ByteDroid: Android Malware Detection Using Deep Learning on Bytecode Sequences

Kewen Zou[1], Xi Luo[2], Pengfei Liu[1], Weiping Wang[1(✉)], and Haodong Wang[3]

[1] School of Computer Science and Engineering, Central South University, Changsha 410083, China
wpwang@csu.edu.cn
[2] Department of Information Technology, Hunan Police Academy, Changsha 410138, China
[3] Department of Electrical Engineering and Computer Science, Cleveland State University, Cleveland, OH 44115, USA

Abstract. The explosive growth of the Android malware poses a great threat to users' privacy and sensitive personal information. It is urgent to develop an effective and efficient Android malware detection system. Existing studies usually require the manual feature engineering for the feature extraction. In fact, the detection performance is heavily relied on the quality of the feature extraction. Additionally, the feature extraction becomes extremely difficult in the malware detection due to the fact that malware developers often deploy the obfuscation techniques. To address this issue, we focus on the Android malware detection using the deep neural networks without the human factors. In this paper, we propose ByteDroid, an Android malware detection scheme that processes the raw Dalvik bytecode using the deep learning. ByteDroid resizes the raw bytecode and constructs a learnable vector representation as the input to the neural network. Then, ByteDroid adopts a Convolutional Neural Networks (CNNs) to automatically extract the malware features and perform the classification. Our experiment results demonstrate that ByteDroid not only can effectively detect Android malware, but also has a great generalization performance given untrained malware. Moreover, ByteDroid maintains resilience to obfuscation techniques.

Keywords: Android malware detection · Dalvik bytecode · Convolutional Neural Networks

1 Introduction

The growth of Android malware has become a crucial security problem for users' privacy and sensitive information. It is impractical to analyze every single application manually given millions of Android applications in the application stores. Many Android malware detection systems heavily rely on human factors, using

W. Han et al. (Eds.): CTCIS 2019, CCIS 1149, pp. 159–176, 2020.
https://doi.org/10.1007/978-981-15-3418-8_12

handcrafted rules to detect the malware from unknown applications and determine the malware type. These rules, however, may not work when the malware equipped with obfuscation techniques. Meanwhile, it is still difficult for malware detector to keep up with the process of malware evolution. To address this issue, we develop a deep learning based malware detection scheme that does not require the domain knowledge and the human factors. In particular, the proposed detection system works in the rapidly variant malware ecosystem.

In recent years, the deep neural networks have achieved great success in the fields of computer vision and natural language processing. The deep neural networks can learn feature representation and classification simultaneously to achieve the best results. Motivated by this, we seek to use the raw bytecode sequences of the Android application as the input to train the deep neural networks. There are three reasons for choosing the Android bytecode sequences instead of the source code. First, studies [1–3] show that malicious applications often contain the similar bytecode sequence due to the fact that malware development usually shares the same libraries or modules. Second, the deep neural networks can learn directly from raw data such as pixels, words and signals. The bytecode sequence is a great form of raw data. Third, malware often use some obfuscation techniques to evade detection by renaming identifiers and inserting junk code. Yet, the bytecode is resilient to these obfuscation techniques and the model we proposed does not rely on the semantics like strings.

Recent work [4–7] strives to extract semantics as the features for the malware detection. Based on the features used to classify the malware, these approaches can be categorized into static analysis and dynamic analysis. The approaches based on static analysis usually obtain the Android application source code through the reverse engineering tools like ApkTool. The features are extracted from the API calls, the permissions, the control flows and the data flows, and then are used for classification tasks. This type of approaches is resource and time efficient because the application does not need execution. The dynamic analysis based detection approaches extract the behavioral characteristics while the application is running and therefore are more effective in detecting the malicious activities even if the evasion techniques, such as native code and dynamic code loading, are used. Nevertheless, these approaches require complicated feature extraction and are not resilient to typical obfuscation techniques.

In this paper, we propose ByteDroid, a deep learning based Android malware detection approach that does not require domain knowledge and manual feature extraction. Different from the studies that extract semantic features, ByteDroid directly processes the raw bytecode for malware detection. The raw bytecode is extracted from the Android application packages (APKs), which is subsequently represented by a series of vectors. The generated vectors are then fed to a Convolutional Neural network (CNN) to learn the bytecode sequential features for the classification.

To evaluate the performance of our model, we implement the Android malware detection model based on the opcode sequences [8]. The results show that bytecode-based method outperforms opcode-based method. In addition, we

evaluate the robustness of ByteDroid. Experimental results indicate that ByteDroid can effectively detect unknown malware. Meanwhile, ByteDroid is resilient to obfuscated Android malware. Our contributions can be summarized as follow:

- We propose ByteDroid that directly processes the raw Dalvik bytecode and automates the Android application bytecode feature extraction through the deep neural networks to detect the Android malware. ByteDroid does not require any domain knowledge and manual feature extraction.
- We implement ByteDroid and the malware detector [8] relying on the Android opcode. The experimental results demonstrate that ByteDroid outperforms the opcode based detector.
- We conduct the extensive experiments on several datasets to evaluate ByteDroid's capability in detecting the malware from the untrained application pool. The results show that ByteDroid has the ability to detect the unknown malware. Moreover, for 10479 malware that applied seven typical obfuscation techniques, ByteDroid successfully detects 92.17% of them.

