Research Article

Android Malware Detection Based on a Hybrid Deep Learning Model

Tianliang Lu, Yanhui Du , Li Ouyang, Qiuyu Chen, and Xirui Wang

College of Information and Network Security, People's Public Security University of China, Beijing, China

Correspondence should be addressed to Yanhui Du; duyanhui@ppsuc.edu.cn

Received 19 June 2020; Revised 22 July 2020; Accepted 30 July 2020; Published 28 August 2020

Academic Editor: Umar M. Khokhar

Copyright © 2020 Tianliang Lu et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

In recent years, the number of malware on the Android platform has been increasing, and with the widespread use of code obfuscation technology, the accuracy of antivirus software and traditional detection algorithms is low. Current state-of-the-art research shows that researchers started applying deep learning methods for malware detection. We proposed an Android malware detection algorithm based on a hybrid deep learning model which combines deep belief network (DBN) and gate recurrent unit (GRU). First of all, analyze the Android malware; in addition to extracting static features, dynamic behavioral features with strong antiobfuscation ability are also extracted. Then, build a hybrid deep learning model for Android malware detection. Because the static features are relatively independent, the DBN is used to process the static features. Because the dynamic features have temporal correlation, the GRU is used to process the dynamic feature sequence. Finally, the training results of DBN and GRU are input into the BP neural network, and the final classification results are output. Experimental results show that, compared with the traditional machine learning algorithms, the Android malware detection model based on hybrid deep learning algorithms has a higher detection accuracy, and it also has a better detection effect on obfuscated malware.

1. Introduction

Mobile phones have become increasingly important tools in people's daily life, such as mobile payment, instant messaging, online shopping, etc., but the security problem of mobile phones is becoming more and more serious. Due to the open source nature of the Android platform, it is very easy and profitable to write malware using the vulnerabilities and security defects of the Android system. This is the main reason for the rapid increase in the number of malware on the Android system. The malicious behaviors of Android malware generally include sending deduction SMS, consuming traffic, stealing user's private information, downloading a large number of malicious applications, remote control, etc., threatening the privacy and property security of mobile phones users.

The number of Android malware is growing rapidly; particularly, more and more malicious software use obfuscation technology. Traditional detection methods of manual analysis and signature matching have exposed some problems,

such as slow detection speed and low accuracy. In recent years, many researchers have solved the problems of Android malware detection using machine learning algorithms and had a lot of research results. With the rise of deep learning and the improvement of computer computing power, more and more researchers began to use deep learning models to detect Android malware. This paper proposes an Android malware detection model based on a hybrid deep learning model with deep belief network (DBN) and gate recurrent unit (GRU). The main contributions are as follows:

- (i) In order to resist Android malware obfuscation technology, in addition to extracting static features, we also extracted the dynamic features of malware at runtime and constructed a comprehensive feature set to enhance the detection capability of malware.
- (ii) A hybrid deep learning model was proposed. According to the characteristics of static features and dynamic features, two different deep learning algorithms of DBN and GRU are used.

- (iii) The detection model was verified, and the detection result is better than traditional machine learning algorithms; it also can effectively detect malware samples using obfuscation technology.

The rest of the paper is organized as follows. Section 2 gives an overview of the previous related work of malware detection and deep learning algorithms. Section 3 describes the extraction process of static and dynamic features of Android malware. Section 4 explains the malware detection process based on the hybrid deep learning model in detail. Section 5 evaluates our approach through experiments. Section 6 concludes the paper.

2. Related Work

Researchers are constantly improving and innovating the detection method of Android malware. Malware analysis technology is mainly divided into static analysis and dynamic analysis, and the detection method evolves from traditional machine learning to deep learning algorithms.

2.1. Android Malware Analysis Technologies

2.1.1. Static Analysis. The static analysis method refers to extracting malicious features through semantic analysis, permission analysis, etc., after decompiling the APK file. The major advantages of static detection are high efficiency and speed, but it is difficult to identify polymorphic deformation technology and code obfuscation [1, 2].

The required permissions of the APK can help gain awareness about the risks. Talha et al. [3] presented a permission-based APK auditor that uses static analysis to characterize and classify Android applications as benign or malicious. Rahul et al. [4] presented WHY-PER, a framework using natural language processing (NLP) techniques to identify sentences that describe the need for a given permission in an application description. To determine whether Android developers follow least privilege with their permissions requests, Felt et al. [5] built Stowaway, a tool that detects overprivilege in compiled Android applications. Arora et al. [6] identified the pairs of permissions that can be dangerous and proposed an innovative detection model, named PermPair.

APIs are also often used as key features in detecting Android malware [7]. API and permission based classification system were constructed as YARA Rule [8], and the API, class, and public methods of each application are extracted from `AndroidManifest.xml`, `classes.dex` and matched with YARA Rule. Chan and Song [9] proposed a feature set containing the permissions and API calls for Android malware static detection, and classifiers that used the proposed feature set outperform those only with the permissions.

2.1.2. Dynamic Analysis. As more and more Android malware avoid static detection through techniques such as repackaging and code obfuscation, dynamic analysis methods based on behavioral characteristics can solve this

problem well [10]. Dynamic analysis refers to monitoring the behavior of Android application software when it is executed. The monitoring scope of most dynamic analysis methods is mainly access to sensitive data and API calls, etc.

