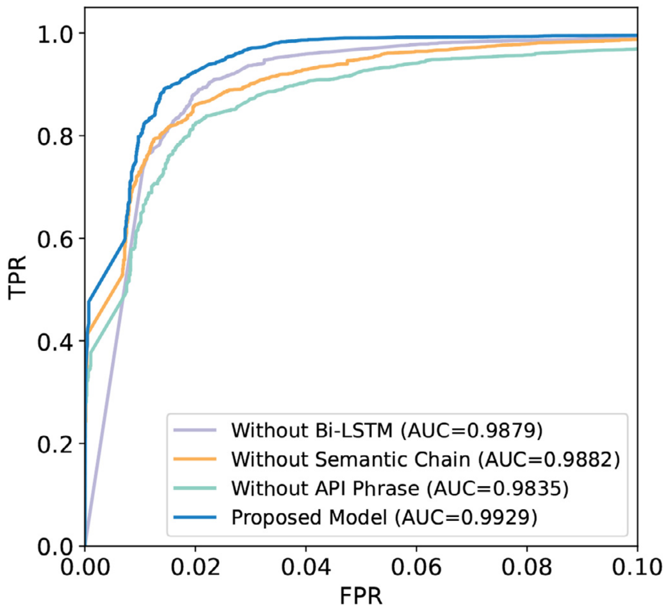

Fig. 7. ROC curve comparisons with different parts of the proposed model.

ing that the model without API Phrase has the worst performance, as the accuracy of the model reduces by 4% on the test set. Therefore, the API Phrase module has a greater impact on the proposed model. Moreover, the Semantic Chain module and Bi-LSTM module improve the F1-score by 2% - 3%. This indicates the importance of API semantic information and relationship extraction.

In order to explore the performance with different model configurations, we fix other structures and only change the lengths of API phrases or the direction of the LSTM. We also evaluate the impact of different MLP structures on detection performance. These experimental results serve as the basis for determining the structure of our final model.

6.2.1. *Lengths of API phrases*

For the proposed model, three parallel convolutional layers are applied to represent API phrases of length 3, 4, and 5. We fix other structures and only change the lengths of API phrases by using different number of convolutional layers with different filter size. Fig. 8 depicts the comparison across different numbers of convolutional layers. We conduct three sets of experiments: two convolution layers with filter size h of 3 and 4 ($h = 3, 4$), three convolution layers with filter size h of 3, 4, and 5 ($h = 3, 4, 5$), and four convolution layers with filter size h of 3, 4, 5, and 6 ($h = 3, 4, 5, 6$). Although the recall of $h = 3, 4, 5$ is similar to $h = 3, 4$, the other three indicators of $h = 3, 4, 5$ are improved. In addition, increasing the number of convolution layer from $h = 3, 4, 5$ to $h = 3, 4, 5, 6$ bring lower performance in accuracy and F1-score. Therefore, the structure of three convolution layers with filter size 3, 4 and 5 is finally applied in our model.

6.2.2. *Direction of LSTM*

The Bi-LSTM layer is applied to represents the relationship between APIs in the proposed model. In order to understand the impact of Bi-LSTM layer, we change or delete this layer and conduct three sets of experiments: without LSTM, undirectional LSTM, and Bi-LSTM. As shown in Fig. 9, the model with Bi-LSTM or undirectional LSTM converges faster and get better performance compared with the model without LSTM. This confirms that the performance of detector can be improved by using the relationship between API calls. Futhermore, the model with Bi-LSTM achieves the best performance, specially in accuracy and F1-score, which indicates the richer relationship information between API calls is helpful to improve the performance of the model.

6.2.3. *Structure of MLP*

The MLP classification module with dense layers and dropout layers is used to make a decision. We change the structure of MLP hidden layers (i.e., the number of *Dense+Dropout* layers) to explore its impact on the detection performance. As shown in Fig. 10, there are three sets of experiments: MLP with one *Dense+Dropout* layer, two *Dense+Dropout* layers, and three *Dense+Dropout* layers. These three sets of experiments have little difference in the final detection results. They all achieve about 97% accuracy and F1-score. However, the model with MLP of two *Dense+Dropout* hidden layers converges faster. In addition, increasing the number of hidden layers from 2 *Dense+Dropout* to 3 *Dense+Dropout* does not bring any performance improvement. Thus, we choose 2 *Dense+Dropout* as the hidden layers of MLP in our proposed model.