2 Related Work

Methods Based on Traditional Machine Learning. Machine learning and data mining based methods have been proposed for Android malware detection [9–15]. In particular, most of studies mainly focus on handcrafted features such as the sensitive APIs, the permissions, the data flow and the control flow. After encoding these features as vectors, the machine learning algorithm is applied for classification.

DroidMat [9] extracts the sensitive API calls, the permission and the intent message through static analysis. Then it uses k-means algorithm for clustering and applies k-NN for classification. DroidMiner [10] also performs static analysis to extract activities, services, broadcast receivers and sensitive API calls to construct a behavioral graph. The malicious patterns can be mined from the behavioral graph and then encoded as vectors to train a classifier. DREBIN [11] is another static-analysis based method that extracts features including the hardware, the permissions, the system API calls and the URLs from the manifest files and the source code. These features is then applied for building a linear SVM. Since DREBIN applies the linear SVM for classification, it can determine the contribution of every features and provide explanation. Crowdroid [12] uses dynamic analysis to extract the system calls. The system calls are clustered by using k-means algorithm to distinguish malware from benign applications. DroidDolphin [13] also utilizes dynamic analysis to extract 13 features of runtime activities, including API calls, network access, file I/O and services. Then a SVM is trained on these features to detect malware. Marvin [14] performs the combination of the static analysis and dynamic analysis to fully capture the behaviors of malware, which uses a linear SVM for binary classification and assesses the risk of unknown apps with a malice score. HinDroid [15] represents the Android applications and the corresponding APIs as a structured

heterogeneous information network, and it uses a meta-path based approach to characterize the semantic information of the application and its corresponding APIs. Ultimately, HinDroid applies the multi-kernel learning to build a malware prediction model.

Methods Based on Deep Learning. Since the deep neural networks perform much better than the traditional machine learning in many application tasks, many studies begin to use the deep neural networks instead of the traditional machine learning algorithms.

Droiddetector [16] associates static analysis and dynamic analysis to extract 192 features and applies Deep Belief Networks (DBNs) whose performance outperform machine learning algorithms such as Random Forest and SVM. Li et al. [17] also build a DBN based model that uses the combinations of sensitive API calls and permissions as the input features. S. Hou et al. [18] propose an approach to categorize the API calls inside a single method and represent the Android application by blocks of API calls. The classification is then performed on API call blocks using Stacked AutoEncoders. Nix et al. [19] design a pseudo-dynamic program analyzer to track a possible execution path and generate a series of the API calls along the path. They build a CNN taking the API call sequences as input to either detect malware or classify the benign applications. Xu et al. [20] propose a malware detection system DeepRefiner consisting of two layers. In the first layer, DeepRefiner classifies the applications into benign, malware and uncertain type based on the features extracted from the manifest files. For the applications that labeled as uncertain, DeepRefiner builds a Long Short Time Memory (LSTM) model to perform refined inspection on the simplified Smali code in the second layer. Mclaughlin et al. [8] take the opcode as the input of a CNN model, which is similar to us. The opcode is extracted from the Smali code dissembled by reverse engineering tools. The author discards the operands in the execution statement, which has a certain impact on the performance of the model. In our experiments, we take this approach as a comparison.

As far as we know, there are little published studies directly process the raw Dalvik bytecode for Android malware detection. Of all the studies described above, though they use the deep neural networks, the manual feature extraction is still required. Consequently, the effectiveness of deep neural networks is limited. Instead, we leverage the potential of the deep neural networks, which use as little human labors as possible to detect Android malware.

3 ByteDroid

In this section, we describe the details of how to process the bytecode and the overall architecture of ByteDroid.

3.1 System Architecture

The system architecture of ByteDroid is shown in Fig. 1. ByteDroid consists of two stages: training and testing. In the training stage, ByteDroid first extracts

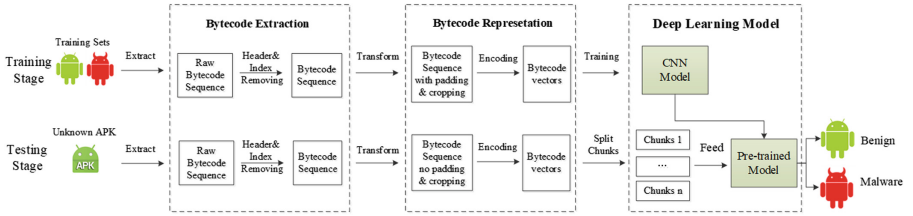

Fig. 1. System architecture of ByteDroid.

the bytecode from the training set containing both malicious and benign applications. Then, the extracted bytecode is resized to a fixed length and encoded to the corresponding vectors to fit the input of CNN. Finally, we train the CNN model using the bytecode vectors. In the testing stage, ByteDroid performs the same operation as in the training stage to deal with the unknown applications. After the bytecode sequence is encoded, ByteDroid splits the bytecode sequence into several chunks to accelerate the malware detection process because the bytecode with larger size will generate larger intermediate results and introduce the extra time consumption. Finally, the chunks are fed to the trained model to complete the classification. Both in training stage and testing stage, there are three components: Bytecode Extraction, Bytecode Representation and the Deep Learning Model. In the rest of this section, we discuss each component in detail.

3.2 Bytecode Extraction

The Android application bytecode is contained in file "classes.dex" and is interpreted by the Dalvik Virtual Machine (DVM) during the execution. Since each APK file is a zipped file, we can easily get the classes.dex file by extraction. After extraction, we use HexDump to obtain hexadecimal bytecode.