Enck et al. [11] proposed TaintDroid, an efficient, system-wide dynamic taint tracking and analysis system capable of simultaneously tracking multiple sources of sensitive data. TaintDroid provides real-time analysis by leveraging Android's virtualized execution environment and has low CPU overhead.

Fu et al. [12] proposed ntLeakSemaic, a framework that can automatically locate abnormal sensitive network transmissions from mobile apps. Compared to existing taint analysis approaches, it can achieve better accuracy with fewer false positives.

Ali-Gombe et al. [13] proposed the DroidScraper system to recover important runtime data structures of application software by enumerating and reconstructing the objects in memory for mobile device forensics and postmortem analysis.

Malware attempts to evade detection by mimicking security-sensitive behaviors of benign apps and suppressing their payload to reduce the chance of being observed. Based on the contexts that trigger security-sensitive behaviors, Yang et al. [14] introduced AppContext, an approach of static program analysis that extracts the contexts of security-sensitive behaviors to assist app analysis in differentiating between malicious and benign behaviors.

2.2. Malware Detection Method

2.2.1. Machine Learning. Antivirus software can effectively detect Android malware, but it needs to manually extract the signature code and update it on the client side after obtaining the malware sample. This method has a high detection accuracy for known malware, but it also has certain limitations. For example, unknown malware that has not been seen before and malware processed by obfuscation techniques cannot be effectively detected. In order to improve the detection accuracy, in recent years, researchers have used machine learning algorithms to detect malware [15].

Kuo et al. [16] proposed an Android malware detection system which combines the machine learning methods (SVM or Random Forest) and hybrid analysis model, and the major feature combines the permissions characteristic and API.

To enhance security of machine learning-based Android malware detection, Chen et al. [17] developed a system called SecureDroid. They presented a novel feature selection method to make the classifier harder to be evaded and proposed an ensemble learning approach by aggregating the individual classifiers.

Combining of supervised learning (KNN) and unsupervised learning (K-Medoids), Arora et al. [18, 19] introduced a hybrid Android malware detection model using permissions and network traffic features. Awad et al. [20] proposed modeling malware as a language and assessed the

feasibility of finding semantics in instances of that language, and they classified malware-documents by applying the KNN.

In traditional machine learning algorithms, the SVM algorithm is often used for Android malware detection, and it has a good classification effect in many cases. Li et al. [21] studied an Android malware detection scheme using an SVM-based approach, which integrates both risky permission combinations and vulnerable API calls and uses them as features in the SVM algorithm.

2.2.2. Deep Learning. Traditional machine learning algorithms are usually shallow structures, so they cannot effectively characterize Android malware through correlation features. Therefore, researchers tried to distinguish Android malware using deep learning models. The deep learning model has a wide range of applications in image recognition, speech recognition, and natural language processing, and its strong fitting ability for nonlinear relationship makes it have a good application prospect in malware detection. Commonly used deep learning networks include stacked autoencoder [22], DBN [23], LSTM [24], and so on.

Deep learning demonstrated excellent performance in image recognition, so malware can be converted into images, and then deep learning algorithms are used for training and detection [25]. Cui et al. [26] converted the malware into grayscale images; then, the images were identified and classified using a convolutional neural network (CNN) that could extract the features of the malware images automatically. Depending on decompiling the Android APK, Zhao and Qian [27] innovatively mapped the opcodes, API packages, and high-level risky API functions into integrated three channels of an RGB image, respectively, and then used convolutional neural network to identify the malware family's features.

Pektaş et al. [28] proposed a deep learning Android malware detection method which examines all possible execution paths and the balanced dataset improves deep neural learning benign execution paths versus malicious paths. Yuan et al. [29] implemented an online Android malware detection engine based on deep learning.

In order to improve the detection accuracy and take advantage of various deep learning algorithms, some researchers have proposed malware detection models with a combination of multiple deep learning algorithms. Luo et al. [30] proposed an Android malware analysis and detection technology based on Attention-CNN-LSTM, which is a type of multi-model deep learning. Safa et al. [31] benchmarked deep learning architectures composed of recurrent and convolutional neural networks and developed an automatic feature extraction component and a hybrid CNN/RNN classification model.

3. Android Malware Features Extraction

The features extraction method combining dynamic analysis and static analysis is adopted, as shown in Figure 1. The static features are obtained by decompiling the APK file, including

FIGURE 1: Android malware features extraction.

resource features and semantic features. The static features generate a binary feature vector through one-hot encoding. The dynamic features are obtained by monitoring the related API calls during the APK running process. For the dynamic features associated with the time series, the entity embedding method is used to generate feature vectors.

3.1. Features Extraction. Static features extraction is high-speed and consumes less system resources, which is suitable for large-scale features extraction, it cannot effectively detect obfuscated Android malware. Therefore, this paper uses a hybrid detection method of static analysis and dynamic analysis; a total of 351 features were extracted, including 303 static features and 48 dynamic features.

