Android Malware Family Classification Based on Deep Learning of Code Images

Yuxia Sun, Yanjia Chen, Yuchang Pan and Lingyu Wu

Abstract—**Android continues to dominate the mobile social devices, and Android applications have become the major target of hackers in social networks. Although millions of Android malware samples are found every year, they can be grouped into a limited number of malware families. To automatically and effectively classify Android malware into the corresponding malware families, a deep-learning based classification approach is proposed by utilizing the code-images converted from the malware's binary bytecodes. To overcome the training issue that only a very limited amount of malware samples are publicly labeled with families, the deep-learning classifier makes use of the low-level layers of a pre-trained convolutional neural network. The empirical studies show that the proposed approach excels the existing code-image based technique in implementation simplicity as well as in classification metrics such as F-measure values, false positive rates, and false negative rates. Furthermore, the implemented classifier can identify malware families of different sizes, including small families.**

Index Terms—**Android malware, code image, deep learning, malware family classification.**

I. Introduction

As mobile social devices becomes increasingly pervasive, it is important to improve the security of applications running on the devices. Due to the overwhelming user amount, Android applications have been the biggest target of malware authors in various social networks. For example, 360 Internet Security Center found 7.57 million new Android malware samples in 2017 [1]. Many researchers focus on distinguishing malicious Android applications from benign ones. For instance, Sachdeva et al. [2] classify Android applications into safe, suspicious and highly suspicious ones. However, in recent years, although the yearly added Android variants are 0.6 times more, the yearly added malware families are 3 times less [3]. This is because the majority of current Android malware are variants that are created by reusing malicious modules or by employing malware generation tools. In view of the ever increasing amount of new malware and the limited number of malware families, it is crucial to automatically classify numerous malware into their corresponding families.

The approaches to characterize and analyze Android malware include static approaches and dynamic ones. Dynamic approaches are also called behavior-based methods [4], which execute the target program, trace and analyze the program behaviors. For examples, Ahmad et al. [5] trace the system call sequences of Android programs and utilize machine learning algorithms to classify them; Madihah et al. [6] trace and analyze system calls and permissions of Android applications to detect camera-exploitation malware. Static approaches and dynamic approaches for malware family classification are complementary. We focus on the static approaches in this paper. Traditional static methods utilize such code features of Android malware as opcode (operational code), API calls, control flow graphs and so on.

Code images were firstly proposed as static features of x86 malware by Nataraj et al. in 2011 [7]. They classify malicious x86 executables by visualizing malware binaries as images and then group the images into families. The approach is based on the following observation: for many malware families, the images belonging to the same family appear very similar in layout and texture. In 2015, Microsoft hosted a competition in Kaggle for x86 malware family classification [8]. The competition winner's solution heavily relied on the malware's code-image features, especially the first 800 pixels' intensities of the images transformed from disassembled codes [9]. The winner utilized a random forest algorithm to classify the code-images. To classify Android malware into families, Y. Yang et al. [10] employ code-images of Android malware. They extract texture features of grayscale images converted from binary bytecodes, and then apply a random forest algorithm to classify the images. However, they cannot detect Android malware families of small sizes (less than 40), and moreover, the detection effectiveness needs to be ameliorated.

Malware family classification is multi-class classification. In recent years, deep Convolutional Neural Networks (CNNs) have yielded significant gains in various multi-class classification tasks of traditional images [11][12][13][14]. The image features used by the CNNs, although not explicitly expressed, are more generic in the early layers and more dataset-specific in the later layers of the CNNs [15]. CNN has been applied to classify various non-image data in applications other than image processing by transforming such data into non-traditional images [16][17][18]. Since malware's code images and traditional images have common generic features such as edges and curves, deep CNNs have potentials to classify the code-images into multi-classes (i.e., malware families). Compared to the traditional machine learning techniques that require manual specification and explicit extracting of image features, deep-learning based family classification techniques automatically learn discriminative features from the malware images, and thus reduce the implementation cost and computation cost.

However, so far, few works have been done for deep-learning based family classification with malware code-images. For x86 malware, Llaurad [19] tries to use a CNN

Manuscript received February 4th, 2019; revised May 30th, 2019. This work was supported by the National Natural Science Foundation (#61402197) of China, Guangdong Province Science and Technology Plan Project (#2017A040405030) in China, Tianhe District Science and Technology Plan Project (#201702YH108) in Guangzhou City of China.

Yuxia Sun is with the Department of Computer Science, Jinan University, Guangzhou, 510000, China e-mail: tyxsun@email.jnu.edu.cn.

Yanjia Chen, Yuchang Pan and Lingyu Wu are with the Department of Computer Science, Jinan University, Guangzhou, China.

for the malware family classification with the code images, but he conducts no controlled experiments to compare the classification results between using machine-learning and deep-learning infrastructures. For Android malware, no deep-learning techniques have been employed to classify malware code-images into families.

To improve the classification effectiveness of Android malware families based on code-images, and to detect Android malware families of varying sizes, we propose a deep-learning based approach in this paper. We construct the deep-learning classifier by reusing the feature-extracting layers of a convolutional neural network that has been successfully trained for a traditional image classification task on a large dataset [20]. The reasons for us to utilize a pre-trained deep-learning network rather than to create a network from scratch are as follows: The publicly available datasets for Android malware family classification are very limited in scale and variety. For example, in the Drebin dataset which is possibly the largest dataset for this purpose, there are only 4 malware families containing over 600 malware samples [21]. However, a large training dataset in size is essential to the success of a deep-learning classifier. Since the dataset currently available for Android malware family classification is small in size, it is not a good idea to train our deep-learning classifier from scratch with such a dataset. Instead, we can utilize a pre-trained network to create our classifier, and then train the top network-layer which contains more dataset-specific features.

We implement the Android malware family classifier using the Google's TensorFlow framework [22], and conduct empirical studies with controlled experiments. As the experimental results demonstrate, our deep-leaning based classifier outperforms the existing classifier with respect to the simplicity of usage and the classification effectiveness. Moreover, our classifier works well even when malicious families only contain few samples for training.

The contributions of this paper are summarized as follows:

1) We propose a family classification approach for Android malware based on deep-learning of malware code-images. The approach extracts malware features automatically, rather than selects features manually.
2) We implement our approach in a tool, and empirically demonstrate its simplicity of usage, detection effectiveness, and applicability to malware families of varying sizes.

