A Multi-class Detection System for Android Malicious Apps Based on Color Image Features

Hua Zhang[1], Jiawei Qin[1(✉)], Boan Zhang[1], Hanbing Yan[2], Jing Guo[2], and Fei Gao[1]

[1] State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China
qinjiawei@bupt.edu.cn

[2] The National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing, China

Abstract. The visual recognition of Android malicious applications(Apps) is mainly focused on the binary classification using gray-scale images, while the multi-classification of malicious App families is rarely studied. If we can visualize the Android malicious Apps as color images, we will get more features than using grayscale images. In this paper, a method of color visualization for Android Apps is proposed and implemented. Based on this, combined with deep learning models, a multi-classifier for the Android malicious App families is implemented, which can classify 131 common malicious App families. Compared with the App classifier based on the gray-scale visualization method, it is verified that the classifier using the color visualization method can achieve better classification results. This paper uses three classes of Android App APK features: classes.dex file, class name collection and API call sequence as input for App visualization, and analyzes the classifier detection accuracy and detection time under each input characteristics. According to the experimental results, we found that using the API call sequence as the color visualization input feature can achieve the highest detection accuracy rate, which is 96.01% in the ten malicious family classification and 100% in the binary classification.

Keywords: Android malicious Apps · Deep learning · Visualization · Multi-class detection

1 Introduction

The openness of the Android system, while helping it win the market, has also brought it huge risks. According to the CVE [2](Common Vulnerabilities Exposures) 2018 annual report, the Android system ranks second in the vulnerability list with 611 vulnerabilities. They have brought more opportunities to malicious App developers. As a large amount of user data is connected to the

D. Wang et al. (Eds.): SPNCE 2020, LNICST 344, pp. 186–206, 2021.
https://doi.org/10.1007/978-3-030-66922-5_13

Internet via mobile phones and spread on the network, the target of hacking is gradually shifting from traditional PCs to mobile devices. As a result, more and more researchs [6,20,21,24,26,30] focused on analyzing Android malicious Apps.

A difficult but important issue in the Android malicious App family classification is how to classify malicious Apps in the presence of a large number of families and achieve high accuracy. With the proliferation of Android malicious Apps, there are more and more Android malicious App families. How to distinguish the endless Android malicious App families has become a greater challenge. Existing research shows that malicious behaviors between malicious App families overlap more and more. The detection standards manually formulated after feature extraction cannot distinguish between families with high similarity, and the accuracy of fingerprint-based methods is getting lower and lower [30].

Using machine learning methods to classify Android malicious Apps has achieved high accuracy [6,8,10,13,15,25]. However, due to its feature generation engineering that relies on expert knowledge, it is difficult for the above-mentioned classifiers to maintain a high accuracy rate after the changes of malware behavior trigger method. Joshua et al. [11] used a machine learning method of classification regression tree to study a family classifier that can classify 33 malicious App families manually labeled in the AMG [29] data set, achieving 95% accuracy. Wang [24] and others proposed the use of deep learning detection methods to implement Android malware detection systems, nonetheless, it did not study the implementation of multi-classification of malware. Andronio [6] analyzed the behavioral characteristics of Android ransomware and implemented a detection model for ransomware.

In exploring the visualization of malicious software, Nataraj et al. [19] drew the gray-scale image of Windows malicious software in a linear way, and Applied GIS T (Gabor filter) to image to obtain features. Then using K-Nearest Neighbor (KNN) algorithm as an automatic classification technology to classify 25 malicious software families and reached a accuracy rate of 98%. Jung et al. [16]used gray-scale image and convolution neural network model to conduct binary classification experiments on Android malware and benign. Nonetheless, no research has been done on color visualization or Android malicious App family classification. It focused on the benefits of visualizing the "data" section of the classes.dex file. And at the end of the article, it was pointed out that the direction of future research is the method of color image visualization.

In the common deep learning model, three-channel color images are used as training samples. For deep learning classifiers, compared to grayscale images, color image visualization theoretically has a higher dimension and more processing flows, so more features are learned and classification accuracy is higher. However, in the existing research, there is no method to classify the Android malicious family using only color image visualization.

In this paper, we classify Android malicious into multiple families by color visualization combined with deep learning. We propose a method of color visualizing Android App, study the features suitable for color visualization, and verify the effect of this method on the classification of a large number of malicious App

families with overlapping malicious behaviors. The specific contributions of this paper as follows:

- A method of color visualization Android App is proposed and applied to malicious App family classification. In view of the better performance of the deep learning classification model on color picture classification tasks, this paper studies the effect of using gray image features and color image features in the Android malicious App family classification, which validates the feasibility of the App of color image visualization to the Android malware families, and proposes a color image visualization method for Android malicious family classification.
- The influence of different features on Android malicious App family classification is studied, and the features that are most suitable for color visualization are obtained. We analyzed the possible collections of Android malicious App features, and selected three more common collections as experimental objects: *classes.dex* file, App class name collection, and App interface call(API) collection. We performed color visualization on each feature, and conducted classification experiments. Using the deep learning method to study the performance of the three features in classification time and accuracy. Finally, according to the experiment results, it is judged that using API call sequence characteristics is the best choice.
- A classifier is implemented for a large number of malicious App families with overlapping malicious behaviors. After analyzing the characteristics of malicious App families, it is found that the increasing number of malicious App families brings difficulties to family classification: the similarity between families increases, and similar malicious behaviors overlap. We used color visualization combined with convolutional neural networks and deep residual networks to classify 131 malicious App families and reached a classification accuracy of 96.36%.

2 Related Work

Android malicious App visualization is a new trend in recent years. One of the common methods for the visualization of binary files comes from the paper by Conti et al. [9]. They used four different ways to visualize binary files. The first method is to draw each byte linearly to generate gray-scale images, where empty bytes are described as black pixels and the 0xff bytes are described as white pixels. The second method is to color a portion of the bytecode to indicate the presence or absence of a particular byte value. This method is especially useful for finding compressed portions or ascii code portions. Third, the traditional hex editor is implemented, which converts the binary to hexadecimal and then colors it. Fourth, using dot plots to show the cross entropy of a file, a dot plot is a way to visualize the similarity or self-similarity of data.