3.1.1. Static Feature Extraction. The extracted static features include resource features and semantic features.

(1) *Resource Features.* Resource features refer to features extracted from resource files stored in APK. The main basis for extracting resource features is the inconsistent structure and inconsistent logic of the APK. Inconsistent structure refers to the artifacts left behind by hiding malicious components, resulting in an abnormal structure of the APK file. Inconsistent logic refers to the fact that when a malicious software is repackaged as a benign application, it usually leaves traces. The types and quantity of resource features are shown in Table 1. A total of 124 resource features were extracted.

(2) *Semantic Features.* Semantic features are extracted from the APK code file. Common static features such as sensitive API and permission are also divided into semantic features. We propose some new semantic features, such as explicit intent and other features mined from meta information. The types and quantity of semantic features are shown in Table 2. A total of 179 semantic features were extracted.

TABLE 1: Types and quantity of resource features.

Types	Specific meaning	Quantity
Certificate	Timestamp	2
	Detail	8
Name	Package name length, entropy	14
	Package name similarity	7
Stealth	Embedded API	15
	Embedded API package name	4
	Embedded permission	26
	Embedded intention	13
Inconsistent	Inconsistent file suffix	4
	Inconsistent resource	5
Native code	System call	19
	Shared library	7

TABLE 2: Types and quantity of semantic features.

Type	Specific meaning	Quantity
DEX	API call	32
	API package	8
	String	13
Intent and permission	Intension	16
	Permission	67
Meta information	Services	6
	Content providers	3
	Broadcast receivers	2
	Activities	25
Escape	Encryption	2
	Reflection	3
	Dynamic code loading	2

3.1.2. Dynamic Features Extraction. Dynamic features are the behavioral characteristics of the Android application when it runs, such as data encryption and decryption, file reading and writing, network data transmission, call, SMS, geographic location, and access to sensitive information. These behaviors can represent the application's functions and intentions. A total of 48 dynamic features are extracted. The extraction of these dynamic features is mainly based on monitoring related API function calls. Each dynamic feature corresponds to several API functions, and the total number of API functions is 141. Some of the dynamic features and corresponding API examples are listed in Table 3.

We select the automatic test tool MonkeyRunner and the dynamic analysis tool Inspeckage to extract dynamic features.

MonkeyRunner is a test tool provided by the Android SDK. It supports writing test scripts to customize data and events and can simultaneously connect to multiple real terminals or emulators to trigger operations of the application software. It can better perform the functions of the Android applications.

The dynamic analysis tool Inspeckage is a simple application software that integrates the commonly used dynamic analysis functions, and a built-in web server can provide a friendly interactive interface for users. Inspeckage can not only obtain basic information such as permissions, components, shared libraries, UID, etc., but also view the behavior of the application in real time. It can customize the

TABLE 3: Types and quantity of dynamic features.

Type	API examples	Quantity
Data encryption and decryption	doFinal	35
	setSeed	
File reading and writing	getString	38
	putString	
Network data transmission	Init	28
	connected	
	getDate	
	Listen	
Call	getServiceStatus	13
	createAppSpecificSmsToken	
SMS	getSmsManagerForSubscriptionId	15
	addNmeaListener	
Geographic location	getAltitude	12
	getLongitude	

hooked API; that is, it can customize the dynamic behavior required for monitoring, and this is also the biggest advantage of the tool.

3.2. Features Encoding

3.2.1. Static Features Encoding. For static features, most of them are binary features, and only a small number of features are discrete features, and there is no relationship between features. Therefore, the deep learning algorithm deep belief network is suitable for static features. Since the input of the deep belief network is a binary vector, one-hot encoding is used to encode the static features into binary vectors.

The process of one-hot encoding is to convert discrete features into corresponding binary sequences. For example, the discrete values of 0, 1, 2, 3 are encoded to binary sequences of 0001, 0010, 0100, 1000. For discrete features with large value range, the feature vector will be sparse if one-hot encoding is used. For this case, the value range can be finely classified to reduce the feature dimension after one-hot encoding. After one-hot encoding, all static features are concatenated into a binary vector, which is the input of the DBN.

3.2.2. Dynamic Features Encoding. After acquiring the dynamic features of the Android application, the dynamic features are formed into an operation sequence in chronological order. Because the dynamic features are correlated in time, the dynamic behavior of the Android application software can be better fitted through the recurrent neural network. The dynamic feature vector is the input of the GRU network.

Entity embedding is a method of data representation. It encodes structured discrete variables and tries to make the data representation retain the continuous relationship between data. In the implementation, the Keras library in Tensorflow is used to implement entity embedding. There are 48 discrete values ($c = 48$) for the dynamic behavior at a certain moment. And, the common choice of embedded size is to use $embedding_size = (c + 1)/2$; therefore, the

embedding size is selected as 24; that is, the dynamic vector input to the GRU at each moment is 24 dimensions. Entity embedding is used to map the values of the discrete variables into a multidimensional space, generate dynamic feature vectors, and provide distance information of different dynamic behaviors in the multidimensional space.

