Ad Hoc Networks 101 (2020) 102098

Contents lists available at ScienceDirect

Ad Hoc Networks

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

End-to-end malware detection for android IoT devices using deep learning

Zhongru Ren[a], Haomin Wu[d], Qian Ning[c], Iftikhar Hussain[e], Bingcai Chen[a,b,*]

[a] School of Computer Science and Technology, Dalian University of Technology, Dalian 116024, China
[b] School of Computer Science and Technology, Xinjiang Normal University, Urumqi 830054, China
[c] College of Electronics and Information Engineering, Sichuan University, Chengdu 610065, China
[d] Electronic Information Engineering Department of Wuhan Polytechnic, Wuhan 430074, China
[e] School of Computer Science and Technology, University of Science and Technology of China, Hefei 230000, China

ARTICLE INFO

Article history:
Received 1 December 2019
Revised 5 February 2020
Accepted 12 February 2020
Available online 14 February 2020

Keywords:
Android malware detection
IoT
End-to-end
Deep learning

ABSTRACT

The Internet of Things (IoT) has grown rapidly in recent years and has become one of the most active areas in the global market. As an open source platform with a large number of users, Android has become the driving force behind the rapid development of the IoT, also attracted malware attacks. Considering the explosive growth of Android malware in recent years, there is an urgent need to propose efficient methods for Android malware detection. Although the existing Android malware detection methods based on machine learning has achieved encouraging performance, most of these methods require a lot of time and effort from the malware analysts to build dynamic or static features, so these methods are difficult to apply in practice. Therefore, end-to-end malware detection methods without human expert intervention are required. This paper proposes two end-to-end Android malware detection methods based on deep learning. Compared with the existing detection methods, the proposed methods have the advantage of their end-to-end learning process. Our proposed methods resample the raw bytecodes of the classes.dex files of Android applications as input to deep learning models. These models are trained and evaluated in a dataset containing 8K benign applications and 8K malicious applications. Experiments show that the proposed methods can achieve 93.4% and 95.8% detection accuracy respectively. Compared with the existing methods, our proposed methods are not limited by input filesize, no manual feature engineering, low resource consumption, so they are more suitable for application on Android IoT devices.

1. Introduction

Internet of Things (IoT) has developed rapidly in recent years, and its widespread use in many fields, such as industrial control, vehicle IoT, smart home, and healthcare has made the importance of IoT security even more prominent. According to reports [1], IoT devices have become a key entry point for targeted attacks. Although routers and webcams currently account for 90% of infected devices, almost every IoT device is vulnerable. In particular, Android-based IoT devices have become one of the major targets of malware attacks due to the openness and universality of the Android platform, as shown in Fig 1. Malicious behaviors, including privacy theft, malicious deduction, traffic consumption, and remote control, seriously threaten the security and privacy of infected devices. According to statistics [2], the total number of Android malware detected worldwide in 2018 reached to 26.61 million, and the monthly increase of malware reached to 520,000. In particular, a large number of Android malware uses various techniques, such as obfuscation and encryption, to evade detection, making malware detection more difficult [3]. The rapid expansion of Android malware has become a huge barrier for malware analysts. Therefore, researching efficient Android malware detection methods has become a hot issue.

Anti-virus software companies, such as Kaspersky, McAfee, TrendMicro, and Comodo have also released their software to protect against Android malware. However, studies have shown that the performance of these anti-virus software is not satisfactory, especially for malware with code obfuscation and encryption [4]. Android malware detection also has a large number of significant achievements in academia. In general, the existing methods for Android malware detection are roughly classified into three categories [5]. The first one is the static detection, including signature-based methods, permission-based methods, bytecode-based methods, and hybrid static analysis methods. These methods check

* Corresponding author.
E-mail address: china@dlut.edu.cn (B. Chen).

https://doi.org/10.1016/j.adhoc.2020.102098
1570-8705/

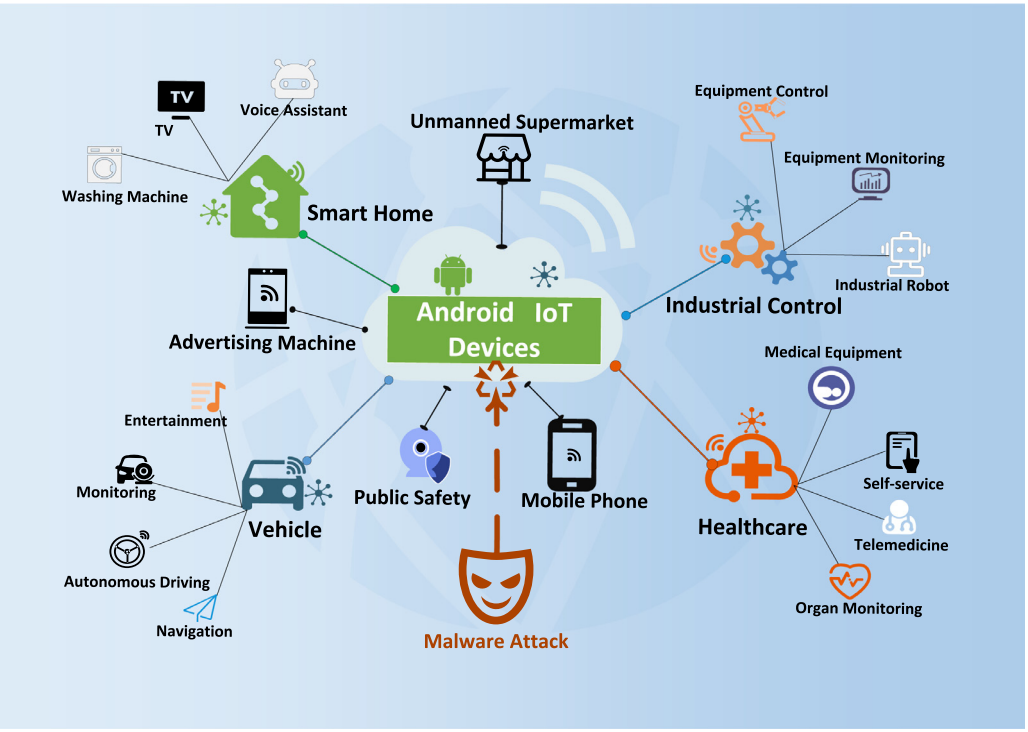


Fig. 1. Android has become the driving force behind the rapid development of the IoT, also attracted malware attacks.

static Android applications to find any potential malicious features without application execution. The second is dynamic detection that executes applications in isolated environments (such as sandbox, simulator, virtual machine) and then determines whether they are malicious or not by monitoring and tracking their behaviors. The combination of static detection and dynamic detection is the third one, called hybrid detection. Traditional malware detection methods rely mainly on the accumulation of signature libraries and human intervention by malware analysts, so it has been difficult to adapt to the explosive growth of Android malware [6].

With the accumulation of data and the continuous improvement of computing power, machine learning technology has also been widely applied to these three types of detection methods, providing another perspective for efficient and automatic Android malware detection. The machine learning based Android malware detection methods mainly consist of four steps: First, datasets are built by collecting benign and malicious Android applications. Second, feature engineering is performed to extract features to characterize Android applications. Third, machine learning models are trained for detecting malware. Fourth, the performance of the trained models are evaluated by predicting test samples [5]. Among them, feature engineering plays a vital role in machine learning and it is also an arduous, time-consuming manual work that usually requires a lot of domain expert knowledge. For example, machine learning based static detection methods often require special decompilation tools to parse binary files, and manual rules are required to extract useful information [7]. Dynamic detection methods are more complicated that the target application needs to be executed in isolation to obtain behavioral features such as function calls, permission requests, and network communications [8–10]. Given the high cost of feature engineering, researchers are increasingly focusing on the techniques that enable automatic feature learning to get rid of the intervention of security experts.

