applied sciences

MDPI

Article

A Malware Detection Framework Based on Semantic Information of Behavioral Features

Yuxin Zhang [1,2], Shumian Yang [1,2], Lijuan Xu [1,2], Xin Li [1,2] and Dawei Zhao [1,2,*]

[1] Key Laboratory of Computing Power Network and Information Security, Ministry of Education, Shandong Computer Science Center (National Supercomputer Center in Jinan), Qilu University of Technology (Shandong Academy of Sciences), Jinan 250014, China; 10431210661@stu.qlu.edu.cn (Y.Z.); yangshm@sdas.org (S.Y.); xulj@sdas.org (L.X.); lixin@sdas.org (X.L.)

[2] Shandong Provincial Key Laboratory of Computer Networks, Shandong Fundamental Research Center for Computer Science, Jinan 250014, China

* Correspondence: zhaodw@sdas.org

Abstract: As the amount of malware has grown rapidly in recent years, it has become the most dominant attack method in network security. Learning execution behavior, especially Application Programming Interface (API) call sequences, has been shown to be effective for malware detection. However, it is troublesome in practice to adequate mining of API call features. Among the current research methods, most of them only analyze single features or inadequately analyze the features, ignoring the analysis of structural and semantic features, which results in information loss and thus affects the accuracy. In order to deal with the problems mentioned above, we propose a novel method of malware detection based on semantic information of behavioral features. First, we preprocess the sequence of API function calls to reduce redundant information. Then, we obtain a vectorized representation of the API call sequence by word embedding model, and encode the API call name by analyzing it to characterize the API name's semantic structure information and statistical information. Finally, a malware detector consisting of CNN and bidirectional GRU, which can better understand the local and global features between API calls, is used for detection. We evaluate the proposed model in a publicly available dataset provided by a third party. The experimental results show that the proposed method outperforms the baseline method. With this combined neural network architecture, our proposed model attains detection accuracy of 0.9828 and an F1-Score of 0.9827.

Keywords: network security; dynamic analysis; API sequences; deep learning; malware detection

check for updates

Citation: Zhang, Y.; Yang, S.; Xu, L.; Li, X.; Zhao, D. A Malware Detection Framework Based on Semantic Information of Behavioral Features. *Appl. Sci.* **2023**, *13*, 12528. https://doi.org/10.3390/app132212528

Academic Editor: Ugo Vaccaro

Received: 18 October 2023
Revised: 8 November 2023
Accepted: 16 November 2023
Published: 20 November 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

A boom in computers and Internet technology has given rise to a growing amount of malware being developed. According to AV-TEST statistics (https://portal.av-atlas.org, accessed on 11 January 2022), over 120 million new malware samples were identified in 2021, a 36.5% surge from the previous year. As of 2022, the total amount of malware had reached a billion, and the quantity of new malware was growing and accelerating year by year, which made the network security situation more and more severe. In today's era, the construction of new infrastructure, such as artificial intelligence, 5G networks, and industrial Internet, has gradually become a hot spot for innovation (Zhao et al. [1]; Xu et al. [2]). Therefore, malware detection, especially for new variants that emerge, is essential in system and network security to prevent further threats to users (Han et al. [3], Korczynski et al. [4]; Xu et al. [5]).

The mainstream approach to detecting malware is divided into two main categories: static analysis, which analyzes the program directly, and dynamic analysis, which requires running the program before starting the analysis (Cesare et al. [6]; Galal et al. [7]). Static analysis methods usually refer to identifying malicious samples by examining the architecture or bytecode files of the program without running them (Ijaz et al. [8]; Zhao et al. [9]).

Normally, static analysis requires tools like IDA Pro and Stud_PE to extract static features, such as byte or opcode sequences, string information, etc. The characteristics obtained by static analysis can be compared with existing signatures saved in the database with malicious intent (Moser et al. [10]; Ye et al. [11]) or by machine learning models. However, those approaches can easily bypass fuzzing techniques (Burnap et al. [12]; Ucci et al. [13]) and are susceptible to interference in packing and deformation, leading to a decrease in accuracy. In addition, pattern matching methods can effectively and accurately detect known malware. However, they require a lot of manual experience for sample analysis and rule extraction, and require continuous updating of the signature database, which greatly limits the ability to detect new malware.

An approach to dynamic analysis uses data on the behavior of malware throughout its execution to address the shortcomings of static analysis. Dynamic analysis methods obtain behavior characteristics like file, network, registry, and process activities by running examples in a secure virtual environment known as a sandbox, which can capture some attributes that reflect behavioral intent. Those approaches can be used to identify similarities in the dynamic behavior of undiscovered malware (Bazrafshan et al. [14]). Therefore, it enables unknown malware detection, which is a capability that signature based approaches lack. Several studies (Cesare et al. [15]; Egele et al. [16]; Ki et al. [17]) have demonstrated in recent years that dynamic analysis can provide greater insight into how malware is generated and implemented than static analysis (Pektaş et al. [18]; Palumbo et al. [19]), and to a certain extent, can provide more reliable detection performance and is more resilient.

Behavioral features, especially Application Programming Interface (API) call sequences, can hold relevant information about programs and behaviors, because it offers access information to the primary resources in the kernel system. A large number of researchers have extracted patterns in API call sequences for malware detection and classification. Some research attempts to identify malware by association rule mining (Galal et al. [7]; Ding et al. [20]; Miao et al. [21]). However, with the growing quantity of malware and the emergence of new types of malware in recent eras (Shalaginov et al. [22]; Xu et al. [23]), the traditional approach to high-volume malware detection consumes a lot of human and computing resources and limits detection efficiency. Thus, more and more researchers are focusing on Machine Learning and Deep Learning. Some studies, for instance, have extracted characteristics from sequences of API calls and used Machine Learning algorithms that include Random Forests, K-Nearest Neighbors, and Support Vector Machines (Tran et al. [24]; Kim et al. [25]; Salehi et al. [26]), and Deep Learning algorithms, like Convolutional Neural Networks (Huang et al. [27]; Pascanu et al. [28]; Zhang et al. [29]) for malware detection. Deep learning algorithms have the advantage of being faster and more accurate than Machine Learning methods in automatically extracting malware features for detection, dramatically improving detection efficiency and increasing detection accuracy while promoting the development of malware detection technology. Moreover, inspired by the Natural Language Processing (NLP) techniques, which are also applied to detect malware (Kang et al. [30]; Amer et al. [31]; Wang et al. [32]), programs can be understood by the sequence of API calls representing the context of the calling program. Although there are many studies based on natural language processing, there are still some problems. For example, most studies only analyze semantic information and ignore the analysis of structural and statistical features.

In this paper, we propose an approach to detect malware with natural language processing and behavioral features, which uses word embedding techniques and feature weighting to learn the the characteristic patterns of API calls in malware. The API sequence feature that describes the context relationship between consecutive API calls and the name feature modeled according to the API name semantic structure information and statistical features. We implement the approach with a deep neural network, namely CNNs-BiGRU. Results illustrate that our proposed work better than other machine learning and deep learning models, achieving an accuracy of 0.9828 and an F1-Score of 0.9827 on the test set.

In summary, the following contributions are given in this paper.

- We propose an approach to malware detection that is based on natural language processing and behavioral features, which can achieve high accuracy just by analyzing API call sequences;
- We design a feature vectorization representation method based on the Skip-Gram algorithm for better learning of contextual behavior information. We demonstrate the results of the experiments and validate the experimental effect of the feature method for word vector representation;
- We design a method that combines API calls' semantic information and statistical characteristics. Numerous experiments verify the effectiveness of the approach we propose;
- We propose a CNNs-BiGRU based deep learning approach to implement malware detection and evaluate it. The results demonstrate that our model superior to other malware detection models.

The rest of this article is organized as follows: Existing research works related to malware detection is discussed in Section 2. Section 3 delineates the proposed approach. Section 4 gives the datasets and the obtained empirical results. Section 5 concludes this article and analyzes the limitations of this framework and future work.

2. Related Work

This section discusses studies of malware detection techniques based on static, dynamic, and natural language processing and deep learning-based techniques. The summary of malware detection systems are included in Table 1.

2.1. Static Analysis

The static analysis method has the advantages of high detection security, fast detection, and low detection overhead, because no samples are executed, but it is sensitive to obfuscation techniques.