6.3. *Impact of proposed model*

To understand the source of the performance gain, we examine the impact of our proposed model. As shown in Fig. 11, according to our proposed model, the input API sequence first passes through the embedding layer, then carries out intrinsic feature encoding,

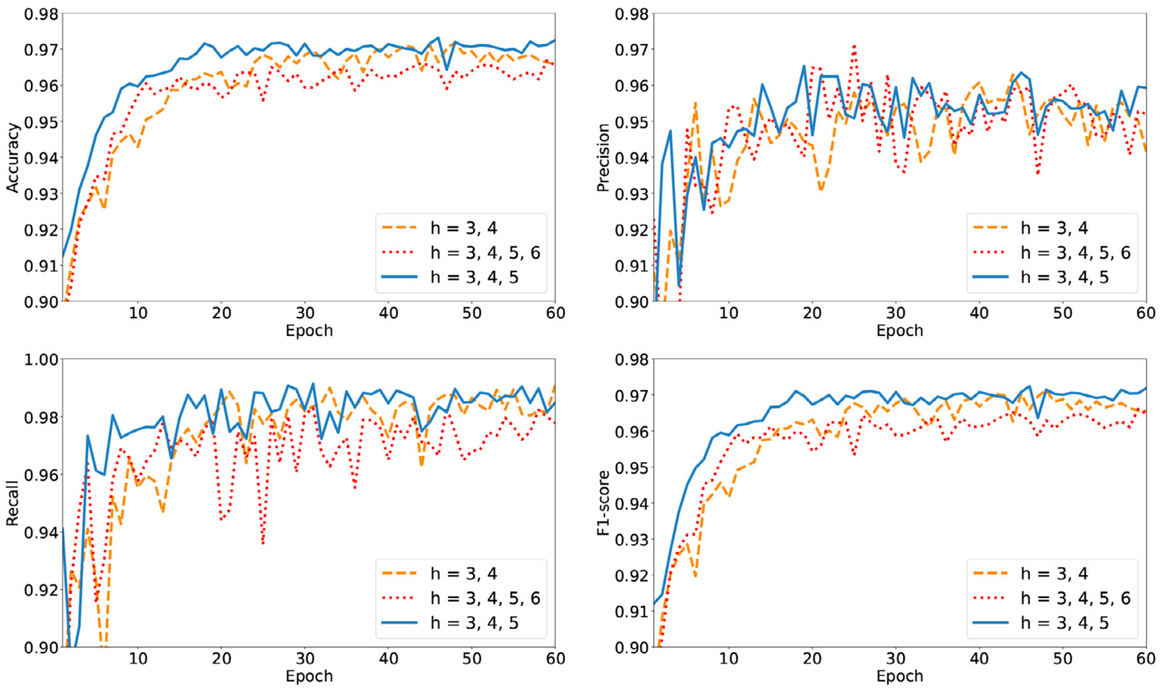

Fig. 8. Comparisons of Accuracy, Precision, Recall, and F1-score of API phrases of different lengths.

