electronics

Article

A Malicious Program Behavior Detection Model Based on API Call Sequences

Nige Li [1,2,*], Ziang Lu [1,2], Yuanyuan Ma [1,2], Yanjiao Chen [3] and Jiahan Dong [4]

1 State Grid Smart Grid Research Institute Co., Ltd., Nanjing 210003, China; luziang@geiri.sgcc.com.cn (Z.L.); mayuanyuan@geiri.sgcc.com.cn (Y.M.)
2 State Grid Laboratory of Power Cyber-Security Protection and Monitoring Technology, Nanjing 210003, China
3 College of Electrical Engineering, Zhejiang University, Hangzhou 310038, China; chenyanjiao@zju.edu.cn
4 State Grid Beijing Electric Power Research Institute, Beijing 100075, China; dongjiahan@bj.sgcc.com.cn
* Correspondence: linige@geiri.sgcc.com.cn

Abstract: To address the issue of low accuracy in detecting malicious program behaviors in new power system edge-side applications, we present a detection model based on API call sequences that combines rule matching and deep learning techniques in this paper. We first use the PrefixSpan algorithm to mine frequent API call sequences in different threads of the same program within a malicious program dataset to create a rule base for malicious behavior sequences. The API call sequences to be examined are then matched using the malicious behavior sequence matching model, and those that do not match are fed into the TextCNN deep learning detection model for additional detection. The two models collaborate to accomplish program behavior detection. Experimental results demonstrate that the proposed detection model can effectively identify malicious samples and discern malicious program behaviors.

Keywords: API call sequences; malicious programs; behavior detection

check for updates

Citation: Li, N.; Lu, Z.; Ma, Y.; Chen, Y.; Dong, J. A Malicious Program Behavior Detection Model Based on API Call Sequences. *Electronics* **2024**, *13*, 1092. https://doi.org/10.3390/electronics13061092

Academic Editors: Lanting Fang and Yubo Song

Received: 14 January 2024
Revised: 4 February 2024
Accepted: 14 February 2024
Published: 15 March 2024

1. Introduction

Compared to traditional power systems, the structure of the new power system is more intricate. The integration of new energy sources, such as renewable energy, into the power system and the growing number of distributed power sources connected to the grid from the user's side are posing greater challenges to the stability and security of the power system. Edge-side applications in the new power system are those that directly interact with power equipment, sensors, controllers, and other components. These applications play a crucial role in collecting, processing, transmitting, and controlling power data, significantly influencing the operational status, fault diagnosis, and dispatch control of the power system. Abnormal or malicious behaviors in edge-side applications, such as data tampering, command errors, or cyber-attacks, can lead to failures, damage, or even paralysis of the power system, with potentially severe socio-economic repercussions [1]. Consequently, real-time behavior detection of edge-side applications is essential to detect and prevent such adverse behaviors, ensuring the safe and stable operation of the new power system.

There are two primary approaches to analyzing malicious programs: static analysis and dynamic analysis [2]. Static analysis does not involve executing the code; instead, it assesses whether a program is malicious based on code attributes, control flow diagrams, function call diagrams, system call sequences, and other characteristics [3–7]. While static analysis offers the benefits of rapid execution and high efficiency, its capacity for behavioral analysis is notably limited. It struggles to detect obfuscation and encryption techniques frequently employed by malicious programs, such as packing, modifications to the PE header, and code obfuscation. In contrast, dynamic analysis evaluates the behavior of an executable file by running it. This method's advantage lies in its resilience to code obfuscation, shelling, and polymorphism, providing the most authentic representation of the program's behavior [8].

Behavior detection technology is a form of dynamic analysis that involves monitoring a program's behavior during execution and determining its maliciousness based on this behavior. An Application Programming Interface (API) provides a means for applications to interact with the system. The sequences of API calls can reveal the functionality and behavioral traits of an application, making them a crucial element in the detection of malicious code [9].

At present, the dynamic features of malicious programs are usually based on API call sequences, with machine learning employed for their detection. However, traditional machine learning methods face challenges in feature selection and struggle to address the anomaly detection issue in high-dimensional massive network traffic, resulting in a high false detection rate. In the detection of malicious programs, rule matching is a commonly used method characterized by its simplicity and low false detection rate. However, it fails to capture the deep relationships within API call sequences and does not leverage the temporal nature of APIs, and the temporal nature of APIs is not utilized, which leads to low accuracy of the model. Deep learning models, known for their robust learning and classification capabilities, have found applications in power systems [10]. These models can automatically extract features and categorize them based on these features, leading to enhanced performance. Consequently, many researchers have explored using deep learning to address the analysis of malicious programs. To ensure the stable operation and data security of edge-side applications in new power systems and to reduce the false positive rate of malicious program detection, we propose a malicious program behavior detection model based on API call sequences by integrating rule matching and deep learning. Initially, the API call sequence is processed to complete the preprocessing of the dataset. Subsequently, the PrefixSpan algorithm is employed to mine frequent API call sequences and establish a rule base for malicious behavior sequences. Finally, the behavior sequence matching model and the TextCNN [11] deep learning detection model are jointly utilized to detect test API call sequences, achieving program behavior detection.

The contributions of this study can be summarized as follows:

- We used the PrefixSpan algorithm to mine the frequent API call sequences from various threads within a malicious program. These sequences serve as a foundation for directly discriminating for malignant API execution actions, obtaining malicious behavior sequences, and constructing a malicious behavior sequence rule base.
- We utilized two models, the behavior sequence matching model and the TextCNN deep learning detection model, to collaboratively detect the tested API call sequences. Firstly, the tested API call sequences are input into the behavior sequence matching model to compare them against the malicious behavior sequence rule base. If a match is found, the sequence is deemed malicious; otherwise, it is passed to the TextCNN neural network model for further analysis to obtain the detection results, thereby enhancing the accuracy of the detection.

2. Related Work

Malicious programs represent a significant security threat to the Internet, with attackers leveraging them for profit through remote control, private information theft, and occasionally targeting network infrastructures. Malware typically disseminates and infects susceptible systems via diverse propagation methods for nefarious purposes, including spam distribution, privacy compromise, system disruption, and denial-of-service attacks. Notable examples of malicious programs include Trojan horses, worms, viruses, ransomware, adware, and spyware [12,13].