In 2011, Nataraj et al. [11] interpret the raw bytecode of Windows malware as grayscale images for the first time. They applied the feature engineering techniques in the image processing field to malware, freeing the malware classification task from the security domain. In recent years, end-to-end deep learning techniques have achieved tremendous breakthroughs in the fields of image identification [12], text classification [13], and speech recognition [14] due to its powerful data mining, learning, and expression capabilities. By replicating these successes to malware detection, several end-to-end malware detection methods have been proposed to alleviate the pressure on malware analysts. Huang et al. [15] converted classes.dex files of Android applications into colored images and developed an Android malware detection system (R2-D2) using Convolutional Neural Network (CNN), but the model they proposed is too large to deploy to the IoT devices. Raff et al. [16] used different ideas to detect Windows malware. They treated the entire Windows executable file as a long text sequence and slightly improved a text classification model (TextCNN) [13] to detect malware, but the input size of their model is limited to 2MB. In this paper, we propose two end-to-end Android malware detection methods (DexCNN and DexCRNN) based on deep learning without manual feature engineering. The deep learning models are trained and evaluated on the resampled classes.dex files of Android applications. Compared with the existing Android malware detection methods, our proposed methods are not limited by the input filesize, no manual feature engineering, so they are more suitable for Android IoT devices.

The main contributions are as follows:

(1) We use the two resampling methods for the first time to preprocess classes.dex files of Android applications into fixed-size sequences. The preprocessed sequences can directly input to deep learning models for training.
(2) We propose two malware detection deep learning models. The first model called DexCNN can achieve a detection accuracy of 93.4%, and the second model called DexCRNN can achieve a detection accuracy of 95.8%.

The rest of this paper is organized as follows. Section 2 presents the background. Section 3 describes the preprocessing methods and malware detection models. Model training and evaluation is given in Section 4. Finally, we conclude the paper in Section 5.

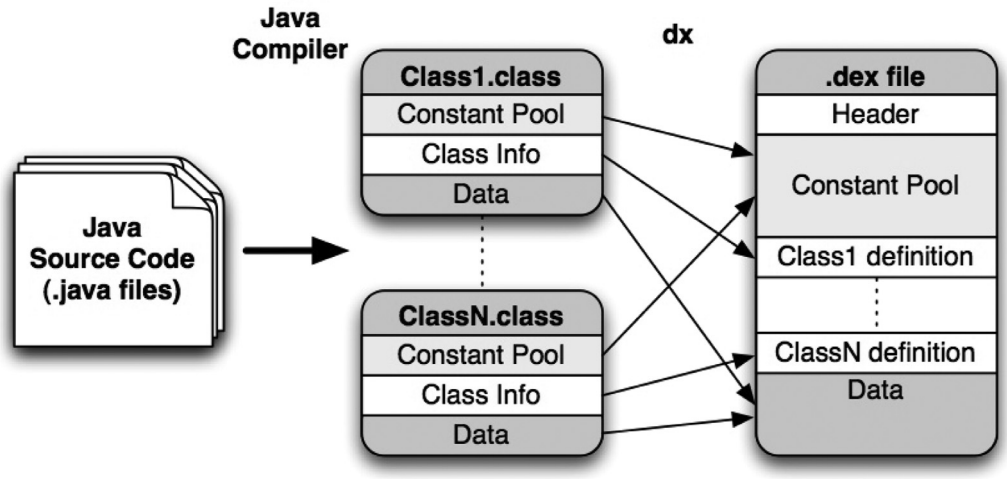


Fig. 2. The process of rebuilding class files into a dex file.

Table 1
APK file structure.

META-INIF/	APK signatures and certificates directory
res /	resource directory
libs /	.so libraries directory
assets /	static file directory
AndroidManifest.xml	APK configuration file
classes.dex	dalvik bytecode file can be executed
resource.asc	the compiled resource file

2. Background

In this section, we briefly introduce the Android application and review the related work of Android malware detection. Android application package (APK) is a zip-formatted archive file that can be installed on Android-based IoT devices. The file structure of an APK is shown in Table 1. The APK author converts the compiled java class files to the dex file via the dx tool of the Android SDK. The dx tool can combine, reorganize, and optimize multiple class files to reduce the size and shorten the running time. In this way, the dex file becomes the core of the entire APK and can be interpreted and executed by the Dalvik virtual machine of the Android platform. The converted dex file, as shown in Fig 2, including the header section, the constant pool section, the class definition section, and the data section, stores information about all classes of the APK. So the methods proposed in this paper use the classes.dex file to represent an APK file.

2.1. Traditional android malware detection

Traditional Android malware detection methods can be roughly classified into three categories: static detection, dynamic detection, and hybrid detection.

Static detection, including signature-based methods, permission-based methods, bytecode-based methods, and hybrid static analysis methods, check static APKs to find any potential malicious features without application execution. The signature-based methods are based on specific character sequences or semantic patterns in the source file to generate accurate and robust signatures [7]. Such kind of methods have been widely used by commercial anti-malware products due to their efficiency. Permissions are the first security gate against malware on the Android platform. The requested permissions must be declared in the AndroidManifest.xml file by the software author. So the permission-based approach primarily analyzes the sensitive or suspicious permissions requested to identify potential malware. Such as Kirin proposed by Enck et al. [17], PUMA proposed by Sanz et al. [10]. The dex bytecode file contains abundant semantic information of application behaviors, such as API (Application Programming Interface) calls, data flows. Therefore, the bytecode-based method mainly analyzes APKs by disassembling dex files, such as DroidAPIMiner [18] and MamaDroid [19]. To further improve the detection accuracy, hybrid static analysis extracts various types of features by analyzing multiple files, such as Drebin [20] and FlowDroid [21]. In summary, static analysis methods are fast and efficient but are powerless for malware with code obfuscation and encryption [22].

Dynamic detection that executes APKs in isolated environments (such as sandbox, simulator, virtual machine) and then determines whether they are malicious or not by monitoring and tracking their behaviors. System-wide Android dynamic analysis system TaintDroid [23] detects malware by tracking the flow of sensitive data. DroidScope [24] reconstructs Linux OS level and Java Dalvik level semantic information to dynamically analyze APKs. Other dynamic features, such as system calls, network traffic, user interaction, system component, are also used by dynamic detection [5]. Dynamic detection methods are effective against code obfuscation and encryption techniques, and can easily detect malicious behaviors that may be missed by static detection [3]. However, the code coverage of dynamic detection is too low to run all branches of one application, thus it is easy to ignore some hidden code sections that will be triggered at certain times or scenes. Most importantly, dynamic detection methods cost more time and resources.

The hybrid detection combines static detection and dynamic detection. It not only analyzes the files of static APKs but also monitors the behaviors of the APKs through execution, such as [25,26]. Therefore, hybrid detection methods can combat code obfuscation or encryption while improving code coverage. However, the large amount of time and resource consumption makes it limited in actual deployment.