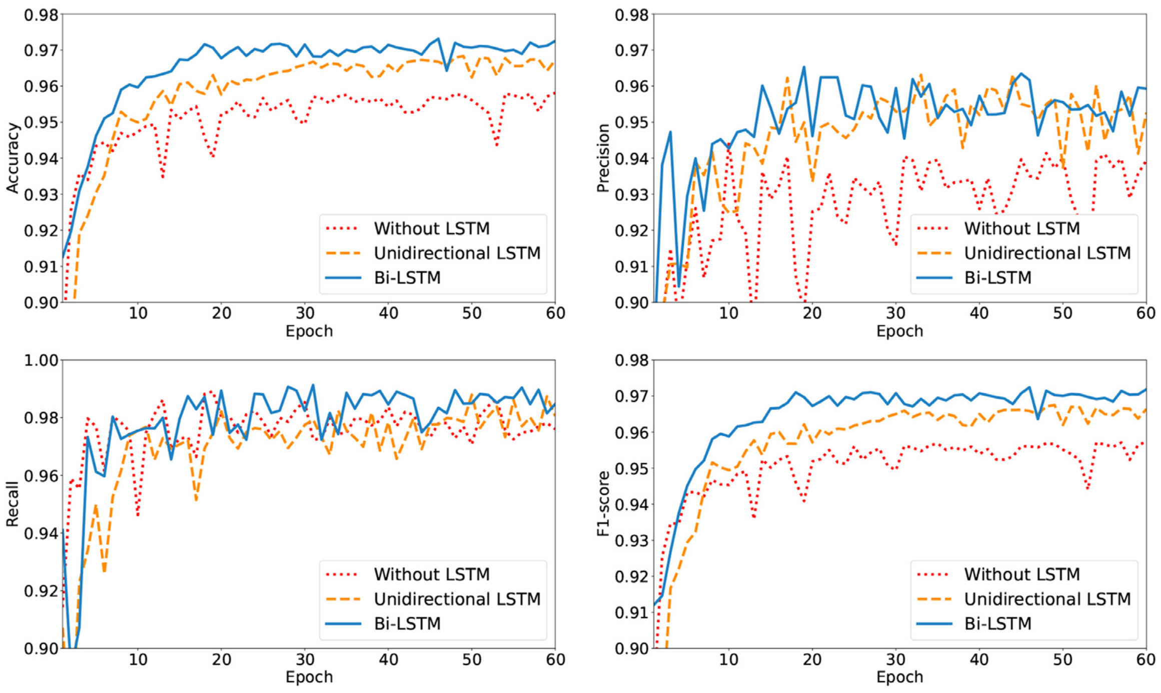

Fig. 9. Comparisons of Accuracy, Precision, Recall, and F1-score of the LSTM with different structures.

and finally passes through the MLP classifier. We define the feature space after the embedding layer (i.e., the embedding layer in the API Phrase module) as the original space. We also define two latent feature spaces: (1) *Latent Space 1* that contains the features after the intrinsic feature encoder and before the MLP. (2) *Latent Space 2* that contains the features after MLP's hidden layers and before the output layer. We present a visualization in Figs. 12–14, which shows the t-SNE plot of the test samples. T-SNE (Van der Maaten and Hinton, 2008) performs non-linear dimensionality reduction to project data samples into a 2-d plot. Compared with Figs. 12 and 13, we can observe that the samples in *Latent Space 1* have formed tighter clusters, making it easier to distinguish between benign software and malware. This indicates the intrinsic feature encoder captures the meaningful features of malware. As shown in Fig. 14, the boundary between malicious and benign clusters is clearer in *Latent Space 2*, which demonstrates the effectiveness of our proposed method for malware detection.

6.4. Longer term malware detection

To verify the effectiveness of our model in long-term malware detection, we widen the time window and establish two new test

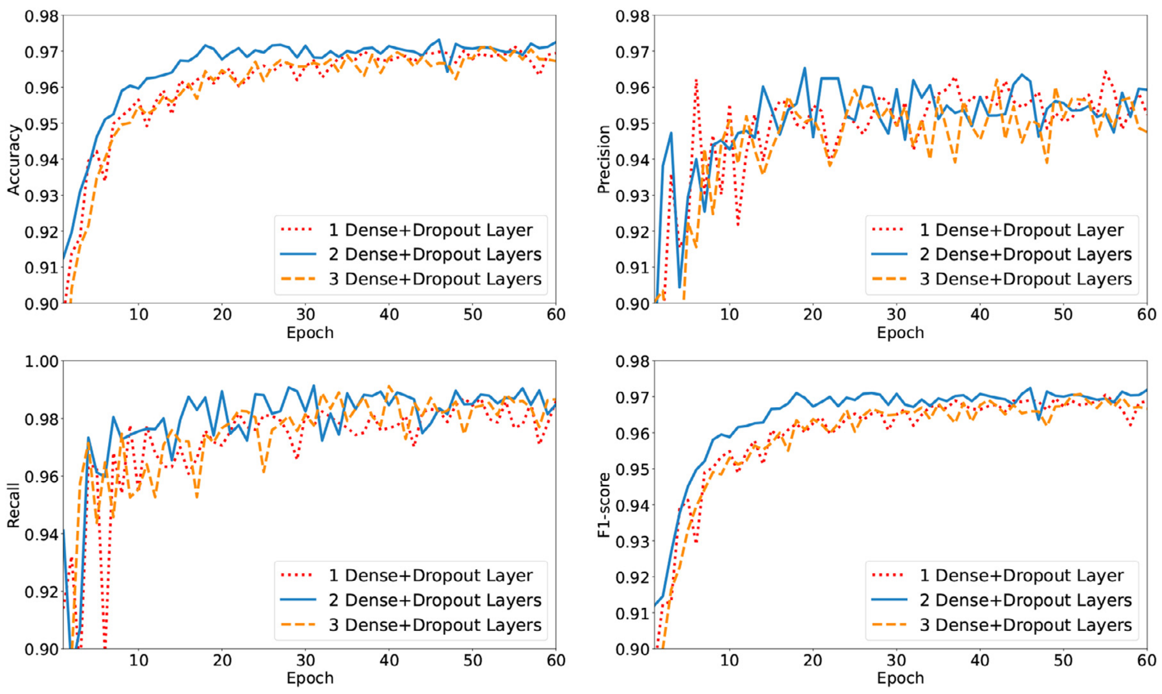

Fig. 10. Comparisons of Accuracy, Precision, Recall, and F1-score of the MLP with different layers.