On the Windows platform, program behaviors are predominantly executed through system API calls, making the use of APIs for dynamic analysis a focal point in malicious behavior detection research. API calls, whereby an application performs services by invoking functions provided by the operating system, encompass activities such as registry operations, process manipulation, accessing network resources, and file reading. Malicious code typically executes specific behaviors by calling a series of APIs, rather than a single API. Analysis of API call sequences reveals that malicious code often invokes fixed sequences to carry out destructive actions, with different malicious behaviors calling distinct sequences rarely used in normal programs [14]. Therefore, API call sequences provide a more accurate representation of program behavior, and malicious code can be identified by analyzing these sequences and extracting subsequence features that differentiate malicious from normal programs [15]. Modeling the API call behavior of malware and benign software, the actual relationship between API functions is represented as a semantic transformation matrix [16]. The author of [17] developed a malware detection model by integrating statistical, contextual, and graph mining features on API call sequences, bridging the gap of dynamic detection methods. The authors of [18] proposed a deep neural network-based malware detection method for the Windows platform, which learns parameter-enhanced API call sequences and employs rule-based and clustering-based classification methods to evaluate the sensitivity of the parameters of the API call sequences to malicious behavior. Kim [19] proposed a malware detection and classification system by generating a chain of behavioral sequences of some malware families and calculating the similarity between the chain of API behavioral sequences and the sequence of target processes. Dynamic malware detection methods like CTIMD [20] utilize Cyber Threat Intelligences (CTIs) to improve the learning of API call sequences with runtime parameters, offering better accuracy and efficiency compared to traditional methods. However, this approach depends on external threat knowledge and may have limited generalization capabilities.

With the ongoing advancements in natural language processing (NLP), API sequences can be viewed as semantically rich text, enabling the application of machine learning (ML) and deep learning techniques to analyze API call sequences [21–23]. K-nearest neighbor (KNN), naive Bayes (NB), decision tree (DT), and support vector machine (SVM) are widely used in the analysis of API sequences [24]. The authors of [25] propose an efficient malware detection system based on deep learning, which uses a reweighted class balance loss function in the final classification layer of the DenseNet model to significantly improve the performance of malware classification by addressing imbalanced data issues. Huang [26] proposed a hybrid visualization approach for malware that integrates static and dynamic analysis, transforming code into images and conducting malicious detection based on the VGG16 network, thereby improving the detection model's performance. However, due to the typically unbalanced distribution of malware and normal software, deep learning-based methods for detecting malicious programs still exhibit a high false detection rate and require further refinement.

A frequent sequence pattern is a subsequence that occurs more than a specified threshold in a sequence database, indicating a regular behavior or trend. PrefixSpan algorithm [27] is a prominent method for mining frequent sequence patterns from sequence data, which employs prefix projection and a depth-first search strategy to efficiently discover frequent patterns in sequence data. This algorithm is highly effective in various fields, including text processing, web log analysis, and bioinformatics. The core of the PrefixSpan algorithm lies in identifying frequent items in each prefix projection, adding them to the prefix to form a new prefix, and continuing this recursive projection and expansion until no further frequent items can be added [28]. Consequently, the PrefixSpan algorithm starts with frequent sequences of length 1, progressively generates longer frequent sequences, and ultimately obtains all frequent sequence patterns in the database. The advantage of the PrefixSpan algorithm is that it does not require generating candidate sequences, which reduces computational and storage requirements, and it leverages the prefix information of the sequence, minimizing unnecessary search space and enhancing efficiency.

3. Materials and Methods

3.1. Model Framework

In this paper, we propose a malicious program detection model based on API call sequences, and its architecture is shown in Figure 1. Initially, the API call sequences to be tested are extracted and subsequently preprocessed. The preprocessed API call sequences are then fed into the behavior sequence matching model for sequence matching. A successful match is denoted as 1, indicating a malicious sequence, while an unsuccessful match is denoted as 0, indicating a normal sequence. Sequences marked as 0 are subsequently input into the TextCNN model for detection, and the detection results from the TextCNN model are considered the final detection results, superseding the results from the behavior sequence matching model.

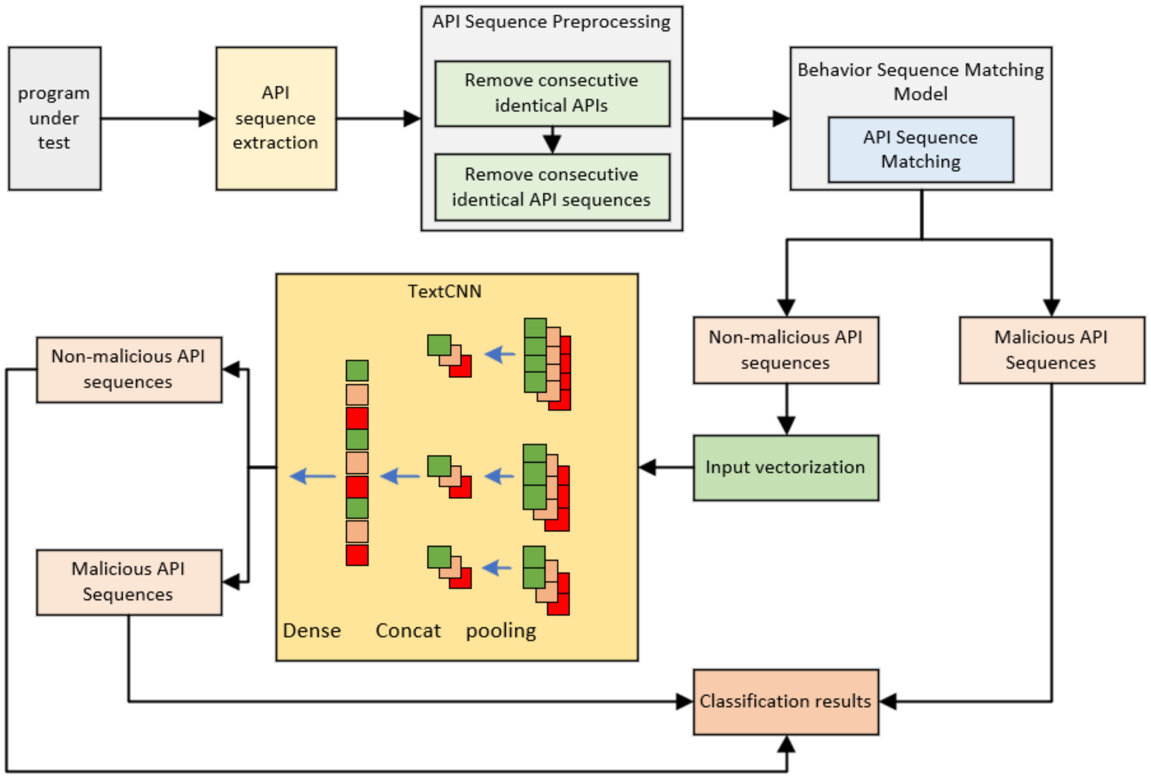

program under test
API sequence extraction
API Sequence Preprocessing
Remove consecutive identical APIs
Remove consecutive identical API sequences
Behavior Sequence Matching Model
API Sequence Matching
TextCNN
Dense
Concat
pooling
Non-malicious API sequences
Malicious API Sequences
Input vectorization
Classification results

Figure 1. Malicious program detection model framework based on API call sequences.

3.1.1. Data Pre-Processing

Preprocessing API call sequences to remove redundant behavior is an important step in sequence mining. Malicious code frequently incorporates numerous redundant behaviors into normal behaviors, resulting in sequences characterized by multiple consecutive identical APIs or API sequence fragments. This increases the length and complexity of the sequences, leading to an increase in the time and space overhead of sequence mining. Moreover, it impacts the precision and interpretability of the mining process, making it difficult to identify the core behaviors of the malicious code.