In the exploration of malware visualization, Gennisse et al. [12] used a partial color visualization method to study Android malicious family classification.

Zhang et al. [27] decompiled the executable file to get the opcode sequence, and then converted these sequences into the form of an image, and finally performed further feature extraction and recognition through CNN. They did not characterize the executable file and directly used all the data, which may lead to false positives in model identification. Kancherla et al. [17] converted the executable files to grayscale images, and then selected the model based on the intensity and texture-based feature selection for malware recognition. Grayscale images retain fewer features than color images, which can reduce the accuracy of malware identification. Nataraj et al. [19] linearly mApped grayscale images of Windows malware in the same way as Conti. The GIST (Gabor filter) was Applied to the image to obtain features. The K-Nearest Neighbor (KNN) algorithm was used as the automatic classification technology, and the classification accuracy rate of the 25 malware families reached 98%. In theory, the characteristics of color images are more abundant than grayscale images, and the accuracy of classification for deep learning should be higher. However, there is a lack of research on the use of color images for the classification of Android malicious families.

In recent years, there are more and more studies focusing on Android malware Apps. However, many studies only focus on the two categories of "malicious" and "benign". DroidDolphin [25] used dynamic analysis techniques such as DroidBox [3] to extract thirteen features from the collected Apps, and constructed a detection system using support vector machine (SVM) model. Crowdroid [8] used dynamic analysis to extract API (App Programming Interface) calls as features and K-means clustering to detect malware. RiskRanker [13] classified Apps into high-risk, medium-risk and low-risk to judge malicious Apps. We find that there are few papers focusing on family classification.

In researchs of multi-family classification, DroidLegacy [10] focused the part of malicious families using piggybacking technology to embed malicious code in benign Apps during repackaging, however this type of malware is not representative of all malware. Dendroid [23] used text mining technology and data flow characteristics to construct a malicious family detection system based on App code structure analysis. It classified 33 families and achieved good results. However, no further research has been done on more families.

In the face of the endless stream of Android malicious App families, how to implement a family classification for most common malicious Apps becomes a problem: as the number of malicious families increases, the malicious behaviors of different families overlap [15]. Different malware families with higher malicious similarity are more difficult to distinguish, and the accuracy of the classifier will also decrease. Due to the lack of reliable manual annotation data sets, some papers use labeled data for a large number of family classification experiments, nevertheless, the results obtained are often questionable. RevealDroid [11] used the classification regression tree algorithm, combined with packet-level and method-level API calls, reflections, and Native code at package and method levels as features, and it successfully classified 33 families on AMC datasets. However, they did not further choose a reliable database by manual classification for more research. Instead, they used the AV [22] classifier to classify and

label the collected unlabeled data, so the accuracy of this machine classifier has been questioned, RevelDroid also pointed this out in the paper.

3 Prerequirement

3.1 Malicious App Behavior of Android

Android malicious Apps refer to Android Apps with malicious intentions, which do great harm to mobile phones and users.Malicious App activities can be divided into four stages, the first is the "infection" stage. Malicious Apps often disguise as normal Apps, the common form is the free version of paid Apps, users often misinstall such malicious Apps. After the "infection" is the "destruction" stage, Apps may cause damage to the system, such as enhancing the permissions of malicious Apps, deleting mobile files, locking mobile phones and modifying passwords, which can prevent the normal use of users. "leak" stage can occur simultaneously with destruction, malicious Apps may collect user information and send it to the designated server. Finally, in the "last propagation" stage, malicious Apps may use infected mobile phones to send links or e-mails, alluring unaware friends to download them to click on or download Apps, so as to achieve the purpose of dissemination of malicious Apps. Generally speaking, an App can be judged as malicious if it has the following behaviors [29]:

- Consume the user's mobile phone fee and occupying the mobile phone system resources, causing other Apps to not work properly and preventing users from using the system normally other Apps unable to work properly, hindering the user's use of the system.
- Record the user's screen (such as screen capture or screen recording) without his or her permissions, and obtain private information such as user account and password.
- Allow others to remotely control the user's mobile phone without the user's permission.
- Intimidate the user, such as setting the lock screen to "You will be jailed" and modify the power-on password.

3.2 Android Malicious App Family

Android malicious App family refers to a kind of malicious Apps with the same behavior, which is the product of the detailed division of malicious Apps according to their behavior. The ten common malicious App families are as follows:

1. *Geinimi*: accept remote instructions, control mobile phones, can read and delete short messages, mute phone ringtone, automatically download files and collect information from mobile phone then pass it back to the server.
2. *FakeInstaller*: send paid short messages to certain numbers and cause user fees to be consumed, which is abundant in repackaged versions of popular Apps.

3. *DroidKungFu*: allow attackers' remote access to the infected phones, and can use the root vulnerability to disguise themselves. Common functions include deleting an executable file, opening a web page, downloading and installing an App, opening a URL, launching other programs and so on.
4. *Plankton*: transmit the user's private information, such as the mobile phone IMEI and the user's browsing history data to the remote server, and modify browser home page, add bookmarked.
5. *Opfake*: forge the interface, let the user think the software is a normal App, and steal user information.
6. *GinMaster*: gain access by rooting the devices, thereby steal sensitive user information and send it to the server, and install other software without the user's permission.
7. *Kmin*: send the IMEI information of the device to the remote server. At the same time, they will further threaten the security of the mobile phone by calling according to the remote command and blocking the short message from the operator, which will consume a lot of money.
8. *BaseBridge*: are similar to the *Kmin* Family, but they can kill anti-virus software processes running in the background.
9. *Adrd*: are similar to the *Geinimi*, but they can change the settings of mobile phones.
10. *DroidDream*: get information through rooting mobile devices, download malicious Apps silently in the background, usually run at night while the device is charging in order to avoid the monitoring of power consumption by the detection software.