4. Hybrid Deep Learning Model

According to the different characteristics of the static and dynamic features of the Android applications, a deep learning model based on a combination of deep belief network and gate recurrent unit is proposed. The advantage of using the DBN is that the learning speed of static features of Android applications is faster and the performance is better. Compared with the traditional RNN model, GRU can perform better in dealing with longer time operation sequences, with fewer parameters, faster training speed, and less data required to achieve good generalization effect. Therefore, the GRU neural network is more suitable for processing the dynamic features of Android applications.

The DBN-GRU hybrid model for Android malware detection is shown in Figure 2. The dynamic feature vectors and the static feature vectors are used to train the DBN and GRU, respectively, and the output vectors are input to the fully connected layer. The softmax function maps the output of multiple neurons to the interval of (0, 1) and outputs classification results in the form of probability. The softmax function is part of the back propagation neural network and is used to fine-tune the parameters of DBN and GRU.

4.1. Deep Belief Network. DBN is a widely used deep learning framework [32]. The deep belief network is divided into two parts. The bottom part is formed by stacking multiple restricted Boltzmann machines. The RBM of each layer is trained by contrastive divergence (CD) algorithm. The upper part is a supervised back propagation neural network, which is used to fine-tune the overall network. Since this paper uses a hybrid model of DBN and GRU, the BP neural network of the two is integrated and will be introduced in Section 4.3.

4.1.1. Network Structure. In Figure 3, V represents the visible layer, and H represents the hidden layer. In the stacked RBMs, except the last RBM, the hidden layer of each RBM is the visible layer of the next RBM. The weight matrix W_i is used to represent the mapping relationship between the visible layer and the hidden layer. V_0 is the initial feature vector of the first RBM, which is the initial input of the DBN network. The hidden layer H_2 of the last RBM represents the output, which is used for classification. In order to make each layer reach the local optimum, the weight matrix W_i needs to be trained by the CD algorithm.

4.1.2. Pretraining. DBN uses CD algorithm to optimize the parameters of each RBM during pretraining; Figure 4 shows the pretraining process. First, input the initial vector to the first RBM. In our model, the one-hot encoded Android static

FIGURE 2: DBN-GRU hybrid model for Android malware detection.

FIGURE 3: Structure of DBN.

feature vector is input to the DBN as the pretraining initial input.

CD algorithm is used to train the weight matrix W_0 of the first RBM, the bias vector A_0 of the visible layer, and the bias vector B_0 of the hidden layer. The output vector H_0 is obtained after the training, and then the output vector is input to the upper layer RBM as the input vector V_1 of the second RBM. The weight matrix W_1 and the bias vectors A_1 and B_1 of the second RBM are calculated, and the above process is repeated until all RBM training is completed. Pretraining makes the parameters of each RBM reach local optimum.

4.1.3. Parameter Selection. During the pretraining process, the effect of the pretraining is greatly related to the relevant

FIGURE 4: DBN pretraining process.

parameters. There are mainly three parameters for pre-training: the learning rate, batch_size, and epochs. Unlike the learning rate in the subsequent fine-tuning stage of the back propagation neural network, in the pretraining of the DBN, the learning rate usually does not need to be changed, but an appropriate value needs to be set. If the learning rate is too large, the model may fail to converge or even diverge, and if it is too small, the gradient descent can be slow.

Using the method proposed by Smith [33], the model is trained with learning rate range from small to large values, and then the change of loss is recorded. As the learning rate increases, the loss will gradually decrease and then increase, and the best learning rate can be selected from the area with the smallest loss. In this paper, when the pretraining learning rate is 0.05, the pretraining effect is better.

A large batch_size can reduce training time and improve stability, but as batch_size increases, the performance of the model will decrease. Therefore, an appropriate batch_size needs to be chosen. Considering the size of the dataset used in this paper, the batch_size is set to 256.

Finally, the epochs of pretraining: according to the value of the learning rate and the value of batch_size, the epochs are set to 10.

4.2. Gate Recurrent Unit. Recurrent neural network (RNN) can better deal with the time series data, so it is suitable for training the dynamic behavioral characteristics of Android applications. The traditional RNN has the problem of disappearing gradients, which is especially serious when the time series is long. To solve this problem, researchers made improvements to the RNN and got a variant, that is, GRU [34].

The GRU neural network is shown in Figure 5. According to the timeline, X^t is the input feature vector, which represents the dynamic behavior of the Android application at the current time. It can be seen that the input X^t can be remembered by the GRU through h^t . This is very suitable for dynamic features with temporal correlation. For example, if there is a correlation between the two dynamic

FIGURE 5: GRU neural network.

behaviors of reading address book and sending network data, it may be related to malicious behavior, and GRU can learn the link between these behaviors of the Android malware.

4.2.1. Internal Structure of GRU. The internal structure of the GRU is shown in Figure 6; X^t is the input of the current unit, Y^t is the output of the current unit, and h^t is the hidden state of the current unit. h^{t-1} is the hidden state output by the previous unit and passed to this unit. The hidden state contains information about the input of the previous unit.