2.2. Machine learning based android malware detection

Traditional malware detection methods rely mainly on the accumulation of signature libraries and expertise of malware analysts, so it has been difficult to adapt to the explosive growth of Android malware. Fortunately, machine learning technology provides another perspective for efficient and automatic Android malware detection. The machine learning based Android malware de-

tection methods mainly includes four steps: dataset constructing, feature engineering, model training, and model evaluation. Feature engineering, which is critical to the validity of the model, extracts informative and robust features to characterize APKs. Typically, features used to characterize APKs include AndroidManifest.xml file features, classes.dex features, and dynamic behavior based features. The AndroidManifest.xml file records basic information about an APK, such as hardware information, requested permissions, APK components, and filtered intents. In [27], AndroidManifest.xml files are parsed to extract key features for malware detection. The classes.dex file can be converted to a Smali format file that contains Dalvik commands (including the opcode and operands). As in [28], n-grams features are extracted from the opcode sequence to train malware classification models. Advanced features that obtained by disassembling classes.dex files, such as control flow diagrams [29], API dependency graphs [30], can also be used to train malware detection models. As in the literature [9,31], dynamic behavioral features such as file operations, network operations, encryption operations, service opening, phone calls, and system calls can be obtained by executing applications in an isolated environment. Using these dynamic features combined with static features to detect malware can achieve higher detection performance. In terms of model selection, both traditional machine learning models (e.g., Support Vector Machines (SVM) [20], Random Forests (RF) [19]) and deep learning models (e.g., CNN [32], Deep Belief Network (DBN) [33], Long Short-Term Memory (LSTM) [34]) are widely used in Android malware detection.

2.3. End-to-end android malware detection

High cost of feature engineering in machine learning based detection methods calling for end-to-end automated feature engineering methods to alleviate the pressure of malware analysts. Huang et al. [15] converted classes.dex files of Android APK into colored images and developed an Android malware detection system (R2-D2) using CNN. Their proposed detection system was tested on a large dataset with a best accuracy of 97.7%. However, their model is too large to be deployed to the IoT devices, so the converted image needs to be uploaded to the GPU server for further detection. Hasegawa et al. [35] developed a light-weight malware detection system that uses one-dimensional CNN (1D-CNN) to analyze a small part of the raw APK file without unpacking. Since the patterns captured by their methods are not included in the key files of the APK, the proposed method requires detailed investigation. There are also some literature on end-to-end detection methods for Windows and Linux malware, such as [16,36]. Since the structure of APK files are more complicated than Windows PE (Portable Executable) files and Linux ELF (Executable and Linkable Format) files, detection methods based on PE and ELF files cannot be directly applied to APK files. Therefore, this paper proposes two preprocessing methods to resample APK files, and then use deep learning models for end-to-end feature learning and classification.

3. Methods

To solve the problem from traditional time-consuming and resource-consuming detection methods, our proposed methods first preprocess the input APKs into fix-size sequences, then train deep learning models on these sequences. In this section, we introduce two preprocessing methods and two deep learning models in detail.

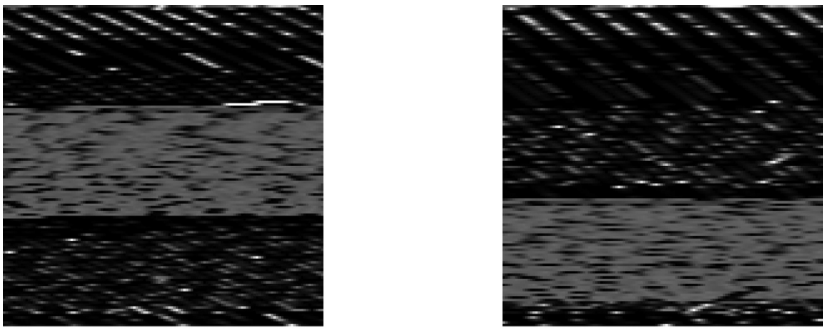
3.1. Preprocessing

Since the classes.dex files in different APKs are different in size and deep learning models usually require fixed-size inputs, these inputs need to be converted to fixed sizes. Short sequences of different lengths can be unified by filling operation, but long sequences (especially larger than 10MB) usually can not directly be used to input to deep learning models. Our preprocessing method first reads the classes.dex file of APK as an unsigned vector, and then converts the vector into a fixed size (L) by resampling [36,37]. use image resampling techniques for preprocessing the original inputs, but they mainly focus on malware family classification. Resampling algorithms commonly used in the field of image processing include Nearest Neighbor Interpolation (NNI), Bilinear Interpolation, and Bicubic Interpolation. Among them, NNI uses the gray value of the pixel closest to the sampling point as the target value, so the calculation is the simplest. Unlike two-dimensional images, here we use similar the idea for resampling one-dimensional sequences. Assuming that the source vector is $X = \{x_0, \ldots, x_n\}, n = L_0$, the NNI resampling vector is $A = \{a_0, \ldots, a_n\}, n = L_1$, then the scaling factor $K = L_0/L_1$ and $a_i = Kx_i$. But K may not be an integer, which means that there is no actual corresponding value in the source vector, so K will be rounded to $\lfloor K \rfloor$ to find the nearest neighbor value. Since the length of the original input vector is usually greater than L, that is, most of the inputs need to be downsampled, we use the Average-pooling (AVGPOOL) downsampling method for comparison, and the upsampling method still uses NNI. First, pad X with zero, $X = \{x_0, \ldots, x_{L_0}, 0, \ldots, x_n = 0\}, n = L_1 * \lceil K \rceil$, then the AVGPOOL downsampling vector is $A = \{a_0, \ldots, a_n\}, n = L_1$, where $a_i = average(x_i + \ldots + x_{(i+1)*K})$. To facilitate visualization, we reshape the converted vectors into 128 $\times$ 128 grayscale images. Fig 3 shows two different APKs of 'Ghost' family malware preprocessed using the NNI resampling. It can be seen that the similar features of the white diagonal stripes at the top of the two figures are very obvious. Fig 4 shows two different APKs of 'Spydealer' family malware preprocessed using the AVGPOOL downsampling and NNI upsampling. Different from the 'Ghost' family malware, the two malicious APKs of 'Spydealer' family have similar dot features. It should be noted that, unlike images, the sequence we are dealing with does not have two-dimensional spatial structure features, so we do not deliberately convert the vector into a 2-dimensional array when model training.

3.2. Model structure

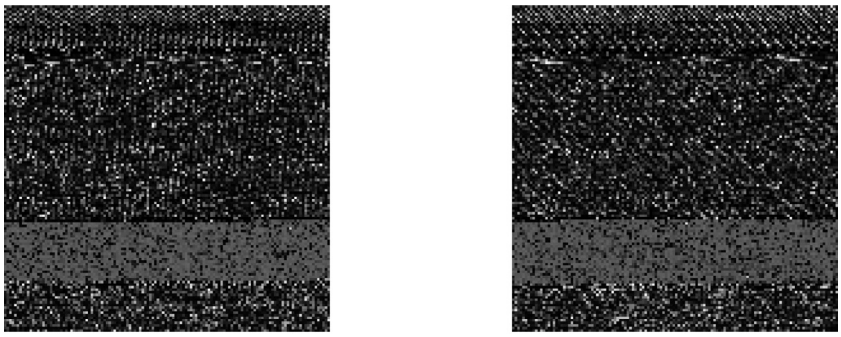
3.2.1. DexCNN

Inspired by TextCNN [13] and Malconv [16], we proposed the detection method of DexCNN. The TextCNN model which use CNN to perform text classification tasks proposed by Yoon Kim et al. in 2014. Multiple convolution kernels of different sizes are used to extract key information in sentences (similar to n-grams with multiple window sizes) leads to encouraging text classification performance. The advantage of TextCNN is that the network structure is simple and requires fewer parameters to train. Similar to the TextCNN structure, Raff et al. [16] proposed the model Malconv for Windows malware detection and achieved 94% detection accuracy. However, the input size of the Malconv is limited to 2MB, which consumes huge memory and causes the model to fail to detect files larger than 2MB. In addition, Malconv proved to be vulnerable to adversarial examples [38]. DexCNN we proposed also adopts a similar model, as shown in Fig 5. But the input of DexCNN is fixed size (L) sequences after preprocessing followed by Embedding layer, One-dimensional Convolution layer, Global Max-pooling layer, Fully Connected layer, and Softmax layer. The Embedding layer is actually a lookup table that maps each input byte to an 8-dimensional vector, so that the meaning of each input byte depends on the context rather than its value. Using the One-dimensional Convolution layer to get feature maps because the input is a long sequence and does not have a two-dimensional spatial structure. The Global Max-pooling layer is used after the Convolutional layer, such that



(a) GhostA (b) GhostB

Fig. 3. Grayscale images converted from 'Ghost' family malware using the Nearest Neighbor Interpolation resampling.