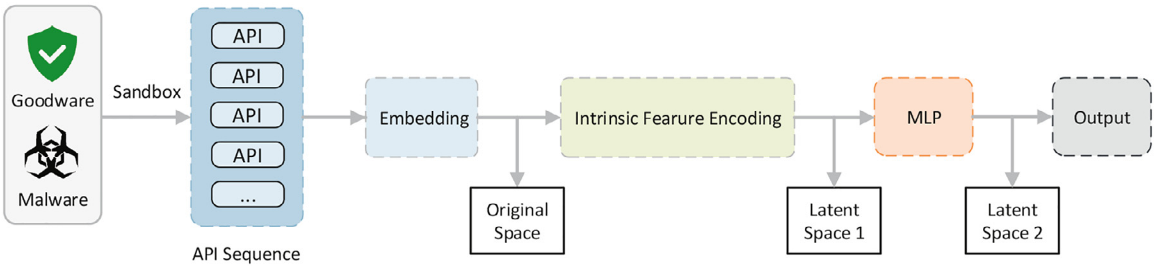

Fig. 11. Feature space definition.

Algorithm 1 GET ACTION AND OBJECT.

```
Input: API Name String $API_i$, Action Dictionary $AD$
Output: $action_i$, $object_i$
1: $action_i \leftarrow NULL$;
2: $object_i \leftarrow NULL$;
3: $ch \leftarrow GetLastCharacter(API_i)$;   // return the last character of $API_i$.
4: if $ch = 'W'$ or $ch = 'A'$ then
5:     $API_i \leftarrow RemoveLastCharacter(API_i)$;   // return the string left after removing the last character from $API_i$.
6: end if
7: for each word $ac \in AD$ do
8:     if $ac$ in $API_i$ then
9:         $action_i \leftarrow ac$;
10:        $object_i \leftarrow API_i.RemoveString(ac)$;   // return the string left after removing the string $ac$ from $API_i$.
11:        break;
12:    end if
13: end for
```

sets: (1) The 9211 malware and 10,074 goodware in the first half of 2020 (i.e., *Jan.-Jun. 2020*). (2) The 9631 malware and 12,213 goodware in the latter half of 2020 (i.e., *Jul.-Dec. 2020*). The experimental results are shown in Table 9. We notice that the F1-score of our model is greater than 90% on all three test sets that contain malware from July 2019 to December 2020, which indicates our proposed method has long-term effectiveness in detecting malware. Another interesting finding is that the performance of the detection model is gradually decreased over time. Compared with *Jul.–Dec. 2019*, the F1-score of the model decreases by about 2% on *Jan.–Jun. 2020*, and about 6% on *Jul.–Dec. 2020*. This is mainly because software tends to evolve over time, resulting in changes in data distribution (i.e., concept drift), which is beyond the scope of this paper.

7. Limitations and future work

In this section, we further discuss the limitations as well as our future work.

7.1. Other file types and operating systems

In this paper, we have only targeted Windows 7. We plan to explore the performance of our proposed methods on other newly released versions of Windows (i.e., Windows 8 and Windows 10). Furthermore, we plan to test the performance of our methods on other API-based operating systems such as Android platforms. In addition, all experiments are conducted using the Cuckoo sandbox. Therefore, this paper is limited to the list of Cuckoo's hooked API calls CuckooAPI (2015). In the future, we plan to design a run-time

Table 9
Longer term malware detection results.