In addition, there are many other malicious App families, such as the Nickyspy family: record dial-in and dial-out information for infected mobile phones, record users' GPS information, and send text messages to other numbers. Zsone family: automatically send text messages to subscribe for paid content, thus achieving the purpose of consuming telephone charges. Obad family: elevate system privilege to prevent being uninstalled and send text messages to value-added service numbers for profit. Zitmo family: steal verification code sent from the bank. The differences between these families vary, and the large overlap of malicious behavior makes it difficult to distinguish some of them.

4 Our Approach

4.1 Select Features

The size of different Android apps varies widely. If the entire App file is visualized, the visualized image sizes may differ by hundreds of times, which will bring a huge burden to the classification task of the images. Therefore, we need to select the features that can represent the behavior of the App and then perform color visualization.

In the internal structure of an Android App, in addition to the *dex* file that stores the code and the *AndroidManifest.xml* file that stores the configuration

information, there is *res* directory that stores resource files such as image files and audio files. This part of the file has nothing to do with the code logic of the App, it is only stored as a resource of the App, and does not affect the behaviers of the program. A small number of malicious Apps may hide malicious code in image files. Such Apps are beyond the scope of this paper, so the resource file is not included in the selection of visualization features.

The basis for our classification of malicious App families is that each Android App has different performance in *classes.dex*, and *AndroidManifest.xml*, which reflect different characteristics and behaviors to distinguish malicious programs from different families [7,10,18].

classes.dex is a bytecode file that compiles Java files into classes and saves them, it contains the package name, classes, methods, variables and Application interfaces (APIs) of the Android App. Most of App's functional behaviors are implemented based on APIs, so we choose it as one of its features.

Every activity component, service component, content provider component, and broadcast receiver component in the Android App needs to be registered in the *AndroidManifest.xml* file. In addition, it also contains some permissions and *SDK* information. So it is part of the features.

4.2 Android App Color Visualization

The purpose of Android App color visualization is to convert the extracted features consist of sequence of APIs of the App and the information of the *AndroidManifest.xml* file into representations of color images. Figure 1 shows the detailed process of color visualization.

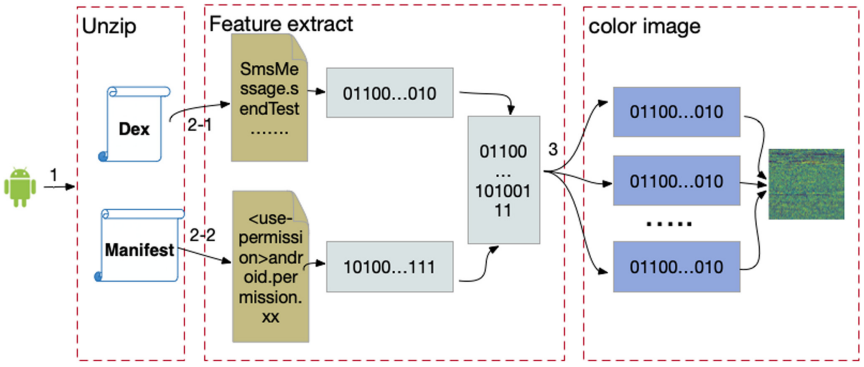


Fig. 1. Android App for color visualization process.

Unzip. As described above, in order to get API call sequence and *Androidmanifest.xml* file of the App, we need to decompress the Apk file. After decompression we get the above two files.

Feature Extraction. We use the tool androguard [1] to reverse the *classes.dex* file, and build the control flow graph(CFG) of the App from the bytecode

obtained by the reverse. We extract the API call sequence from each node of the CFG, that is, the API call sequence for the i-th node is

$$NAPI_i = [API_1, API_2, \ldots API_n] \tag{1}$$

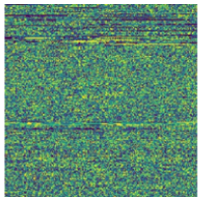
Then we map the connection relationship of nodes in CFG to the API call sequences from all nodes. In this way, API call sequence of the entire App is

$$AApp = [NAPI_1, NAPI_2, \ldots NAPI_n] \tag{2}$$

For the *Androidmanifest.xml* file, we use the method of bytecode restoration to restore it to readable text content. In order to convert the above two features into a visualized image, we need to convert the features of the string into numeric values. We use the linear rendering visualization method [9] to visualize them.

Color Visualization. The common binary file visualization method is to convert each byte to a value between 0 and 255, each value corresponds to a pixel in the image (0 is black, 255 is white). For image classification, more image channels mean that more pixels and more features that can be learned. The color visualization conversion method used in this paper is to represent a byte value with three channels of pixels. We use a "blue-green-yellow" color image instead of "black-white" in a grayscale image to represent a range of pixels. The generated image is no longer a single-channel grayscale image, but a three-channel color image, and the value of each channel is not simply repeated.

(a) Grayscale image.



(b) Color image.

Fig. 2. Grayscale image and color image of the same App. (gray image (Figure a on the left) and color image (Figure b on the right)).

As shown in Fig. 2, gray image (Fig. 2a) and color image (Fig. 2b) are generated from the same Android malicious App. the color image successfully maps the original "black-white" of the gray image to the "blue-green-yellow" color range. By analyzing the image file, the original single-layer channel gray image is transformed into three-layer channel color image, which contains more abundant information.

4.3 Malware Detection

Figure 3 shows the classification process of the Android malware multi-classifier. We roughly divide this process into two parts, which are the color visualization

of the application and classification process using machine learning models. The details are described as follows.

Color Visualization. For an App to be detected, we need to decompress it and to get the *classes.dex* file and *Androidmanifest.xml*file. Then through the feature conversion process described in the previous section, the App file is color visualized to a color image, which is the input to the image classifier in the next process.