II. Related Work

A. Malware family classification using code-images

1) For x86 programs: To classify x86 malware into the corresponding families, Nataraj et al. [7] initiate a code-image based technique. It extracts texture features of grayscale code-images with the GIST (Generalized Search Trees) [23] method and classifies the images by employing the KNN (k-nearest neighbors) algorithm. Dey et al. [24] propose an improved KNN-based approach by adding an entropy filter while extracting the GIST features. Han et al. [25] convert malicious PE (Portable Executable) files into texture images, and then use a clustering technique to classify and tag malware families. The image texture features are extracted with the GLCM (Gray Level Co-occurrence Matrix) method. In the Microsoft contest of malware family classification in 2015, the winner extracted the following features from malware: code-images, segment line count of the disassembled code, and n-gram opcodes. They utilized a random forest algorithm for classification [9]. Gibert [19] also utilizes the code-images of disassembled codes of x86 executables, and employs a CNN for malware family classification. He et al. [26] draw disassembly codes from x86 malicious executables too. They extract code-image texture features using the GIST method, and further obtain and optimize the local features of the code-images using the SIFT (Scale Invariant Feature Transform) method [27] and the BoW (Bag of Words) model, respectively. Based on the above features, they utilize a random forest algorithm to classify the code-images into families.

In conclusion, since Nataraj et al. proposed the code-images based technique for family classification of x86 malware, researchers have improved the technique by using more features like SIFT, or by employing other machine-learning algorithms for classification, e.g., clustering, random forests and SVM (Support Vector Machine) algorithms.

Gibert [19] utilizes a deep-learning infrastructure to implement the family classification of x86 malware, and shows that its classification effectiveness is worse than that of the competition winner's approach (which employs a machine-learning infrastructure). However, Gibert's work has the following limitations: (1) His deep-learning infrastructures is primitive and needs be further ameliorated; (2) His experiments are not controlled ones. For example, the deep-learning based implementation only uses the code-images features of malware, but for comparison, the machine-learning based implementation uses not only code-images but also other features such as opcode n-grams. In conclusion, Gibert's work cannot demonstrate that deep-learning approaches are less suitable for malware family classification than machine-learning ones. Thus, in this paper, we will conduct controlled experiments to explore whether or not deep-learning based approaches are promising for malware family classification, and focus on Android malware.

To classify the malware on non-Android platforms, Cui et al. [28] convert malicious code into grayscale images, and Ni et al. [29] transform the SimHash value of the opcode sequence of malware to grayscale images, and both of them utilize CNN to construct the classifiers. Different from them, this paper aims to classify malware on Android platforms.

There are some works on visual analysis of binary codes on x86 platforms. For example, Han et al. [30] transform disassembly information of binary x86 programs into RGB colored pixels on image matrices, and then uses a selective area matching method to calculate the similarities between image matrices. The similarities can be further utilized for malware family classification.

2) For Android programs: Few works have been done to utilize code-images for family classification of malicious Android programs. Yang et al. [10] convert binary bytecodes of Android malware into grayscale images, extract image texture features using the GIST method, and then apply a random forest algorithm to classify the images. They conduct experiments on a subset of Drebin dataset and show that the approach works. However, there is still a considerable margin of improvement on the classification results.

B. Android family classification based on machine-learning

Most works of malware family classification for Android programs are based on traditional machine-learning techniques. As mentioned in previous subsection, Yang et al. [10] utilize the code-image features of Android malware based on a random forest algorithm. Some recent machine-learning based works utilize other features of Android malware for family classification. For examples, DroidLegacy [31] uses API calls and agglomerative clustering algorithms to extract familial signatures of malicious code module that are produced by piggybacking. Dendroid [32] extracts static features from code chunks and uses a text-mining based method to find similarities in malware of the same family. Droidscribe [33] extracts malware features at different levels, including system calls, decoded Binder communication and abstract behavioral patterns, and then employs an SVM algorithm for multi-class classification. Massarelli et al. [34] extract features of resource consumption metrics of malware through detrended fluctuation analysis, and then employ an SVM to classify malware into families. Chakraborty et al. [35] extract both static features from the manifest files and dynamic features from operation logs, and then utilize traditional classification and clustering algorithms for family classification. Garcia et al. [36] utilize a CART decision tree to classify Android malwares based on the features of API usage, reflection, and native code. Recently, Zhang et al. [37] use an online passive-aggressive (PA) classifier for binary classification of malware based on fingerprint n-gram features.

Although traditional machine-learning techniques are effective in malware classification for Android malware, they have the following limitations: malware features must be specified manually by experts with professional background knowledge, and the classification effectiveness need be further improved. In this paper, we propose to automatically learn discriminative features of Android malware for family classification using a deep-learning based approach, and show its improvement on classification effects with controlled experiments.

Moreover, unlike many existing static methods, our classification method does not require any disassembly analysis of Android malware, and thus more resilient to obfuscation techniques.

III. METHODOLOGY

A. Method overview

To classify unknown Android malware into malware families, we employ a deep-learning based classification approach with the code images of the malware. We construct the deep-learning classifier by reusing the feature-extracting layers of a CNN which has been successfully trained for other image classification tasks on a large dataset. Our malware classifier makes use of code images transformed from binary bytecodes of Android malware, and employs a pre-trained CNN to learn discriminative patterns from the images.

In the remaining parts of Section III, we will expatiate how to determine our deep-learning architecture and how to construct our malware family classifier.

TABLE I
Comparison of Two Network Architectures

Network	#Layers	#Parameters	Top-5 Error
VGGNet	19	140 M	7.3%
GoogLeNet Inception-v3	47	23 M	3.46%

B. Decision on deep-learning architecture

Our code-image based malware classification approach aims to utilize a deep-learning network architecture to classify malware families. Because very deep convolutional networks have been successfully applied to various computer-vision related tasks since 2014 [11][12][13][14], our approach will utilize such networks.

VGGNet [38] and GoogLeNet Inception [39] are two most popular convolutional network architectures. VGGNet has the advantage of simple architecture, while the Inception architecture performs well with low costs of computation and memory. Inception-V3 is a popular version of the Inception architecture and its pre-trained model is publicly available. Inception-V3 utilizes factorized convolutions and aggressive regularization to efficiently scale up convolution networks [40].

We compare the architectures of VGGNet and Inception-V3 in respect of network depth (i.e. layer count), parameter count (measured in M, Million) and top-5 error rate on the ImageNet ILSVRC-2012 dataset [40] and demonstrate the results in Table I. As Table I shows, compared to VGGNet, Inception-V3 has 2.5x more network depth but 6x less parameters and 2x reduction of top-5 error. It means that, compared to VGGNet, Inception-V3 network might represent more complex functions with smaller empirical margin error and much less computational cost [41] [42].

Based on the above analysis, we choose GoogLeNet Inception-V3 as the deep-learning network architecture of our malware classifier.

C. Construction of our malware classifier

We construct the malware family classifier in the following four steps, as shown in Figure 1.