Test Set	Malware	Goodware	Accuracy	Precision	Recall	F1-score
Jul.–Dec. 2019	10,454	11,245	0.9731	0.9617	0.9835	0.9724
Jan.–Jun. 2020	9211	10,074	0.9519	0.9433	0.9666	0.9548
Jul.–Dec. 2020	9631	12,213	0.9108	0.9017	0.9318	0.9165

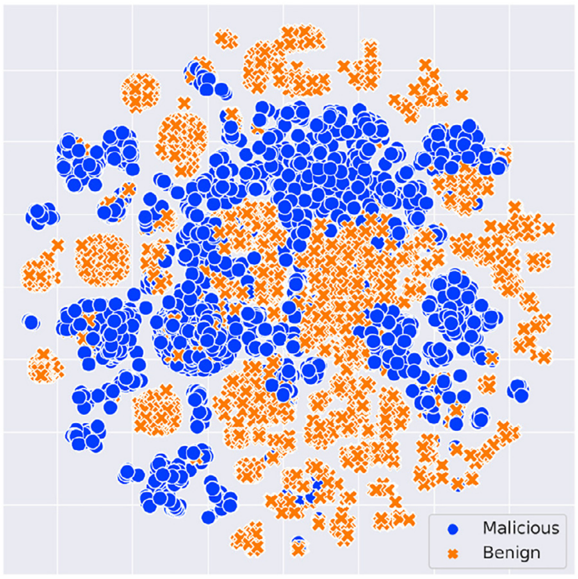

Fig. 12. T-SNE plot in the original space.

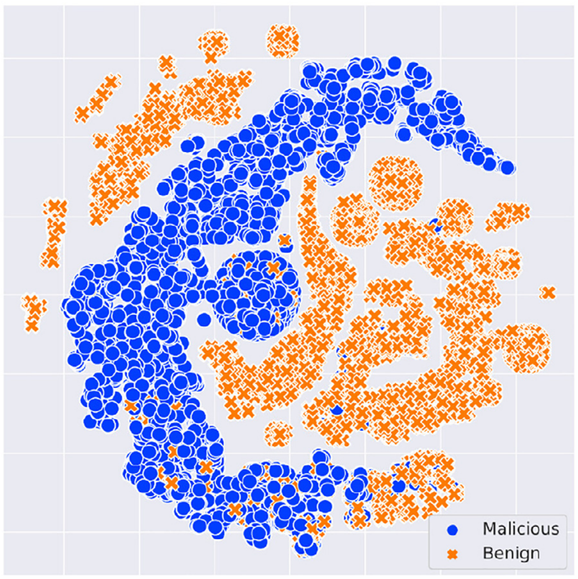

Fig. 14. T-SNE plot in the latent space 2.

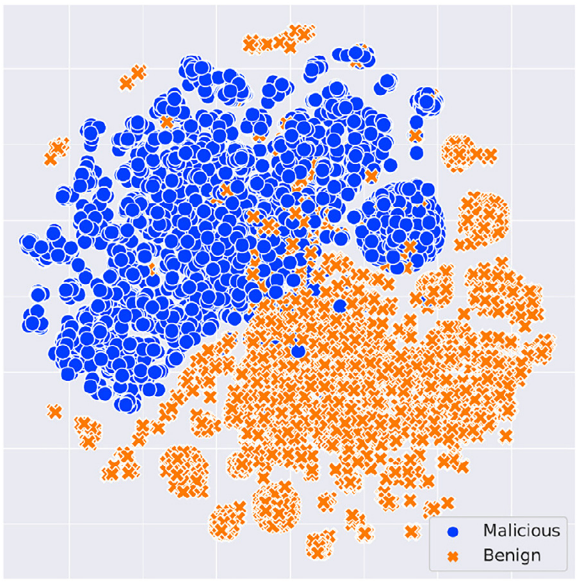

Fig. 13. T-SNE plot in the latent space 1.

malware analysis and detection tool that can extract the API calls simultaneously while the program is running.

7.2. Robustness to adversarial attacks

Recent studies have shown that many API sequence based malware detection methods are vulnerable to adversarial attacks (Rosenberg et al., 2020; 2018). Hence, in the future, we plan to use some methods like sequence squeezing (Rosenberg et al., 2019) to make our solution more robust against the adversarial attacks.

7.3. The problem of concept drift

Concept drift is the problem of the changing underlying relationships in the data. This will result in the decay of the prediction quality of malware detectors and classifiers over time as malware evolves and new variants appear (Pendlebury et al., 2019), which has been reflected in Section 6.4. In the future, we plan to modify our method and build a sustainable model for malware detection.