Classification. Algorithms that are commonly used in image classification include support vector machines(SVM), K nearest neighbors, neural networks, random forests. However, based on previous experiments results, deep residual network (ResNet) [14] have get better performance in image classification than the above algorithms. Therefore, we choose ResNet to process color image features. We use a network structure with fewer hidden layers, it contains two convolutional layers, two residual modules and two fully connected layers.

Result. The purpose of this paper is to achieve multi-classification of Android malware, therefore, the output of our system is the malicious family name of the app to be detected.

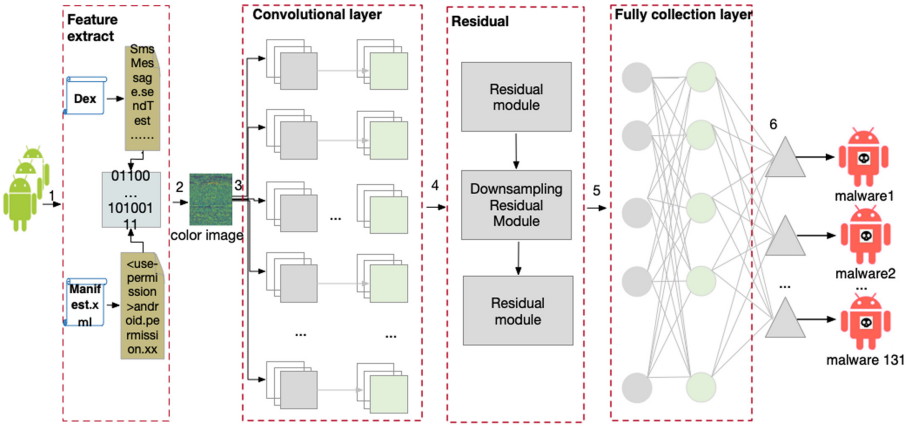


Fig. 3. Overview of malicious app multi-classification system based on color visualization.

5 Experiments

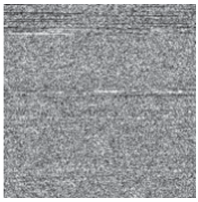
5.1 Characterization of Gray and Color Images

We select the *FakeInstaller* [4] and *Plankton* [5] families in the Drebin [7] data set as experimental data, a total of 1120 Apps, and form a training set and a test set according to the ratio of about 8:2. The size of training sample set of *FakeInstaller* is 403, test sample set size is 100. The size of training sample set of *Plankton* is 497, test sample set size is 120. A single-channel grayscale image

and color image are generated for each App in the training set and the test set according to the steps of extracting features, and visualization. After obtaining image features, they are trained and classified by using ResNet. Also the number of training rounds is 100.

Binary Classification of Single Channel Grayscale Image and Three-Channel Grayscale Image. In order to make a comprehensive comparison with the grayscale visualized images, we manually add three-channel grayscale images. We copy the single-channel grayscale image twice and superimpose them with the original image to form a three-channel grayscale image. As shown in Fig. 4, 4(a) is a single-channel grayscale image, and 4(b) is a three-channel grayscale image.

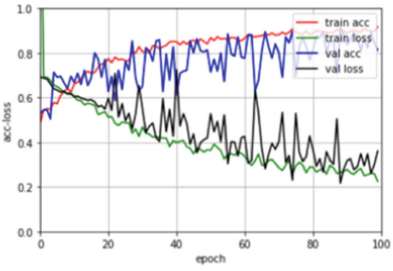
(a) Single channel grayscale image.



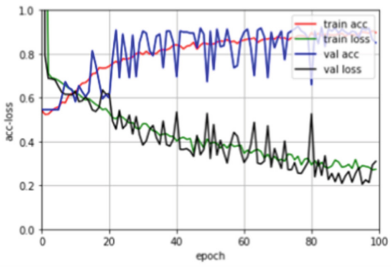
(b) Three-channel grayscale image.

Fig. 4. Grayscale images of different channels of the same App.

The single-channel grayscale image classification result is shown in Fig. 5(a), and the three-channel grayscale image classification result is shown in Fig. 5(b). The abscissa is the number of training rounds, and the ordinate is the accuracy and loss. The accuracy of single-channel grayscale images is 81.36%, and the classification accuracy of three-channel grayscale images is 85.00%.



(a) The experiment results of single channel grayscale image classification.



(b) The experiment results of three-channel grayscale image classification.

Fig. 5. The experiment results of different channel gray image classification.

The accuracy of three-channel grayscale image is higher than single-channel grayscale image. It proves that multi-channel image is more effective for identifying Android malware Apps. Although the classification accuracy has been improved, the improvement of accuracy is only increased by 3.64%. It is proved that the simple repetition of single-channel images does not contribute much to the classification effect.

Binary Classification of Three-Channel Color Images. The results of the three-channel color image classification are shown in Fig. 6. The classification accuracy rate is 90.91%. The classification accuracy is improved by 9.55% compared with the single-channel grayscale image, also the accuracy compared with the three-channel grayscale image is increased by 5.91%. In the case of the same number of channels, the color image can help the CNN model to learn and classify better than the purely superimposed gray image. It has more features than the grayscale visualization image, and is more suitable for classification.

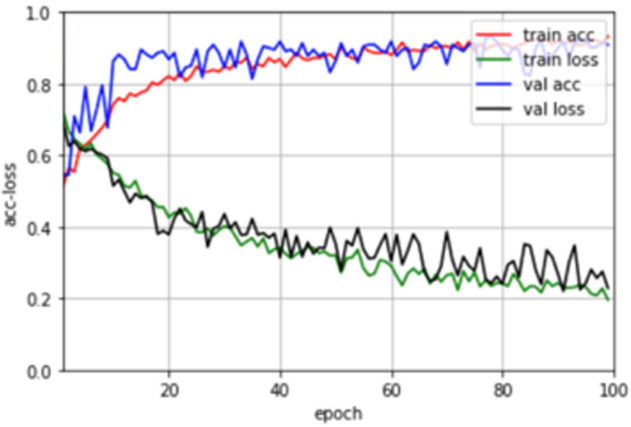


Fig. 6. The results of three-channel color image classification.