(a) SpydealerA (b) SpydealerB

Fig. 4. Grayscale images converted from 'Spydealer' family malware using the Average-pooling downsampling and Nearest Neighbor Interpolation upsampling.

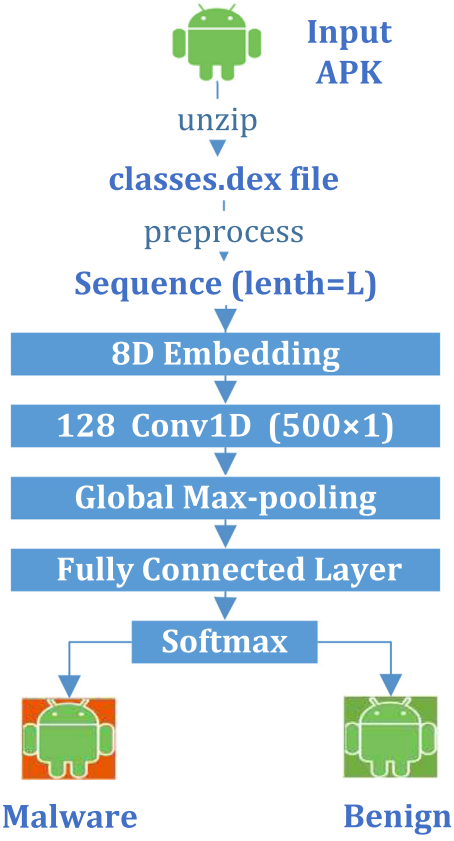


Fig. 5. The structure of DexCNN.

all feature maps are reduced to a fixed-size vector before entering into a fully-connected layer. Finally, the binary classification results are output by the Softmax layer. Since the DexCNN model uses the pre-processed fixed vector as the input of the Convolution layer, there is no limit to the size of the detected file.

3.2.2. DexCRNN

The DexCRNN proposed in this paper combines two types of neural networks, CNN and Recurrent Neural Network (RNN). RNN is used for speech recognition and Natural Language Processing (NLP) tasks because of its expertise in processing sequence data. However, naive RNN has the problem of short-term memory, which means that if a sequence is sufficiently long, the information will be hard to pass from the earlier time step to the later time step. LSTM and Gated Recurrent Unit (GRU) [39], which are the two most popular RNNs at present, can solve the short-term memory problem of naive RNN. They have structural improvements to the hidden layers of naive RNN, specifically because they have an internal mechanism called 'Gate' that regulates the flow of information. These 'Gates' can know which important data in the sequence needs to be learned and which ones need to be forgotten. In this way, information can be transferred along long sequences for prediction. Xu et al. [34] uses LSTM to train on the semantic structure of the Android dex file and achieves 97.74% accuracy. As end-to-end methods, we want to be able to use bytecode files directly, eliminating the need for file analysis steps. But the bytecode files are too large and of different sizes, so we reduce the dimension of the preprocessed sequence through the convolutional layer and then input it to the Recurrent layer. GRU is a variant of LSTM, and the performance of both is comparable in many tasks, so we also use GRU as a comparison. Both LSTM and GRU can only predict the output of the next moment based on the timing information of the previous moment. However, in some tasks, the

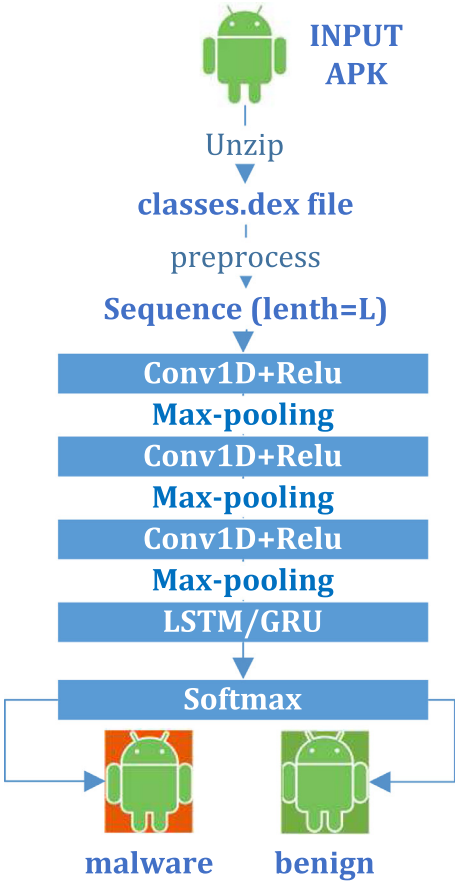


Fig. 6. The structure of DexCRNN.

output of the current moment is not only related to the previous state but also related to the future state [40]. Therefore, we also try Bi-directional LSTM (BiLSTM) and Bi-directional GRU (BiGRU) to accommodate the possible bi-directional dependencies of the code in dex files. The structure of DexCRNN is similar to DexCNN, as shown in Fig 6. Different from DexCNN, DexCRNN does not require the initial Embedding layer but add a Recurrent layer after the Convolutional layer.

After resampling Windows executable files, [36,37] use CNN for malware family classification and achieve encouraging classification results. So we also tried to remove the Recurrent layer and just use CNN for training. The experiment proved that there is almost no detection effect. Malware detection tasks are different from malware family classification tasks. The same family of malware tends to have similarities in filesize, structure, and content. Therefore, the classification model only needs to learn the common characteristics of the same family and the differences of different families, while the malware detection seeks to find the malicious features and benign features of all samples in the dataset to distinguish between benign software and malware. The image classification task directly uses CNN for feature learning because the meaning of the gray value of the pixel is fixed and does not need to be learned. Similarly, bytecodes in the same family of malware have similar meanings, so we believe that CNN can learn some advanced local features that can be used for malware classification but not enough for malware detection. The elements of our preprocessed malware sequence are not independent, so we need to learn the relationships between different values of the input sequence through the Embedding layer, just like DexCNN. Moreover, our pre-processed sequence still retains the timing information of the source file, so we can effectively capture the differences of different samples by adding the Recurrent layer to the CNN layer.

4. Experiments

4.1. Dataset and evaluation measures

In this article, the dataset consists of malware and benign applications, including 8000 benign APKs collected from the Google Play store [41] and 8000 malicious APKs collected from Virusshare [42]. The filesize of APK we collected is between 20KB and 50MB.

We use the following measures to evaluate the performance of the proposed methods:

(1) Accuracy, which is defined as Eq. (1), where T_p F_p T_n F_n indicates true positives, false positives, true negatives, and false negatives respectively.

$$Accuracy = \frac{T_p + T_n}{T_p + T_n + F_p + F_n} \tag{1}$$

(2) TPR (True Positive Rate, Recall), FPR (False Positive Rate), Precision, and F1-score, as shown in Eq. (2).