8. Conclusion

In this paper, we propose a novel malware detection framework based on API sequence intrinsic features. An intrinsic feature encoder consists of 3 modules is designed to represent and combine the intrinsic features of API sequence. The API Phrase module can well depict the actual software behaviors. Semantic Chain module expresses semantic information of the API calls. Bi-LSTM module is applied to capture the relationship between APIs. Based on the feature encoder, a deep neural network model is implemented to detect whether the software is malicious or not. We evaluate the model with a large real software dataset. The experiments show that our solution performs better than all the baselines. The ablation studies over multiple architectural changes prove the effectiveness of our intrinsic features and validates our model design decisions. Our proposed framework has a good performance in API sequence analysis and malware detection. Since all the samples in the training are temporally precedent to the testing ones, our model has a good ability for detecting unknown malware to a certain degree.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Ce Li: Conceptualization, Methodology, Software. **Qiujian Lv:** Writing – original draft, Visualization. **Ning Li:** Data curation, Investigation. **Yan Wang:** Supervision. **Degang Sun:** Resources, Validation. **Yuanyuan Qiao:** Writing – review & editing.

Acknowledgments

This research has been partially funded by the Institute of Information Engineering, Chinese Academy of Science, Project E0H0041, and is supported by the University of Chinese Academy of Science.