5.2 Multi-classification of Color Images Using Different DL Models

The Drebin [7] dataset collected a total of 5,560 samples from 179 different malicious App families. Some Apps cannot be opened with androguard [1], and further inspection revealed these Apps have been corrupted, possibly due to improper storage or file corruption during decompression. In the end, we get 131 malicious App families, each of which has a sample size greater than 2. We choose Convolutional Neural Network (CNN) [28] and Deep Residual Network (ResNet) [14] for multi-classification respectively. Among them, the CNN uses a 5-layer network structure and uses ReLU as the activation function for training. ResNet is trained by using a 20-layer network structure.

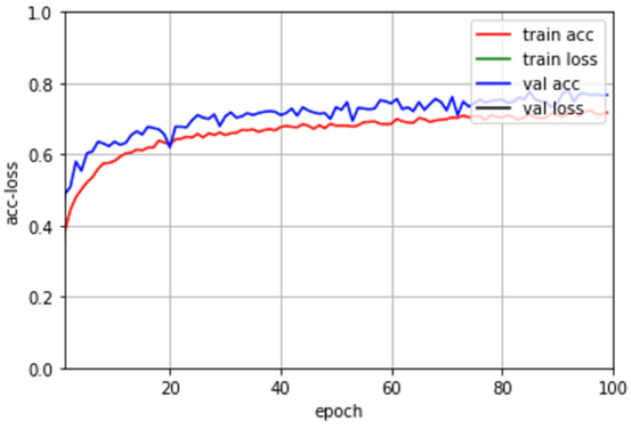


Fig. 7. Experimental results of 131 family classification by using CNN.

CNN. Figure 7 shows the results of a CNN classification experiment on 131 Android malicious families. The accuracy of classification can reach 76.68%. Since the loss function values are all greater than 1, so they are not shown in the figure.

ResNet. It can be seen from Fig. 8 that when the number of training iterations is small, the ResNet is not able to classify the image well, and the phenomenon of overfitting appears. After 80 complete trainings, the model can well distinguish most families, the accuracy and loss curve have stabilized, and the model classification accuracy has reached 96.36%.

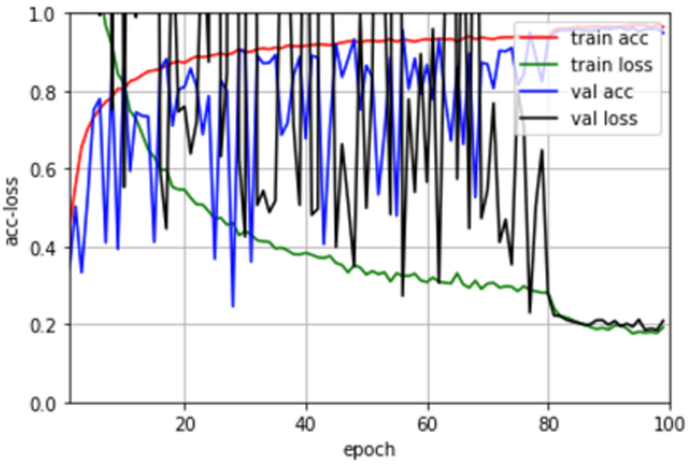


Fig. 8. Classification experimental process for 131 malicious sample families by DRNN(Deep residual neural network).

The details of classification accuracy of each family are shown in Table 1. It can be seen from the table that the ResNet can maintain a high accuracy rate, and identify more families.

Table 1. Accuracy of classification of 131 malicious families.