In this paper, we deduplicate the sequence of API call sequences to reduce their complexity. The deduplication process involves retaining only one instance of APIs that are repeated consecutively within the sequence. The specifics of this process are outlined in Algorithm 1.

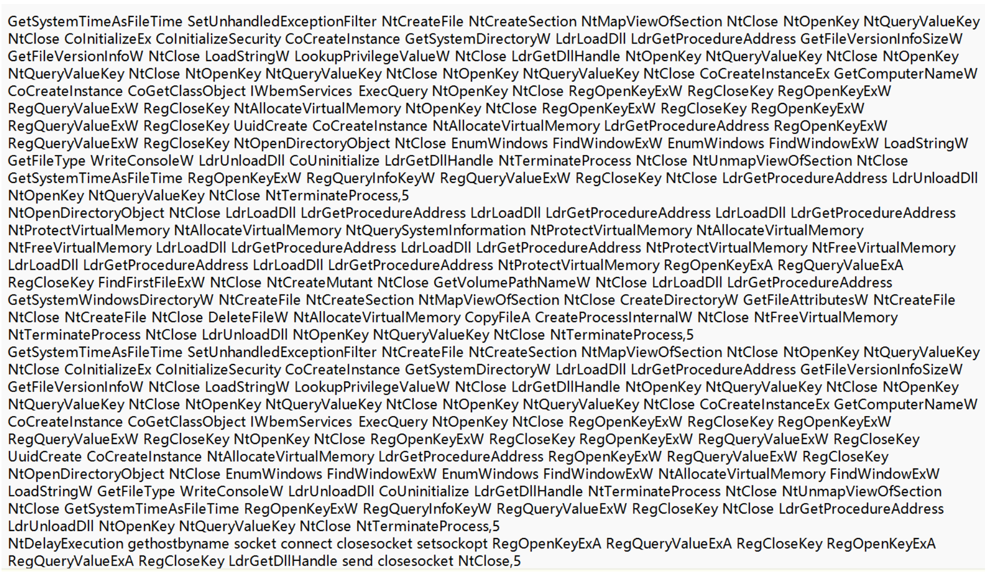
Algorithm 1: API deduplication algorithm

```
Input: API sequence set S, new API list
Output: new API sequence
foreach API seq in S do
    foreach API in API seq do
        if $isEmpty(new\ API\ list)$ or $API \neq last(new\ API\ list)$ then
            Append(new API list, API)
    new API sequence = join(new API list, " ")
return new API sequence
```

3.1.2. Model Construction

Building a malicious program detection model based on the API call sequence is the key to this study. In the process of API sequence matching, we employ the PrefixSpan algorithm to mine frequent API call sequences and establish a rule base for malicious behavior sequences. This rule base, containing malicious behavior sequences, serves as the foundation for subsequent detection.

In this paper, we consider API call sequences from different processes of the same program. Initially, the PrefixSpan algorithm is employed to mine frequent API call sequences within each process of a malicious program, identifying key API call sequences for each program. Subsequently, the PrefixSpan algorithm is further applied to mine frequent sequences from the key API call sequences of all programs, yielding malicious API call sequences that are stored in the malicious behavior sequence repository. This repository is a collection of sequence sets that encapsulate the typical behavioral characteristics of various malicious programs and can be used to assess the behavior of unknown programs. By analyzing the API call sequences of different processes, we can more comprehensively capture the behavior patterns of malicious programs, enhancing the accuracy and interpretability of the detection model. Examples of malicious API call sequences are illustrated in Figure 2.



```
GetSystemTimeAsFileTime SetUnhandledExceptionFilter NtCreateFile NtCreateSection NtMapViewOfSection NtClose NtOpenKey NtQueryValueKey
NtClose CoInitializeEx CoInitializeSecurity CoCreateInstance GetSystemDirectoryW LdrLoadDll LdrGetProcedureAddress GetFileVersionInfoSizeW
GetFileVersionInfoW NtClose LoadStringW LookupPrivilegeValueW NtClose LdrGetDllHandle NtOpenKey NtQueryValueKey NtClose NtOpenKey
NtQueryValueKey NtClose NtOpenKey NtQueryValueKey NtClose NtOpenKey NtQueryValueKey NtClose CoCreateInstanceEx GetComputerNameW
CoCreateInstance CoGetClassObject IWbemServices ExecQuery NtOpenKey NtClose RegOpenKeyExW RegCloseKey RegOpenKeyExW
RegQueryValueExW RegCloseKey NtAllocateVirtualMemory NtOpenKey NtClose RegOpenKeyExW RegCloseKey RegOpenKeyExW
RegQueryValueExW RegCloseKey UuidCreate CoCreateInstance NtAllocateVirtualMemory LdrGetProcedureAddress RegOpenKeyExW
RegQueryValueExW RegCloseKey NtOpenDirectoryObject NtClose EnumWindows FindWindowExW EnumWindows FindWindowExW LoadStringW
GetFileType WriteConsoleW LdrUnloadDll CoUninitialize LdrGetDllHandle NtTerminateProcess NtClose NtUnmapViewOfSection NtClose
GetSystemTimeAsFileTime RegOpenKeyExW RegQueryInfoKeyW RegQueryValueExW RegCloseKey NtClose LdrGetProcedureAddress LdrUnloadDll
NtOpenKey NtQueryValueKey NtClose NtTerminateProcess,5
NtOpenDirectoryObject NtClose LdrLoadDll LdrGetProcedureAddress LdrLoadDll LdrGetProcedureAddress LdrLoadDll LdrGetProcedureAddress
NtProtectVirtualMemory NtAllocateVirtualMemory NtQuerySystemInformation NtProtectVirtualMemory NtAllocateVirtualMemory
NtFreeVirtualMemory LdrLoadDll LdrGetProcedureAddress LdrLoadDll LdrGetProcedureAddress NtProtectVirtualMemory NtFreeVirtualMemory
LdrLoadDll LdrGetProcedureAddress LdrLoadDll LdrGetProcedureAddress NtProtectVirtualMemory RegOpenKeyExA RegQueryValueExA
RegCloseKey FindFirstFileExW NtClose NtCreateMutant NtClose GetVolumePathNameW NtClose LdrLoadDll LdrGetProcedureAddress
GetSystemWindowsDirectoryW NtCreateFile NtCreateSection NtMapViewOfSection NtClose CreateDirectoryW GetFileAttributesW NtCreateFile
NtClose NtCreateFile NtClose DeleteFileW NtAllocateVirtualMemory CopyFileA CreateProcessInternalW NtClose NtFreeVirtualMemory
NtTerminateProcess NtClose LdrUnloadDll NtOpenKey NtQueryValueKey NtClose NtTerminateProcess,5
GetSystemTimeAsFileTime SetUnhandledExceptionFilter NtCreateFile NtCreateSection NtMapViewOfSection NtClose NtOpenKey NtQueryValueKey
NtClose CoInitializeEx CoInitializeSecurity CoCreateInstance GetSystemDirectoryW LdrLoadDll LdrGetProcedureAddress GetFileVersionInfoSizeW
GetFileVersionInfoW NtClose LoadStringW LookupPrivilegeValueW NtClose LdrGetDllHandle NtOpenKey NtQueryValueKey NtClose NtOpenKey
NtQueryValueKey NtClose NtOpenKey NtQueryValueKey NtClose NtOpenKey NtQueryValueKey NtClose CoCreateInstanceEx GetComputerNameW
CoCreateInstance CoGetClassObject IWbemServices ExecQuery NtOpenKey NtClose RegOpenKeyExW RegCloseKey RegOpenKeyExW
RegQueryValueExW RegCloseKey NtOpenKey NtClose RegOpenKeyExW RegCloseKey RegOpenKeyExW RegQueryValueExW RegCloseKey
UuidCreate CoCreateInstance NtAllocateVirtualMemory LdrGetProcedureAddress RegOpenKeyExW RegQueryValueExW RegCloseKey
NtOpenDirectoryObject NtClose EnumWindows FindWindowExW EnumWindows FindWindowExW NtAllocateVirtualMemory FindWindowExW
LoadStringW GetFileType WriteConsoleW LdrUnloadDll CoUninitialize LdrGetDllHandle NtTerminateProcess NtClose NtUnmapViewOfSection
NtClose GetSystemTimeAsFileTime RegOpenKeyExW RegQueryInfoKeyW RegQueryValueExW RegCloseKey NtClose LdrGetProcedureAddress
LdrUnloadDll NtOpenKey NtQueryValueKey NtClose NtTerminateProcess,5
NtDelayExecution gethostbyname socket connect closesocket setsockopt RegOpenKeyExA RegQueryValueExA RegCloseKey RegOpenKeyExA
RegQueryValueExA RegCloseKey LdrGetDllHandle send closesocket NtClose,5
```