Moskovitch et al. [33] proposed a malicious code classification methodology on the basis of opcode sequences, using N-gram sequences with different dimensions, 1-g, 2-g, ... and 6-g, to construct feature sets. Using different methods, document frequency, information gain and Fisher Score, they finally determined that the 2-g-based opcode sequences obtain better classification performance. This method solved the problem of low accuracy when the data were unbalanced. Shabtai et al. [34] focused on a detection method based on extracting static feature of malicious codes, then classified them by a machine learning model, achieved high accuracy while keeping a low false positive rate. The method extracted the instruction layer and byte-level features of malicious codes through the N-grams algorithm, trained them with multiple classifiers. An active learning mechanism, as well as a weighting algorithm, were used for the classification results of these classifiers to retain a high detection accuracy. However, this method was the result of combining several simple machine learning classification models, and the accuracy of detection needed to be improved. Chai et al. [35] proposed a dynamic prototype network (DPNSA) that can learn adaptively based on samples to detect malware. The method took images from fixed-size malware binaries and used them as input to the embedding module. Then, introduced the dynamic convolutional neural network, which employed the SE module in the convolution. They proposed a new activation function based on the dynamic activation function, i.e., the dual-sample dynamic activation. The distance between malware and query samples was calculated using a metric approach to achieve malware detection. Sami et al. [36] introduced data mining into malware detection. The method analyzed and extracted the API calls of PE files, then selected the most distinguishing API calls by Fisher score to improve the classification accuracy. Random forest was used for classification, which achieved an accuracy of 0.983. Although the method achieved good performance in an unbalanced dataset, handling unreachable code and fake API calls needed further improvement.

2.2. Dynamic Analysis

The dynamic analysis approach entails running the test program in a secure simulation environment as a way to capture the behavior of the program's execution. Activities like registry, API call, and network communications are examined to identify if the program is malicious.

Christodorescu et al. [37] used a data mining algorithm to capture the difference between malicious and benign programs from the dependency graph. This method was able to detect the obfuscated malware, but it was time consuming. Tobiyama et al. [38] used Long Short Term Memory (LSTM) to construct a behavioral model to transform APIs into feature vectors and generate feature images and then used Convolutional Neural Network (CNN) to perform malware detection. It was observed that API call sequences could cause performance loss if they were too long. However, the dataset used in this method is too small, and the model tends to be overfitted. Ndibanje et al. [39] dynamically obtained API calls in a virtual environment and performed similarity calculations on API sequences to distinguish different malware with similar behavioral contexts. Zhang et al. [40] obtained API-Graph to characterize the internal relations among entities by constructing API relationship diagrams. Then, they transformed the API in the relationship graph into the embeddings of Random Forests, Support Vector Machine (SVM), and Deep Neural Networks (DNNs), so that the semantic feature similarity was preserved even if the malware evolves. Rosenberg et al. [41] separated the original API sequence into several subsequences and used Recurrent Neural Network (RNN) to detect each subsequence separately. The whole sequence would be identified as malicious whenever a subsequence was marked as malicious. Zhang et al. [29] extracted features from API names and runtime parameters, used a hashing method to encode API names, categories, and parameters separately, and set different dimension sizes based on the feature characteristics. Then, these characteristics were further connected and fed into the CNN model and applied Bi-LSTM to capture the connections between APIs. The results showed that the method achieved an accuracy of over 0.96 with an AUC of 0.99, which is higher than other similar methods. Chen et al. [42] suggested a DNN-based detection of malware for learning parameter-enhanced API calls. For part of the API, a set of empirical rules containing four categories totaling more than 100 were constructed manually, together with statistical rules to identify if the parameter is malicious. Different categories were classified based on the GSDMM clustering algorithm for the rest of the calls to obtain their sensitivity labels.

2.3. Natural Language Processing and Deep Learning for Malware Detection

Recently, many studies have combined natural language processing with deep learning to apply it to feature processing of malicious code, using word embedding to vectorize the features.

David et al. [43] implemented malware classification by using deep belief networks (DBN) through a deep stack of denoising autoencoders. The sandbox-generated log files were transformed into fixed-size strings using the most common method in natural language processing, the 1-g approach. The commonly used 20,000 words were extracted to form a dictionary, and then a binary vector was produced by checking which of these words existed for each text sample. Arzu et al. [44] designed a framework that used a weighted random walk method to obtain a sequence of opcodes and performed feature learning on the opcode graph. After that, a vectorized representation was performed using Word2Vec. By applying the random walk method for edge and node selection to avoid handling lengthy operation code sequences, a vector space consisting of low-dimensional sequential opcode embeddings was constructed with extremely high accuracy. Kang et al. [30] employed an LSTM-based word embedding method, combining the strengths of signature-based and heuristic approaches, to analyze opcode and API names in fewer dimensions and to made an evaluation of malware detection and differentiation. Experimental results show that the time for learning was about 10 min faster and the performance was improved by about 0.5% compared to the One-Hot encoding method when Word2Vec was used.

Liu et al. [45] considered that consecutive repetitive API calls were unnecessary, so the same APIs that appeared more than twice in a row in the collected API sequences were culled, and the API sequences were sorted according to the process call time to extract the valid API sequences. Then, each API call is transformed into a unique binary vector using Word2Vec, and here they set the vector length to 50. Finally, by comparing the detection accuracy of GRU, BGRU, LSTM, BLSTM, and SimpleRNN, the highest accuracy of malware detection is determined by using BLSTM neural network. Amer and Zelinka [31] used a Markov chain approach to the detection of malware. The semantic similarity of each API in benign and malicious software was calculated in the initialization phase, and the APIs were clustered using K-means. After setting the API to its cluster index, a sequence of transfer matrices was constructed. They determined whether the sequence was malicious by calculating the API sequence's transfer probability. Wang et al. [32] presented a SIMPLE model to identify novel malware families with limited samples, which significantly solved the problems of sample scarcity and dynamic identification. They took the original API sequence using N-gram to enhance the local feature representation, and word embedding was applied to initialize the embedding matrix.

Table 1. Summary of malware detection systems.

Paper	Year	Analysis Type	Feature Info	Method
Moskovitch et al. [33]	2008	Static	N-grams	ML
Shabtai et al. [34]	2009	Static	N-gram	ML
Sami et al. [36]	2010	Static	frequency characteristic	ML
Kang et al. [30]	2019	Static	semantic information	DL
Chai et al. [35]	2022	Static	malware images	DL
Arzu et al. [44]	2022	Static	semantic information	DL
Christodorescu et al. [37]	2007	Dynamic	dependency graph	Data Mining
David et al. [43]	2015	Dynamic	1-g	DL
Tobiyama et al. [38]	2016	Dynamic	relationship between API calls	DL
Kolosnjaji et al. [46]	2016	Dynamic	One-Hot	DL
Rosenberg et al. [41]	2018	Dynamic	frequency characteristic	DL
Ndibanje et al. [39]	2019	Dynamic	statistical information	ML
Liu et al. [45]	2019	Dynamic	semantic information	DL
Amer and Zelinka [31]	2020	Dynamic	Markov	DL
Zhang et al. [40]	2020	Dynamic	API-Graph	DL
Wang et al. [32]	2021	Dynamic	semantic information	DL
Zhang et al. [29]	2020	Dynamic	hash encoding	DL
Catak et al. [47]	2020	Dynamic	relationship between API calls	DL
Chen et al. [42]	2022	Dynamic	semantic clustering	DL
Li et al. [48]	2022	Dynamic	intrinsic features	DL

It is known from previous work that word vectors generated by malware can be used for malware detection, thus providing crucial insights for malware analysis. As far as we know, most deep learning-based malware detection methods work with static features, and only a few studies consider dynamic features. Moreover, most existing dynamic analysis methods only analyze a single feature or analyze the feature insufficiently, resulting in information loss and affecting accuracy. Therefore, our approach exploits CNNs using dynamic features to discover malware. Experimental findings demonstrate that the proposed detection method based on CNNs-BiGRU can efficiently detect malware.

3. Methodology

API calls usually describe the semantic execution of a program. The program's behavior can be described by analyzing the relationships between API calls and determining whether a file is malicious. Therefore, in this article, we represent the semantics of the procedure on the basis of API calls. In this section, a description of the working principle of our proposed malware detection method is presented. The system we propose is shown in Figure 1 and has three phases: data preprocessing phase, feature extraction

and generation phase, and model detection phase. We shall describe these phases in the following subsections in detail.

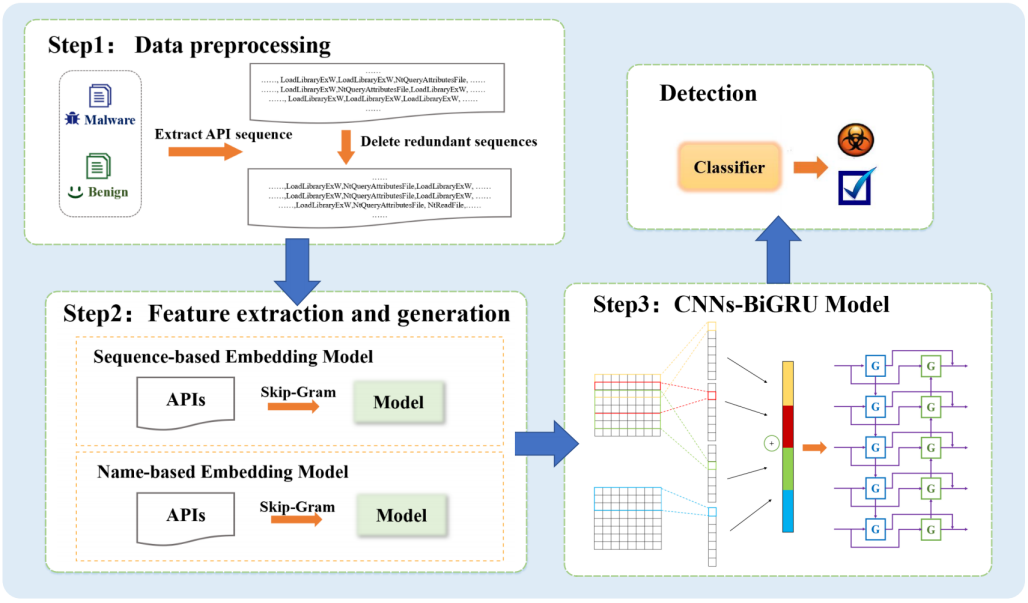

Figure 1. Overview of proposed model.