The dex file is mainly composed of three parts: the header section, the index section, and the data section. The header contains the basic information such as magic, check sum and file size. The index section stores the offset attribute of string, type, proto, field and method. The data section contains the actual executable code and data where we are most concerned about. The bytes of the header section and the index section are constants that vary in different applications. These bytes normally cannot contribute to malware detection. Therefore, ByteDroid removes the header section and the index section, only preserving the data section of an application.

3.3 Bytecode Representation

Size Padding and Cropping. The Android application bytecode has a highly variable size. Since CNN requires a fixed-size input, we need to pad the bytecode of a smaller APK to a pre-determined size and crop the bytecode of a larger APK

to the same size so that all bytecode sequences have the same size for training the neural networks. We do not select the pre-determined size to be the bytecode size of the largest APK because that would significantly incur the computational complexity. Note that padding the bytecode will not cause additional false positives because adding zeros at the end of the bytecode is equivalent to adding a bunch of NOPs. Unfortunately, bytecode cropping does impact the false negatives because the bytecode containing the malicious operations may be cut off and therefore cannot be "learned" during the training stage.

In this work, we carefully select the pre-determined size in the training stage to keep the false negative low. Let α be the maximum size of the sequence. The value of α in our system is determined to 1,500 KB by our experiments (as will be discussed in Sect. 4.5). In the testing stage, considering the unknown application may have large size of bytecode, we split the input sequence into multiple chunks and then feed to our CNN model respectively. At last, we take the summation of all chunks' output as the results. The size of each chunk can be arbitrary and is limited by the physical memory size.

Bytecode Encoding. Since the operation instruction of Android bytecode is strictly limited to one byte, the bytecode can be considered as a sequence of single bytes. Assuming that the pre-determined bytecode size is α, then a given bytecode sequence B can be represented as $B = \{b_1, b_2, b_3, ..., b_\alpha\}, b_i \in [0, 255]$, where b_i refers to a single of the bytecode.

We encode each byte using a one-hot vector. A one-hot vector for byte is a vector of length 256 with a single element equals one and other elements being zero. Therefore, the respective dimension of each bytecode value is set to one and others are zeros. Using the one-hot vector representation, an Android application's bytecode sequence is constructed as the input, a sparse matrix $B^{'}$ of size $\alpha \times 256$ to our CNN.

3.4 Deep Learning Model

ByteDroid builds a CNN model consisting of embedding layer followed by a convolutional layer, a global max pooling layer, a fully connected layer and an output layer. The proposed CNN architecture is shown in Fig. 2.

Bytecode Vectors. Given a bytecode sequence $B^{'} = \{b_1^{'}, ..., b_i^{'}, ..., b_\alpha^{'}\}$ representing an APK file and $b_i^{'}$ refers to a one-hot form bytecode, our task is to construct a classifier to label the APK to be either benign or malicious.

Embedding Layer. The bytecode in the one-hot form would result in very large vector space. For reducing the dimensions of the vector space, ByteDroid chooses embedding layer to transform the vectors of one-hot form into the dense vectors with P dimensions. The dense vectors not only reduce the dimensions and therefore lower the computational cost, but also capture the bytecode sequence information surrounding each vector. In particular, if a sub-sequence consisting of two or more vectors appears frequently in the bytecode sequence, the similarity score between these vectors is higher than any other vector pairs. For example,

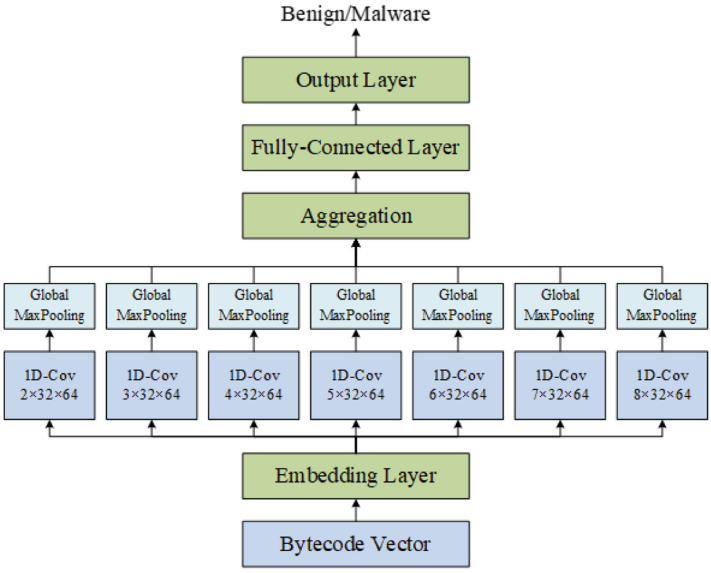

Fig. 2. The architecture of CNN.

if the vector of "invoke-virtual" frequently follows the vector of "iget-object", the similarity score between "iget-object" vector and "invoke-virtual" vector is higher than the similarity score between "iget-object" and any other vectors. Note that the size (P) of dense vector is a hyperparameter that requires to be tuned. A larger P can capture longer sub-sequence patterns, but does introduce much higher computational cost. Actually, our experiment results (as will be discussed in Sect. 4.5) show that the system only achieves a marginal performance improvement when P is set to 64 and 128. Therefore, we set the value of P to 32 for the practical reason. We project each of bytecode in $B^{'}$ into a P-dimensional space by multiplying a weight matrix $W_E \in R^{256\times P}$:

$$b_i^E = b_i^{'} W_E, i \in [1, \alpha]. \tag{1}$$

Thus, a bytecode sequence is mapped into a matrix $B_E \in R^{P\times\alpha}$. The initial value of W_E is pre-trained by using Word2Vec [21] and can be fine-tuned during training.