Figure 2. Partial malicious API call sequences.

In this study, we employ regular expressions for sequence matching. Regular expressions utilize metacharacters, quantifiers, grouping, assertions, and additional grammatical constructs to formulate complex matching rules, catering to a wide range of matching requirements. Sequence matching is conducted on the API call sequences of the test set, using the malicious behavior sequence repository. This results in a label of 1 for successful matches, denoting a malicious sequence, and a label of 0 for unsuccessful matches, signifying a normal sequence.

After API sequence matching for the API call sequences to be tested, those sequences marked as 0 are further detected using the TextCNN model. The detection process of malicious programs in the neural network model is shown in Figure 3.

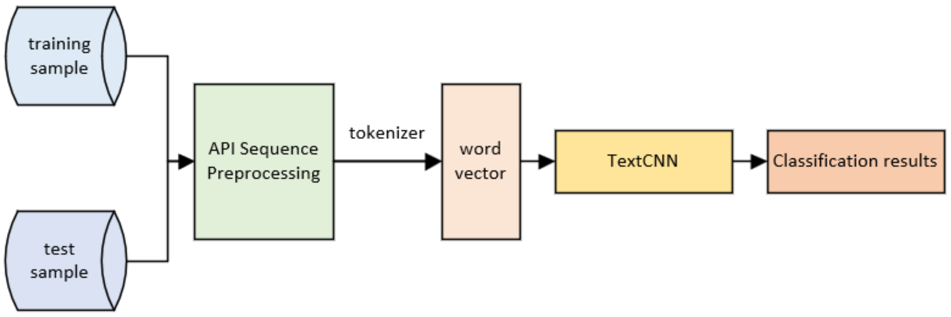


Figure 3. Neural network model malicious program detection process.

The specific steps for constructing the TextCNN malicious program detection model are as follows:

1. Data preprocessing: Remove redundant APIs that appear consecutively in the sequence, retaining only one instance of each API;
2. Data vectorization: The Keras toolkit is used for word vectorization. Treating the API sequence as text, the "fit_on_texts" method is employed to convert the text into integer sequences. Subsequently, the "texts_to_sequences" method transforms the integer sequences into vectors. Finally, the "pad_sequences" method is used to pad or truncate different-length text sequences to a uniform length, ensuring consistency in input data, with the length set to 5000;
3. Training model: The detailed structure of the TextCNN model is shown in Figure 4.

The training set data are input to the Input layer, and word embedding is performed in the Embedding layer to obtain the vector $X_i = \{x_1, x_2, \cdots, x_{5000}\}$ $(1 \leq i \leq n)$, where X_i denotes the vector representation of the ith API sequence, x_i denotes the vector represented by the ith API in a sample of API call sequences, and n denotes the number of samples in the training set. The vector X_i is input into the dropout layer to prevent overfitting, and then input to the TextCNN module. Firstly, feature extraction is performed by three convolutional modules with convolutional kernel sizes of 1, 3, and 5, and then MaxPooling1D is used for pooling, respectively, followed by Concat for feature fusion to obtain the vector $M_i = \{m_1, m_2, \cdots, m_{5000}\}$, and finally, the vector F_i is obtained by using the flatten layer for flattening.

The vector F_i is input into a dropout layer to obtain vector D_i, preventing over-fitting. The vector is then input into a dense layer.

Subsequently, the vector D_i is input into both a dropout layer and a dense layer. The softmax function is employed as the classifier to obtain the classification results.

The TextCNN configuration is shown in Table 1.

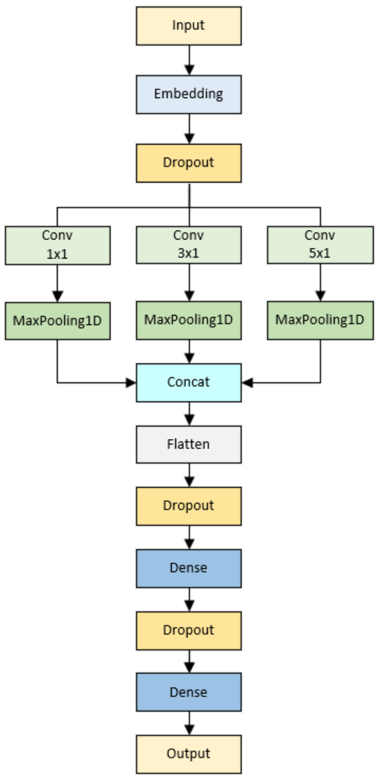


Figure 4. Detailed structure of TextCNN model.

Table 1. TextCNN configuration table.