No	Family	DRNN	CNN	No	Family	DRNN	CNN	No	Family	DRNN	CNN
0	Fjcon	100.00%	0.00%	50	CellSpy	100.00%	0.00%	100	SpyBubble	100.00%	0.00%
1	Fakengry	100.00%	0.00%	51	Tesbo	100.00%	0.00%	101	Geinimi	98.91%	23.91%
2	Sakezon	87.50%	0.00%	52	Ceshark	71.43%	0.00%	102	Koomer	100.00%	0.00%
3	Placms	83.33%	0.00%	53	Fakelogo	100.00%	0.00%	103	Jifake	66.67%	0.00%
4	Generic	100.00%	0.00%	54	Nyleaker	88.89%	0.00%	104	CgFinder	0.00%	0.00%
5	Gonca	100.00%	0.00%	55	Gmuse	100.00%	0.00%	105	Nisev	100.00%	0.00%
6	GGtrack	100.00%	0.00%	56	Coogos	87.50%	0.00%	106	Stiniter	62.50%	0.00%
7	Anti	100.00%	0.00%	57	SeaWeth	83.33%	0.00%	107	Imlog	100.00%	23.26%
8	Mobilespy	85.71%	0.00%	58	Dialer	100.00%	0.00%	108	Replicator	100.00%	0.00%
9	FakeTimer	100.00%	0.00%	59	YcChar	100.00%	0.00%	109	Kiser	100.00%	0.00%
10	Antares	100.00%	0.00%	60	Yzhc	100.00%	89.19%	110	Spitmo	100.00%	100.00%
11	SheriDroid	100.00%	0.00%	61	TrojanSMS.H	100.00%	0.00%	111	EICAR-Test	33.33%	0.00%
12	FinSpy	100.00%	0.00%	62	Dabom	100.00%	0.00%	112	Iconosys	98.03%	91.45%
13	RootSmart	100.00%	0.00%	63	MobileTX	100.00%	100.00%	113	GinMaster	97.05%	92.63%
14	DroidDream	98.77%	100.00%	64	Plankton	99.04%	89.92%	114	Luckycat	100.00%	0.00%
15	KsApp_	100.00%	0.00%	65	ISmsHider	50.00%	0.00%	115	Kidlogger	100.00%	0.00%
16	DroidKungFy	100.00%	95.50%	66	Adsms	100.00%	0.00%	116	SMSZombie	90.00%	0.00%
17	SendPay	100.00%	94.92%	67	DroidRooter	100.00%	0.00%	117	FakePlayer	93.75%	0.00%
18	Gamex	100.00%	0.00%	68	FakeDoc	96.21%	75.00%	118	Moghava	100.00%	0.00%
19	LifeMon	100.00%	0.00%	69	NickyRCP	50.00%	0.00%	119	TheftAware	100.00%	0.00%
20	Saiva	100.00%	0.00%	70	Boxer	100.00%	100.00%	120	Vidro	100.00%	0.00%
21	Copycat	100.00%	0.00%	71	Dogowar	100.00%	0.00%	121	Fidall	100.00%	0.00%
22	Penetho	100.00%	0.00%	72	FakeRun	93.44%	55.74%	122	Steek	71.43%	0.00%
23	SmsWatcher	100.00%	0.00%	73	TrojanSMS.C	100.00%	0.00%	123	Typstu	100.00%	0.00%
24	RediAssi	100.00%	0.00%	74	TigerBot	100.00%	0.00%	124	SpyHasb	92.31%	0.00%
25	Aks	100.00%	0.00%	75	Hispo	100.00%	0.00%	125	AccuTrack	50.00%	0.00%
26	Raden	100.00%	0.00%	76	Fsm	0.00%	0.00%	126	Stealthcell	100.00%	0.00%
27	Mobinauten	25.00%	0.00%	77	Loozfon	50.00%	0.00%	127—	Hamob	92.86%	21.43%
28	BaseBridge	99.39%	87.84%	78	Lemon	100.00%	0.00%	128	Pirates	100.00%	0.00%
29	Dougalek	100.00%	100.00%	79	BeanBot	50.00%	0.00%	129	DroidSheep	100.00%	0.00%
30	Nickspy	81.82%	0.00%	80	Xsider	100.00%	0.00%	130	Gapev	100.00%	0.00%
31	Kmin	97.96%	60.54%	81	GAppusin	98.28%	55.17%				
32	ExploitLinux	95.59%	80.88%	82	Rooter	100.00%	0.00%				
33	Proreso	100.00%	0.00%	83	SpyPhone	100.00%	0.00%				
34	Vdloader	93.75%	0.00%	84	Glodream	98.53%	64.71%				
35	SmForw	100.00%	0.00%	85	Spyset	25.00%	0.00%				
36	Stealer	100.00%	0.00%	86	Spyoo	100.00%	0.00%				
37	FaceNiff	100.00%	0.00%	87	Mania	100.00%	0.00%				
38	SMSreg	100.00%	24.39%	88	Fakelnstaller	99.78%	97.19%				
39	Tapsnake	100.00%	0.00%	89	Opfake	100.00%	97.83%				
40	Bige	100.00%	0.00%	90	Zsone	75.00%	0.00%				
41	CrWind	0.00%	0.00%	91	Nandrobox	100.00%	30.77%				
42	GPSpy	100.00%	0.00%	92	QPlus	100.00%	0.00%				
43	Cosha	100.00%	0.00%	93	Fauxcopy	0.00%	0.00%				
44	FarMap	80.00%	0.00%	94	Trackplus	100.00%	0.00%				
45	FoCobers	100.00%	80.00%	95	Ackposts	50.00%	0.00%				
46	SerBG	100.00%	14.29%	96	Fatakr	100.00%	100.00%				
47	PdaSpy	100.00%	0.00%	97	Zitmo	92.86%	21.43%				
48	SpyMob	100.00%	0.00%	98	Flexispy	50.00%	0.00%				
49	FakeFlash	100.00%	0.00%	99	Adrd	100.00%	79.12%				

5.3 Color Visualization Experiments with Different Features

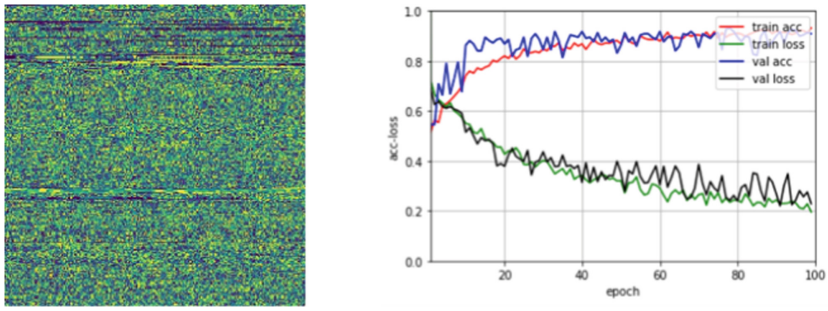
The features selected in this paper are (1) *classes.dex* file obtained by decompilation of the APK file; (2) the set of class name extracted from the class.dex file;

(3) API call sequence extracted from the App. We will measure the impact of different visualization features on malicious application family classification by detecting accuracy and efficiency.

Binary Classification Experiment. We selected *FakeInstaller* and *Plankton* families with a total of 1,550 Apps. 80% of the samples are classified into the training set and 20% into the test set, so the number of *FakeInstaller* training samples is 740, the number of test samples is 185. the number of *Plankton* training samples is 500, and the number of test samples is 125.

Table 2. Android malware classification experiment results with different features for color visualization features.

Feature	Time(s)			Accuracy
	Feature extraction	Color visualization	Total	
classes.dex	1.16	1	2.16	90.91%
Set of class name	1.19	1	2.19	100%
Sequence of API call	1.20	1	2.20	100%



(a) Visualized image of the classes.dex file.

(b) Experimental results of class.dex visual classification.