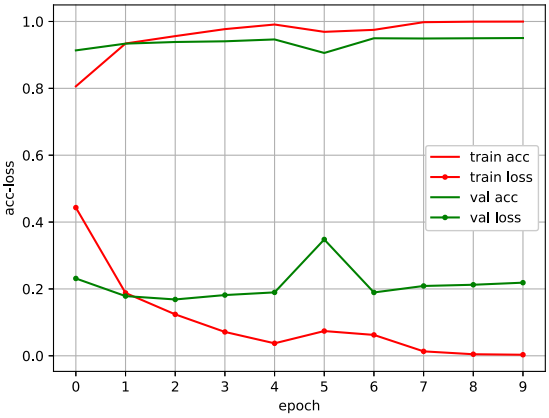
$$TPR = \frac{T_p}{T_p + F_n}, \quad FPR = \frac{F_p}{T_n + F_p}, \quad Pre = \frac{T_p}{T_p + F_p},$$
$$F1 = \frac{2T_p}{2T_p + F_p + F_n} \tag{2}$$

(3) ROC (Receiver Operating Characteristic) curve: The curve with FPR as the abscissa and TPR as the ordinate. The closer the curve is to the (0,1), the more accurate the model classification. AUC (Area under Curve) indicates the area under the ROC curve, which is between 0.1 and 1.0. AUC score is the probability that a classification model will rank a randomly chosen positive sample higher than a random negative one. A larger value indicates a better classification model.

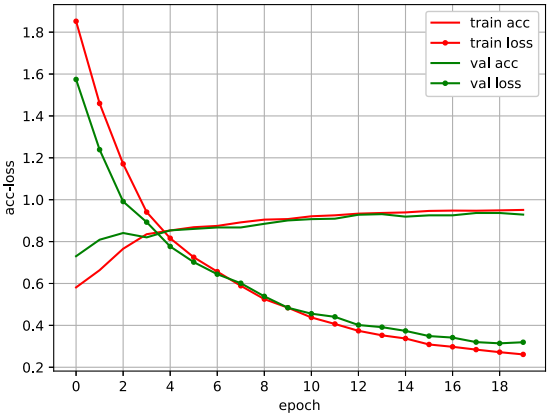
4.2. Results discussion

The dataset is split to training set, validation set, and test set account for 80%, 10%, and 10%, respectively. The proposed methods are trained and evaluated on the constructed dataset. The hyperparameters of the DexCNN model are as follows: the size of sequences L is 80000, the convolution filter size is 500, the stride is 500, and the batchsize is set to 96. The hyperparameters of DexCRNN are as follows: L is 16384, the convolution filter size of three One-dimensional Convolution layers is 6, and the number of convolution filters are 32, 64, 128 respectively. The size of the Max-pooling layer is 6 and the output size of LSTM/GRU is 128. The batchsize is set to 640. All models are implemented using the Keras library in Python. The NNI resampling is used as the default preprocessing method. Accuracy is used as a metric for model training. Binary_crossentropy is used as the loss function and Adam is used as the optimizer. All experiments were performed on a NVIDIA Tesla K80 GPU server using CUDA.

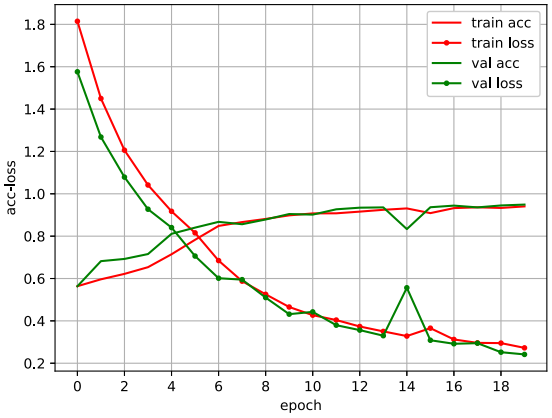
Fig 7 visualizes the loss and the accuracy on the training dataset and validation dataset for the different models. As shown in Fig 7 a, it can be seen that DexCNN has the fastest training speed, and the training accuracy and validation accuracy can exceed 92% after 2 epochs training. However, in the subsequent training, the loss curve and accuracy curve of the validation set show small fluctuations. After 10 epoch, DexCNN achieves a validation loss of 0.21 and a validation accuracy of 93.33%. As shown in Fig 7 b and c, it can be seen that the accuracy curve of DexCRNN_GRU and DexCRNN_LSTM converges slowly, and gradually stabilizes after 10 epochs. The curve of DexCRNN_GRU is smoother, and the curve of DexCRNN_LSTM fluctuates at 14th epoch, which



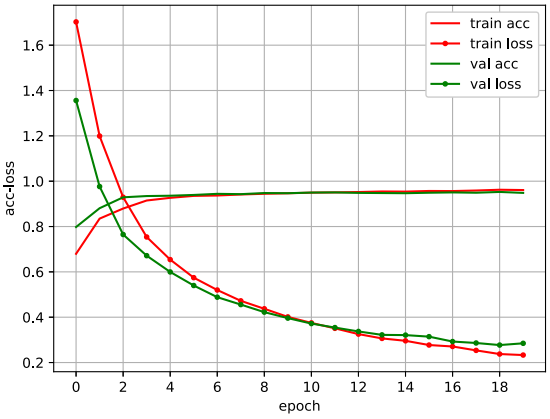
(a) DexCNN



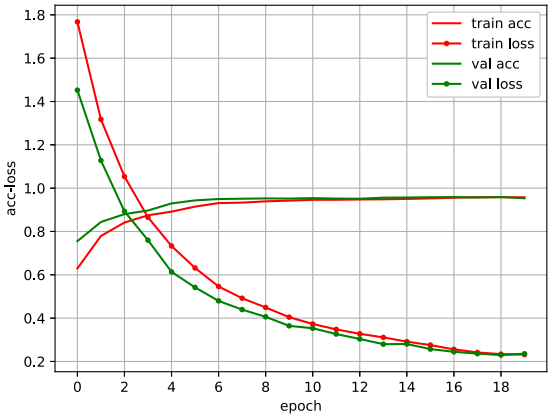
(b) DexCRNN_GRU



(c) DexCRNN_LSTM



(d) DexCRNN_BiGRU



(e) DexCRNN_BiLSTM

Fig. 7. Loss and accuracy on the training dataset and validation dataset for the different models.

indicates that over-fitting occurs. In Fig 7 d and e, it can be seen that the accuracy curve of DexCRNN_BiGRU and DexCRNN_BiLSTM converges faster than the two uni-directional models. The accuracy curve of these two bi-directional models is stable after 4 epochs, and can converge to a higher accuracy than unidirectional models. After 20 epochs training, DexCRNN_BiLSTM was able to converge to the highest accuracy in all models, with a validation loss of 0.2356 and a validation accuracy of 95.33%. It should be noted that for the purpose of reflecting the generalization ability of our proposed method, we did not deliberately adjust the hyperparameters (such as learning rate, batchsize, number of neurons, etc.) of the model by fitting the accuracy of the validation set. We detect the occurrence of overfitting by comparing the accuracy and loss of the validation set with the training set so that the optimizer can be stopped early.

Experiments are performed by 10 times, each time the dataset are shuffled and split to training set, validation set, and test set. DexCNN and DexCRNN are evaluated on the test set after 10 epochs and 20 epochs training respectively. Accuracy, precision, recall, and F1-score are used as evaluation metrics. As shown in Table 2, all the results listed are the average of 10 experiments. It can be seen that the accuracy of DexCNN is 93.6%, and the

Table 2
Performance comparison of different methods by using NNI preprocessing.

Method	Accurancy	Precision	Recall	F1-score
DexCNN	93.6%	91.8%	**95.7%**	0.937
DexCRNN_GRU	93.7%	91.7%	96.1%	0.939
DexCRNN_LSTM	93.4%	93.7%	93.1%	0.934
DexCRNN_BiGRU	94.4%	**94.9%**	93.8%	0.944
DexCRNN_BiLSTM	**94.9%**	93.7%	96.4%	**0.950**

Table 3
Performance comparison of different methods by using NNI upsampling and AVGPOOL downsampling.