Layer	Output Shape	Param	Connected to
input_1 (InputLayer)	(None, 5000)	0	
embedding (Embedding)	(None, 5000, 20)	6080	input_1
spatial_dropout1d (SpatialDropout)	(None, 5000, 20)	0	embedding
conv1d (Conv1D)	(None, 5000, 32)	672	spatial_dropout1d
conv1d_1 (Conv1D)	(None, 5000, 32)	1952	spatial_dropout1d
conv1d_2 (Conv1D)	(None, 5000, 32)	3232	spatia_dropout1d
max_pooling1d (MaxPooling1D)	(None, 2500, 32)	0	conv1d
max_pooling1d_1 (MaxPooling1D)	(None, 2500, 32)	0	conv1d_1
max_pooling1d_2 (MaxPooling1D)	(None, 2500, 32)	0	conv1d_2
concatenate(Concatenate)	(None, 2500, 96)	0	max_pooling1d max_pooling1d_1 max_pooling1d_2
flatten (Flatten)	(None, 240,000)	0	concatenate
dropout (Dropout)	(None, 240,000)	0	flatten
dense (Dense)	(None, 256)	61,440,256	dropout
dropout_1 (Dropout)	(None, 256)	0	dense
dense_1 (Dense)	(None, 8)	2056	dropout_1

3.2. Datasets

In this paper, we utilize the dataset of AliCloud Security Malicious Program Detection Challenge [29], and the data provided by the competition are derived from the API command sequence of the Windows binary executable program after simulated execution in a sandbox environment, with a total of 13,887 files and nearly 90 million call records, of which 4978 are normal files and 8909 are malicious files. There are a total of 7 types of malicious files, which are worms, infection viruses, Trojans, mining programs, ransomware, backdoor programs, and DDoS Trojans. The distribution of sample types and specific numbers are presented in Table 2.

Table 2. Experimental dataset.

Label	Type	Number
0	Normal sample	4978
1	Ransomware	502
2	Mining program	1196
3	DDoS Trojan	820
4	worm	100
5	infection virus	4289
6	backdoor program	515
7	Trojan	1487

The statistics of API sequence length before and after deduplication are shown in Figure 5.

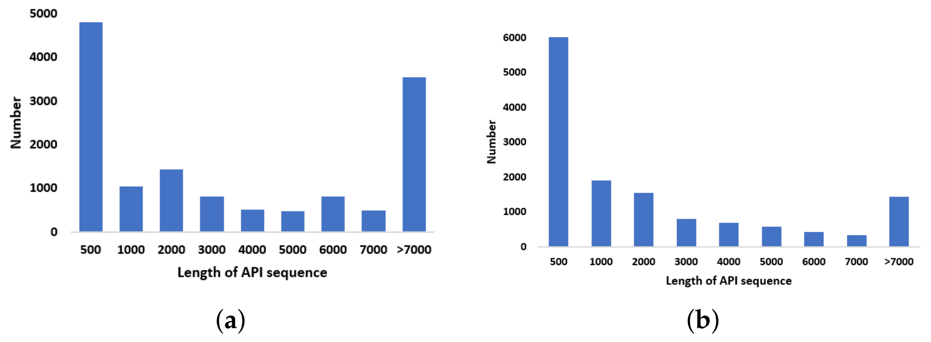


Figure 5. API sequence length statistics before and after deduplication. (**a**) API sequence length statistics before deduplication. (**b**) API sequence length statistics after deduplication.

We quantified the number of APIs in the samples before and after deduplication. As shown in the figure, before deduplication, 65% of the samples exhibitedhad an API sequence length of less than 5000. Following deduplication, this proportion increased to 83%. The deduplication process effectively shortened the length of the API sequence, thereby enhancing the efficiency of subsequent analyses.

3.3. Performance Metrics Used

For model evaluation, we employed Accuracy, Precision, Recall, F1-score, and execution time as metrics. The specific formula is as follows:

$$Accuracy = \frac{TP}{TP + TN + FP + FN}, \tag{1}$$

$$Precision = \frac{TP + TN}{TP + FP}, \tag{2}$$

$$Recall = \frac{TP}{TP + FN}, \tag{3}$$

$$F1\ score = \frac{2 \times Recall \times Precision}{Recall + Precision}, \tag{4}$$

where *TP* (True Positive) represents the number of instances that are actually positive and are correctly predicted as positive, *TN* (True Negative) represents the number of instances that are actually negative and are correctly predicted as negative, *FP* (False Positive) represents the number of instances that are actually negative but are predicted as positive, *FN* (False Negative) represents the number of instances that are actually positive but are predicted as negative. Accuracy represents the ratio of correctly classified samples to all samples; Precision represents the proportion of true positive instances among the instances predicted as positive in the classification model; Recall represents the proportion of correctly predicted positive instances among all positive instances; F1 score is the harmonic mean of Precision and Recall, which balances both precision and recall. A higher value of the metric indicates better classification performance.

4. Experimental Setup

Experiments were performed on a PC with an Intel Core i5-1240P CPU running at 1.70 GHz. We completed the experiment based on Python 3.8, Keras, Scikit-learn and Matplotlib. The experimental hyper-parameter settings for the TextCNN in this paper are shown in Table 3.

Table 3. Hyper-parameter settings for TextCNN.

Parameter	Value
Optimizer	Adam
Batch_size	1000
Epoch	100

5. Results

To verify the performance of the model, in the case of the same dataset, we conducted comparisons with the following models using the same dataset:

- TextCNN model: This model employs multiple convolutional kernels of different sizes (1, 3, 5) to extract key information from the program;
- Sequence Matching: We use the Prefixspan algorithm to mine the API call sequences of all malicious programs, obtaining the malicious API call sequences, and then perform sequence matching using these sequences;
- CNN model: We use Conv1D with convolutional kernel size 3 to extract local features by sliding the convolutional kernel over the input sequences;
- Our model: We initially employ the behavioral sequence matching model to match the API call sequences with malicious sequences, marking successful matches as 1 and unsuccessful matches as 0. Subsequently, sequences marked as 0 are input into the TextCNN model for detection. The detection result from the TextCNN model is considered the final detection result, overriding the outcome from the behavioral sequence matching model.

For the neural network model, this paper carried out 5-fold cross-validation on the training data, divided the training set into training and verification sets, and trained the model for each fold.

From the comparative results in Table 4, it can be observed that the Recall value of sequence matching is relatively high, indicating its effectiveness in capturing malicious samples. Utilizing a network with multiple convolutional layers allows for a more comprehensive extraction of API features, resulting in better detection performance compared to models with a single convolutional layer. Hence, the CNN model's performance is not as strong as the TextCNN model. By combining sequence matching and the TextCNN model, high values for Accuracy, Precision, Recall, and F1-score are achieved. However, since sequence matching is performed one by one using regular expressions, it results in a longer detection time. Our model, which combines sequence matching and a deep learning model, exhibits the longest execution time.

Table 4. Comparison of evaluation results of the models mentioned above.