Convolutional Layer. ByteDroid then applies one-dimensional convolutional layer whose input is matrix B_E. As each row in B_E represents a bytecode, the width of convolutional kernels should be same as the embedding size P. In the convolutional layer, we select the multiple-sizes of kernels, ranging from 2 to 8, and apply them in parallel to improve the receptive field of the CNN model. Formally, let W_{cov} be the convolutional kernels which are size of $K \times P$. K is the height of convolutional kernel, which is the number of bytecode bytes participating in each convolution operation. For different size of the convolutional

kernels, we arrange to use 64 different filters. Each convolutional kernel produces a feature map of size $\alpha \times 1$ as we use the same-padding in convolution operation. Since there are 64 filters for each kernel, the feature maps can be stacked to a matrix of size $\alpha \times 64$. Let out^j_{cov} be the feature maps produced by a kernels of size j:

$$out^j_{cov} = CC^{64}_{i=1}(ReLU(Cov(W^i_{cov}, B_E) + b^i_{cov})), j \in [2, 8], \tag{2}$$

where $ReLU = max\{0, x\}$ is the rectified linear unit function. $W^j_{cov} \in R^{P \times j}$ is the weight parameter and b^j_{cov} is the bias parameter. CC is the stack operation.

Global Max Pooling. Global max pooling is applied to make our CNN work regardless of the location of detected features, and fix the size of feature maps given any arbitrary size of bytecode. Then, we have:

$$out^j_{mp} = max(out^j_{cov}). \tag{3}$$

After global max pooling, we aggregate all the features in a matrix of size 7×64 as the following:

$$out_{aggr} = \{out^2_{mp}, out^3_{mp}, ..., out^8_{mp}\}, \tag{4}$$

where out_{aggr} refers to the results of the aggregation operation.

Fully Connected Layer and Output Layer. Finally, out_{aggr} is flattened and fed to the fully connected layer and the Output layer to obtain the label of the bytecode sequence. In the fully connected layer, 512 neurons are used. We also use dropout between the fully connected layer and the output layer to reduce over-fitting. Therefore, we have the fully connected layer as following:

$$out_{fc} = ReLU(W_{fc} out_{aggr} + b_{fc}), \tag{5}$$

where W_{fc} and b_{fc} respectively represent the weight and bias of the fully connected layer.

The output layer uses the output of the fully connected layer to calculate the probabilities of a bytecode sequence being benign and being malicious. Thus, there are two neurons in this layer. The Softmax function is then applied for normalizing the probabilities. The description of output layer is as following:

$$P(out_{fc}) = Softmax(W_{out} out_{fc} + b_{out}), \tag{6}$$

where W_{out} and b_{out} respectively represent the weight and the bias of the output layer. The classification result of the bytecode sequence is expressed in Eq. 7 and the result with the probability greater than 0.5 is used as the prediction.

$$Predict(B_E) = argmax(P(out_{fc})), \tag{7}$$

We denote the label of the bytecode sequence by L_i which is a two-dimensional vector (i.e., [0, 1] representing a malware, [1, 0] representing a benign

application). Given M training samples, we use the categorical cross entropy as the loss function:

$$Loss = -\frac{1}{M}\sum_{i=1}^{M} L_i \log Predict(B_i) + \frac{\lambda}{2M}\sum_{W} W^2, \tag{8}$$

where $\frac{\lambda}{2M}\sum_W W^2$ is the L2 regularization and λ is the attenuation coefficient. The loss function is minimized by using Adam [22] optimizer. In our system, we set the value of λ to 0.01. The learning rate is set to 0.01 and the dropout rate is 0.5.

4 Experiments

In this section, we show the experimental result of our proposed method. In the first set of experiments, we evaluate the detection performance of ByteDroid. In the second set of experiments, we evaluate the Generalization performance of ByteDroid using real-world datasets. In the third set of experiments, we evaluate the robustness of ByteDroid against typical obfuscation techniques. In the last set of experiments, we evaluate the hyper-parameters of our CNN.

4.1 Datasets

We use four datasets in our experiments, as depicted in Table 1.

Table 1. Description of the datasets.

Datasets	# Benign	# Malware	Source from
Dataset A	6420	4554	Kang et al. [23]
Dataset B	6982	6575	FalDroid [24]
Dataset C	-	9389	VirusShare [25]
Dataset D	-	979	VirusShare [25]
Dataset E	-	10479	Android PRAGuard [28]

Dataset A. This dataset is obtained from Kang et al. [23], with 10,874 samples in total. Among them, there are 6,420 benign samples and 4,554 malware samples. All of the benign samples are scanned by VirusTotal [26]. We only keep the applications that are not reported as malware by any anti-virus scanners.

Dataset B. This dataset is provided by FalDroid [24]. It contains 6,982 benign samples and 6,575 malware samples. Benign samples are obtained from Google Play Stores and are scanned by VirusTotal. All malware samples are labeled into 30 malware families as described in FalDroid.