1) Generating code images of malware: For each Android malware sample, we firstly decompress its application package to get the executable file in Dex (Dalvik executable) format. A Dex file is in bytecode format that contains compiled code written for Android and can be interpreted by the Dalvik virtual machine. Next, we convert the binary code of each Dex file into its equivalent hexadecimal form, where every two hexadecimal digits, ranging from 0X00 to 0XFF (namely from 0 to 255), denote one byte of the malware code. Finally, we utilize an image-pixel matrix to represent a Dex file by mapping each hexadecimal byte into one grayscale value of an image-pixel. As a result, the grayscale code-images of malware samples are derived from the above image-pixel matrixes.

2) Generating feature vectors of code images: Firstly, we download the instance of Google Inception V3 architecture which was trained for the ImageNet project [43]. Then, we remove the top convolutional layer, namely the fully-connected layer, of the neural network. The remaining

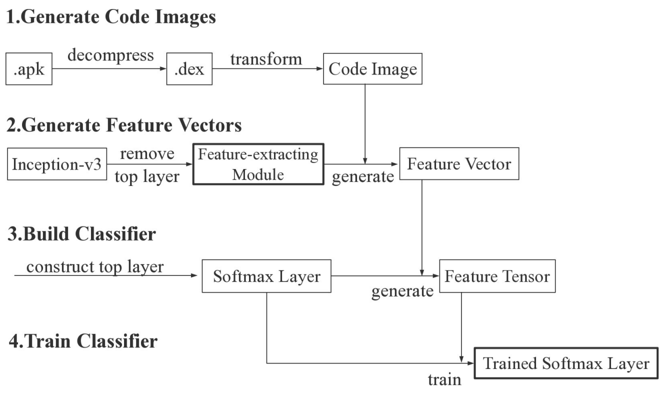


Fig. 1. Construction process of our malware family classifier

convolutional layers of the neural network were used for extracting image features in the ImageNet project. We keep all the parameters in the remaining layers unchanged and get the feature-extracting module. Next, we run the feature-extracting module on the code images of the malware, as mentioned in Subsection B, and consequently obtain the 2048-dimentional feature vectors of the images. Finally, the feature vectors of all the aforementioned code images are stored in a feature-vector file.

3) Building the malware classifier: Firstly, we construct a top convolutional layer, namely a fully-connected layer, as follows: The Softmax layer of the fully-connected layer reads the feature vectors of code images from the feature-vector file, and then converts them into the tensor data format that can be processed directly by the Softmax layer. The Softmax layer has N output nodes, where N denotes the total number of the malware families. Next, we add the top convolutional layer to the lower feature-extracting layers obtained previously, and thus get our convolutional neural network for malware family classification.

4) Training the malware classifier: When training the malware classifier, we only train the top layer of the deep-learning neural network without altering the parameters of all the remaining layers. In other words, we only train the Softmax layer but keep the feature-extracting module unchanged. A training dataset is a group of code images labeled with malware families. During the training of the malware classifier, we primarily adjust the following three parameters: the optimizer, the learning rate and the count of iterations of the deep-learning neural network.

As a result of the above steps, we obtain our familial Android malware classifier that consists of the trained Softwax layer and the feature-extracting module, as the two nodes with thick border-lines show in Figure 1.

IV. EMPIRICAL STUDIES

In this section, we conduct experiments to evaluate our malware family classification technique and compare it to the existing technique proposed by Yang et al [10]. The experiments aim to answer the following research questions:

Q1: How to determine the classifier's parameters in our technique? Is the parameter-tuning simpler than that in the existing technique?

Q2: Is the classification effectiveness of malware families using our technique better than that using the existing technique?

Q3: How about the classification results of our technique for malware families of varying sizes?

A. Dataset

Drebin [21] is the largest public dataset of Android malware with labeled families. Yang et al [10] propose a machine-learning based technique for Android-malware family-classification by utilizing code-images. They conduct 10-fold cross-validation on a Drebin subset, namely Subset1 shown in Table II. The Subset1 contains 3962 Android malicious Apps of 14 malware families.

However, the existing technique [10] cannot work with small families. The Subset1 does not include small families whose sizes (namely numbers of family members) are less than 40. To evaluate the classification results of our technique on malware families of varying sizes, including small families, we extend the Dataset1 with the Dataset2, as shown in Table II. The Subset2 includes 16 malware families, and 10 of them are small families whose sizes are less than 40. In summary, the entire dataset used in our experiments consists of the existing Subset1 and the new Subset2. As Table II lists, our dataset includes 30 malware families and 4892 malware. Column 2 and 4 list the malware family names, while column 3 and 5 indicate the number of malware in each family.

To answer the above research questions RQ1 and RQ2, we compare our technique to the existing technique [10] on the same dataset (namely Subset1) that are used by Yang et al. To answer the research question RQ3, we evaluate our technique on the entire dataset (including Subset1 and Subset2) which involves families of varying sizes, including small families.

TABLE II
DATASET OF MALWARE FAMILIES

Dataset	Malware family	#Malware	Malware family	#Malware
Subset1	BaseBridge	330	Imlog	43
	DroidDream	81	Kmin	147
	DroidKungFu	667	MobileTx	69
	FakeDoc	132	Opfake	613
	FakeInstaller	925	Plankton	625
	FakeRun	61	SendPay	59
	Iconosys	152	Gappusin	58
Subset2	GinMaster	339	Geinimi	92
	Adrd	91	ExploitLinux Lotoor	70
	Glodream	69	SMSreg	41
	Yzhc	37	Jifake	29
	Hamob	28	Boxer	27
	Penetho	19	Fakelogo	19
	Xsider	18	Fatakr	17
	Dougalek	17	FakePlayer	17

B. Experiment setup

We implemented our classifier tool in Python language by utilizing the open-source libraries of TensorFlow 0.12.0-rc0 [44] and PrettyTensor 0.7.1 [45]. We performed experiments on a PC with Window 10 platform, Intel Core i7-7500U CPU and 8G RAM.

To perform 10-fold cross-validation [46], we partition one dataset into 10 roughly equal-sized subsets at random. The cross-validation process is repeated 10 times, namely 10 rounds. In each round, one subset is chosen as the testing data (namely validation data) and the remaining 9 subsets are used as the training data. Each subset can be used as the validation data in only one round. At last, the average results generated from the 10 rounds are used as the final experiment results.

The cross-validation process is repeated 10 times (namely 10 rounds) with each subset used exactly once as the validation data. In other words, in one round, one subset is chosen as the testing data and the remaining 9 subsets are used as the training data. At last, the average results generated from the 10 rounds are used as the final experiment results.