References

Agrawal, R., Stokes, J.W., Marinescu, M., Selvaraj, K., 2018. Neural sequential malware detection with parameters. In: Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP. IEEE, pp. 2656–2660. doi:10.1109/ICASSP.2018.8461583.
Ahmadi, M., Ulyanov, D., Semenov, S., Trofimov, M., Giacinto, G., 2016. Novel feature extraction, selection and fusion for effective malware family classification. In: Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy, CODASPY 2016, New Orleans, LA, USA. ACM, pp. 183–194. doi:10.1145/2857705.2857713.
Alazab, M., Venkatraman, S., Watters, P.A., Alazab, M., 2011. Zero-day malware detection based on supervised learning algorithms of API call signatures. In: Proceedings of the Ninth Australasian Data Mining Conference, AusDM 2011, Ballarat, Australia. Australian Computer Society, pp. 171–182.
Amer, E., Zelinka, I., 2020. A dynamic windows malware detection and prediction method based on contextual understanding of API call sequence. Comput. Secur. 92, 101760. doi:10.1016/j.cose.2020.101760.
Çatak, F.Ö., Yazi, A.F., Elezaj, O., Ahmed, J., 2020. Deep learning based sequential model for malware analysis using windows exe API calls. PeerJ Comput. Sci. 6, e285. doi:10.7717/peerj-cs.285.
Christodorescu, M., Jha, S., 2003. Static analysis of executables to detect malicious patterns. In: Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA. USENIX Association.
CuckooAPI. Hooked APIs and categories in cuckoo. https://github.com/cuckoosandbox/cuckoo/wiki/Hooked-APIs-and-Categories.
CuckooSandbox. Cuckoo sandbox automated malware analysis. https://cuckoosandbox.org/.
Damodaran, A., Troia, F.D., Visaggio, C.A., Austin, T.H., Stamp, M., 2017. A comparison of static, dynamic, and hybrid analysis for malware detection. J. Comput. Virol. Hacking Tech. 13 (1), 1–12. doi:10.1007/s11416-015-0261-z.
David, O.E., Netanyahu, N.S., 2015. Deepsign: deep learning for automatic malware signature generation and classification. In: Proceedings of the International Joint Conference on Neural Networks, IJCNN. IEEE, pp. 1–8. doi:10.1109/IJCNN.2015.7280815.
Ding, Y., Zhu, S., 2019. Malware detection based on deep learning algorithm. Neural Comput. Appl. 31 (2), 461–472. doi:10.1007/s00521-017-3077-6.
Elhadi, A.A.E., Maarof, M.A., Barry, B.I.A., Hentabli, H., 2014. Enhancing the detection of metamorphic malware using call graphs. Comput. Secur. 46, 62–78. doi:10.1016/j.cose.2014.07.004.
Fan, M., Liu, J., Luo, X., Chen, K., Tian, Z., Zheng, Q., Liu, T., 2018. Android malware familial classification and representative sample selection via frequent subgraph analysis. IEEE Trans. Inf. Forensics Secur. 13 (8), 1890–1905. doi:10.1109/TIFS.2018.2806891.
Galal, H.S., Mahdy, Y.B., Atiea, M.A., 2016. Behavior-based features model for malware detection. J. Comput. Virol. Hacking Tech. 12 (2), 59–67. doi:10.1007/s11416-015-0244-0.
Han, W., Xue, J., Wang, Y., Huang, L., Kong, Z., Mao, L., 2019. Maldae: detecting and explaining malware based on correlation and fusion of static and dynamic characteristics. Comput. Secur. 83, 208–233. doi:10.1016/j.cose.2019.02.007.
Hart, J.M., 1997. Win32 Systems Programming. Win32 Systems Programming.
Ketkar, N., Santana, E., 2017. Deep Learning with Python, 1. Springer.
Ki, Y., Kim, E., Kim, H.K., 2015. A novel approach to detect malware based on API call sequence analysis. Int. J. Distrib. Sens. Netw. 11, 659101:1–659101:9. doi:10.1155/2015/659101.
Kim, C.W., 2018. Ntmaldetect: a machine learning approach to malware detection using native API system calls. CoRRabs/1802.05412.
Kim, H., Kim, J., Kim, J., Kim, I., Chung, T., 2016. Feature-chain based malware detection using multiple sequence alignment of API call. IEICE Trans. Inf. Syst. 99-D (4), 1071–1080. doi:10.1587/transinf.2015CYP0007.
Kim, Y., 2014. Convolutional neural networks for sentence classification. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, EMNLP. ACL, pp. 1746–1751. doi:10.3115/v1/d14-1181.
Kolosnjaji, B., Zarras, A., Webster, G.D., Eckert, C., 2016. Deep learning for classification of malware system call sequences. In: Proceedings of the AI 2016: Advances in Artificial Intelligence - 29th Australasian Joint Conference, Hobart, TAS, Australia. Springer, pp. 137–149. doi:10.1007/978-3-319-50127-7_11.
Li, B., Roundy, K.A., Gates, C.S., Vorobeychik, Y., 2017. Large-scale identification of malicious singleton files. In: Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy, CODASPY. ACM, pp. 227–238. doi:10.1145/3029806.3029815.
Lin, Y., Lai, Y., Lu, C., Hsu, P., Lee, C., 2015. Three-phase behavior-based detection and classification of known and unknown malware. Secur. Commun. Netw. 8 (11), 2004–2015. doi:10.1002/sec.1148.
Lin, Z., Xiao, F., Sun, Y., Ma, Y., Xing, C., Huang, J., 2018. A secure encryption-based malware detection system. KSII Trans. Internet Inf. Syst. 12 (4), 1799–1818. doi:10.3837/tiis.2018.04.022.
Liu, W., Ren, P., Liu, K., Duan, H.-x., 2011. Behavior-based malware analysis and detection. In: Proceedings of the First International Workshop on Complexity and Data Mining. IEEE, pp. 39–42.
Van der Maaten, L., Hinton, G., 2008. Visualizing data using t-sne. J. Mach. Learn. Res 9 (11).
Miramirkhani, N., Appini, M.P., Nikiforakis, N., Polychronakis, M., 2017. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In: IEEE Symposium on Security and Privacy, San Jose, CA, USA. IEEE Computer Society, pp. 1009–1024. doi:10.1109/SP.2017.42.
Mylonas, A., Gritzalis, D., 2012. Practical malware analysis: the hands-on guide to dissecting malicious software. Comput. Secur. 31 (6), 802–803. doi:10.1016/j.cose.2012.05.004.
Nair, V.P., Jain, H., Golecha, Y.K., Gaur, M.S., Laxmi, V., 2010. Medusa: metamorphic malware dynamic analysis using signature from API. In: Proceedings of the International Conference on Security of Information & Networks.
Pascanu, R., Stokes, J.W., Sanossian, H., Marinescu, M., Thomas, A., 2015. Malware classification with recurrent networks. In: Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP. IEEE, pp. 1916–1920. doi:10.1109/ICASSP.2015.7178304.
Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L., 2019. TESSERACT: eliminating experimental bias in malware classification across space and time. In: Proceedings of the 28th USENIX Security Symposium, Santa Clara, CA, USA. USENIX Association, pp. 729–746.
Pichotta, K., Mooney, R.J., 2016. Learning statistical scripts with LSTM recurrent neural networks. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, Arizona, USA. AAAI Press, pp. 2800–2806.
Portableapps.com, URL https://portableapps.com/.
Qiao, Y., Yang, Y., Ji, L., He, J., 2013. Analyzing malware by abstracting the frequent itemsets in API call sequences. In: Proceedings of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). IEEE Computer Society, pp. 265–270. doi:10.1109/TrustCom.2013.36.
Raff, E., Nicholas, C., 2020. A survey of machine learning methods and challenges for windows malware classification. CoRRabs/2006.09271.
Rosenberg, I., Shabtai, A., Elovici, Y., Rokach, L., 2019. Defense methods against adversarial examples for recurrent neural networks. CoRRabs/1901.09963.
Rosenberg, I., Shabtai, A., Elovici, Y., Rokach, L., 2020. Query-efficient black-box attack against sequence-based malware classifiers. In: Proceedings of the ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA. ACM, pp. 611–626. doi:10.1145/3427228.3427230.
Rosenberg, I., Shabtai, A., Rokach, L., Elovici, Y., 2018. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In: Proceedings of the Research in Attacks, Intrusions, and Defenses - 21st International Symposium, RAID 2018, Heraklion, Crete, Greece. Springer, pp. 490–510. doi:10.1007/978-3-030-00470-5_23.
Salehi, Z., Sami, A., Ghiasi, M., 2017. MAAR: robust features to detect malicious activity based on API calls, their arguments and return values. Eng. Appl. Artif. Intell. 59, 93–102. doi:10.1016/j.engappai.2016.12.016.
Sami, A., Yadegari, B., Rahimi, H., Peiravian, N., Hashemi, S., Hamzeh, A., 2010. Malware detection based on mining API calls. In: Proceedings of the ACM Symposium on Applied Computing (SAC), Sierre, Switzerland. ACM, pp. 1020–1025. doi:10.1145/1774088.1774303.
Softonic.com, URL https://en.softonic.com/.
Sourceforge.net, URL https://sourceforge.net/.
Staudemeyer, R. C., Morris, E. R., 2019. Understanding LSTM - a tutorial into long short-term memory recurrent neural networks. CoRRabs/1909.09586.
Tian, R., Islam, M.R., Batten, L.M., Versteeg, S., 2010. Differentiating malware from cleanware using behavioural analysis. In: Proceedings of the 5th International Conference on Malicious and Unwanted Software, MALWARE, Nancy, France. IEEE Computer Society, pp. 23–30. doi:10.1109/MALWARE.2010.5665796.
Tran, T.K., Sato, H., 2017. Nlp-based approaches for malware classification from API sequences. In: Proceedings of the 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES).
Vasan, D., Alazab, M., Wassan, S., Safaei, B., Zheng, Q., 2020. Image-based malware classification using ensemble of CNN architectures (IMCEC). Comput. Secur. 92, 101748. doi:10.1016/j.cose.2020.101748.
VirusShare. Virusshare dataset, URL https://virusshare.com/.
VirusTotal. Virustotal scanner, URL https://www.virustotal.com/.