Method	Accurancy	Precision	Recall	F1-score
DexCNN	93.4%	89.7%	**98.1%**	0.937
DexCRNN_GRU	95.4%	95.2%	95.5%	0.954
DexCRNN_LSTM	93.7%	92.1%	95.6%	0.938
DexCRNN_BiGRU	**95.8%**	**95.4%**	96.2%	**0.958**
DexCRNN_BiLSTM	95.1%	94.3%	96.0%	0.951

F1-score is 0.937, which is comparable to the performance of the two uni-directional models (DexCRNN_GRU and DexCRNN_LSTM) of DexCRNN. The performance of the bi-directional models (DexCRNN_BiGRU and DexCRNN_BiLSTM) is better than that of the uni-directional models, where DexCRNN_BiLSTM performs best with a detection accuracy of 94.9% and an F1-score of 0.950. For comparison, we also used the AVGPOOL downsampling and NNI upsampling to preprocess the input samples, and all the models were trained and evaluated under the same hyperparameter settings again. The evaluation results are shown in Table 3. Compared with the NNI downsampling method, by using AVGPOOL downsampling, DexCNN achieves the same F1-score, the performance of DexCRNN_GRU slightly improved while DexCRNN_LSTM decreased slightly. DexCRNN_BiGRU achieves the highest performance improvement with the accuracy of 95.8% and the F1-score of 0.958. In addition, regardless of which preprocessing method is used, DexCNN always has the highest recall rate and low precision, indicating that this model can detect the most malware but will generate more false positives.

Sample distribution is often unbalanced in actual malware detection tasks, and the distribution of positive and negative samples may change over time. The Precision-Recall (PR) curve is also widely used to evaluate the classification performance of machine learning algorithms. The curve with TPR as the abscissa and Precision as the ordinate. The closer the curve is to the (1,1), the more accurate the model classification. However, when the sample distribution in the test set is transformed, the PR curve often changes greatly, but the ROC curve can remain unchanged. Fig 8 shows the ROC curves and PR curves of DexCRNN_BiGRU in test sets with different distributions. It can be seen that the ratios of benign and malicious APKs in the three different distribution test sets are 1:1, 1:4, and 1:16 respectively. The more disparate the ratio, the sharper the PR curve decline, but the ROC curves is unchanged. The ROC curves of different methods are shown in Fig 9. The larger the area under the ROC curve, that is, the higher the AUC score, the easier it is to detect malware from the sample to be tested. It can be seen that the ROC curves of the 4 DexCRNN models are very close, and the AUC scores are between 0.976 and 0.984. This shows that the effectiveness of our method is minimally affected by changes in the RNN models. The ROC curve of the DexCNN model is also very similar to that of DexCRNN. The AUC score is 0.982, which shows that its performance is comparable to that of DexCRNN.

Table 4 compares the proposed method with the state-of-art deep learning based detection methods. The 'Dataset' column indicates the number of samples in the dataset. 'M' column indicates whether manual feature engineering is required. 'L' column indicates whether there is an input size limit. It can be seen that the end-to-end method proposed by us can achieve a detection accuracy of 95.8% and a F1-score of 0.958, which is not inferior to most methods in terms of performance. The DroidDeepLearner [33] uses API calls and risky permission to build a DBN detection model, which achieves a F1-score of 0.945. The DeepClassifyDroid [32] uses multi features, including API calls, permissions, and intent filters to bulid an Android malware detection system based on CNN, achieving 97.4% accuracy and 0.974 F1-score. The HADM [43] is a DNN-based hybrid Android malware detection system that uses 9 static features and system calls obtained by running APKs. It can be seen that most of the existing methods focus on feature engineering techniques to improve detection performance, and the advantage of our approach is the end-to-end learning process without the need for manual feature engineering [15,35]. are two end-to-end detection methods that achieve high detection ac-

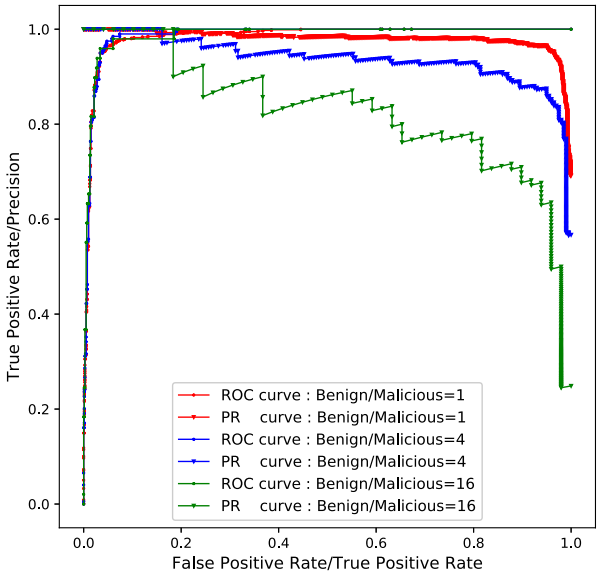


Fig. 8. Comparison of ROC curves and PR curves of DexCRNN_BiGRU in test sets with different distributions.

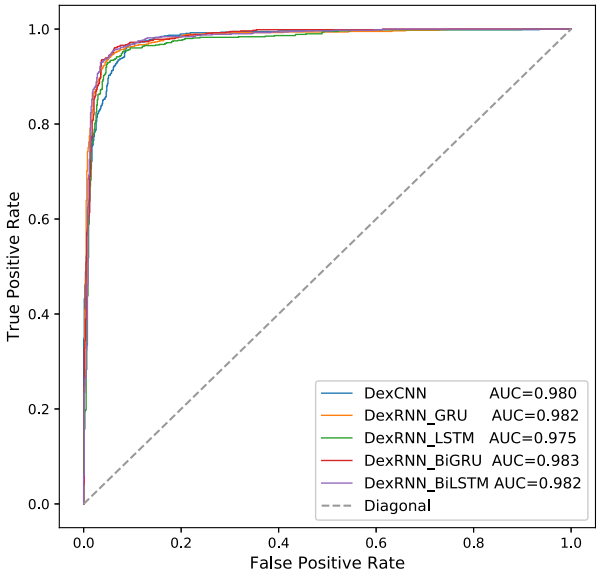


Fig. 9. Average ROC curve of 10 experiments with different methods.

Table 4
Performance comparison with other works .

Ref	PubTime	Features	Dataset	Acc	Precision	Recall	F1	M	L
3.2.1	N/A	end-to-end	16,000	95.8%	95.4%	96.2%	0.958	no	no
3.2.2	N/A	end-to-end	16,000	93.4%	89.7%	98.1%	0.937	no	no
[33]	2016	API&p	6,334	N/A	93.09%	94.5%	0.945	yes	no
[32]	2018	multi	10,770	97.4%	96.6%	98.3%	0.974	yes	no
[34]	2018	multi	110,440	97.74%	N/A	N/A	N/A	yes	no
[15]	2018	end-to-end	829,356	97.7%	N/A	N/A	N/A	no	yes
[35]	2018	end-to-end	7,000	95.4%	N/A	N/A	N/A	no	no
[44]	2017	system calls	14,231	93.1%	95.75%	N/A	0.866	yes	no
[43]	2016	multi	5,888	93.4%	N/A	N/A	N/A	yes	no