To evaluate the malware classification effectiveness, for each malware family, we collect the following classification data: true positive (TP), true negative (TN), false negative (FN) and false positive (FP). To determine the parameters of our classifier, we compute the classification Accuracy of our technique with the following formula:

$$Accuracy = \frac{TP+TN}{FP+TP+FN+TN} \quad (1)$$

To compare the classification results of our technique with those of the existing one, we employ the following evaluation metrics that were used by the existing technique: F-measure, false positive rate (FPR) and false negative rate (FNR). F-measure is the harmonic mean of Precision and Recall, where the Precision is computed as TP/(TP+FP); the Recall, namely the detection rate of malware, is computed as TP/(TP+FN). In summary, the three metrics are calculated as follows:

$$F-Measure = \frac{2*Precision*Recall}{Precision+Recall} \quad (2)$$

$$FPR = \frac{FP}{FP+TN} \quad (3)$$

$$FNR = \frac{FN}{TP+FN} \quad (4)$$

C. Study1: Parameter-tuning

In this section, we will show how to determine the parameters of our malware classifier. We will also demonstrate our technique's simplicity on parameter-tuning, compared to that of the existing technique.

1) Optimizer: The convergence speed of a neural network may heavily rely on its optimizer. The optimizer determines the update rules of a neural network, which accelerate the process of finding a minimum loss function.

In order to achieve an optimized convergence speed of our neural network, we compare the acceleration effects of the following four commonly-used optimizers in the Tensorflow framework: AdadeltaOptimizer, AdagradOptimizer GradientDescentOptimizer, and AdamOptimizer. We train the malware classifier with the four optimizers on a same training dataset, respectively, and collect the classification accuracies along with the numbers of training iterations. Figure 2 plots the classification accuracies (in percentage) using the above four optimizers, respectively. Four types of scatterplots are drawn with different point types. For example, for the AdamOptimizer, the scatterplot consists of hollow circle points.

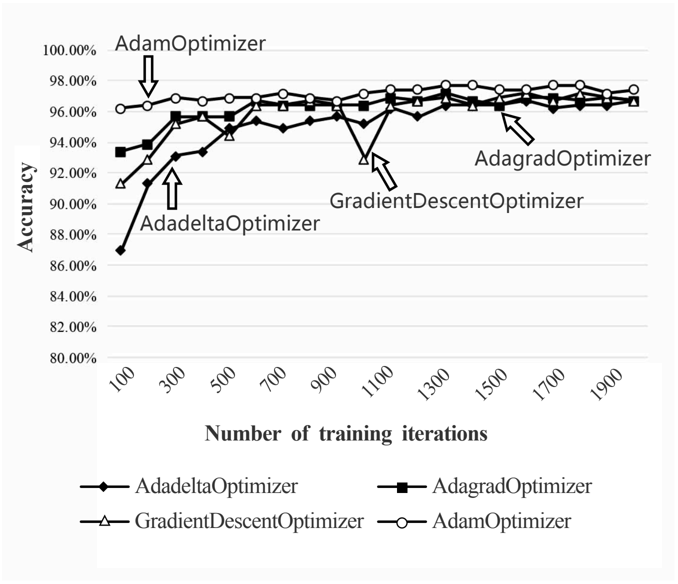


Fig. 2. Impacts of optimizer on convergence speed

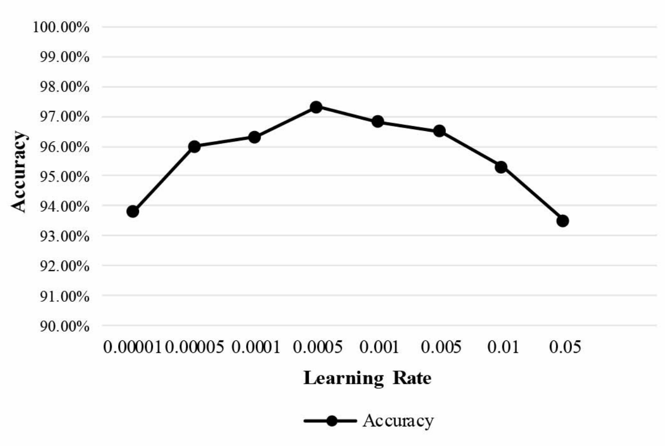


Fig. 3. Impacts of learning-rate on classification accuracy

As Figure 2 shows, the classifier using the AdamOptimizer achieves the classification accuracy of 96.

The above results might be explained as follows: GradientDescentOptimizer employs a gradient descent algorithm that needs to process all (but not parts of) the training samples, thus resulting in slower training [47]. AdagradOptimizer, AdadeltaOptimizer and AdamOptimizer use similar algorithms which do well in similar circumstances. But due to a bias-correction strategy, when gradients become sparse, the AdamOptimizer outperforms the other two optimizers towards the end of optimization [48].

In conclusion, based on the above empirical results, we set the neural network optimizer of our final malware classifier to AdamOptimizer.

2) Learning rate: During the training of a deep-learning neural network, the learning rate is an important parameter that controls the magnitude of the updates to the final layer. If the learning rate is too low, the convergence will be very slow; and if the learning rate is too high, the training precision might be low.

We tune the learning rate of a deep-learning neural network that has been pre-trained and able to be convergent fast on the original image dataset, namely ImageNet [40]. To avoid distorting the network's weights too quickly or too much, we keep both our learning rate and learning rate decay very small [49]. In this way, the tuned network might also be fast convergent on our dataset. We tune the learning rate from a very small value, gradually to a big one. As Figure 3 shows, we use six values of the learning rate, ranging from a very small value of 0.00001 to a big one of 0.05, and experimentally observe the corresponding classification accuracy when the neural network converges. As Figure 3 demonstrates, when the learning rate is 0.0005, the neural network achieves the best convergence accuracy.

Next, we empirically evaluate the convergence speed of our malware classifier on our testing data with the learning rate of 0.0005. We run the 10-fold cross-validation on different training data subsets for 10 rounds. Figure 4 shows the running results of the former 5 and the latter 5 rounds, respectively, with two sub-figures for clarity. As Figure 4 illustrates, in all the 10 rounds of runs, the malware classifier converges to the best accuracies (around 96%), and the count of optimization iterations is between 800 and 1400. Thus, with the above learning rate, our malware classifier can converge fast on the testing datasets.

In conclusion, based on the above empirical results, we set the neural network learning rate of our final malware classifier to 0.0005.

3) Comparison of two techniques' parameters: To train our malware classifier, only 3 hyper-parameters, namely the optimizer, the learning rate and the number of iterations, need to be determined by the trainer.