3.1. Data Preprocessing

Firstly, extract API call sequences from the datasets. After obtaining the API sequences, we find that there are many consecutive repetitive APIs, because a program usually calls some of the same APIs repeatedly when performing some loops or some file-related work. To improve performance, we remove those same call subsequences, which are considered redundant, and only keep those most distinguishing features. Since the convolutional neural network needs to guarantee that the input matrix has the same dimensionality while the length of API sequences in every sample are different, API sequences are treated uniformly for all samples. Here, the API sequence length is set to a fixed length, the APIs exceeding this length are truncated, and those with insufficient length are complemented by '0'.

3.2. Feature Extraction and Generation

3.2.1. Sequence-Based Embedding

Word embedding [49] is the representation of a word in an n-dimensional vector space. Within the context of malware detection, the sequence of calls to API in malware is not arbitrarily present. It reflects some contextual pattern of malicious activity. These patterns have similar manifestations across different malware. We can identify the contextual relationships among API function sequences by learning these patterns from a mount of malicious sequences.

Since the possible contextual relationships between API calls cannot be captured by statistical methods alone, inspired by word embeddings used in NLP, we initialize the embedding matrix with pre-trained word embeddings rather than random initialization to give semantic meaning to the word vectors. There are various models used in Word Embeddings, one of them is the Word2Vec model, which represents words as vectors based on several features they have, such as window size and vector size. Word2Vec captures the similarity values between words from the training of a large corpus. The resulting similarity values are computed from the word vector values then using the cosine similarity formula. Thus, similar words tend to have the same vector values and are grouped into the same block. Word2Vec is considered to be one of the most commonly used forms of word embeddings (Rezaeinia et al. [50]; Alami et al. [51]; Martinčić-Ipšićet al. [52]), which contains both CBOW and Skip-Gram models. The core idea of the CBOW model is to use

the context before and after a keyword to predict the probability of the occurrence of that keyword, as shown in Equation (1). Skip-Gram model is just the opposite; it is the use of a keyword to predict the probability of the words that appear before and after the keyword, and the model's target formula is shown in Equation (2). Unlike the CBOW model, the Skip-Gram will output multiple probability distributions from the hidden layer to the output layer.

$$P(\omega_c|\omega_{o_1},\ldots,\omega_{o_2m}) = \frac{exp\left(\frac{1}{2m}u_c^\top(v_{o_1}+\ldots+v_{o_2m})\right)}{\sum_{i\in v} exp\left(\frac{1}{2m}u_i^\top(v_{o_1}+\ldots+v_{o_2m})\right)} \quad (1)$$

$$P(\omega_o|\omega_c) = \frac{exp(u_o^\top v_c)}{\sum_{i\in v} exp(u_i^\top v_c)} \quad (2)$$

where ω_o, ω_c represent context words and center words, respectively, and u_o, v_c represent context word vectors and center vectors. Here, the window size is m and V is the whole lexicon.

In this paper, we use the Skip-Gram model (as shown in Figure 2), a shallow single-layer neural network, to capture the connection of hidden contexts between features and to efficiently model API sequence patterns. In this study, the pre-processed API call sequences are used as a corpus for training, which is trained by predicting the C words of the word's context by the target word to obtain the encoding of all APIs.

First, each API call is vectorized using the One-Hot algorithm, and each vector's length V is the amount of unique APIs counted. Row number of the element corresponding to the index position of a word in the vocabulary is 1, and other elements are 0. Then, word embedding is performed. According to the index mapping, each word is mapped to the N-dimensional space so that all words can be mapped to the matrix $W \in R^{V\times N}$, where every column in the matrix matches a word. Finally, the output is multiplied with the weight matrix $W' \in R^{N\times V}$, to acquire a probability distribution. The ultimate goal of model training is to obtain a weight matrix W, in which each row represents a lower dimensional vector of different APIs. In this way, the sequence r in which each API is expressed as a lower dimensional vector $\Phi(V_i)$. And for out-of-vocabulary (OOV) words, we use '0' as padding.

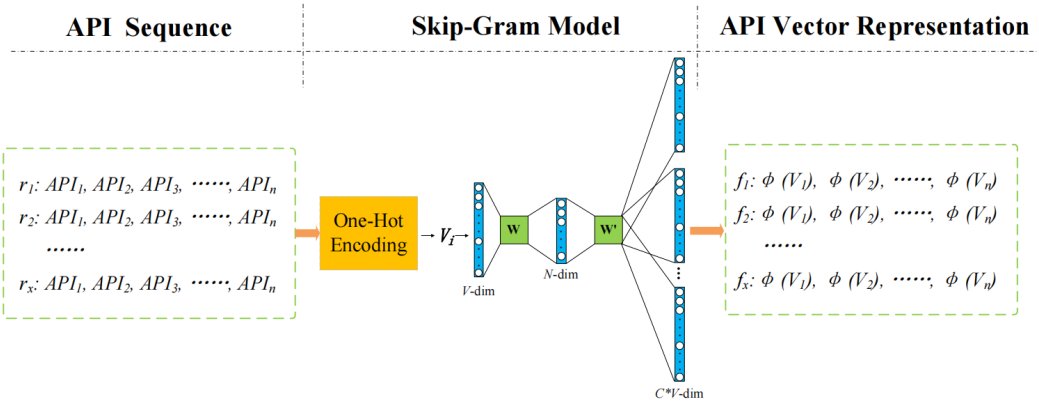

Figure 2. Word embedding model based on API sequence.

3.2.2. Name-Based Embedding

Since operating system developers usually adopt a semantic based naming approach for APIs (Hart et al. [53]), each API name string contains certain semantic information. In this chapter, an encoding method for feature representation on the basis of API names is defined.

An efficient way to describe the semantic information of API names was presented by Li et al. [48], who designed an 'action' dictionary to extract its operations from each call, and

the remaining part after extraction is used as the 'object' of its operations. We improve this by redescribing the API based on the feature representation of Li. Each API name consists of multiple words, and we extract from the API name the string that reflects the API operation and the object of the operation. The cuckoo sandbox (https://github.com/cuckoosandbox/cuckoo/wiki/Hooked-APIs-and-Categories, accessed on 11 January 2022) tracks a total of 312 API calls belonging to 17 categories. We obtain their categories based on the API classification criteria offered by the cuckoo sandbox, and divide the API calls into 18 categories, as shown in Equation (3). In this way, the action, object, and category for each API name are sequentially arranged to form a sequence, which is used as input to generate API name features with the same idea as API sequence embedding. Thus, the name embedding will be closed for APIs that exhibit related behaviors in the sequence.

$$\begin{aligned} category_i \in \{ &'notification', 'certificate', 'crypto', 'exception', 'file', 'iexplore', 'misc', \\ &'netapi', 'network', 'ole', 'process', 'registry', 'resource', 'services', \\ &'synchronisation', 'system', 'ui', 'other'\} \end{aligned} \tag{3}$$

TF-IDF, a statistical method, is a weighting technique frequently employed in text mining and information retrieval to estimate the significance of a word for documents in a corpus. If a word or phrase frequently appears in one document and seldom in others, it is deemed to possess excellent category differentiation ability and is appropriate for classification. We also analyze the TF-IDF values of each API, as shown in Equation (4), to incorporate statistical characteristics into the name embeddings.

$$TF - IDF = TF \times IDF = \frac{n_{i,j}}{\sum_k n_{k,j}} \times \log \frac{|D|}{|j : t_i \in d_j|} \tag{4}$$

where $n_{i,j}$ represents the occurrences of the word in document d_j, $n_{k,j}$ represents the overall number of occurrences of all words in document d_j. D refers to the overall count of documents included in the corpus, and j is the count of documents that contain the word.

We combine the word vectors of API name features obtained from the word embedding model with their statistical features, and they provide a vectorized representation of API names. Finally, we link the sequence-based embedding long with the name-based embedding into a complete embedding, which is illustrated in Figure 3. In this way, the embeddings generated for sequence-related APIs or name-related APIs are close, which enables the CNNs-BiGRU model to perceive the feature information of the APIs.