Dataset C and D. There two datasets are obtained from VirusShare [25]. There are 9,389 samples from 2014 and 979 samples from 2017 to 2018.

Dataset E. is obtained from Android PRAGuard [28] that contains 10479 malware obfuscated by seven different obfuscation techniques. Particularly, each obfuscation category contains 1497 samples, which are from the Mal-Genome [2] and the Contagio Minidump [27] datasets.

We use Dataset A and Dataset B as the training sets, and Dataset C and Dataset D are used to evaluate the Generalization performance of ByteDroid. Then, we evaluate the performance of ByteDroid against obfuscation techniques using the Dataset E. The proposed CNN model is implemented by using the Python package of Tensorflow [30]. We train the CNN model with 4 Nvidia Titan X. As depicted in Table 2, we evaluate the performance of ByteDroid using accuracy, precision, recall and f1 score.

Table 2. Evaluation metrics

Metrics	Abbr.	Description
True Positive	TP	# of malware correctly detected
True Negative	TN	# of benign samples correctly classified
False Positive	FP	# of benign samples predicted as malware
False Negative	FN	# of malware predicted as benign sample
Accuracy	Acc.	(TP+TN)/(TP+TN+FP+FN)
Precision	Pre.	TP/(TP+TN)
Recall	Rec.	TP/(TP+FN)
F1-score	F1	2*Precision*Recall/(Precision + Recall)

4.2 Detection Performance

In this experiment, we measure the detection performance of ByteDorid using ten-fold cross validation based on the Dataset A and the Dataset B. To prevent over-fitting, cross validation is performed in ten rounds. In each round, we split the dataset into the training sets, validation sets and testing sets. The split ratio is 8:1:1.

Single Dataset Cross Validation. We firstly evaluate the performance of ByteDroid in single dataset. At the same time, we implement the method of N. Mclaughlin et al. [8] as a comparison and we refer this method as DAMD in the following experiments. Table 3 shows the detection results of ByteDroid and DAMD based on Dataset A and Dataset B. In addition, the result threshold is set to 0.5.

We can see that ByteDroid outperforms DAMD with the accuracy improvement as much as 6%. The reason can be explained as the following. First, ByteDroid utilizes both the opcode and the operands of the applications, while

Table 3. Single dataset cross validation of ByteDroid and DAMD.

Method	Data type	Datasets	Acc.	Pre.	Rec.	F1
Ours	Bytecode	A	0.984	0.972	0.991	0.981
DAMD	Opcode	A	0.968	**0.996**	0.927	0.960
Ours	Bytecode	B	**0.995**	0.992	**0.998**	**0.995**
DAMD	Opcode	B	0.936	0.957	0.887	0.920

DAMD only relies on the opcode. We believe the operands do carry the partial malware features and that is the reason why DAMD is less accurate than ByteDroid. Second, ByteDroid uses the multiple sizes of the convolutional kernels to improve the performance of the model architecture, while DAMD only has a single sized convolutional kernel.

Cross-Dataset Testing. It is commonly believed that the deep learning based malware detection cross the datasets is more challenging because the malware from different datasets may not share the features as the malware from the same dataset does. In this experiment, we test ByteDroid's detection accuracy on different datasets. In particular, we conduct the cross-dataset testing. We first train ByteDroid with Dataset A, and then use Dataset B as the testing set to evaluate ByteDroid's performance. We repeat the same experiment by swapping the training set and the testing set. Similarly, we also perform the same test for DAMD for the comparison purpose.

Table 4. Cross-dataset testing of ByteDroid and DAMD.

Method	TrainSet	TestSet	Acc.	Pre.	Rec.	F1
Ours	A	B	0.854	0.857	0.837	0.847
DAMD	A	B	0.682	**0.997**	0.551	0.710
Ours	B	A	**0.969**	0.974	**0.952**	**0.963**
DAMD	B	A	0.885	0.971	0.828	0.894

Table 4 shows the results of cross-dataset tests. Obviously, ByteDroid achieves the significant improvement in all metrics. It indicates that ByteDroid has much better cross-dataset testing performance than DAMD. In addition, it is interesting to find that both methods have much better detection accuracy when dataset B is used as the training set and the dataset A is used as the testing set. By examining the both datasets, we find the dataset B contains approximately 500 more benign applications and 2,000 more malicious applications than dataset A. It demonstrates that the size of the training set does affect the malware detection performance of both methods. When the training set is larger, the deep learning model has better chances to capture more malware features and therefore achieves higher accuracy in the detection stage. Nevertheless, the

testing results show ByteDroid is much more effective in the malware detection than DAMD.

Manual Analysis. When we use dataset B as the training set, ByteDroid correctly classifiers 96.9% of 10,874 applications (Dataset A), with 69 benign samples being misclassified as malware and 236 malware samples being misclassified as benign applications. To investigate the actual reasons that cause the above false positive and false negative, we perform the following manual analysis.

We use VirusTotal to rescan 69 benign samples that are reported incorrectly as malware. The results show that all of them are clean. After manual analysis, we find that 52 benign samples have the behaviors that are commonly performed in the malware, such as reading contact, reading SMS, collecting device ID, obtaining geographical location and device IMEI. We believe the behaviors' corresponding bytecode sequences are detected during the test and thus cause the false positive.