The internal calculation process of GRU is as follows:

- $\odot$ is an element-wise multiplication
- $+$ is an element-wise addition
- r is the reset gate, and z is the update gate
- W^r , W^z , and W are weight matrixes
- Activation function is tanh:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \quad (1)$$

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Step 1. Calculate the states of gate r and gate z :

$$\begin{aligned} r &= \sigma(W^r X^t + W^r h^{t-1}), \\ z &= \sigma(W^z X^t + W^z h^{t-1}). \end{aligned} \quad (2)$$

Step 2. Reset using reset gate r , and calculate h^t :

$$\begin{aligned} h^{t-1'} &= h^{t-1} \odot r, \\ h^t &= \tanh(W X^t + W h^{t-1'}). \end{aligned} \quad (3)$$

Step 3. Update the memory. The closer the gate signal is to 1, the more information it memorizes, and the closer it is to 0, the more information it forgets:

$$h^t = z \odot h^{t-1} + (1 - z) \odot h'. \quad (4)$$

As mentioned above, combining X^t and h^{t-1} , the GRU will get the output Y^t of the current unit and pass it as the

FIGURE 6: Internal structure of GRU.

hidden state h^t to the next unit, where Y^t and h^t are the same in value.

4.2.2. Number of Network Layers. Our model uses a single-layer GRU. If a second layer is added, it can capture higher-level correlation of dynamic behaviors theoretically, but this is based on the large number of Android applications and high-dimensional input vectors. For the input 24-dimensional dynamic feature vectors and the training samples that have just passed 10,000, the multilayer GRU neural network will not produce good results and may even cause overfitting.

4.2.3. Parameter Initialization. Parameter initialization refers to the process of initializing the weights and biases before the network training. The initialization of parameters is related to whether the network training can yield good results and converge at a faster speed. If the parameter is too small, the input signal of neuron will be too small, and the signal will slowly disappear after multiple layers. If the parameter is too large, the input signal will be too large, and the activation value is saturated causing the gradient to be close to zero.

The Gaussian distribution is used to initialize the weight matrix parameters. The parameters follow a Gaussian distribution with a fixed mean and a fixed variance. Assuming the number of neurons of a certain layer is n_i , the Gaussian distribution of the initialized weight matrix parameters has a mean value of 0 and a variance of $\sqrt{1/n_i}$.

4.3. Back Propagation Neural Network. Back propagation neural network uses supervised learning methods to compare the classification results with the labels of Android applications to fine-tune the hybrid deep learning model. This paper uses the BP neural network combined with the softmax classifier, and the gradient descent algorithm is applied to fine-tune both DBN and GRU networks.

4.3.1. Softmax. Softmax is used in the multiclassification process to map the output of multiple neurons to the (0, 1) interval for classification. Softmax is calculated as

$$S_i = P_i = \frac{e^{f_i}}{\sum_j e^{f_j}}. \quad (5)$$

where S_i represents the probability that the sample is divided into the i -th category. During classification, the category with the highest probability is selected as the classification result.

4.3.2. Fine-Tune the Parameters. The main function of the BP neural network is to fine-tune the parameters of the deep learning model, and the main basis in the fine-tuning process is to find the global minimum value of the loss function of the model. At this time, the parameters of the corresponding weight matrix are the global optimal. For the softmax classifier, the cross-entropy loss function is selected, as shown in

$$L = - \sum_i y_i \ln S_i, \quad (6)$$

where y_i represents the correct label value of each category corresponding to the sample. If it belongs to the i -th category, then, $y_i = 1$; otherwise, $y_i = 0$. Since only one label is 1, the other label is 0, so formula (6) is simplified to formula (7), and its gradient is calculated as

$$L = -y_i \ln S_i = -\ln S_i, \quad (7)$$

$$\frac{\partial L}{\partial f_i} = S_i - 1. \quad (8)$$

The cross-entropy loss function not only can effectively measure the similarity between the calculated value and the actual value, but also has a simple form and is easy to calculate and partial derivative, which is very convenient in the gradient calculation.

4.3.3. Optimization Algorithm. In the process of back propagation, the most commonly used optimization algorithm is to fine-tune the parameters by gradient descent. In order to solve the problems in the gradient descent method, an improved method mini-batch gradient descent is proposed, which can reduce the fluctuation of parameter update and finally get better results and more stable convergence. However, there are still some problems. For example, it is difficult to choose a suitable learning rate and it is easy to fall into the local optimum. Therefore, some algorithms for further optimization of gradient descent are proposed, such as Momentum, Adagrad, and RMSprop.

In this paper, the Adam (Adaptive Moment Estimation) algorithm is selected for fine-tuning. It is a combination of RMSProp algorithm and Momentum algorithm. The main feature of Adam is the adjustment strategy of the learning rate. The first moment estimation (the mean) and second moment estimation (the uncentered variance) of the gradients are used to adjust the learning rate of each parameter dynamically. The main advantage of Adam is that, after the bias correction, the learning rate of each iteration has a certain range, which makes the parameters relatively stable

and has low memory requirements. And in actual operation, the Adam algorithm is simple to use and does not require manual parameter adjustment.