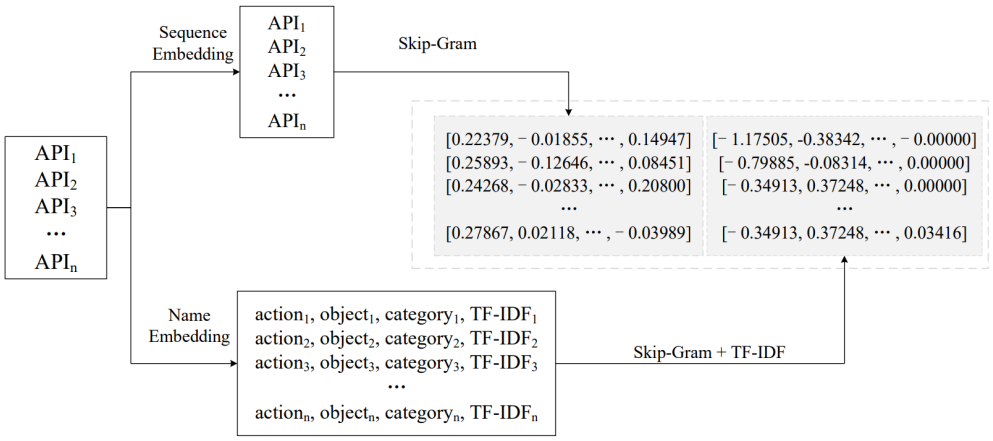

Figure 3. API embedded module.

3.3. CNNs-BiGRU Model

We apply the CNNs-BiGRU model to build a malware detection model, which includes a deep learning architecture consisting of convolutional layers with multiple convolutional

kernels of different sizes and bidirectional GRUs. Figure 4 shows a detailed description of the CNN-BiGRU layers used for malware detection.

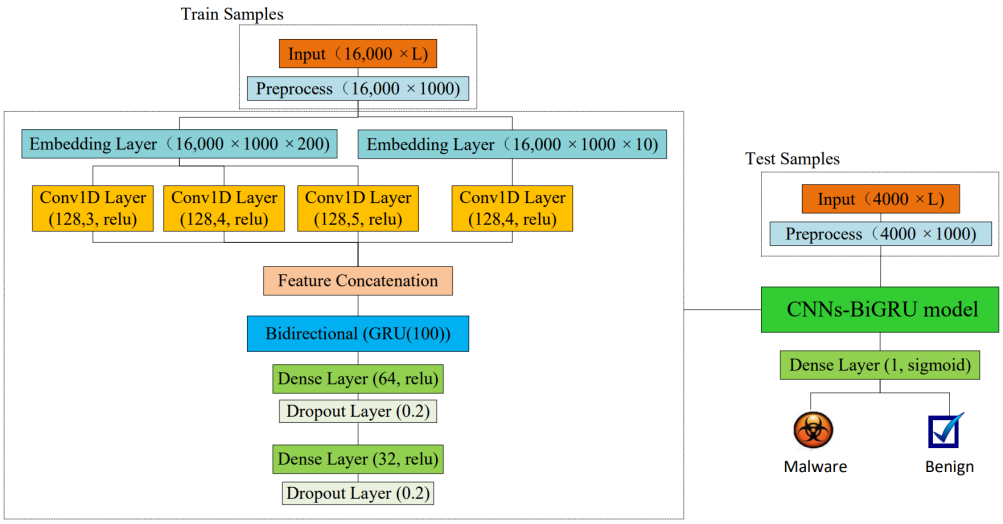

Figure 4. CNNs-BiGRU model architecture.

Input layer: This layer has malware or benign software samples for training and testing, it consists of a sample set of datasets with length varying between 0 and L as input. We select 80% of the dataset as training set in a randomized manner and keep the remaining 20% as test set with a data size of 20,000. The preprocessing block is a part of the input layer, where the length of the sequence of all the samples is uniformly processed. Here, it is set to a fixed length of 1000 to improve performance and reduce training time while maximizing the inclusion of useful data.

Embedding layer: The embedding layer converts each call in the API sequence into a vector representation. We determine the embedding vectors based on the word vector model with semantic information of the API sequence and the feature vector model consisting of API name features processed into the Skip-Gram model in Section 3.2. The output dimensions of the embedded layers are 200 and 10 to represent API calls of malware or benign files, respectively.

Convolution 1D layer: The convolution layer is the main building block of the CNN. We use multiple convolution kernels of different scales in parallel. Multi-layer convolution applies several convolution layers with different filter sizes to extract API calls of different lengths, where different lengths of sequences represent different granularity of behavioral features. The API sequence module sets up three parallel convolution layers with filter sizes of 3, 4, and 5, respectively, and the API name module applies an embedding layer and a convolution layer with filter size 4. The API name module applies an embedding layer and a convolutional layer with a filter size of 4. "Relu" is used as the activation function of the layer.

Bidirectional GRU layer: after concatenating all the outputs of the convolutional layer, the BiGRU module is applied to process the data in both the forward and reverse directions to capture the relationship between the API calls.

A gated recurrent neural network is a simplified and improved neural network for LSTM (Tang et al. [54]). For the gated recurrent unit, in order to reduce the parameters and tensor of the model and improve the information processing efficiency, the gated recurrent unit will replace the three gating units of the LSTM with the Reset gate (r_t) and the Update gate (z_t); thus, it is more concise and efficient than LSTM. The gates are formalized as Formulas (5) and (6). The GRU unit is shown in Figure 5.

The GRU obtains the information of the two gates through a current input x_t and the hidden state h_{t-1} passed down from the previous node.

$$Reset\ gate: r_t = \sigma(W_r \cdot [h_{t-1}, x_t]) \tag{5}$$

$$Update\ gate: z_t = \sigma(W_z \cdot [h_{t-1}, x_t]) \tag{6}$$

After the GRU model obtains the gating information, it needs to splice the reset data and the input x_i, and then use the tanh function to complete the activation task, so that the output from the hidden node can be obtained. The formula is shown in Equation (7).

$$\tilde{h}_t = \tanh(W \cdot [r_t h_{t-1}, x_t]) \tag{7}$$

Finally, in the "update" stage, the expression is shown in Equation (8).

$$h_t = (1 - z_t)h_{t-1} + z_t \tilde{h}_t \tag{8}$$

In the above formulas, W_r, W_z, W represent the corresponding weight matrix.

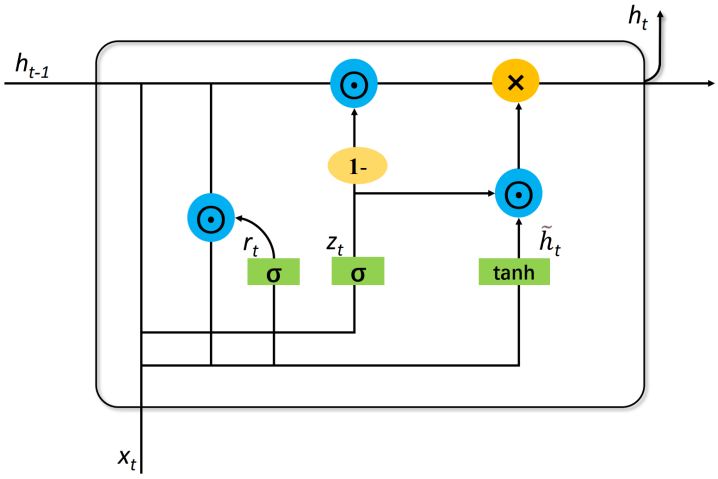

Figure 5. Gated Recurrent Unit (GRU).

The bidirectional GRU consists of two GRUs for processing sequential data. One of the GRU receives input sequence forward and the other receives backward, so that the bidirectional GRU is able to have the previous and future features in the sequence data learned. Bidirectional GRUs can be applied to data sequences to solve problems related to natural language processing. The advantage of the bidirectional model is that they produce greater accuracy while reducing the quantity of parameters necessary to train the model. The BiGRU network has fewer parameters than the Bi-LSTM, and the computation process is relatively simple and takes less time.

Dense layer: By using a dense connected layer, the features are further downscaled to 64 and 32 dimensions sequentially. The dense layer considers all the values in the bi-directional GRU output, applies the "Relu" activation function, and generates the output. We further use a dropout layer for regularization purposes.

Dropout layer: The dropout layer reduces the overfitting of the model to the training data by randomly deactivating some neurons with a certain probability. And, it makes the network more robust by forcing the model not to depend on specific neurons. We use multiple dropout layers in the proposed model to reduce the overfitting problem. Each dropout layer drops 20% (0.2) of the output of the previous layer.

The last dense layer of the activation function of the proposed model is adapted according to the classification of the expected results of the model. For malware detection,

the Sigmoid function is used and the output dimension 2 is chosen to predict whether a given file is malware or not.

4. Dataset and Evaluation Methods

Throughout this section, a description of the dataset and experimental setup employed in the experiment are given, and the results of our experiments are presented. We also show the comparison results with baseline models and the ablation studies to verify that each part of our method is effective in improving detection.