Model	Accuracy	Precision	Recall	F1-Score	Execution Time (s)
TextCNN [11]	0.9279	0.9278	0.9280	0.9276	41.98
Sequence matching	0.8636	0.8681	0.9938	0.9267	292.3
CNN [30]	0.8779	0.8784	0.8779	0.8761	29.69
Our model	0.9288	0.9246	0.9993	0.9605	366.7

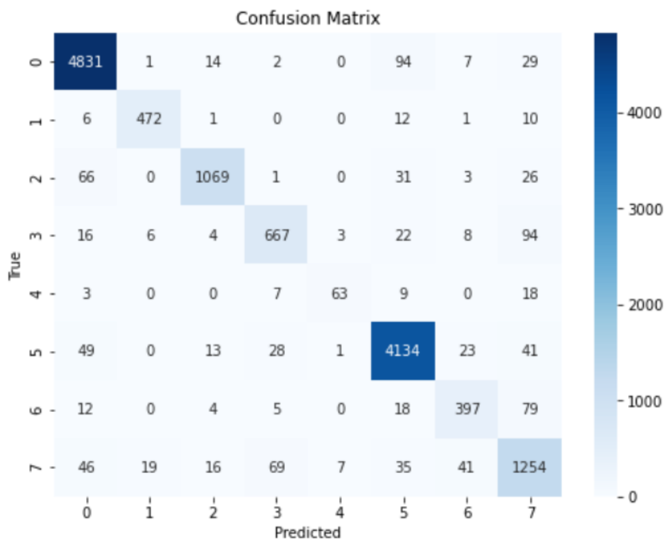
To obtain optimal classification performance, determining the appropriate size of the convolutional kernel in the TextCNN model is essential. The effectiveness of one-dimensional convolutional neural networks for text categorization lies in employing convolutional kernels of varying lengths to extract local features of the sequence. Utilizing multiple sizes of convolutional kernels allows for a more comprehensive capture of the information between APIs. In this paper, we fuse multiple convolutional kernels of different sizes to generate various TextCNN models. The experimental results are presented in Table 5.

Table 5. Comparison of evaluation results of the models mentioned above. The contents of the parentheses refer to the size of convolutional kernels.

Model	Accuracy	Precision	Recall	F1-Score
TextCNN (1 3 3 5 5)	0.9073	0.9057	0.9073	0.9047
TextCNN (1 3 5)	0.9279	0.9278	0.9280	0.9276
TextCNN (1 2 4)	0.9179	0.9173	0.9179	0.9163
TextCNN (2 3 4)	0.9165	0.9169	0.9165	0.9160

From the comparison results in the above table, it can be seen that the TextCNN model achieves its highest accuracy of 0.9279 when the convolutional kernel sizes are 1, 3, and 5. Therefore, in this paper, we choose the TextCNN model with the convolutional kernel sizes 1, 3, and 5 for the detection of malicious programs.

The confusion matrix of the TextCNN model (1, 3, 5) is shown in Figure 6.



True \ Predicted	0	1	2	3	4	5	6	7
0	4831	1	14	2	0	94	7	29
1	6	472	1	0	0	12	1	10
2	66	0	1069	1	0	31	3	26
3	16	6	4	667	3	22	8	94
4	3	0	0	7	63	9	0	18
5	49	0	13	28	1	4134	23	41
6	12	0	4	5	0	18	397	79
7	46	19	16	69	7	35	41	1254

Figure 6. Confusion matrix of the TextCNN model (1, 3, 5).

The AUC value represents the area under the ROC curve. It serves as an evaluation metric to assess the quality of a classification model and effectively describes the model's overall performance. The ROC curves for each class of the TextCNN model with convolutional kernel sizes of 1, 3, and 5 are illustrated in Figure 7.

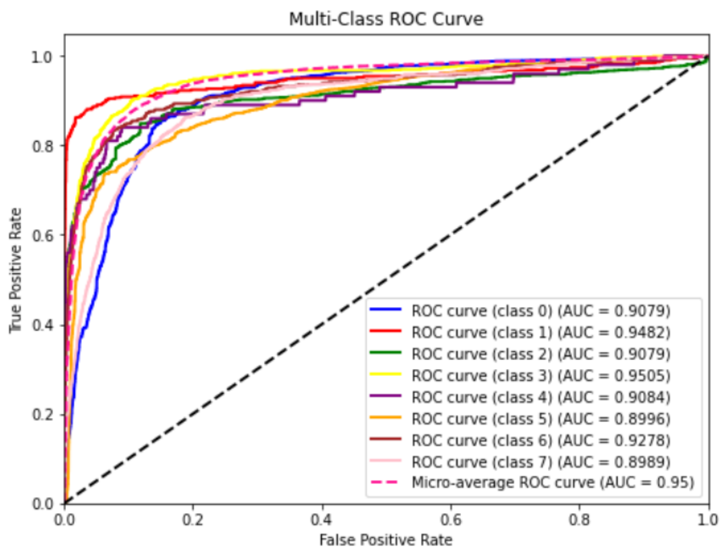


Figure 7. ROC curves for each class of TextCNN model (1, 3, 5).

Figure 8 shows the ROC curves of the four TextCNN models, from which it can be seen that the TextCNN model with convolutional kernel sizes 1, 3, and 5 exhibits the largest AUC value, indicating superior performance compared to the other models.

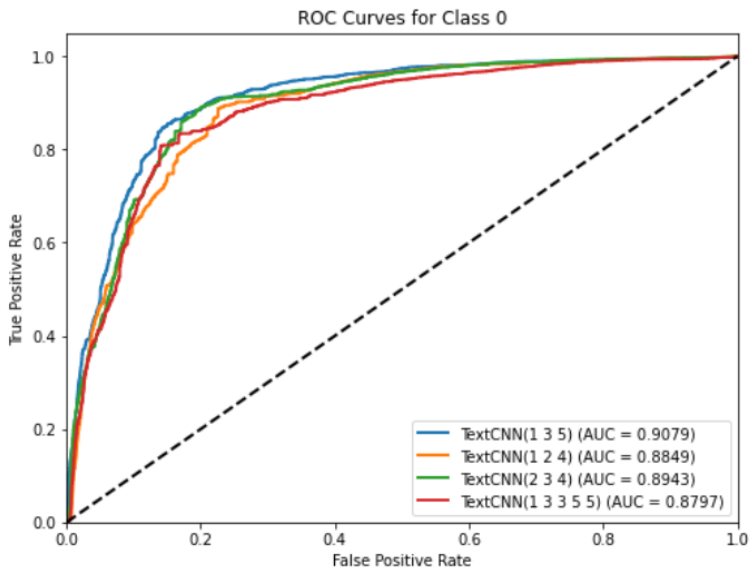


Figure 8. ROC curves of the four TextCNN models.

Furthermore, to validate the efficacy of deep learning, a machine learning algorithm is employed. The N-Gram algorithm is utilized for feature extraction of API call sequences, and the optimal size of n in the N-Gram algorithm must be determined to achieve the best classification performance. The value of n is set to 2, 3, and 4, and the random forest algorithm (the number of decision trees is set to 500), logistic regression and KNN algorithm are selected for model training.

The results of machine learning experiments with different values of n in the N-Gram algorithm are presented in Table 6.

Table 6. Results of machine learning experiments with different values of n in N-Gram.

Value of n	Model	Accuracy	Precision	Recall
2	Random Forest	0.9005	0.9022	0.9010
	Logistic Regression	0.8928	0.8950	0.8928
	KNN	0.8856	0.8856	0.8830
3	Random Forest	0.9028	0.9037	0.9048
	Logistic Regression	0.8938	0.8964	0.8959
	KNN	0.8882	0.8882	0.8902
4	Random Forest	0.8988	0.9029	0.9003
	Logistic Regression	0.8910	0.8933	0.8949
	KNN	0.8856	0.8882	0.8558

The detection results of each machine learning algorithm are shown in Figure 9.