curacy, but still have limitations [15]. has a limit on the filesize of the input samples. Moreover, the Inception-v3 model they use has parameters of up to 25 millions, which is too large to run on normal IoT devices and can only be deployed on the GPU server. The lightweight malware detection system proposed in [35] only analyzes a small part of the original APK zip file (512–4096 Bytes). The small part usually does not contain the most important files such as classes.dex and AndroidManifest.xml. Therefore, their method is difficult to explain reasonably and remains to be further studied. In our experiments, we used one GPU to train the 5 different models on 12800 training samples. With only 2 hours of training, DexCNN can achieve validation accuracy of over 92%. The DexCRNN models require longer training time but can achieve higher accuracy. For example, the DexCRNN_BiGRU achieves validation accuracy of over 92% taking about 6 hours but can achieve validation accuracy of over 95% in about 14 hours. The models we proposed are also lightweight, with 0.53 million of parameters for DexCNN and 0.26 million for DexCRNN_BiGRU. We run our trained model on a computer with one Intel Xeon E5-2630 processor to detect malware. The time required for DexCNN and DexCRNN_BiGRU to detect a 10MB APK dex file is only about 0.1s and 0.2s respectively. This is because the proposed methods have simple a model structure and a small number of parameters, so they are more suitable for lightweight IoT devices. Apart from this, the commonly used operating system platforms (such as Linux and Windows) run executable file formats (PE and ELF) similar to the dex file of APK. Therefore, our detection methods can be directly applied to PE files or ELF files without the intervention of security experts.

5. Conclusions

Due to the openness and universality of the Android platform, the malware detection of Android IoT devices has become a hot topic in recent years. Existing methods require security experts to spend a lot of time and effort building features for training detection models, so these methods are difficult to apply in practice. Therefore, this paper proposes two end-to-end malware detection methods without manual feature engineering. We first use the resampling method to preprocess the classes.dex file in Android APK. Then we use two deep learning models, DexCNN and DexCRNN, to train the preprocessed sequences. We evaluated the two methods on a dataset containing 8000 benign APKs and 8000 malicious APKs. The experiment proved that DexCNN can achieve 93.4% detection accuracy, and the DexCRNN can achieve 95.8% detection accuracy. Compared with the existing Android malware detection methods, our proposed methods are not limited by input filesize, no manual feature engineering, low resource consumption, so they are more suitable for Android IoT devices. Since security expert intervention is not necessary, the proposed methods can also be easily extended to other similar malware detection tasks, which will also be a future work of our research on cross-platform malware detection. Another future work is to add other files (e.g., AndroidManifest.xml) in the APK as model input and repeat the proposed methods on a larger production-scale dataset. Beside this, the patterns learned by the proposed detectors are still difficult to understand, how to make these patterns correspond to benign or malicious areas in dex files still requires further research in the future.

Declaration of Competing Interest

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

Acknowledgments

This work was supported in part by the Natural Science Foundation of China under Grant 61771089 and Grant 61961040, and in part by Tianshan Youth Plan Project of Xinjiang Uyghur Autonomous Region of China under Grant 2018Q024.

References

[1] Symantec, The 2019 internet security threat report (ISTR), 2019. https://www.symantec.com/content/dam/symantec/docs/reports/.
[2] statista, Development of new android malware worldwide from june 2016 to may 2019 (in millions), 2019. https://www.statista.com/statistics/680705/global-android-malware-volume/.
[3] K. Tam, A. Feizollah, N.B. Anuar, R. Salleh, L. Cavallaro, The evolution of android malware and android analysis techniques, ACM Comput. Surv. 49 (4) (2017) 76:1–76:41, doi:10.1145/3017427.
[4] V. Rastogi, Y. Chen, X. Jiang, Catch me if you can: evaluating android anti–malware against transformation attacks, IEEE Trans. Inf. Forensics Secur. 9 (1) (2013) 99–108.
[5] J. Qiu, S. Nepal, W. Luo, L. Pan, Y. Tai, J. Zhang, Y. Xiang, Data-driven android malware intelligence: A survey, in: Machine Learning for Cyber Security - Second International Conference, ML4CS 2019, Xi'an, China, September 19–21, 2019, Proceedings, 2019, pp. 183–202, doi:10.1007/978-3-030-30619-9_14.
[6] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, J. Hoffmann, Mobile-sandbox: having a deeper look into android applications, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, ACM, 2013, pp. 1808–1815.
[7] Y. Feng, S. Anand, I. Dillig, A. Aiken, Apposcopy: semantics-based detection of android malware through static analysis, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16, - 22, 2014, 2014, pp. 576–587, doi:10.1145/2635868.2635869.
[8] V. Rastogi, Y. Chen, W. Enck, Appsplayground: automatic security analysis of smartphone applications, in: Third ACM Conference on Data and Application Security and Privacy, CODASPY'13, San Antonio, TX, USA, February 18–20, 2013, 2013, pp. 209–220, doi:10.1145/2435349.2435379.
[9] A. Reina, A. Fattori, L. Cavallaro, A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors, EuroSec, April (2013).
[10] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P.G. Bringas, G.Á. Marañón, PUMA: permission usage to detect malware in android, in: International Joint Conference CISIS'12-ICEUTE'12-SOCO'12 Special Sessions, Ostrava, Czech Republic, September 5th-7th, 2012, 2012, pp. 289–298, doi:10.1007/978-3-642-33018-6_30.
[11] L. Nataraj, S. Karthikeyan, G. Jacob, B.S. Manjunath, Malware images: visualization and automatic classification, in: 2011 International Symposium on Visualization for Cyber Security, VizSec '11, Pittsburgh, PA, USA, July 20, 2011, 2011, p. 4, doi:10.1145/2016904.2016908.
[12] C. Szegedy, S. Ioffe, V. Vanhoucke, A.A. Alemi, Inception-v4, inception-resnet and the impact of residual connections on learning, in: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San