4.1. Experimental Design

4.1.1. Dataset

To verify our approach, we use a dataset generated through Sandbox and openly available in Github (https://github.com/kericwy1337/Datacon2019-Malicious-Code-DataSet-Stage1, accessed on 11 January 2022) provided by a third party. It contains execution traces of PE files with 20K entries in total. Among them, 10K are from malicious Windows PE files and the rest are benign. Figure 6 displays a snippet extracted from the XML file output generated after subjecting an executable (exe) file to the sandbox environment.


```
▼<action api_name="LoadLibraryExW" call_name="1.exe" call_pid="396" call_time="18:45:41.071" ret_value="0">
  ▼<apiArg_list count="3">
      <apiArg value="0"/>
      <apiArg value="0"/>
      <apiArg value="0"/>
    </apiArg_list>
  ▼<exInfo_list count="1">
      <exInfo value="C:\WINDOWS\system32\imm32.dll"/>
    </exInfo_list>
  </action>
▼<action api_name="NtQueryAttributesFile" call_name="1.exe" call_pid="396" call_time="18:45:41.077" ret_value="c0000034">
  ▼<apiArg_list count="2">
      <apiArg value="12eae8"/>
      <apiArg value="12eac0"/>
    </apiArg_list>
  ▼<exInfo_list count="1">
      <exInfo value="C:\program\LPK.DLL"/>
    </exInfo_list>
  </action>
▼<action api_name="LoadLibraryExW" call_name="1.exe" call_pid="396" call_time="18:45:41.080" ret_value="0">
  ▼<apiArg_list count="3">
      <apiArg value="0"/>
      <apiArg value="0"/>
      <apiArg value="0"/>
    </apiArg_list>
  ▼<exInfo_list count="1">
      <exInfo value="C:\WINDOWS\system32\lpk.dll"/>
    </exInfo_list>
  </action>
```


Figure 6. Example sample snippet.

Figure 7 shows the distribution of the extracted API sequence length for each file. The API sequence length of approximately 60% of all files was less than 1000, and the other lengths are scattered over a wide range and memory overflow occurs during training as the API sequence length increases. Therefore, in this study, the maximum length of API sequences was limited to 1000.

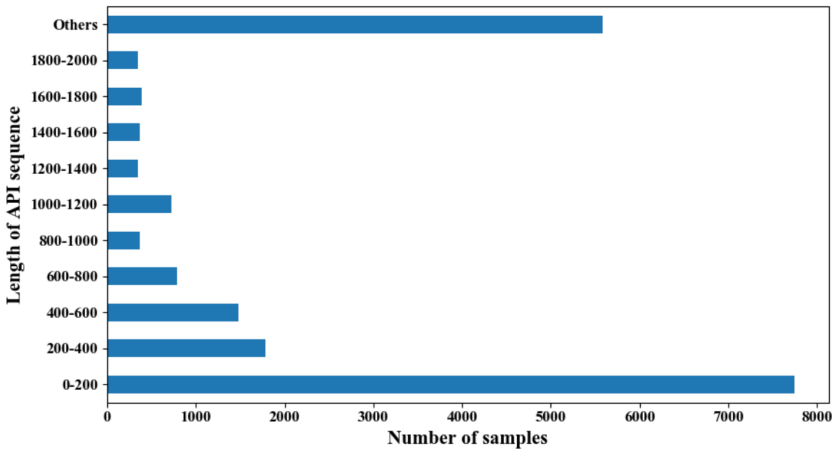

Figure 7. Distribution of API sequence lengths extracted from each file.

4.1.2. Experimental Setup

The proposed model is on a system with Intel Core i5-10210U. The experimental program is written in Python language and uses PyTorch 1.10.0 as the backend. The hyper-parameters in experiments are listed in Table 2. The experiment batch is to 64, and the training period is 80.

Table 2. Hyper-parameters in experiments.

Hyper-Parameter	Value
Window sizes in Skip-Gram model	5
The minimum count in Skip-Gram model	3
Units of embedding layer in Sequence-based model	200
Units of embedding layer in Name-based model	10
Activation function of convolution layers	Relu
Bi-GRU layer	1
Units of Bi-GRU layer	100
Units of the first dense layer	64
Units of the second dense layer	32
Activation function of dense layers	Relu
Dropout Ratio	0.2
Learning rate	10^{-3}

Table 3 shows the accuracy and F1-Score of detection with different embedding models and embedding vector size settings. We performed a comparison of multiple word embedding models with embedding vector sizes set to 100, 128, and 200, respectively. From the tabular data, it can be seen in the table that the accuracy of detection using the Skip-Gram model is always higher than the accuracy obtained with the CBOW model. In addition, the highest accuracy and F1-Score are obtained when the embedding vector size is set to 128.

Table 3. Various architectures' performance comparison based on the embedding model.

Embedding Model	Embedding Vector Size	Accuracy	Precision	Recall	F1-Score
CBOW	100	0.9770	0.9793	0.9743	0.9780
	128	0.9720	0.9724	0.9715	0.9719
	200	0.9715	0.9709	0.9719	0.9714
Skip-Gram	100	0.9775	0.9843	0.9706	0.9773
	128	0.9828	0.9870	0.9789	0.9827
	200	0.9755	0.9827	0.9680	0.9754

4.2. Evaluation Metrics

In our experiments, we select 80% data from the dataset for training, while keeping the remaining 20% for testing. We assess the performance of our proposed model depending on the following metrics, including accuracy, precision, recall, F1-Score, and Receiver Operating Characteristic (ROC) curves. The assessment indicators are calculated as shown in Equations (9)–(12).

$$Accuracy = \frac{TP+TN}{TP+FP+TN+FN} \tag{9}$$

$$Precision = \frac{TP}{TP+FP} \tag{10}$$

$$Recall = \frac{TP}{TP+FN} \tag{11}$$

$$F1-Score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{12}$$

4.3. Experimental Results

4.3.1. Comparison with Baselines

We assess the performance of three traditional Machine Learning algorithms, which are Naïve Bayes (NB), K-Nearest Neighbors (KNN), and Decision Tree (DT), and three Deep Learning models, which include BiLSTM, Text-CNN, and BiGRU. All algorithms measure results using accuracy, precision, recall, and F1-Score metrics.

As shown in Table 4, the performance of deep learning models is generally superior to that of the machine learning model, indicating that the features of API sequences cannot be fully exploited based on frequency features alone. The accuracy of the model proposed in this paper, namely CNNs-BiGRU, is 0.9828, precision is 0.9870, recall is 0.9789, and F1-Score is 0.9827 on the test dataset, which is the best performance among all models of machine learning and deep learning. This indicates multiple CNN networks can extract local features at different scales, local and higher-order features of the call can be better captured, and BiGRU models can use contextual information more effectively to increase the detection performance on the model.

Table 4. Comparison with baseline models.

	Accuracy	Precision	Recall	F1-Score
NB	0.8198	0.8518	0.7768	0.8126
KNN	0.9175	0.9556	0.8767	0.9145
DT	0.9178	0.9184	0.9180	0.9182
Text-CNN	0.9443	0.9587	0.9298	0.9440
BiLSTM	0.9743	0.9766	0.9723	0.9758
CNNs-BiGRU	0.9828	0.9870	0.9789	0.9827

Figure 8 plots the ROC curves [55] of various approaches over the test dataset, which represents the false positive rate (FPR) on the x-axis and the true positive rate (TPR) on the y-axis. Area under ROC curve is defined as AUC; the larger its value, the better the classifier works. It can be observed that our model obtains the best AUC score among all the baseline models of the test dataset, and the proposed model obtains a high TPR even with a low FPR.

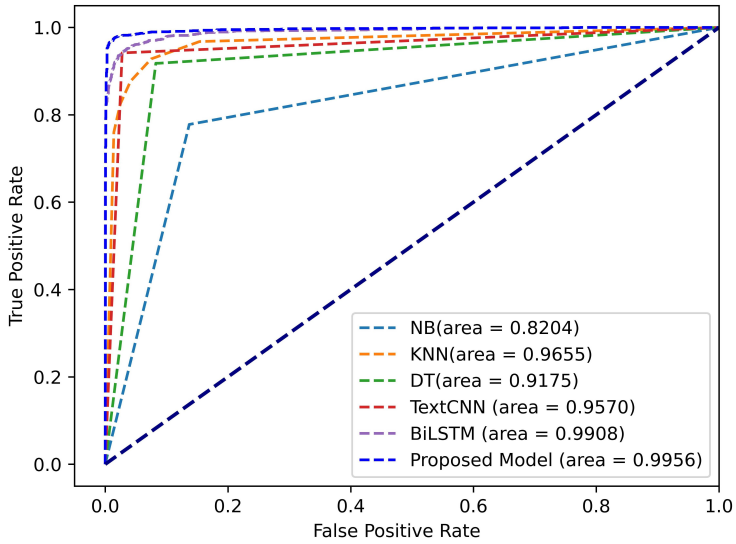

Figure 8. Comparisons of ROC curves of different models.