5. Experimental Results and Analysis

5.1. Collation of Android Malware and Benign Samples.

The dataset is divided into benign samples dataset and malware samples dataset. The type, source, and quantity of samples in the dataset are shown in Table 4. The total number of benign samples is 7,000, downloaded from the Google Play and APKpure mobile application markets through web crawler. The number of samples in the malware dataset is 6,298, all downloaded from public malware sharing websites. According to whether the samples use obfuscation technology, the malware samples dataset is divided into two parts: one part is the nonobfuscated malware dataset downloaded from VirusShare [35] and the other part is the obfuscated malware dataset downloaded from PRAGuard [36], obtained by obfuscating the MalGenome and the Contagio Minidump datasets with seven different obfuscation techniques.

5.2. Experimental Results and Analysis

5.2.1. Features with High Frequency of Use. The frequently used features between obfuscated malware samples and non-obfuscated malware samples are analyzed. Figures 7 and 8 show the frequently used top ten features of the two types of samples.

Permission-related features (such as Read_SMS, Write_SMS, etc.) of both sample types are used frequently, because permission features are difficult to be obfuscated, and obfuscating the permission features will destroy the inherent structure of APK. However, some sensitive API features (such as Telephonymanager_Getdeviceid, etc.) are used frequently in nonobfuscated malware samples but are used very rarely in obfuscated malware samples, which shows that the malware samples after obfuscation can avoid related detections when calling sensitive APIs.

It is worth noting that Stat_Cert_Diff, the top-ranked feature in the obfuscated malware samples, is a resource feature related to the certificate. It detects whether the time when the certificate is generated and the certificate is used to sign the APK is the same time. The frequency of this feature is high, which indicates that most of the obfuscated malware samples are generated through automatic repackaging. It also shows that the features extracted in this paper (such as Stat_Cert_Diff, Stat_Reflection) are very effective in detecting obfuscated malware samples.

5.2.2. Detection Effect of the Hybrid Deep Learning Model.

Evaluate the detection effect of the hybrid deep learning model (DBN-GRU) on Android malware through the indicators of precision, recall, and accuracy; the results are shown in Table 5. Deep learning models (such as DBN, GRU, DBN-GRU) are significantly better than traditional machine learning models (such as SVM, Naïve Bayes, KNN). For deep learning models, the DBN-GRU hybrid model is superior to the separate DBN or GRU.

TABLE 4: Android malware and benign samples.

Type	Source	Number	Total
Benign	Google Play	5000	7000
	APKpure	2000	
Malware	VirusShare (nonobfuscated)	4038	6298
	PRAGuard (obfuscated)	2260

FIGURE 7: Top 10 features of nonobfuscated malware.

FIGURE 8: Top 10 features of obfuscated malware.

The DBN model uses static features. Although the antiobfuscation capability of the static features proposed in this paper has been greatly improved, it is still insufficient in capturing the dynamic behaviors of malware. The GRU model uses dynamic features, which has advantages in dynamic behavior analysis but is insufficient in the types of features. According to the experimental results, based on the combination of dynamic features and static features, the hybrid model of DBN and GRU can improve the detection ability and achieve a better detection effect.

TABLE 5: Detection effects of different algorithms.

	Benign samples		Malware samples		Accuracy
	Precision	Recall	Precision	Recall	
SVM	92.74	88.91	88.38	92.38	90.57
KNN	83.83	85.65	83.90	81.90	83.86
Naïve Bayes	86.54	78.26	78.45	86.67	82.27
DBN	95.98	93.49	93.06	95.71	94.55
GRU	93.78	91.74	91.16	93.33	92.50
DBN-GRU	97.79	96.09	95.79	97.62	96.82

5.2.3. Detection Effect of Different Training Datasets. Based on the collected samples, three sample datasets are constructed to evaluate the detection effect of the Android malware detection model on obfuscated samples and non-obfuscated samples. The composition of these three datasets is as follows:

Nonobfuscated dataset: benign samples + VirusShare

Obfuscated dataset: benign samples + PRAGuard

Mixed dataset: benign sample + VirusShare + PRAGuard

Each dataset is divided into a training dataset and a testing dataset, with 2/3 of the samples as the training dataset and 1/3 of the samples as the testing dataset.

Table 6 shows the detection accuracy of using different training datasets. Using the nonobfuscated dataset for both training and testing, an accuracy rate of 96.89% is obtained. Using the obfuscated dataset for both training and testing, an accuracy rate of 96.58% is obtained. In both cases, the detection accuracy is relatively high.

Then, analyze the situation when the training dataset and the testing dataset are different. Using the nonobfuscated dataset for training and the obfuscated dataset for testing, the accuracy rate drops significantly, being 89.51%. Using the obfuscated dataset for training and the nonobfuscated dataset for testing, the accuracy rate also drops to 92.32%. Using the mixed dataset for training, no matter whether the obfuscated dataset or the nonobfuscated dataset is used for testing, the accuracy rates are higher, 96.78% and 96.24%, respectively.

The experimental results show that the richer the sample types of the training dataset, the higher the detection accuracy. The mixed training dataset contains nonobfuscated malware and obfuscated malware. The detection accuracy of all types of testing datasets is high, which can meet the needs of malware detection in actual network.