Francisco, California, USA., 2017, pp. 4278–4284. http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14806.
[13] Y. Kim, Convolutional neural networks for sentence classification, in: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL, 2014, pp. 1746–1751. http://aclweb.org/anthology/D/D14/D14-1181.pdf.
[14] Y. Zhang, W. Chan, N. Jaitly, Very deep convolutional networks for end-to-end speech recognition, in: 2017 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2017, New Orleans, LA, USA, March 5–9, 2017, 2017, pp. 4845–4849, doi:10.1109/ICASSP.2017.7953077.
[15] T.H. Huang, H. Kao, R2-D2: color-inspired convolutional neural network (cnn)-based android malware detections, in: IEEE International Conference on Big Data, Big Data 2018, Seattle, WA, USA, December 10–13, 2018, 2018, pp. 2633–2642, doi:10.1109/BigData.2018.8622324.
[16] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, C.K. Nicholas, Malware detection by eating a whole EXE, in: The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018., 2018, pp. 268–276. https://aaai.org/ocs/index.php/WS/AAAIW18/paper/view/16422.
[17] W. Enck, M. Ongtang, P.D. McDaniel, On lightweight mobile phone application certification, in: Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9–13, 2009, 2009, pp. 235–245, doi:10.1145/1653662.1653691.
[18] Y. Aafer, W. Du, H. Yin, Droidapiminer: Mining api-level features for robust malware detection in android, in: Security and Privacy in Communication Networks - 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25–28, 2013, Revised Selected Papers, 2013, pp. 86–103, doi:10.1007/978-3-319-04283-1_6.
[19] E. Mariconti, L. Onwuzurike, P. Andriotis, E.D. Cristofaro, G.J. Ross, G. Stringhini, Mamadroid: Detecting android malware by building markov chains of behavioral models, 24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017, 2017. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/mamadroid-detecting-android-malware-building-markov-chains-behavioral-models/.
[20] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, DREBIN: effective and explainable detection of android malware in your pocket, 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014, 2014. https://www.ndss-symposium.org/ndss2014/drebin-effective-and-explainable-detection-android-malware-your-pocket.
[21] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y.L. Traon, D. Octeau, P.D. McDaniel, Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014, 2014, pp. 259–269, doi:10.1145/2594291.2594299.
[22] Y. Ye, T. Li, D.A. Adjeroh, S.S. Iyengar, A survey on malware detection using data mining techniques, ACM Comput. Surv. 50 (3) (2017) 41:1–41:40, doi:10.1145/3073559.
[23] W. Enck, P. Gilbert, B. Chun, L.P. Cox, J. Jung, P.D. McDaniel, A. Sheth, Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones, in: 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4–6, 2010, Vancouver, BC, Canada, Proceedings, 2010, pp. 393–407.
[24] L. Yan, H. Yin, Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis, in: Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012, 2012, pp. 569–584. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/yan.
[25] R. Mahmood, N. Mirzaei, S. Malek, Evodroid: segmented evolutionary testing of android apps, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16, - 22, 2014, 2014, pp. 599–609, doi:10.1145/2635868.2635896.
[26] T. Vidas, J. Tan, J. Nahata, C.L. Tan, N. Christin, P. Tague, A5: automated analysis of adversarial android applications, in: Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, SPSM@CCS 2014, Scottsdale, AZ, USA, November 03 - 07, 2014, 2014, pp. 39–50, doi:10.1145/2666620.2666630.
[27] K. Jamrozik, A. Zeller, Droidmate: a robust and extensible test generator for android, in: Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft '16, Austin, Texas, USA, May 14–22, 2016, 2016, pp. 293–294, doi:10.1145/2897073.2897716.
[28] B. Kang, S.Y. Yerima, K. McLaughlin, S. Sezer, N-opcode analysis for android malware classification and categorization, in: 2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security), London, United Kingdom, June 13–14, 2016, 2016, pp. 1–7, doi:10.1109/CyberSecPODS.2016.7502343.
[29] M.A. Atici, S. Sagiroglu, I.A. Dogru, Android malware analysis approach based on control flow graphs and machine learning algorithms, in: 2016 4th International Symposium on Digital Forensic and Security (ISDFS), IEEE, 2016, pp. 26–31.
[30] M. Zhang, Y. Duan, H. Yin, Z. Zhao, Semantics-aware android malware classification using weighted contextual api dependency graphs, in: Proceedings of the 2014 ACM SIGSAC conference on computer and communications security, ACM, 2014, pp. 1105–1116.
[31] S. Rasthofer, S. Arzt, M. Miltenberger, E. Bodden, Harvesting runtime values in android applications that feature anti-analysis techniques., NDSS, 2016.
[32] Y. Zhang, Y. Yang, X. Wang, A novel android malware detection approach based on convolutional neural network, in: Proceedings of the 2nd International Conference on Cryptography, Security and Privacy, ICCSP 2018, Guiyang, China, March 16–19, 2018, 2018, pp. 144–149, doi:10.1145/3199478.3199492.
[33] Z. Wang, J. Cai, S. Cheng, W. Li, Droiddeeplearner: Identifying android malware using deep learning, in: 2016 37th IEEE Sarnoff Symposium, Newark, NJ, USA, September 19–21, 2016, 2016, pp. 160–165, doi:10.1109/SARNOF.2016.7846747.
[34] K. Xu, Y. Li, R.H. Deng, K. Chen, Deeprefiner: Multi-layer android malware detection system applying deep neural networks, in: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24–26, 2018, 2018, pp. 473–487, doi:10.1109/EuroSP.2018.00040.
[35] C. Hasegawa, H. Iyatomi, One-dimensional convolutional neural networks for android malware detection, in: 2018 IEEE 14th International Colloquium on Signal Processing & Its Applications (CSPA), IEEE, 2018, pp. 99–102.
[36] J. Su, D.V. Vargas, S. Prasad, D. Sgandurra, Y. Feng, K. Sakurai, Lightweight classification of iot malware based on image recognition, in: 2018 IEEE 42nd Annual Computer Software and Applications Conference, COMPSAC 2018, Tokyo, Japan, 23–27 July 2018, Volume 2, 2018, pp. 664–669, doi:10.1109/COMPSAC.2018.10315.
[37] Q. Le, O. Boydell, B. Mac Namee, M. Scanlon, Deep learning at the shallow end: malware classification for non-domain experts, Digital Invest. 26 (2018) S118–S126.
[38] B. Chen, Z. Ren, C. Yu, I. Hussain, J. Liu, Adversarial examples for cnn-based malware detectors, IEEE Access 7 (2019) 54360–54371.
[39] J. Chung, C. Gulcehre, K. Cho, Y. Bengio, Empirical evaluation of gated recurrent neural networks on sequence modeling, (2014) arXiv:1412.3555.
[40] M. Schuster, K.K. Paliwal, Bidirectional recurrent neural networks, IEEE Trans. Signal Process. 45 (11) (1997) 2673–2681.
[41] Googleplay, play.google.com, 2019. Accessed: 2019-05-10. https://play.google.com/.
[42] VirusShare, Virusshare.com, 2019. Accessed: 2019-04-05. https://virusshare.com/.
[43] L. Xu, D. Zhang, N. Jayasena, J. Cavazos, Hadm: Hybrid analysis for detection of malware, in: Proceedings of SAI Intelligent Systems Conference, Springer, 2016, pp. 702–724.
[44] H. Liang, Y. Song, D. Xiao, An end-to-end model for android malware detection, in: 2017 IEEE International Conference on Intelligence and Security Informatics, ISI 2017, Beijing, China, July 22–24, 2017, 2017, pp. 140–142, doi:10.1109/ISI.2017.8004891.

Zhongru Ren received his B.S. degree in Computer Science and Technology from Zhengzhou University, Zhengzhou, China, in 2011. Currently he is a graduate student at the School of Computer Science and Technology, Dalian University of Technology, Dalian, China. His current research interests include Information Security and Deep Learning.

Haomin Wu received her B.S. degree in Korean from ShanDong University in 2011 and the M.S. degree in Pedagogy from Yonsei University, Seoul, Korea in 2013. Currently she is a teaching assistant in the Electronic Information Engineering Department of Wuhan Polytechnic, Wuhan, China. Her current research interests include Pedagogy and Articial Intelligence.

Qian Ning is now a Professor with the College of Electronics and Information Engineering, Sichuan University, Chengdu, 610065 China and also with the School of Physics and Electronics, Xinjiang Normal University, Urumqi, 830054 China. Her current research interests include Wireless Ad hoc network and pattern recognition.

Iftikhar Hussain received the B.S. degree in Computer Science from Islamia College University of Peshawar in 2013 and the M.S degree in Computer Science and Technology from Dalian University of Technology, Dalian, China in 2018. Currently he is working towards the Ph.D. degree at School of Computer Science and Technology at University of Science and Technology of China, Hefei, China. His research interest includes Information Security and Wireless Networks.

Bingcai Chen received his MS degree and Ph.D. degree in Information and Communication Engineering from Harbin Institute of Technology (HIT), Harbin, China, in 2003 and 2007, respectively. He has been a visiting scholar in University of British Columbia, Canada, in 2015. He is currently a Professor with the school of computer science and technology, Dalian University of Technology, and also with the School of Computer Science and technology, Xinjiang Normal University. His research interests include wireless ad hoc network, network and information security and computer vision. He received National Science Foundation Career Award of China in 2009. He is severing as a reviewer for project proposals to National science foundation of China, Ministry of Education of China. He is also serving as a reviewer for some refereed Journals including IEEE/ACM Transactions on Circuits and Systems for Video Technology, IEEE Transactions on Vehicular Technology, IEEE Wireless Communications Letters, IEEE Access, Journal of Electronics, Journal of Communication, etc. He won the best paper awards in IEEE VTC2017-Spring. Additionally, he served as a TPC member for many conferences.