4.3.2. Ablation Studies

The proposed model is made up of several components that can be flexibly tuned, such as feature processing, CNNs, and BiGRU. To explore the effects of different configurations, we conduct several groups of comparative experiments by fixing other structures and

changing only the test components. The results of these experiments are the foundation for determining the structure of the final model.

- **Feature processing:** a set of experiments is used to verify the detection effectiveness of various feature processing methods, networks only the contextual semantic information about the sequence of API calls, networks only the semantic structure for the names of API calls and statistical features, and networks with both.
- **CNNs:** by a set of experiments to observe the model effects of API sequences of different lengths, two convolutional layers with the first filter stride of 3 and the second filter stride of 5; three convolutional layers with kernel strides of 3, 4, and 5; three convolutional layers with kernel strides of 3, 5, and 7; and four convolutional layers with kernel strides of 3, 4, 5, and 6.
- **BiGRU:** a set of experiments are to understand the impact of the pact of the BiGRU layer, the model has no BiGRU, only one unidirectional GRU, and BiGRU.

As shown in Figure 9, the absence of any part causes the model's performance to degrade, indicating that each feature we deal with has a will impact on our test model. The model that combines both features achieves the highest performance, especially with respect to accuracy and F1-Score, significantly better than the other two sets of experiments. It is significant to note that the module lacking information regarding the semantics of the API sequence obtained the worst performance, indicating the importance of semantic information about API call context.

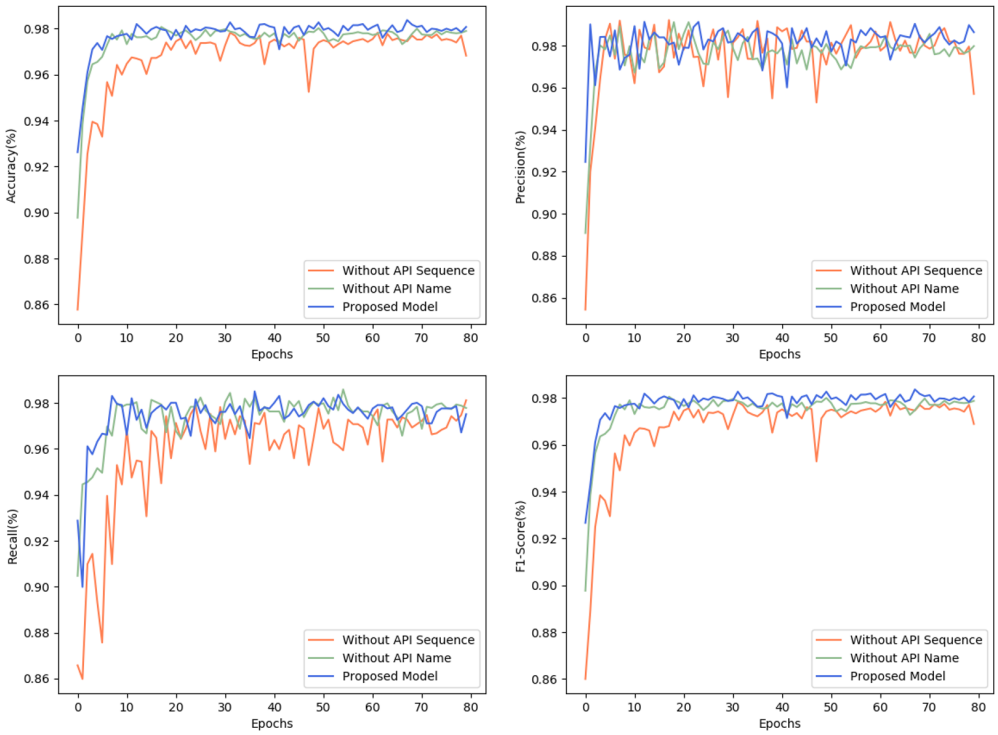

Figure 9. Comparison analysis of accuracy, precision, recall, and F1-Score of processing methods.

For the application of the number of convolutional layers, Figure 10 illustrates the performance for each configuration. The convergence is slower and the final performance is poorer when the convolutional layer size is 2. The model performance is higher when the number of convolutional layers is 3 than when it is 2. Among them, the patterns with filter sizes of 3, 4, and 5 have higher performance than those with filter sizes of 3, 5, and 7. In addition, the performance does not bring improvement as the convolution layer count grows from 3 to 4. Therefore, we chose a three-layer convolutional layer structure in our model with applied filter sizes of 3, 4, and 5.

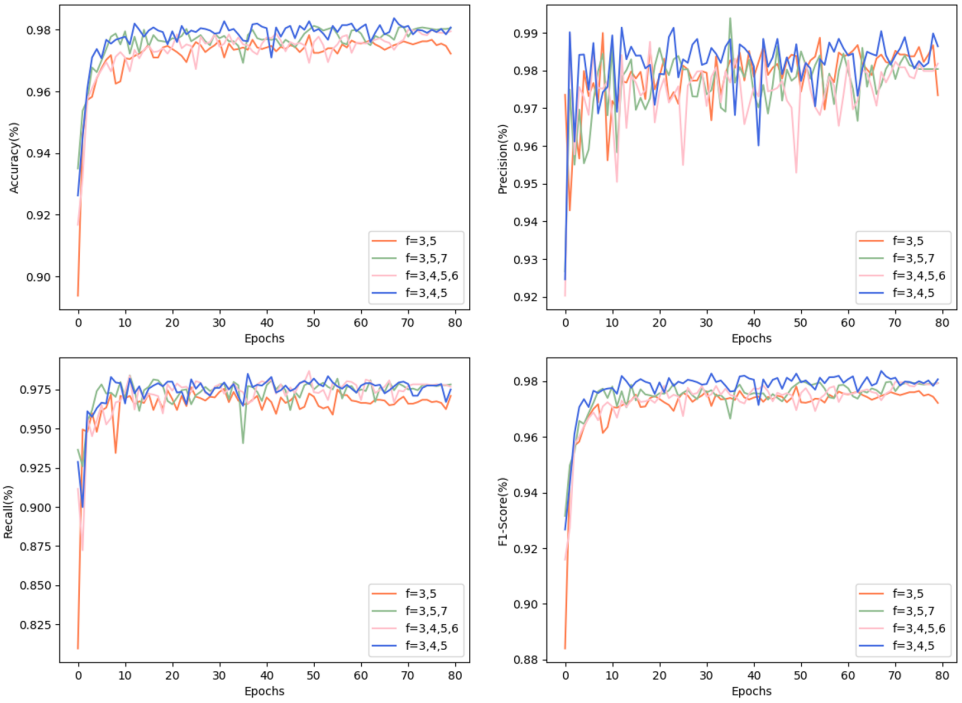

Figure 10. Comparison analysis of accuracy, precision, recall, and F1-Score of the number of convolutional layers.

Figure 11 illustrates a comparison of various numbers of GRUs. It is clear that BiGRU model has significantly better accuracy, recall, and F1-Score than the other two models, which indicates the importance of BiGRU for this detection model. It is noticeable that as the quantity of GRUs grows, the convergence of the model gradually accelerates and the performance improves, reflecting that learning the contextual relationship of API calls helps improve the model's performance.

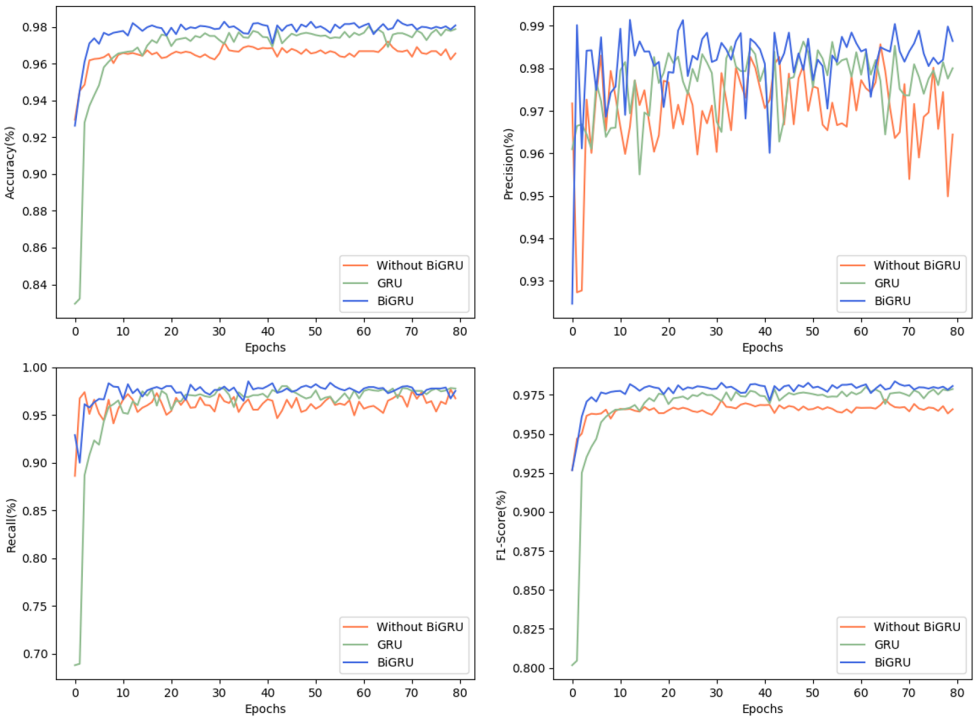

Figure 11. Comparative analysis of accuracy, precision, recall, and F1-Score of BiGRU model.