There are also 236 malware samples misclassified as benign applications. We find 194 malware out of 236 have their APK sizes less than 500 KB. These APKs have the short bytecode sequences, and thus lack the sufficient sequence patterns that match the malicious ones. The rest of the misclassified malware either hide the malicious code in its resources or disguise itself as an image such as install.png. The detection of these behaviors is out of the scope of our malware detection.

4.3 Generalization Performance

In this experiment, we evaluate the generalization performance of our proposed method in realistic scenarios. Based on the model trained using Dataset B, we use the Dataset C and the Dataset D to measure the Generalization performance of ByteDroid. Again, we take DAMD as a comparison. Table 5 shows the detection rates of ByteDroid and DAMD using the Dataset C and the Dataset D.

Table 5. Detection rates of ByteDroid and DAMD using Dataset C and D.

Method	Datasets	Years	# of malware	# of detected	Detected rate
Ours	C	2014	9389	7923	**84.38**%
DAMD	C	2014	9389	7253	77.24%
Ours	D	2017–2018	979	822	**83.96**%
DAMD	D	2017–2018	979	611	62.41%

We can clearly see that the malware detection rate of ByteDroid is at least 7% higher than DAMD. One may notice that the overall detection rate of ByteDroid is less than 85%. The reason is that the dataset (B) used for training is relatively old, and the datasets (C&D) under the test are relatively newer. We believe many newer malicious patterns of the malware in dataset C and D are not learned in

the training stage. To further investigate the reason of the false negative, we perform the following manual analysis.

Manual Analysis. In Dataset C, ByteDroid fails to detect 1,466 malware samples. We discover that 191 samples only detected by 1 out of 62 anti-scanners in VirusTotal. There are 146 samples used native code or dynamic code loading to evade the detection. The malicious behaviors of these malware samples are not obviously under the static analysis.

In Dataset D, ByteDroid misses 157 malware samples. One-half of the 157 samples contain the dynamically load code from a library. It is obvious that the obfuscation techniques become more popular in the newer malware. In addition, 10 malware samples use MultiDex technique to generate multiple DEX files in their APKs.

Nevertheless, experiment results demonstrate that ByteDroid is capable of catching the various malicious patterns once the malicious behaviors are present during the training.

4.4 Against the Obfuscation

In this experiment, we evaluate the robustness of ByteDroid in catching the Android malware with the obfuscation techniques. Currently, obfuscation techniques on Android platform have been very mature, and there are many automated obfuscation frameworks [28,29] available, which allow attackers to reduce the labor cost or even achieve the obfuscation capability without understanding the detailed obfuscation techniques.

We use the datasets (E) from the Android PRAGuard [28] for the evaluation. The Dataset E contains seven obfuscation techniques including trivial obfuscation, class encryption, string encryption, reflection and their combinations. We also train the CNN model using the Dataset B. Table 6 shows the detection results.

Table 6. Detection rate of ByteDroid against obfuscated malware.

ID	Obfuscation method	Detected rate	Miss rate
1	Trivial obfuscation	94.07%	5.93%
2	String encrption	88.02%	11.98%
3	Class encryption	89.90%	10.10%
4	Reflection	93.52%	6.48%
5	Combined 1, 2	90.48%	9.52%
6	Combined 1, 2, 4	90.40%	9.60%
7	Combined 1, 2, 3, 4	98.81%	1.19%
Average	-	92.17%	7.83%

The results shown in Table 6 indicate that ByteDroid effectively detect Android malware despite the use of different types of obfuscation techniques.

ByteDroid is more effective for detecting the malware applied with trivial obfuscation. As described in Android PRAGuard [28], the trivial obfuscation only affects the string and does not change the instructions in the bytecode. Yet, ByteDroid learns the malicious behaviors by the bytecode sequences rather than by the semantics of the strings.

ByteDroid has a poorer performance in defending other obfuscation techniques such as string encryption, class encryption and reflection because these techniques affect both the strings and the bytecode sequence. They introduce some noises may break the key bytecode sequence of a malware, which makes ByteDroid difficult to classify them correctly.

4.5 Hyperparameter

In this experiment, we evaluate the effectiveness of hyperparameters in ByteDroid, including embedding vector size, the maximum size of the bytecode sequence, the convolutional kernel sizes and the number of epochs. In the validation process, we explore these hyperparameters with ten-fold validation using the Dataset B.

Embedding Size P. In Fig. 3(a), the embedding size of the outermost ROC curve is 128. The ROC curves corresponding to the embedding size exceeding 32 are close to each other. Since the larger embedding vector size requires the longer training time and the higher computational cost, we choose to set the embedding size to 32 to make a balance between performance and efficiency.

Convolutional Kernel Sizes. In our CNN, the convolutional kernel with sizes are ranging from 2 to 8, and the stride is set to 1. Compared with the kernel sizes of 3, 5 and 7, which are commonly used in other CNN-based schemes, we find that the sizes with the powers of 2 (i.e., 2, 4, and 8) achieve better performance. Figure 3(b) shows the ROC curves with different convolution kernel sizes.

Maximum Length of the Bytecode Sequence α. Figure 3(c) shows the distribution of the bytecode size of the dataset. Nearly 90% of the bytecode samples are less than 2,500 KB. Consider that 2,500 KB is still a large number that incurs much computational cost, we search for a shorter size that fits the best for our system. To do that, we test the malware detection true positive rate with the bytecode sizes starting from 500 KB and gradually increasing at an interval of 500 KB. It can be seen from Fig. 3(d) that the receiver operating characteristic (ROC) curve does not improve much when the bytecode size is 1,500 KB or more. Based on the above result, we set 1,500 KB as the size of the bytecode sequence in the training stage.