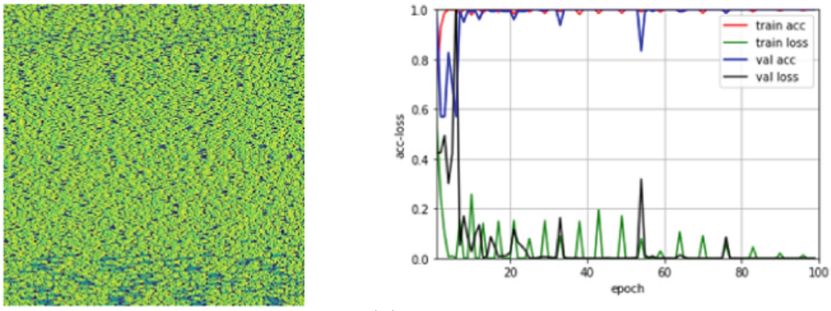
Fig. 9. Android malware classification experiment results by using the classes.dex file with color visualization features.

Color Visualization of Classes.dex File. Figure 9(a) shows the color visualization image of the *classes.dex* file. Since the it contains all the code of the Android App, the visualization image has more details. There are obvious texture in the figure and different colors that represent different binary numbers. The pictures generated by malicious Apps of the same family have certain similarities in texture and color, which is the basis for image characterization as a method of classification of Android malicious Apps.

Accuracy. As shown in Fig. 9(b), the classification accuracy rate can reach 90.91%. The loss value is relatively high and can reach 20%. This is because

the **classes.dex** file includes the code from the third-party libraries, and same third-party libraries code will cause the same characterization results in different apps, which is one of the factors affecting the accuracy of classification.

Efficiency. As shown in Table 2, the average time required to decompile an App file into a *classes.dex* file is 1.16 s. Depending on the size of App files, the maximum decompilation time is 12.93 s and the minimum is 0.10 s. Due to the limitation of image size, instead of visualizing all bytes in large files, we chose to partially visualize up to 89401 bytes. The visualization process takes an average of 1 s per App.



(a) Color visualization of class name collection

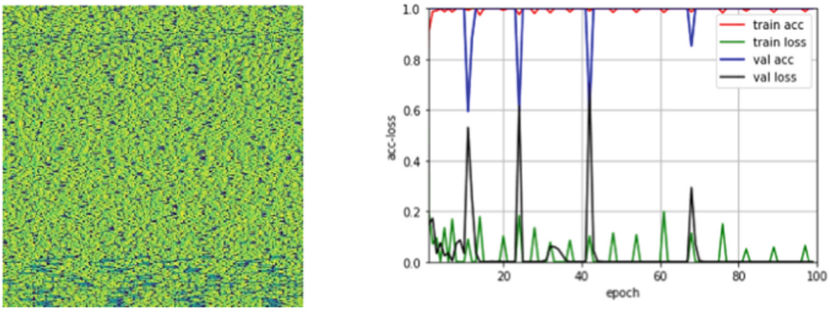
(b) Experimental results of visual classification of class name sets

Fig. 10. The experiment results of Android malware classification by using the set of class name with color visualization features.

Color Visualization of Set of Class Names in Apps. The set of class names in the App is a type of code extracted from the App file, which explain the class invocation of the App. A class name can be used as a description of the App's single behavior, and a collection of invocations can represent behaviors of the entire App in macro. Therefore, it can be used as a feature of the App for color visualization. The image after the feature visualization is shown in Fig. 10(a).

Accuracy. The classification results for the two families *FakeInstaller* and *Plankton* are shown in Fig. 10(b). The results show that the classification accuracy rate reaches 100%. It can be seen that the visualization result using the class name set as input is more conducive to learn from the image which is useful information and which is useless information.

Efficiency. As shown in Table 2, the average time to extract all the App class names from the App is 1.19 s, and the time taken depends on the size of the App. The minimum is less than 1 s and the highest is 5.54 s. The time-consuming aspect of color visualization of features is that the extracted set of class name are relatively small, and the time-consuming is 1 s.



(a) Color visualization image of API call sequence.

(b) Experimental results of the classification of color visualization features by using the sequence of APIs.

Fig. 11. The experiment results of the sequence of APIs.

Color Visualization of API Call Sequence. We use the API call sequence as a visual feature input. APIs can better reflect the internal logical structure of the Android App, which has a positive impact on the improvement of classification accuracy. Due to the need to analyze the internal code structure of the Android App, it takes slightly more time than simply extracting the App class name. The color visualization image of the API call sequence is shown in Fig. 11(a).

Accuracy. The classification results for the two families are shown in Fig. 11(b). The classification result using the API sequence as the input of visualization can reach 100%. The occasional accuracy fluctuations in the figure may be due to the increase in similarity of different Apps to a certain extent due to the third-party libraries.

Efficiency. As shown in Table 2, the average time taken by the App analysis to extract the features of the API call sequence is 1.20 s. The time taken by the API call is determined by the size of the App. The maximum time is 13.16 s and the shortest time is 0.1 s. It also takes 1 s in color visualization.

An Experiment on the Classification of ten Malicious Families. In order to further verify the differences between the two types of features of the class name set and the API call sequence, we conducted multiple classification experiments using these two features. The malicious App families used include *FakeInstaller*, *DroidKungFu*, *Plankton*, *Opfake*, *GinMaster*, *BaseBridge*, *Iconosys*, *Kmin*, *FakeDoc*, *Geinime*, a total of 4005 malicious Apps. The number of samples in the training data set is 80%, and the rest are used in the test data set.

Color Visualization of Set of Class Names in Apps. This experiment mainly verifies the accuracy of classification. The results in the experiment is shown in Fig. 12. After 100 complete iterations, the classification accuracy reaches 91.40%.