4.3.3. Model Evaluation

We compare the performance of the algorithms in this experiment with the performance of other state-of-the-art malware detection methods in the existing literature to better evaluate the method proposed in this work. Kolosnjaji et al. [46] built a detection model based on a sequence of system calls, transferring the deep learning over. A unique binary vector was encoded for each API from the dataset using One-Hot coding. A deep neural network with two convolutions and one recursive neural network were used for malware classification. Liu et al. [45] proposed a BLSTM-based malware detection method. The same APIs that appear more than twice in a row are removed and sorted by call time. Then, Word2Vec is transformed into a unique binary vector for each API call, and finally a BLSTM neural network is used to implement malware detection. Catak et al. [47] built a malware detection model using deep learning methods. The sequences of API calls they collected were detected with an embedding layer and a two-layer LSTM. Li et al. [48] presented an architecture for malware detection basing on the intrinsic features with API sequences. They designed a feature encoder consisting of three modules to express and assemble the inherent characteristics of API sequences. On the basis of the encoder, they implemented a deep learning model as a means of detecting whether a program is malware or not.

We replicate the above methodology and conduct experiments using the same dataset. We mainly consider four metrics including accuracy, precision, recall, and F1-Score.

As shown in Table 5, the model we proposed scores the highest in all metrics compared to all the other detection models. In comparison to the models based on the traditional approaches (Kolosnjaji et al. [46] and Catak et al. [47]), our proposed approach learns more features and the model performance is significantly more efficient, which shows the importance of learning both semantic and structural information of APIs through word embedding models. Liu et al. [45], using only semantic features, and Li et al. [48], using only structural features, both obtain good results, but neither is the best-performing method, suggesting that a combination of structural semantic information and statistical features of APIs can be used to learn API features more adequately.

Table 5. Various architectures' performance comparison based on the embedding model.

	Accuracy	Precision	Recall	F1-Score	Structure	Semantic
Kolosnjaji et al. [46]	0.9660	0.9569	0.9775	0.9671	✗	✗
Liu et al. [45]	0.9668	0.9763	0.9575	0.9668	✗	✓
Catak et al. [47]	0.9685	0.9720	0.9642	0.9681	✗	✗
Li et al. [48]	0.9738	0.9774	0.9712	0.9743	✓	✗
Proposed Model	0.9828	0.9870	0.9789	0.9827	✓	✓

5. Conclusions and Future Work

Malware has grown to be a common problem in computer security. Therefore, it is crucial to have an effective malware detection method to detect malware precisely. While many malware detection techniques have decent performance, some still need to improve in semantic information analysis. This work introduces a new approach to dynamic malware detection on the basis of behavioral features. First, a vectorized representation is performed on the basis of API call sequences, which is trained with the Skip-Gram model to acquire the contextual semantic information of APIs. Then, the API call names are encoded by analyzing the semantic information of the name structure and the statistical features. Finally, the embedded API sequences are fed into the CNNs-BiGRU network to train the detector for malware. Multiple sets of experiments demonstrate that this proposed model superior to all the proposed baselines and can effectively achieve malware detection, reaching an accuracy of 0.9828 and an F1-Score of 0.9827.

There are some potential problems with our framework, which may limit its practical application. For example, encoding API names manually suffers from a lack of flexibility. In order to improve the ability to learn semantic information with a high degree of flexibility, there is an urgent need to work on developing a more automated way of encoding API

names. This will help the system understand the semantic meaning of APIs more accurately. In addition, many recent studies have shown that API parameters can effectively improve malware detection. In the next step, we plan to analyze API parameters and further modify our approach to analyze malware more comprehensively. Some new malware variants emerge as new loopholes are discovered, which lead to concept drift, a phenomenon where the incoming test distribution deviates from the original training distribution [56]. That will result in the classifier's performance to start degrading over time. In the future, we will modify our model to develop a more robust feature space and adapt to the drift.

Author Contributions: Conceptualization, S.Y. and D.Z.; methodology, Y.Z.; software, Y.Z.; validation, S.Y. and L.X.; formal analysis, L.X. and X.L.; writing—original draft preparation, Y.Z.; writing—review and editing, D.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the Natural Science Foundation of Shandong Province (ZR2021MF132 and ZR2020YQ06), in part by the National Natural Science Foundation of China (62172244), in part by the National Major Program for Technological Innovation 2030-New Generation Artifical Intelligence (2020AAA0107700), in part by the Taishan Scholars Program (tsqn202211210), in part of by the Graduate Education and Teaching Reform Research Project of Shandong Province (SDYJG21177), in part by the Education Reform Project of Qilu University of Technology (2021yb63).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Restrictions apply to the availability of these data. Data was obtained from Github and are available from the authors with the permission of Github.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Zhao, D.; Xiao, G.; Wang, Z.; Wang, L.; Xu, L. Minimum dominating set of multiplex networks: Definition, application, and identification. *IEEE Trans. Syst. Man Cybern. Syst.* **2020**, *51*, 7823–7837. [CrossRef]
2. Xu, L.; Wang, B.; Wu, X.; Zhao, D.; Zhang, L.; Wang, Z. Detecting Semantic Attack in SCADA System: A Behavioral Model Based on Secondary Labeling of States-Duration Evolution Graph. *IEEE Trans. Netw. Sci. Eng.* **2021**, *9*, 703–715. [CrossRef]
3. Han, W.; Xue, J.; Wang, Y.; Huang, L.; Kong, Z.; Mao, L. MalDAE: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics. *Comput. Secur.* **2019**, *83*, 208–233. [CrossRef]
4. Korczynski, D.; Yin, H. Capturing malware propagations with code injections and code-reuse attacks. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017; pp. 1691–1708.
5. Xu, L.; Wang, B.; Yang, M.; Zhao, D.; Han, J. Multi-Mode Attack Detection and Evaluation of Abnormal States for Industrial Control Network. *J. Comput. Res. Dev.* **2021**, *58*, 2333–2349.
6. Cesare, S.; Xiang, Y.; Zhou, W. Control flow-based malware variantdetection. *IEEE Trans. Dependable Secur. Comput.* **2013**, *11*, 307–317. [CrossRef]
7. Galal, H.S.; Mahdy, Y.B.; Atiea, M.A. Behavior-based features model for malware detection. *J. Comput. Virol. Hacking Tech.* **2016**, *12*, 59–67. [CrossRef]
8. Ijaz, M.; Durad, M.H.; Ismail, M. Static and dynamic malware analysis using machine learning. In Proceedings of the 2019 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST), Islamabad, Pakistan, 8–12 January 2019; pp. 687–691.
9. Zhao, Z.; Yang, S.; Zhao, D. A New Framework for Visual Classification of Multi-Channel Malware Based on Transfer Learning. *Appl. Sci.* **2023**, *13*, 2484. [CrossRef]
10. Moser, A.; Kruegel, C.; Kirda, E. Limits of static analysis for malware detection. In Proceedings of the Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), Miami Beach, FL, USA, 10–14 December 2007; pp. 421–430.
11. Ye, Y.; Li, T.; Adjeroh, D.; Iyengar, S.S. A survey on malware detection using data mining techniques. *ACM Comput. Surv. (CSUR)* **2017**, *50*, 1–40. [CrossRef]
12. Burnap, P.; French, R.; Turner, F.; Jones, K. Malware classification using self organising feature maps and machine activity data. *Comput. Secur.* **2018**, *73*, 399–410. [CrossRef]
13. Ucci, D.; Aniello, L.; Baldoni, R. Survey of machine learning techniques for malware analysis. *Comput. Secur.* **2019**, *81*, 123–147. [CrossRef]
14. Bazrafshan, Z.; Hashemi, H.; Fard, S.M.H.; Hamzeh, A. A survey on heuristic malware detection techniques. In Proceedings of the 5th Conference on Information and Knowledge Technology, Shiraz, Iran, 22–24 May 2013; pp. 113–120.
15. Cesare, S.; Xiang, Y. *Software Similarity and Classification*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2012.