5.2.4. Detection Effect of Repackaged Malware. 200 samples chosen from the malware sample dataset are repackaged. The process of repackaging is very simple; just uncompress and reassemble the application software without changing any functions. In that case, the MD5 or SHA hash value of the repackaged application software will be different from the original value. Then, using mainstream antivirus software and the DBN-GRU hybrid model proposed in this article for detection, the results are shown in Figure 9. It can be seen that the detection accuracy of the repackaged APKs by

TABLE 6: Accuracy of different training datasets.

Training dataset	Testing dataset	
	Nonobfuscated dataset	Obfuscated dataset
Nonobfuscated dataset	96.89	89.51
Obfuscated dataset	92.32	96.58
Mixed dataset	96.78	96.24

FIGURE 9: Detection effect of repackaged Android malware.

antivirus software has dropped significantly. This is because the antivirus software may use a hash check method to identify the known Android malware, due to the fact that the hash value of the repackaged APK has changed, causing the detection accuracy to decrease. The detection results of the DBN-GRU hybrid model are not affected by the application software repackaging, because repackaging does not affect the extracted static and dynamic features and the process of model training, which is the major advantage of the hybrid model in detecting Android malware.

6. Conclusion

Due to the widespread use of obfuscation techniques in malware, the effect of traditional detection methods is greatly affected. This paper combines dynamic analysis technology and static analysis technology for Android malware detection and builds a hybrid deep learning model based on DBN and GRU. In order to deal with the obfuscation technology, new static features with strong anti-obfuscation capabilities are added, and the dynamic features of the application software at runtime are extracted to enrich the Android malware feature set. According to the different characteristics of static features and dynamic features, a hybrid deep learning model with DBN and GRU is used for learning, and the detection effect of this model is verified through comparative experiments.

Due to the limited computing resources of mobile devices, and the fact that deep learning is a compute-intensive task, the Android malware detection model proposed in this paper is suitable for running on high-performance computers. In order to solve this problem, the cloud antivirus technology is recommended, the mobile phone client is

responsible for uploading suspicious files, and the cloud-based server is responsible for sample analysis and detection.

The research has the following deficiencies and needs to be improved in future research work. First, the number of samples, especially malware samples, is not enough, and the representativeness of the obtained malware features is still not strong. It is necessary to constantly enrich the types and number of samples in the malware sample dataset; second, the calculating consumption of the hybrid model is larger than that of the separate model and the traditional machine learning algorithm, so further improvement and optimization is needed to reduce the time cost.

Data Availability

The Android malware samples used to support the findings of this study can be downloaded from Virus-Share (available at <https://virusshare.com>) and Android PRAGuard (available at <http://pralab.diee.unica.it/en/AndroidPRAGuardDataset>).

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Cryptography Development Fund of China (grant no. MMJJ20180108) and the Fundamental Research Funds for the Central Universities of PPSUC (grant no. 2020JKF101).

References

- [1] K. Bakour, H. M. Nver, and R. Ghanem, "The Android malware static analysis: techniques, limitations, and open challenges," in *Proceedings of the 3rd International Conference on Computer Science and Engineering (UBMK'18)*, pp. 586–593, Sarajevo, Bosnia-Herzegovina, September 2018.
- [2] K. Riad and L. Ke, "RoughDroid: operative scheme for functional android malware detection," *Security and Communication Networks*, vol. 2018, Article ID 8087303, 10 pages, 2018.
- [3] K. A. Talha, D. I. Alper, and C. Aydin, "APK Auditor: permission-based Android malware detection system," *Digital Investigation*, vol. 13, pp. 1–14, 2015.
- [4] P. Rahul, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: towards automating risk assessment of mobile applications," in *Proceedings of the 22nd Usenix Security Symposium*, pp. 527–542, Washington, DC, USA, August 2013.
- [5] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, pp. 627–636, Chicago, IL, USA, October 2011.
- [6] A. Arora, S. K. Peddoju, and M. Conti, "PermPair: android malware detection using permission pairs," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1968–1982, 2020.
- [7] W. Wang, J. Wei, S. Zhang, and X. Luo, "LSCDroid: malware detection based on local sensitive API invocation sequences," *IEEE Transactions on Reliability*, vol. 69, no. 1, pp. 174–187, 2020.
- [8] J. Park, H. Chun, and S. Jung, "API and permission-based classification system for Android malware analysis," in *Proceedings of the International Conference on Information Networking (ICOIN 2018)*, pp. 930–935, Chiang Mai, Thailand, January 2018.
- [9] P. P. K. Chan and W. K. Song, "Static detection of Android malware by using permissions and API calls," in *Proceedings of the International Conference on Machine Learning and Cybernetics (ICMLC 2014)*, pp. 82–87, Lanzhou, China, July 2014.
- [10] H. Cai, N. Meng, B. Ryder, and D. Yao, "DroidCat: effective Android malware detection and categorization via app-level profiling," *IEEE Transactions on Information Forensics & Security*, vol. 14, no. 6, pp. 1455–1470, 2018.
- [11] W. Enck, P. Gilbert, S. Han et al., "TaintDroid," *ACM Transactions on Computer Systems*, vol. 32, no. 2, pp. 1–29, 2014.
- [12] H. Fu, Z. Zheng, S. Bose, M. Bishop, and P. Mohapatra, "Leaksemantic: identifying abnormal sensitive network transmissions in mobile applications," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM 2017)*, pp. 1–9, Atlanta, GA, USA, May 2017.
- [13] A. Ali-Gombe, S. Sudhakaran, A. Case, and G. G. Richard, "DroidScraper: a tool for Android in-memory object recovery and reconstruction," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*, pp. 547–559, Beijing, China, October 2019.
- [14] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: differentiating malicious and benign mobile app behaviors using context," in *Proceedings of the International Conference on Software Engineering (ICSE 2015)*, pp. 303–313, Florence, Italy, May 2015.
- [15] M. A. Kadri, M. Nassar, and H. Safa, "Transfer learning for malware multi-classification," in *Proceedings of the 23rd International Database Applications & Engineering Symposium (IDEAS'19)*, pp. 1–7, Athens, Greece, June 2019.
- [16] W. C. Kuo, T. P. Liu, and C. C. Wang, "Study on Android hybrid malware detection based on machine learning," in *Proceedings of the IEEE International Conference on Computer and Communication Systems (ICCCS 2019)*, pp. 31–35, Singapore, February 2019.
- [17] L. Chen, S. Hou, and Y. Ye, "Securedroid: enhancing security of machine learning-based detection against adversarial android malware attacks," in *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC 2017)*, pp. 362–372, Orlando, FL, USA, December 2017.
- [18] A. Arora, S. K. Peddoju, V. Chouhan, and A. Chaudhary, "Hybrid Android malware detection by combining supervised and unsupervised learning," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom 2018)*, pp. 798–800, October 2018.
- [19] A. Arora and S. K. Peddoju, "NTPDroid: a hybrid android malware detector using network traffic and system permissions," in *Proceedings of the 2018 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/12th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE)*, pp. 808–813, New York, NY, USA, August 2018.
- [20] Y. Awad, M. Nassar, and H. Safa, "Modeling malware as a language," in *Proceedings of the 2018 IEEE International Conference on Communications (ICC 2018)*, pp. 1–6, Kansas City, MO, USA, May 2018.
- [21] W. Li, J. Ge, and G. Dai, "Detecting malware for Android platform: an SVM-based approach," in *Proceedings of the*