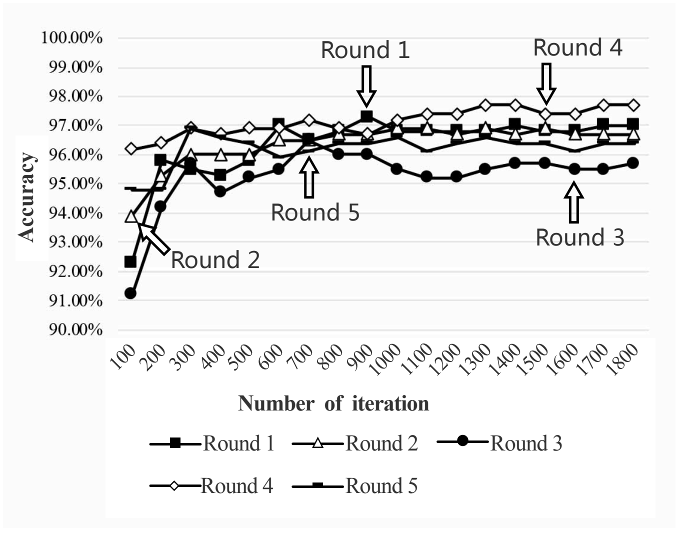
The existing technique employs a GIST algorithm to extract the features of malware code images and utilizes a random forest algorithm to train its malware classifier [10]. Thus, during the training of the classifier, the existing technique needs to tune at least 8 parameters. The tuned parameters include those of the random forest algorithm as follows: the number and the maximum depth of the decision trees (i.e., n_estimators and max_depth), the maximum number of features used by a single decision tree (i.e., max_features), the minimum number of samples for splitting inner nodes (i.e., min_samples_split), and the minimum number of samples in leaf nodes (i.e., min_samples_leaf). The parameters to be determined also include those of the GIST algorithm as follows: the number of filters (i.e., prefilter), the count of orientations per scale (i.e., orientationsPerScal), and the number of image blocks (i.e., numberBlocks).

In conclusion, it is much simpler for users to determine the classifier parameters by using our classification technique than using the existing one.

D. Study2: Comparison of two techniques' effectiveness

In this subsection, to compare to the classification results reported by the existing technique in [10], we will perform 10-fold cross-validation of malware classification on the same dataset (namely Dataset1) and with the same metrics, by employing our technique.

We run the cross-validation of our technique for 10 rounds. In each round, we output a 14*14 confusion matrix which is the most intuitive method to evaluate the classification



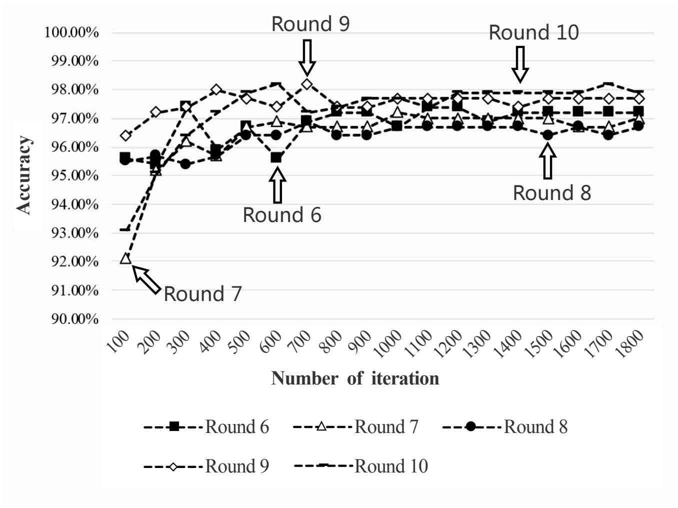


Fig. 4. Convergence speed on testing dataset with the learning rate of 0.005

model. Each column represents a predicted family name, and each row represents the actual category. Table III shows a confusion matrix for one round of experiments. As shown in the first line, it is the classification result of the malware family BaseBridge. The first number 29 is the TP value representing the number of samples correctly classified to BaseBridge. The values of the other columns are all 0 except the third column filled 1. It displays that one sample actually belongs to BaseBridge is classified to the wrong family DroidKungFu. Thus the FN value is 1. Overall, in this round of experiments, there are 382 samples correctly classified in the test set, and the total number of samples in the test set was 391. The accuracy of this round reaches 97.7%.

Based on the confusion matrix, we firstly collect the classification results of TP, TN, FP and FN of each malware family, and then compute the three metrics of F-measure, FPR and FNR according to the formulas (2)-(4), where F-measure is computed based on Precision and Recall, as depicted in Subsection B. Next, we compute the mean metric values of the 10 rounds for each malware family, as Table IV shows. In Table IV, the first column indicates the fourteen malware families, the next two columns show the mean precisions and the mean recalls of the classification for each malware family. The next three columns demonstrate

TABLE III
CONFUSION MATRIX FOR ONE ROUND OF EXPERIMENTS

	BaseBridge	DroidDream	DroidKungFu	FakeDoc	FakeInstaller	FakeRun	Iconosys	Imlog	Kmin	MobileTx	Opfake	Plankton	SendPay	Gappusin
BaseBridge	29	0	1	0	0	0	0	0	0	0	0	0	0	0
DroidDream	0	7	0	0	0	0	0	0	1	0	0	0	0	0
DroidKungFu	2	0	65	0	1	0	0	0	0	0	0	0	0	0
FakeDoc	0	0	0	11	0	0	0	0	1	0	0	0	0	0
FakeInstaller	0	0	0	0	92	0	0	0	0	0	0	0	0	0
FakeRun	0	0	0	0	0	5	0	0	0	0	0	0	0	0
Iconosys	0	0	0	0	0	0	15	0	0	0	0	0	0	0
Imlog	0	0	0	0	0	0	0	4	1	0	0	0	0	0
Kmin	0	0	0	0	0	0	0	0	15	0	0	0	0	0
MobileTx	0	0	0	0	0	0	0	0	0	6	0	0	0	0
Opfake	0	0	0	0	0	0	0	0	0	0	62	0	0	0
Plankton	0	0	0	0	1	0	0	0	0	0	0	62	0	0
SendPay	0	0	0	0	0	0	0	0	0	0	0	0	5	0
Gappusin	0	0	0	0	0	0	0	0	1	0	0	0	0	4

the mean F-measure, the mean FPR and the mean FNR, respectively, for each malware family. The last column shows the AUC value of the classification for each family. Finally, we compute the average F-measure, the average FPR, the average FNR, and the average AUC for all the fourteen malware-families and list the average values at the last row in the table. As comparison, Table V shows the classification results of the existing technique. The columns and rows in Table V have the same meaning as those in Table IV.

To visually compare the malware classification results of our technique and the existing one, we draw scatterplots in Figure 5 to show the results of the two techniques. The thick solid lines that connect solid diamond scatter-points denote the results of our technique, while the thin solid lines that connect hollow round scatter-points represent the results of the existing technique. Figure 5 consists of three sub-figures that show the results measured in the three metrics of F-measure, FPR and FNR, respectively. As comparison, Figure 5 also demonstrates the average results of our technique and the existing one with thick and thin dot lines, respectively.