16. Egele, M.; Scholte, T.; Kirda, E.; Kruegel, C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv. (CSUR)* **2008**, *44*, 1–42. [CrossRef]
17. Ki, Y.; Kim, E.; Kim, H.K. A novel approach to detect malware based on API call sequence analysis. *Int. J. Distrib. Sens. Netw.* **2015**, *11*, 659101. [CrossRef]
18. Pektaş, A.; Acarman, T. Classification of malware families based on runtime behaviors. *J. Inf. Secur. Appl.* **2017**, *37*, 91–100. [CrossRef]
19. Palumbo, P.; Sayfullina, L.; Komashinskiy, D.; Eirola, E.; Karhunen, J. A pragmatic android malware detection procedure. *Comput. Secur.* **2017**, *70*, 689–701. [CrossRef]
20. Ding, Y.; Yuan, X.; Tang, K.; Xiao, X.; Zhang, Y. A fast malware detection algorithm based on objective-oriented association mining. *Comput. Secur.* **2013**, *39*, 315–324. [CrossRef]
21. Miao, Q.; Liu, J.; Cao, Y.; Song, J. Malware detection using bilayer behavior abstraction and improved one-class support vector machines. *Int. J. Inf. Secur.* **2016**, *15*, 361–379. [CrossRef]
22. Shalaginov, A.; Franke, K. Automated intelligent multinomial classification of malware species using dynamic behavioural analysis. In Proceedings of the 2016 14th Annual Conference on Privacy, Security and Trust (PST), Auckland, New Zealand, 12–14 December 2016; pp. 70–77.
23. Xu, L.; Wang, B.; Wang, L.; Zhao, D.; Han, X.; Yang, S. PLC-SEIFF: A programmable logic controller security incident forensics framework based on automatic construction of security constraints. *Comput. Secur.* **2020**, *92*, 101749. [CrossRef]
24. Tran, T.K.; Sato, H. NLP-based approaches for malware classification from API sequences. In Proceedings of the 2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES), Hanoi, Vietnam, 15–17 November 2017; pp. 101–105.
25. Kim, C.W. Ntmaldetect: A machine learning approach to malware detection using native api system calls. *arXiv* **2018**, arXiv:1802.05412.
26. Salehi, Z.; Sami, A.; Ghiasi, M. MAAR: Robust features to detect malicious activity based on API calls, their arguments and return values. *Eng. Appl. Artif. Intell.* **2017**, *59*, 93–102. [CrossRef]
27. Huang, X.; Ma, L.; Yang, W.; Zhong, Y. A method for windows malware detection based on deep learning. *J. Signal Process. Syst.* **2021**, *93*, 265–273. [CrossRef]
28. Pascanu, R.; Stokes, J.W.; Sanossian, H.; Marinescu, M.; Thomas, A. Malware classification with recurrent networks. In Proceedings of the 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), South Brisbane, QLD, Australia, 19–24 April 2015; pp. 1916–1920.
29. Zhang, Z.; Qi, P.; Wang, W. Dynamic malware analysis with feature engineering and feature learning. In Proceedings of the AAAI Conference on Artificial Intelligence, New York, NY, USA, 7–12 February 2020; Volume 34, pp. 1210–1217.
30. Kang, J.; Jang, S.; Li, S.; Jeong, Y.S.; Sung, Y. Long short-term memory-based malware classification method for information security. *Comput. Electr. Eng.* **2019**, *77*, 366–375. [CrossRef]
31. Amer, E.; Zelinka, I. A dynamic Windows malware detection and prediction method based on contextual understanding of API call sequence. *Comput. Secur.* **2020**, *92*, 101760. [CrossRef]
32. Wang, P.; Tang, Z.; Wang, J. A novel few-shot malware classification approach for unknown family recognition with multi-prototype modeling. *Comput. Secur.* **2021**, *106*, 102273. [CrossRef]
33. Moskovitch, R.; Feher, C.; Tzachar, N.; Berger, E.; Gitelman, M.; Dolev, S.; Elovici, Y. Unknown malcode detection using opcode representation. In Proceedings of the European Conference on Intelligence and Security Informatics, Esbjerg, Denmark, 3–5 December 2008; Springer: Berlin/Heidelberg, Germany, 2008; pp. 204–215.
34. Shabtai, A.; Moskovitch, R.; Elovici, Y.; Glezer, C. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *Inf. Secur. Tech. Rep.* **2009**, *14*, 16–29. [CrossRef]
35. Chai, Y.; Du, L.; Qiu, J.; Yin, L.; Tian, Z. Dynamic prototype network based on sample adaptation for few-shot malware detection. *IEEE Trans. Knowl. Data Eng.* **2022**, *35*, 4754–4766. [CrossRef]
36. Sami, A.; Yadegari, B.; Rahimi, H.; Peiravian, N.; Hashemi, S.; Hamze, A. Malware detection based on mining API calls. In Proceedings of the 2010 ACM Symposium on Applied Computing, Sierre, Switzerland, 22–26 March 2010; pp. 1020–1025.
37. Christodorescu, M.; Jha, S.; Kruegel, C. Mining specifications of malicious behavior. In Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Dubrovnik, Croatia, 3–7 September 2007; pp. 5–14.
38. Tobiyama, S.; Yamaguchi, Y.; Shimada, H.; Ikuse, T.; Yagi, T. Malware detection with deep neural network using process behavior. In Proceedings of the 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), Atlanta, GA, USA, 10–14 June 2016; Volume 2, pp. 577–582.
39. Ndibanje, B.; Kim, K.H.; Kang, Y.J.; Kim, H.H.; Kim, T.Y.; Lee, H.J. Cross-method-based analysis and classification of malicious behavior by api calls extraction. *Appl. Sci.* **2019**, *9*, 239. [CrossRef]
40. Zhang, X.; Zhang, Y.; Zhong, M.; Ding, D.; Cao, Y.; Zhang, Y.; Zhang, M.; Yang, M. Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual, 9–13 November 2020; pp. 757–770.
41. Rosenberg, I.; Shabtai, A.; Rokach, L.; Elovici, Y. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses, Crete, Greece, 10–12 September 2018; Springer: Berlin/Heidelberg, Germany, 2018; pp. 490–510.

42. Chen, X.; Hao, Z.; Li, L.; Cui, L.; Zhu, Y.; Ding, Z.; Liu, Y. CruParamer: Learning on Parameter-Augmented API Sequences for Malware Detection. *IEEE Trans. Inf. Forensics Secur.* **2022**, *17*, 788–803. [CrossRef]
43. David, O.E.; Netanyahu, N.S. Deepsign: Deep learning for automatic malware signature generation and classification. In Proceedings of the 2015 International Joint Conference on Neural Networks (IJCNN), Killarney, Ireland, 12–17 July 2015; pp. 1–8.
44. Kakisim, A.G.; Gulmez, S.; Sogukpinar, I. Sequential opcode embedding-based malware detection method. *Comput. Electr. Eng.* **2022**, *98*, 107703. [CrossRef]
45. Liu, Y.; Wang, Y. A robust malware detection system using deep learning on API calls. In Proceedings of the 2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), Chengdu, China, 15–17 March 2019; pp. 1456–1460.
46. 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, Hobart, TAS, Australia, 5–8 December 2016; Springer: Berlin/Heidelberg, Germany, 2016; pp. 137–149.
47. Catak, F.O.; Yazı, A.F.; Elezaj, O.; Ahmed, J. Deep learning based Sequential model for malware analysis using Windows exe API Calls. *PeerJ Comput. Sci.* **2020**, *6*, e285. [CrossRef]
48. Li, C.; Lv, Q.; Li, N.; Wang, Y.; Sun, D.; Qiao, Y. A novel deep framework for dynamic malware detection based on API sequence intrinsic features. *Comput. Secur.* **2022**, *116*, 102686. [CrossRef]
49. Ketkar, N.; Santana, E. *Deep Learning with Python*; Springer: Berlin/Heidelberg, Germany, 2017; Volume 1.
50. Rezaeinia, S.M.; Rahmani, R.; Ghodsi, A.; Veisi, H. Sentiment analysis based on improved pre-trained word embeddings. *Expert Syst. Appl.* **2019**, *117*, 139–147. [CrossRef]
51. Alami, N.; Meknassi, M.; En-nahnahi, N. Enhancing unsupervised neural networks based text summarization with word embedding and ensemble learning. *Expert Syst. Appl.* **2019**, *123*, 195–211. [CrossRef]
52. Martinčić-Ipšić, S.; Miličić, T.; Todorovski, L. The influence of feature representation of text on the performance of document classification. *Appl. Sci.* **2019**, *9*, 743. [CrossRef]
53. Hart, J.M. *Windows System Programming*; Pearson Education—Addison-Wesley Professional: San Francisco, CA, USA, 2010.
54. Tang, C.; Xu, L.; Yang, B.; Tang, Y.; Zhao, D. GRU-Based Interpretable Multivariate Time Series Anomaly Detection in Industrial Control System. *Comput. Secur.* **2023**, *127*, 103094. [CrossRef]
55. Invernizzi, L.; Miskovic, S.; Torres, R.; Kruegel, C.; Saha, S.; Vigna, G.; Lee, S.J.; Mellia, M. Nazca: Detecting malware distribution in large-scale networks. In Proceedings of the NDSS, San Diego, CA, USA, 23–26 February 2014; Volume 14, pp. 23–26.
56. Moreno-Torres, J.G.; Raeder, T.; Alaiz-Rodríguez, R.; Chawla, N.V.; Herrera, F. A unifying view on dataset shift in classification. *Pattern Recognit.* **2012**, *45*, 521–530. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.