- IEEE International Conference on Cyber Security and Cloud Computing (CSCloud 2015)*, pp. 464–469, New York, NY, USA, November 2015.
- [22] X. Yuan, J. Zhou, B. Huang, Y. Wang, C. Yang, and W. Gui, “Hierarchical quality-relevant feature representation for soft sensor modeling: a novel deep learning strategy,” *IEEE Transactions on Industrial Informatics*, vol. 16, no. 6, pp. 3721–3730, 2020.
 - [23] Y. Wang, Z. Pan, X. Yuan, C. Yang, and W. Gui, “A novel deep learning based fault diagnosis approach for chemical process with extended deep belief network,” *ISA Transactions*, vol. 96, pp. 457–467, 2020.
 - [24] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, “LSTM: a search space odyssey,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, 2017.
 - [25] F. Mercaldo and A. Santone, “Deep learning for image-based mobile malware detection,” *Journal of Computer Virology and Hacking Techniques*, vol. 16, no. 2, pp. 157–171, 2020.
 - [26] Z. Cui, F. Xue, X. Cai, Y. Cao, G.-g. Wang, and J. Chen, “Detection of malicious code variants based on deep learning,” *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3187–3196, 2018.
 - [27] Y. L. Zhao and Q. Qian, “Android malware identification through visual exploration of disassembly files,” *International Journal of Network Security*, vol. 20, no. 6, pp. 1061–1073, 2018.
 - [28] A. Pektaş and T. Acarman, “Learning to detect Android malware via opcode sequences,” *Neurocomputing*, vol. 396, pp. 599–608, 2020.
 - [29] Z. Yuan, Y. Lu, and Y. Xue, “Droiddetector: android malware characterization and detection using deep learning,” *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 114–123, 2016.
 - [30] S. Luo, Z. Liu, B. Ni, H. Wang, H. Sun, and Y. Yuan, “Android malware analysis and detection based on Attention-CNN-LSTM,” *Journal of Computers*, vol. 14, no. 1, pp. 31–43, 2019.
 - [31] H. Safa, M. Nassar, and W. A. R. A. Orabi, “Benchmarking convolutional and recurrent neural networks for malware classification,” in *Proceedings of the 15th International Wireless Communications & Mobile Computing Conference (IWCMC 2019)*, pp. 561–566, Tangier, Morocco, June 2019.
 - [32] G. E. Hinton, S. Osindero, and Y.-W. Teh, “A fast learning algorithm for deep belief nets,” *Neural Computation*, vol. 18, no. 7, pp. 1527–1554, 2006.
 - [33] L. N. Smith, “Cyclical learning rates for training neural networks,” pp. 1–10, 2017, <http://arxiv.org/abs/1506.01186v6>.
 - [34] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” pp. 1–9, 2014, <http://arxiv.org/abs/1412.3555>.
 - [35] VirusShare, 2020, <https://virusshare.com>.
 - [36] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, “Stealth attacks: an extended insight into the obfuscation effects on Android malware,” *Computers & Security*, vol. 51, pp. 16–31, 2015.