As Table IV and Figure 5 reveal, our malware classification technique is effective: The average F-measure value is 95.2%, indicating that our technique can classify malware with a high harmonic mean of the precision and the recall rate; The average FNR value is 4.8%, showing that our technique fails to report malware family in a low rate; The average FPR value is 0.2%, implying that our technique classifies malware wrongly in a very low rate. The AUC of each families is close to 1 and the average AUC achieves 99.9%, indicating that our technique has strong generalization ability.

In average, our technique has higher F-measure value, lower false negative rate, lower false positive rate, and higher AUC value than the existing technique, as the last rows in Table IV and Table V demonstrate. On F-measure, our technique obviously excels the existing one on half of the malware families (namely DroidDream, DroidKungFu, Iconosys, Imlog, Opfake, SendPay and Gappusin), and has comparable results on the other half. On FNR, our technique obviously outperforms the existing one on 12 malware families and has comparable results on 2 malware families (namely Kmin and MobileTx). For FPR, our technique obviously exceeds the existing one on 5 malware families (namely DroidKungFu, FakeInstaller, Iconosys, Opfake and Plankton) and has comparable results on all the other malware families. For the AUC indicator, our technique surpasses the existing one on 8 malware families (namely BaseBridge, DroidDream, DroidKungFu, FakeDoc, Imlog, Plankton, SendPay and Gappusin) and has comparable AUC values on all the other families. In conclusion, our malware classification technique can classify unknown Android malware into malware families effectively, and excels the existing technique, with respect to all the three metrics of F-measure, false negative rate and false positive rate. In addition, our classification technique has better generalization ability than the existing technique, in view of the AUC value.

TABLE IV
CLASSIFICATION RESULTS USING OUR TECHNIQUE

Malware family	Precision	Recall	F-Measure	FNR	FPR	AUC
BaseBridge	0.952	0.882	0.916	0.118	0.003	0.999
DroidDream	0.908	0.899	0.903	0.101	0.002	0.999
DroidKungFu	0.95	0.967	0.958	0.033	0.01	0.998
FakeDoc	0.967	0.969	0.968	0.031	0.001	0.999
FakeInstaller	0.985	0.989	0.987	0.011	0.004	0.999
FakeRun	0.923	0.986	0.953	0.014	0.001	1
Iconosys	0.994	0.983	0.988	0.017	0.0002	0.999
Imlog	1	0.897	0.946	0.103	0.0002	1
Kmin	0.952	0.979	0.965	0.021	0.001	1
MobileTx	0.964	1	0.982	0	0.0002	1
Opfake	0.997	0.998	0.997	0.002	0.0006	0.999
Plankton	0.983	0.987	0.985	0.013	0.002	0.999
SendPay	1	0.963	0.981	0.037	0.0003	1
Gappusin	0.786	0.821	0.803	0.179	0.001	1
(Average)	0.954	0.951	0.952	0.048	0.002	0.999

TABLE V
CLASSIFICATION RESULTS USING THE OLD TECHNIQUE

Malware family	Precision	Recall	F-Measure	FNR	FPR	AUC
BaseBridge	0.965	0.839	0.898	0.161	0.002	0.984
DroidDream	0.882	0.827	0.854	0.173	0.002	0.983
DroidKungFu	0.733	0.934	0.821	0.066	0.057	0.989
FakeDoc	0.977	0.962	0.969	0.038	0.001	0.995
FakeInstaller	0.956	0.969	0.962	0.031	0.011	0.999
FakeRun	1	0.951	0.975	0.04	0	1
Iconosys	0.885	0.961	0.921	0.039	0.004	0.999
Imlog	1	0.837	0.911	0.163	0	0.984
Kmin	0.973	0.98	0.976	0.02	0.001	1
MobileTx	0.986	1	0.993	0	0	1
Opfake	0.951	0.977	0.964	0.023	0.008	0.999
Plankton	0.965	0.974	0.97	0.026	0.005	0.998
SendPay	0.947	0.915	0.931	0.085	0.001	0.989
Gappusin	1	0.534	0.697	0.466	0	0.954
(Average)	0.944	0.904	0.917	0.096	0.006	0.991