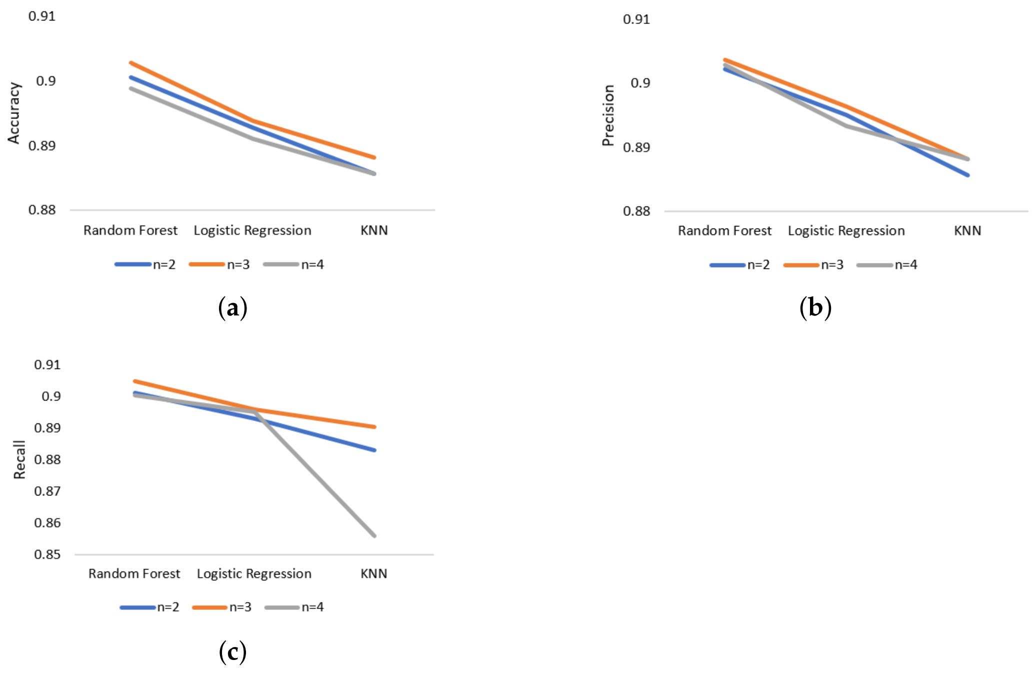


Figure 9. The detection results of three machine learning algorithms. (**a**) The statistical chart of Accuracy. (**b**) The statistical chart of Precision. (**c**) The statistical chart of Recall.

According to Figure 9 and Table 6, it can be seen that for the same value of n, the random forest algorithm outperforms logistic regression and KNN in terms of accuracy, precision and recall, and when n is taken as 3, the random forest algorithm achieves the highest performance in terms of Accuracy, Precision and Recall. All three algorithms exhibit slightly better detection results when $n = 3$, compared to when $n = 2$ or 4. However, their performance is still inferior to that of deep learning. The comparison chart of the detection results of machine learning algorithms and TextCNN and our model is shown in Figure 10.

According to Figure 10, it can be seen that the model we proposed outperforms other machine learning algorithms and TextCNN in terms of performance.

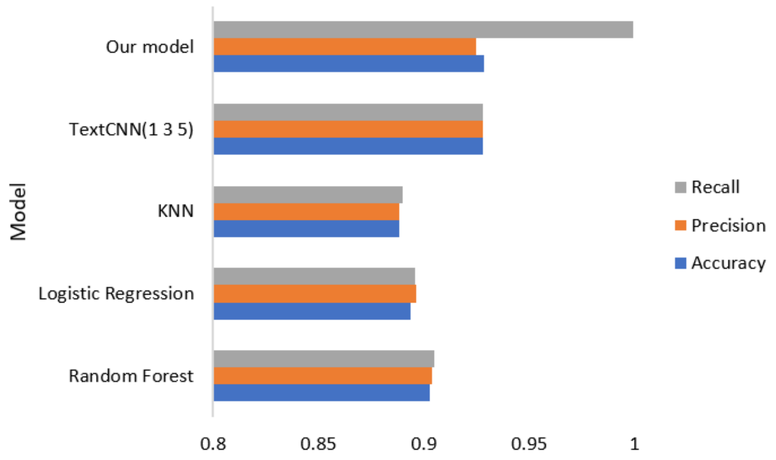


Figure 10. Comparison chart of performance metrics of different classifiers.

6. Discussion

In this paper, we use the public dataset from the AliCloud Malicious Program Detection Challenge. This dataset is derived from the API instruction sequence of Windows executable programs after sandbox simulation. The imbalance between categories exists in real-world scenarios, so an unbalanced dataset can better simulate the actual situation. However, the disparity between different categories and the relatively small size of this dataset compared to the expanding family of malicious code may impact detection results.

In this paper, we focus on dynamic API call sequences as the primary subject of study. However, dynamic analysis relies on the samples executing in the sandbox and producing an execution report. As malicious code countermeasures improve, anti-virtualization techniques used by malicious code can evade detection, preventing the samples from successfully executing in the sandbox. In the future, we consider employing static analysis for further feature extraction and combining both dynamic and static approaches for malicious code detection.

Although the malicious samples in the training set encompass various types of malicious code, their behavior remains constrained. In the future, a more comprehensive analysis of program behaviors can be conducted manually to uncover potential connections among multiple APIs. Enriching the behavioral dataset will be more conducive to the detection of malicious programs.

7. Conclusions

In this paper, we initially mitigate the impact of repetitive information in API call sequences, then analyze the API call sequences in the program to extract behavioral characteristics. Based on these characteristics, we employ a combination of rule matching and deep learning to detect malicious programs. Firstly, malicious sequences are filtered out using behavior sequence matching. Subsequently, the remaining sequences are examined using the TextCNN model. Finally, the detection results from the TextCNN model are used as the final outcomes, superseding the results from the behavioral sequence matching model, to achieve more effective detection of malicious sequences. Since this study only considers the names of the APIs, disregarding information such as parameters, a future direction is to incorporate API parameter information to enhance the accuracy of malicious program detection. In future work, we will continue to collect samples to balance the amount of data across categories. Moreover, we will further consider the parameters of API call sequences to augment their expressive capability. For instance, parameter information of API call sequences related to file operations could include file names, paths, sizes, permissions, etc. This information can aid in identifying potential malicious behaviors in the program, such as deleting, modifying, and hiding important files. Additionally, we will address the imbalance of dataset categories, which may lead to the model favoring categories with more data while overlooking the characteristics of categories with less data.

Author Contributions: Conceptualization, N.L., Z.L. and Y.M.; methodology, N.L., Z.L. and Y.M.; writing—review and editing, N.L., Z.L. and Y.M.; investigation, Y.C. and J.D.; software, Y.C. and J.D.; writing—original draft preparation, Y.C. and J.D.; validation: N.L., Z.L. and Y.M.; supervision: N.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the science and technology project of State Grid Corporation of China: "Research on Key Technologies of Multi-agent Trusted Interaction and Monitoring Response for New-type Power System User Side Business" (Grand No. 5108-202218280A-2-405-XG).