Number of Epochs. We train the model for 20 Epochs and record the accuracy, the precision, the recall and the f1 score of every single epoch. The results are plotted in Fig. 4. It can be observed that all of the metrics keep a small range fluctuation after 9 epochs. Consequently, the CNN model can be trained to achieve good performance quickly after several epochs.

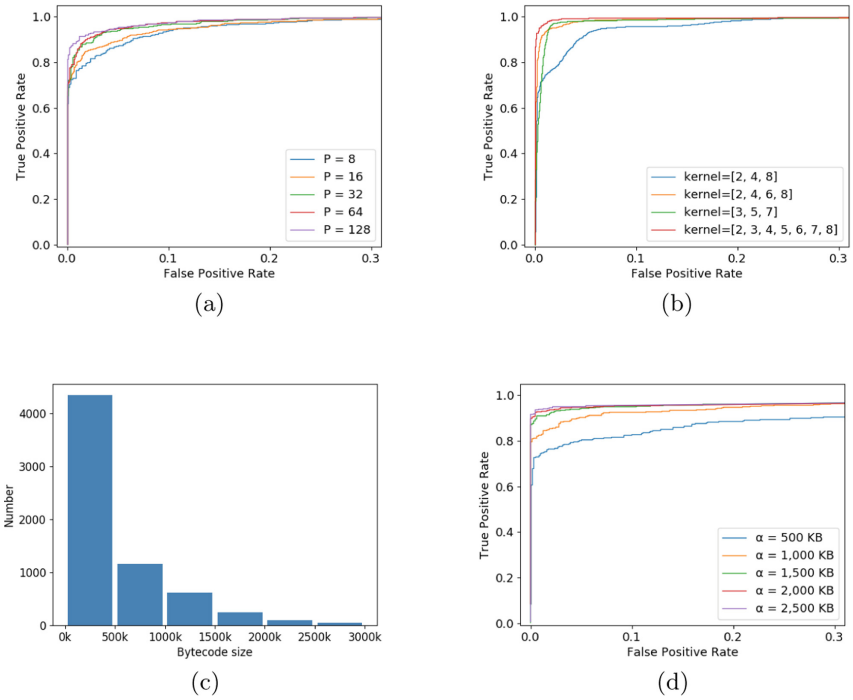

Fig. 3. Effectiveness of different hyperparameters. Figure 3(a) shows the ROC curves with different embedding vector sizes. Figure 3(b) shows the ROC curves for different convolutional kernel sizes. Figure 3(c) shows the size distribution of the bytecode. Figure 3(d) shows the ROC curves for different α.

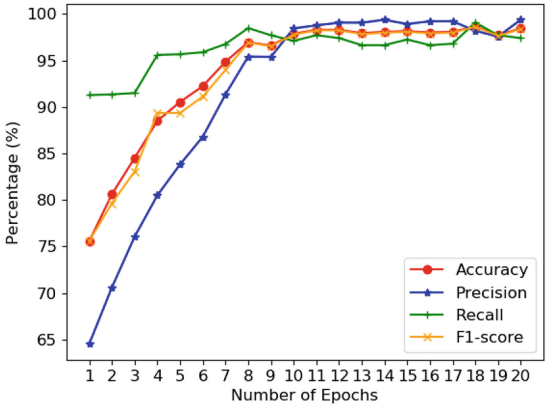

Fig. 4. The metrics versus the number of epochs.

5 Limitations

Since ByteDroid is a static analysis based malware detection scheme, it inherently fails to detect the malware that dynamically launches the malicious Dalvik bytecode or the native code from a library. We will consider adopting a dynamic analysis module in our future work.

One may notice that ByteDroid limits that bytecode sequence to 1,500 KB during the training stage. Attackers may take the advantage of this knowledge by inserting the malicious code beyond the 1,500 KB of the bytecode to evade the feature extraction. In fact, this bytecode size limitation can be easily relaxed given the actual dataset. In practice, we can group the APKs into different sets by their bytecode sizes and perform the training separately with the different bytecode size limitations. Note that the different datasets can be trained in a serialized fashion so that the training result is naturally aggregated.

Obviously, the performance of ByteDroid depends on the completeness of the Android application datasets. While our experiments demonstrate ByteDroid has a good generalization performance, the false positive or the false negative rate does increase when the application datasets are very different from the training datasets. We believe a dynamic malware detection module in our future work will significantly improve the ByteDroid's detection rate in this scenario.

6 Conclusion

In this paper, we propose ByteDroid, an automatic Android malware detection system using Convolutional Neural Network. ByteDroid eliminates the need of manual feature extraction. It applies multiple convolutional kernels to learn the sequential patterns of the bytecode. The effectiveness and generalization performance of ByteDorid are evaluated in our experiments. In addition, ByteDroid is robust against several typical obfuscation techniques.

In future work, we plan to explore more complex network architecture and consider dynamic analysis to detect dynamically loaded code. Moreover, to better understand malware behaviors, the interpretability of the neural networks is also what we concerned about.