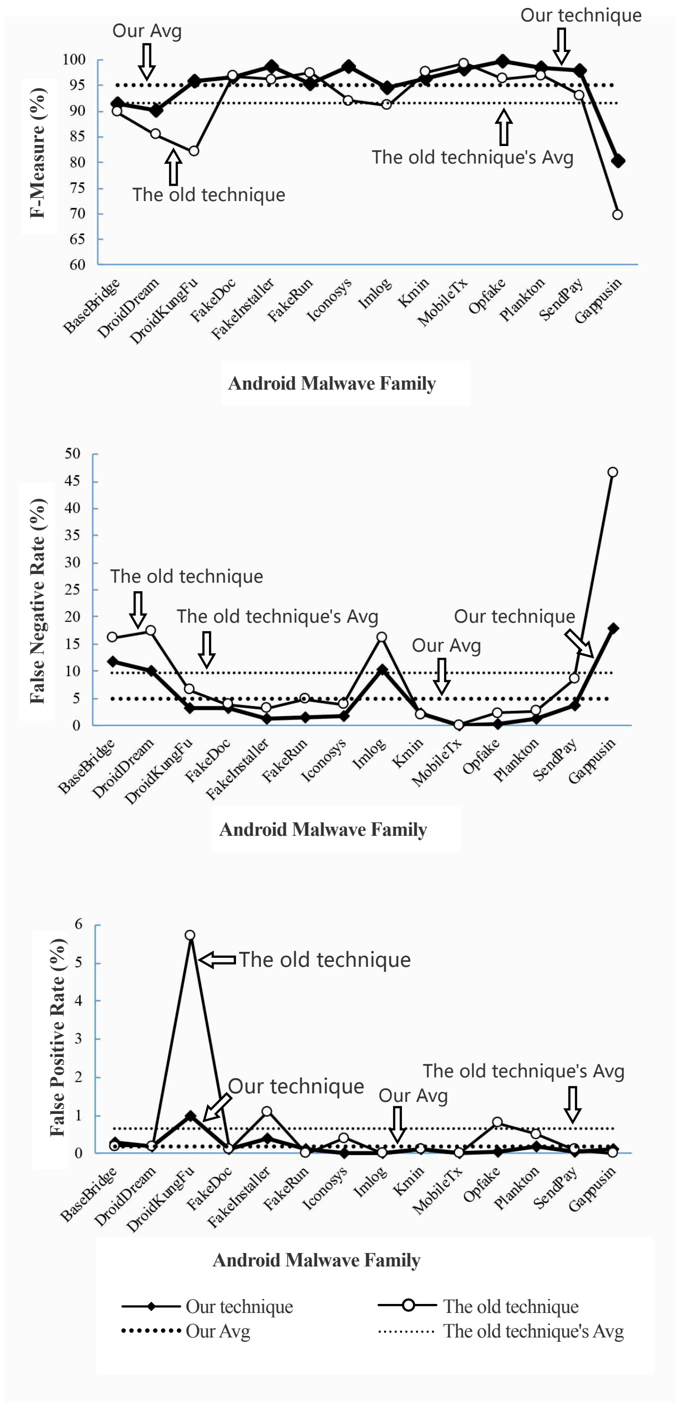


Fig. 5. Comparison of classification effects using two techniques

E. Study3: Effectiveness for malware families of varying sizes

In this subsection, we will demonstrate the classification effectiveness of our technique on malware families of varying sizes, ranging from large to small families. We employ our technique on the entire dataset in Table II, including both Dataset1 and Dataset2. We run the cross-validation for 10 rounds, compute the mean metric values for each malware family using the same method depicted in Subsection D, and show the classification results in Table VI.

As Table VI demonstrates, the classification results of our technique are as follows: The average F-measure value is 90.5%, the average FNR is 4.8%, and the average FPR is 0.2%. Thus, our technique can classify malware into families with high harmonic mean of the precision and the recall rate, very low false negative rate, and very low positive rate.

In summary, our malware family classification technique is effective even for malware families of varying sizes, including small families.

TABLE VI
AVERAGE CLASSIFICATION RESULTS OF 20 MALWARE FAMILIES USING OUR TECHNIQUE

For 20 Malware families	Precission	Recall	F-Measure	FNR	FPR
(Average)	0.927	0.889	0.905	0.111	0.002

V. CONCLUSION AND FUTURE WORK

Due to Android malware' ever-increasement and the limited variety of malware families, it is important to automatically categorize Android malware into families of close variants. In this paper, we presented an automatic and effective malware family classification approach based on deep learning of code images of Android malware. The code images were grayscale images converted from binary bytecodes of the malware. A CNN was constructed for the image classification based on a pre-trained GoogLeNet Inception-v3 instance. The results of our experiments show that the proposed approach is effective to identify the families of Android malware, including small families.

In the future, we plan to conduct the following further studies: (1) Utilize localized as well as global features of Android code images to tackle against such malware countermeasures as relocating binary sections. (2) Construct a large-scale dataset of Android malware with family labels, and then evaluate and improve the effectiveness of our approach on the dataset.

ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their valuable and constructive comments.

REFERENCES

[1] "Mobile security reports in china of 2017," [Online]. Available: http://zt.360.cn/1101061855.php?dtid=1101061451&did=491106049. [Accessed: March-2018].
[2] S. Sachdeva, R. Jolivot, and W. Choensawat, "Android malware classification based on mobile security framework." *IAENG International Journal of Computer Science*, vol. 45, no. 4, pp. 514–522, 2018.
[3] "Internet security threat report of 2017," [Online]. Available: https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf. [Accessed: March-2018].
[4] K. A. Ryusei Fuji, Shotaro Usuzaki *et al.*, "Investigation on sharing signatures of suspected malware files using blockchain technology," in *Lecture Notes in Engineering and Computer Science: Proceedings of The International MultiConference of Engineers and Computer Scientists 2019*, Hong Kong, 13-15 March 2019, pp. 94–99.
[5] I. N. Ahmad, F. Ridzuan, M. M. Saudi, S. A. Pitchay, N. Basir, and N. Nabila, "Android mobile malware classification using a tokenization approach," in *Lecture Notes in Engineering and Computer Science: The World Congress on Engineering and Computer Science 2017*, San Francisco, USA, 25-27,October 2017, pp. 271–285.

[6] L. H. Z. Madihah Mohd Saudi *et al.*, "A new mobile malware classification for camera exploitation based on system call and permission," in *Lecture Notes in Engineering and Computer Science: Proceedings of the World Congress on Engineering and Computer Science 2017*, San Francisco, USA, 25-27 October 2017, pp. 95–100.

[7] L. Nataraj, S. Karthikeyan, G. Jacob, and B. Manjunath, "Malware images: visualization and automatic classification," in *Proc. the 8th international symposium on visualization for cyber security*. ACM, 2011.

[8] "Microsoft malware classification challenge (big 2015)," [Online]. Available: https://bindog.github.io/blog/2015/08/20/microsoft-malware-classification. [Accessed: March-2018].

[9] "Kaggle microsoft malware," [Online]. Available: https://github.com/xiaozhouwang/kaggle_Microsoft_Malware. [Accessed: March-2018].

[10] Y. Yang and T. Chen, "Android malware family classification method based on the image of bytecode," *Chinese Journal of Network and Information Security*, vol. 2, no. 6, pp. 38–43, 2016.

[11] M. Oquab, L. Bottou, I. Laptev, and J. Sivic, "Learning and transferring mid-level image representations using convolutional neural networks," *Computer Vision and Pattern Recognition*, pp. 1717–1724., 2014.

[12] Z. Yan, H. Zhang, R. Piramuthu, V. Jagadeesh, D. DeCoste, W. Di, and Y. Yu, "Hd-cnn: hierarchical deep convolutional neural networks for large scale visual recognition," in *Proc. the IEEE international conference on computer vision*. IEEE, 2015, pp. 2740–2748.

[13] Y. Wei, W. Xia, M. Lin, J. Huang, B. Ni, J. Dong, Z. Yao, and S. Yan, "Hcp: A flexible cnn framework for multi-label image classification," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 9, pp. 1901–1907, 2016.

[14] E. Maggiori, Y. Tarabalka, G. Charpiat, and P. Alliez, "Convolutional neural networks for large-scale remote-sensing image classification," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 55, no. 2, pp. 645–657, 2017.

[15] "Transfer learning and fine-tuning deep convolutional neural networks," [Online]. Available: http://blog.revolutionanalytics.com/2016/08/deep-learning-part-2.html. [Accessed: March-2018].

[16] T. S. S. Rokrakthong and N.Tumrukwatthana, "Applying cnn to infrared thermography for preventive maintenance of electrical equipment," in *Lecture Notes in Engineering and Computer Science: Proceedings of The International MultiConference of Engineers and Computer Scientists 2019*, Hong Kong, 13-15 March 2019, pp. 367–370.

[17] F. Z. Yuanyuan Zhang, Ning Wu and Rehan, "Design of multifunctional convolutional neural network accelerator for iot endpoint soc," in *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering and Computer Science 2018*, San Francisco, USA, 23-25 October 2018, pp. 16–19.