You, I., Yim, K., 2010. Malware obfuscation techniques: a brief survey. In: Proceedings of the Fifth International Conference on Broadband and Wireless Computing, Communication and Applications, BWCCA. IEEE Computer Society, pp. 297–300. doi:10.1109/BWCCA.2010.85.

Zhang, Y., Wallace, B.C., 2017. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. In: Proceedings of the Eighth International Joint Conference on Natural Language Processing, IJCNLP 2017, Taipei, Taiwan. Asian Federation of Natural Language Processing, pp. 253–263.

Zhang, Z., Qi, P., Wang, W., 2020. Dynamic malware analysis with feature engineering and feature learning. In: Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence, New York, NY, USA. AAAI Press, pp. 1210–1217.

Ce Li received bachelor from Ocean University of China, in 2019. He is presently pursuing the Ph.D. degree at School of Cyber Security, University of Chinese Academy of Sciences. His current research interests include mlaware detection and machine learning.

Qiujian Lv received Ph.D. degree from information and communication engineering from the School of Information and Communication Engineering, Beijing University of Posts and Telecommunications. She is now an associate senior engineer in the institute of information engineering, Chinese Academy of Sciences. Her research interests include anomaly detection and data mining.

Ning Li received master's degree from Beijing University of Posts and Telecommunications, Beijing, China. He is now an associate senior engineer in the institute of information engineering, Chinese Academy of Sciences. His research interests include anomaly detection, user behavior analysis, and network security operation and maintenance technology.

Yan Wang received master's degree from Beijing University Of Technology, Beijing, China. She is now an associate senior engineer in the institute of information engineering, Chinese Academy of Sciences. Her research interests include anomaly detection and network security situational awareness.

Degang Sun is a research professor, PhD supervisor of Institute of Information Engineering, Chinese Academy of Sciences. He is also a professor in the School of Cyber Security, University of Chinese Academy of Sciences. His research interests include electromagnetic leakage protection, wireless communication technology and high security level information system protection technology.

Yuanyuan Qiao received the BE degree from Xidian University in 2009. She is an assistant professor in the School of Information and Communication Engineering, BUPT. Her research focuses on broadband IP network, traffic measurement and classification, mobile Internet traffic analysis and cloud computing optimization and management.