Acknowledgements. We would like to thank the anonymous reviewers for their insightful comments to improve our paper. This work was supported by the National Natural Science Foundation of China under Grant No. 61672543, by the Open Research Fund of Key Laboratory of Network Crime Investigation of Hunan Provincial Colleges Grant No. 2017WLFZZC002.

References

1. Comparetti, P.M., Salvaneschi, G., Kirda, E., Kolbitsch, C., Kruegel, C., Zanero, S.: Identifying dormant functionality in malware programs. In: 2010 IEEE Symposium on Security and Privacy, pp. 61–76. IEEE (2010)

2. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy, pp. 95–109. IEEE (2012)
3. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party Android marketplaces. In: Proceedings of the Second ACM Conference on Data and Application Security and Privacy, pp. 317–326. ACM (2012)
4. Chen, S., Xue, M., Tang, Z., Xu, L., Zhu, H.: Stormdroid: a streaminglized machine learning-based system for detecting Android malware. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, pp. 377–388. ACM (2016)
5. Narudin, F.A., Feizollah, A., Anuar, N.B., Gani, A.: Evaluation of machine learning classifiers for mobile malware detection. Soft Comput. **20**(1), 343–357 (2016)
6. Milosevic, N., Dehghantanha, A., Choo, K.K.R.: Machine learning aided Android malware classification. Comput. Electr. Eng. **61**, 266–274 (2017)
7. Li, J., Sun, L., Yan, Q., Li, Z., Srisa-an, W., Ye, H.: Significant permission identification for machine-learning-based Android malware detection. IEEE Trans. Industr. Inform. **14**(7), 3216–3225 (2018)
8. McLaughlin, N., et al.: Deep Android malware detection. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, pp. 301–308. ACM (2017)
9. Wu, D.J., Mao, C.H., Wei, T.E., Lee, H.M., Wu, K.P.: Droidmat: Android malware detection through manifest and API calls tracing. In: 2012 Seventh Asia Joint Conference on Information Security, pp. 62–69. IEEE (2012)
10. Yang, C., Xu, Z., Gu, G., Yegneswaran, V., Porras, P.: DroidMiner: automated mining and characterization of fine-grained malicious behaviors in Android applications. In: Kutyłowski, M., Vaidya, J. (eds.) ESORICS 2014. LNCS, vol. 8712, pp. 163–182. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11203-9_10
11. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: effective and explainable detection of Android malware in your pocket. In: NDSS, vol. 14, pp. 23–26 (2014)
12. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: behavior-based malware detection system for Android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 15–26. ACM (2011)
13. Wu, W.C., Hung, S.H.: Droiddolphin: a dynamic android malware detection framework using big data and machine learning. In: Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems, pp. 247–252. ACM (2014)
14. Lindorfer, M., Neugschwandtner, M., Platzer, C.: Marvin: efficient and comprehensive mobile app classification through static and dynamic analysis. In: 2015 IEEE 39th Annual Computer Software and Applications Conference (COMPSAC), vol. 2, pp. 422–433. IEEE (2015)
15. Hou, S., Ye, Y., Song, Y., Abdulhayoglu, M.: Hindroid: an intelligent Android malware detection system based on structured heterogeneous information network. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1507–1515. ACM (2017)
16. Yuan, Z., Lu, Y., Xue, Y.: Droiddetector: Android malware characterization and detection using deep learning. Tsinghua Sci. Technol. **21**(1), 114–123 (2016)
17. Li, W., Wang, Z., Cai, J., Cheng, S.: An Android malware detection approach using weight-adjusted deep learning. In: 2018 International Conference on Computing, Networking and Communications (ICNC), pp. 437–441. IEEE (2018)

18. Hou, S., Saas, A., Chen, L., Ye, Y., Bourlai, T.: Deep neural networks for automatic android malware detection. In: Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, pp. 803–810. ACM (2017)
19. Nix, R., Zhang, J.: Classification of Android apps and malware using deep neural networks. In: 2017 International Joint Conference on Neural Networks (IJCNN), pp. 1871–1878. IEEE (2017)
20. Xu, K., Li, Y., Deng, R.H., Chen, K.: Deeprefiner: multi-layer Android malware detection system applying deep neural networks. In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 473–487. IEEE (2018)
21. Le, Q., Mikolov, T.: Distributed representations of sentences and documents. In: International Conference on Machine Learning, pp. 1188–1196 (2014)
22. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
23. Kang, H., Jang, J.W., Mohaisen, A., Kim, H.K.: Detecting and classifying Android malware using static analysis along with creator information. Int. J. Distrib. Sens. Netw. **11**(6), 479174 (2015)
24. Fan, M., et al.: Frequent subgraph based familial classification of Android malware. In: 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), pp. 24–35. IEEE (2016)
25. Roberts, J.M.: Virus share (2011). https://virusshare.com
26. Total, V.: Virustotal-free online virus, malware and URL scanner (2012). https://www.virustotal.com
27. Mobile, C.: Mobile malware mini dump (2013). http://contagiominidump.blogspot.com
28. Maiorca, D., Ariu, D., Corona, I., Aresu, M., Giacinto, G.: Stealth attacks: an extended insight into the obfuscation effects on Android malware. Comput. Secur. **51**, 16–31 (2015)
29. Rastogi, V., Chen, Y., Jiang, X.: Droidchameleon: evaluating Android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, pp. 329–334. ACM (2013)
30. Abadi, M., et al.: Tensorflow: a system for large-scale machine learning. In: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 2016), pp. 265–283 (2016)