[18] Y. S. Ma Jin and I. McLoughlin, "End-to-end dnn-cnn classification for language identification," in *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering 2017*, London, U.K., 5-7 July 2017, pp. 199–203.

[19] D. Gibert, "Convolutional neural networks for malware classification," *University Rovira i Virgili, Tarragona, Spain*, 2016.

[20] "Large scale visual recognition challenge 2012," [Online]. Available: http://www.image-net.org/challenges/LSVRC/2012. [Accessed: March-2018].

[21] "The drebin dataset," [Online]. Available: https://www.sec.cs.tu-bs.de/ danarp/drebin/. [Accessed: March-2018].

[22] "Large scale visual recognition challenge 2012," [Online]. Available: https://www.tensorflow.org/. [Accessed: March-2018].

[23] A. Oliva and A. Torralba, "Modeling the shape of a scene: a holistic representation of the spatial envelope," *International Journal of Computer Vision*, vol. 42, no. 3, pp. 145–175, 2001.

[24] A. Dey, S. Bhattacharya, and N. Chaki, "Byte label malware classification using image entropy," in *Advanced Computing and Systems for Security*. Springer, 2019, pp. 17–29.

[25] X. Han, X. YAO, and W. Qu, "Malicious code family tagging based on image texture clustering technology," *Journal of PLA University of Science and Technology*, vol. 15, no. 05, pp. 440–449, 2014.

[26] S. He and J. Liu, "Malicious code family tagging based on malicious code image fingerprint," *Communications Technology*, vol. 50, no. 3, pp. 545–549, 2017.

[27] D. Lowe, "Object recognition from local scale-invariant features," vol. 2. IEEE, 1999, pp. 1150–1157.

[28] Z. Cui, F. Xue, X. Cai, Y. Cao, G.-g. Wang, and J. Chen, "Detection of malicious code variants based on deep learning," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3187–3196, 2018.

[29] S. Ni, Q. Qian, and R. Zhang, "Malware identification using visualization images and deep learning," *Computers & Security*, vol. 77, pp. 871–885, 2018.

[30] K. Han, J. Lim, and E. Im, "Malware analysis method using visualization of binary files," *Research in Adaptive and Convergent Systems*, pp. 317–321., 2013.

[31] L. Deshotels, V. Notani, and A. Lakhotia, "Droidlegacy: Automated familial classification of android malware," no. 3. ACM, 2014.

[32] G. Suarez-Tangil, J. Tapiador, and P. Peris-Lopez, "Dendroid: A text mining approach to analyzing and classifying code structures in android malware families," *Expert Systems with Applications*, vol. 41, no. 4, pp. 1104–1117, 2014.

[33] S. Dash, G. Suarez-Tangil, and S. Khan, "Droidscribe: Classifying android malware based on runtime behavior," *Security and Privacy Workshops*, pp. 252–261., 2016.

[34] L. Massarelli, L. Aniello, C. Ciccotelli, L. Querzoni, D. Ucci, and R. Baldoni, "Android malware family classification based on resource consumption over time," *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*, no. 17652649, 2017.

[35] T. Chakraborty, F. Pierazzi, and V. Subrahmanian, "Ec2: Ensemble clustering and classification for predicting android malware families," *IEEE Transactions on Dependable and Secure Computing*, pp. 1545–5971., 2017.

[36] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of android malware," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 26, no. 3, p. 11, 2018.

[37] L. Zhang, V. L. Thing, and Y. Cheng, "A scalable and extensible framework for android malware detection and family attribution," *Computers & Security*, vol. 80, pp. 120–133, 2019.

[38] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," no. 1409.1556, 2014.

[39] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions." IEEE, 2015, pp. 1–9.

[40] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computervision." IEEE, 2016, pp. 2818–2826.

[41] S. Sun, W. Chen, L. Wang, X. Liu, and T. Liu, "On the depth of deep neural networks: A theoretical view." AAAI, 2016, pp. 2066–2072.

[42] M. Bianchini and F. Scarselli, "On the complexity of neural network classifiers: A comparison between shallow and deep architectures," *IEEE Transactions on Neural Networks*, vol. 25, no. 8, pp. 1553–1565, 2014.

[43] "Google inception v3 architecture trained for imagenet project," [Online]. Available: http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz. [Accessed: March-2018].

[44] "Tensorflow 0.12.0-rc0," [Online]. Available: https://pypi.org/project/tensorflow/0.12.0rc0/ [Accessed: March-2018].

[45] "Prettytensor 0.7.1," [Online]. Available: https://github.com/google/prettytensor. [Accessed: March-2018].

[46] P. Refaeilzadeh, L. Tang, and H. Liu, "Cross-validation," pp. 532–538, 2009.

[47] S. Ruder, "An overview of gradient descent optimization algorithms," no. 1609.04747, 2016.

[48] D. Kingma and J. Ba, "Adam:a method for stochastic optimization," no. 1412.6980, 2014.

[49] "How to retrain inception's final layer for new categories," [Online]. Available: https://www.tensorflow.org/tutorials/image_retraining?hl=zh-cn. [Accessed: March-2018].

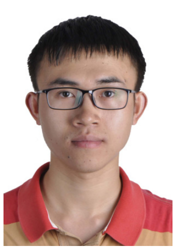
Yuxia Sun was born in Hubei province, China. She received the B.S. degree in computer software from Huazhong University of Science and Technology, Wuhan, China and the Ph.D. degree in computer software and theory from Sun Yat-sen University, Guangzhou, China.

She was a Research Associate at the University of Hong Kong in 2007. From 2009 to 2010, she was a Research Scholar in the College of Computing at Georgia Institute of Technology. She was a Research Associate at the Hong Kong Polytechnic University in 2017. She is currently an Associate Professor in the Department of Computer Science at Jinan University, Guangzhou, China. Her research interests include software engineering, software safety and system safety.

Dr. Sun's awards include Ministry of Education Nominated State Science and Technology Award in China, and Guangdong Province Science and Technology Award in China.

Yanjia Chen was born in Guangdong province, China, in 1994. She received the B.S. degree from Jinan University, Guangzhou, China.

She is now studying for the M.S. degree at Jinan University and her research focuses on software safety.



Yuchang Pan was born in Guangdong province, China, in 1993. He received the B.S. degree in computer science and technology from Guangdong University of Technology, Guangzhou, China and the M.S. degree in computer technology from Jinan University, Guangzhou, China.

He is currently a research assistant at Jinan University, Guangzhou, China His research focuses on software safety.

Lingyu Wu was born in Guangdong province, China, in 1991. received his B.S. degree in Computer Science and Technology from Jinan University.

He is currently a research assistant at Jinan University, Guangzhou, China. His research focuses on software testing and software safety.