Data Availability Statement: The data presented in this study are available in this article.

Conflicts of Interest: The authors N.L., Z.L. and Y.M., were employed by State Grid Smart Grid Research Institute Co., Ltd. The author J.D., was employed by the State Grid Beijing Electric Power Research Institute. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

1. Wang, B.; Zhang, J.; Luo, C.; Yang, L.; Chen, J.; Ma, H. Research on Deep Detection Technology of Abnormal Behavior of Power Industrial Control System. In Proceedings of the 2022 IEEE 6th Information Technology and Mechatronics Engineering Conference (ITOEC), Chongqing, China, 4–6 March 2022; Volume 6, pp. 1256–1261. [CrossRef]
2. Aboaoja, F.A.; Zainal, A.; Ghaleb, F.A.; Al-Rimy, B.A.S.; Eisa, T.A.E.; Elnour, A.A.H. Malware Detection Issues, Challenges, and Future Directions: A Survey. *Appl. Sci.* **2022**, *12*, 8482. [CrossRef]
3. Ghillani, D.; Gillani, D.H. A perspective study on Malware detection and protection, A review. *Authorea Prepr.* **2022** . [CrossRef]
4. Singh, J.; Singh, J. A survey on machine learning-based malware detection in executable files. *J. Syst. Archit.* **2021**, *112*, 101861. [CrossRef]
5. Gao, H.; Cheng, S.; Zhang, W. GDroid: Android malware detection and classification with graph convolutional network. *Comput. Secur.* **2021**, *106*, 102264. [CrossRef]
6. Cesare, S.; Xiang, Y.; Zhou, W. Control Flow-Based Malware VariantDetection. *IEEE Trans. Dependable Secur. Comput.* **2014**, *11*, 307–317. [CrossRef]
7. Hassen, M.; Chan, P.K. Scalable function call graph-based malware classification. In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, Scottsdale, AR, USA, 22–24 March 2017; pp. 239–248. [CrossRef]
8. Garcia, D.E.; DeCastro-Garcia, N. Optimal feature configuration for dynamic malware detection. *Comput. Secur.* **2021**, *105*, 102250. [CrossRef]
9. Muzaffar, A.; Hassen, H.R.; Lones, M.A.; Zantout, H. An in-depth review of machine learning based android malware detection. *Comput. Secur.* **2022**, *121*, 102833. [CrossRef]
10. Xu, P.; Duan, J.; Zhang, J.; Pei, Y.; Shi, D.; Wang, Z.; Dong, X.; Sun, Y. Active power correction strategies based on deep reinforcement learning—Part I: A simulation-driven solution for robustness. *CSEE J. Power Energy Syst.* **2021**, *8*, 1122–1133. [CrossRef]
11. Wang, Q.; Qian, Q. Malicious code classification based on opcode sequences and textCNN network. *J. Inf. Secur. Appl.* **2022**, *67*, 103151. [CrossRef]
12. Gopinath, M.; Sethuraman, S.C. A comprehensive survey on deep learning based malware detection techniques. *Comput. Sci. Rev.* **2023**, *47*, 100529. [CrossRef]
13. Faruk, M.J.H.; Shahriar, H.; Valero, M.; Barsha, F.L.; Sobhan, S.; Khan, M.A.; Whitman, M.; Cuzzocrea, A.; Lo, D.; Rahman, A.; et al. Malware detection and prevention using artificial intelligence techniques. In Proceedings of the 2021 IEEE International Conference on Big Data (Big Data), Orlando, FL, USA, 15–18 December 2021; pp. 5369–5377. [CrossRef]
14. 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]
15. Lu, X.; Jiang, F.; Zhou, X.; Yi, S.; Sha, J.; Pietro, L. ASSCA: API sequence and statistics features combined architecture for malware detection. Computer Networks. *Comput. Netw.* **2019**, *157*, 99–111. [CrossRef]
16. 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]
17. Amer, E.; Zelinka, I.; El-Sappagh, S. A multi-perspective malware detection approach through behavioral fusion of api call sequence. *Comput. Secur.* **2021**, *110*, 102449. [CrossRef]
18. 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. 10.1109/TIFS.2022.3152360. [CrossRef]
19. Kim, H.; Kim, J.; Kim, Y.; Kim, I.; Kim, K.J.; Kim, H. Improvement of malware detection and classification using API call sequence alignment and visualization. *Clust. Comput.* **2019**, *22*, 921–929. [CrossRef]

20. Chen, T.; Zeng, H.; Lv, M.; Zhu, T. CTIMD: Cyber threat intelligence enhanced malware detection using API call sequences with parameters. *Comput. Secur.* **2024**, *136*, 103518. [CrossRef]
21. 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. [CrossRef]
22. Kim, C.W. Ntmaldetect: A machine learning approach to malware detection using native api system calls. *arXiv* **2018**, arXiv:1802.05412. https://doi.org/10.48550/arXiv.1802.05412.
23. Lin, Z.; Xiao, F.; Sun, Y.; Ma, Y.; Xing, C.C.; Huang, J. A secure encryption-based malware detection system. In Proceedings of the KSII Transactions on Internet and Information Systems (TIIS), Online, 23 December 2018; Volume 12, pp. 1799–1818. [CrossRef]
24. Fan, M.; Liu, J.; Luo, X.; Chen, K.; Tian, Z.; Zheng, Q.; Liu, T. Android malware familial classification and representative sample selection via frequent subgraph analysis. *IEEE Trans. Inf. Forensics Secur.* **2018**, *13*, 1890–1905. [CrossRef]
25. Hemalatha, J.; Roseline, S.A.; Geetha, S.; Kadry, S.; Damaševičius, R. An efficient densenet-based deep learning model for malware detection. *Entropy* **2021**, *23*, 344. [CrossRef]
26. 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]
27. Mane, R.V. A comparative study of Spam and PrefixSpan sequential pattern mining algorithm for protein sequences. In Proceedings of the International Conference on Advances in Computing, Communication and Control, Mumbai, India, 18–19 January 2013; pp. 147–155. [CrossRef]
28. Han, J.; Pei, J.; Mortazavi-Asl, B.; Pinto, H.; Chen, Q.; Dayal, U.; Hsu, M. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In Proceedings of the 17th International Conference on Data Engineering, Heidelberg, Germany, 2–6 April 2001; pp. 215–224. [CrossRef]
29. AliCloud Security Malicious Program Detection Dataset. Available online: https://tianchi.aliyun.com/dataset/dataDetail?dataId=137262 (accessed on 13 September 2022).
30. Ganesh, M.; Pednekar, P.; Prabhuswamy, P.; Nair, D.S.; Park, Y.; Jeon, H. CNN-based android malware detection. In Proceedings of the 2017 International Conference on Software Security and Assurance (ICSSA), Altoona, PA, USA, 24–25 July 2017; pp. 60–65. [CrossRef]