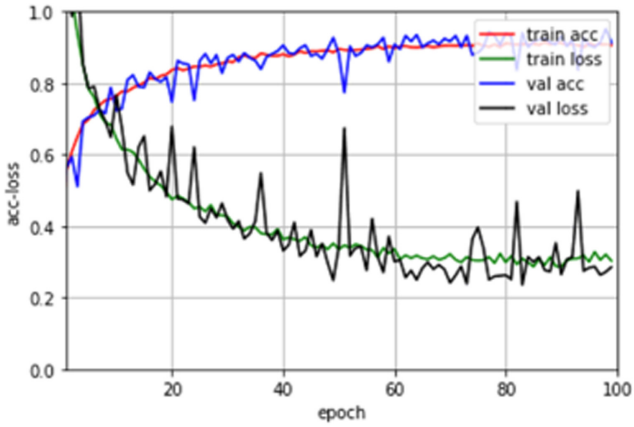


Fig. 12. Experimental results of the classification of ten malicious families visualized by the collection of class name.

There may be more features that overlap in different families, which leads to a decrease in accuracy.

Color Visualization of API Call Sequences. As shown in Fig. 13, we use API call sequence as visualization. The accuracy of family classification is 96.01%. Compared with the results of binary classification, the accuracy is reduced.

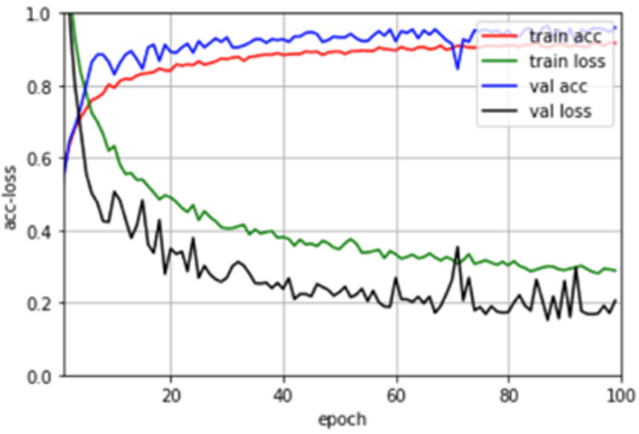


Fig. 13. Experimental results of the classification of ten malicious families visualized by the Apps' API call sequence)

In the above two experiments, the accuracy of each family is shown in Table 3. API call sequence has higher accuracy in most families, except in *FakeDoc* and

FakeInstallers. We compared the malicious behavior of the two families and found that Apps in *FakeDoc* download the malware without the user's consent, Also Apps in *FakeInstallers* send the SMS without the user's permission. The behavior of the two malicious families is similar. This phenomenon leads to low accuracy.

Table 3. Accuracy of classification of ten malicious families.

Family	Class name	API call
DroidKungFu	99.2%	99.2%
PlanKton	91.1%	92.2%
FakeDoc	89.8%	84.5%
Geinimi	72.1%	87.5%
Iconosys	88.1%	95.2%
GinMaster	81.1%	87.2%
BaseBridge	81.1%	98.3%
Kmin	89.5%	92.2%
FakeInstaller	98.2%	94.2%
Opfake	99.1%	99.1%

From the two aspects of classification accuracy and time consumption, the feature of API call sequence has the best effect. It achieves the highest accuracy in the two classifications and ten classes of experiments, and the time spent is relatively short, so that it has become the current optimal choice.

6 Discussions and Limitations

Obfuscation. More and more Apps are used obfuscation. For malicious Apps, they are used obfuscation to hide malicious behaviors. Here are: (1) encoding classes and methods into meaningless strings; (2) adding some useless APIs to Apps; (3) storing malicious APIs in the form of ascii code. We extract APIs belong to the Android system, these APIs cannot be obfuscated, so for the first two obfuscation techniques, our method can get the APIs. For the third obfuscation technique, we cannot get the APIs.

Packer. Some malicious Apps use packing technology to hide malicious code. In our feature extraction, there is no unpacking process for these Apps, so we can not analysis these Apps. But for our current research on the unpacking method, we can already use the method of memory insertion to realize the automatic unpacking process. So in the future researchs, we will implement detection for packed malicious Apps.

7 Conclusion

We present a method for multi-classification of Android malicious App families using color visualization. Experiments in this paper proves that compared to single-channel images, deep learning models can more easily learn features from three-channel images, thereby achieving higher classification accuracy. We use ResNet to implement a multi-classification of 131 malicious families, and find that the best classification results can be achieved when using API calls as features of malicious App color visualization. All in all, in terms of using the deep learning model to classify Android malicious App families, compared to the traditional single-channel gray-scale image for App visualization, the color image visualization method has obvious advantages.

Acknowledgement. This work was supported in part by the National Key R&D Program of China under Grant No. 2018YFB0804703.

References

1. Androguard. https://github.com/androguard/androguard. Accessed 8 Jan 2019
2. Common vulnerabilities and exposures. https://cve.mitre.org/. Accessed 4 June 2018
3. Droidbox. https://github.com/pjlantz/droidbox. Accessed 10 July 2019
4. Fakeinstaller. https://www.mcafee.com/blogs/other-blogs/mcafee-labs/fakeinstaller-leads-the-attack-on-android-phones/. Accessed 18 Aug 2019
5. Plankton. https://news.ncsu.edu/2011/06/wms-android-plankton/. Accessed 18 Aug 2019
6. Andronio, N., Zanero, S., Maggi, F.: Heldroid: dissecting and detecting mobile ransomware. In: Bos, H., Monrose, F., Blanc, G. (eds.) RAID 2015. LNCS, pp. 382–404. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-319-26362-5_18
7. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: effective and explainable detection of Android malware in your pocket. Ndss **14**, 23–26 (2014)
8. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: behavior-based malware detection system for Android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 15–26 (2011)
9. Conti, G., Dean, E., Sinda, M., Sangster, B.: Visual reverse engineering of binary and data files. In: Goodall, J.R., Conti, G., Ma, K.-L. (eds.) VizSec 2008. LNCS, vol. 5210, pp. 1–17. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85933-8_1
10. Deshotels, L., Notani, V., Lakhotia, A.: Droidlegacy: automated familial classification of Android malware. In: Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop, pp. 1–12 (2014)
11. Garcia, J., Hammad, M., Malek, S.: Lightweight, obfuscation-resilient detection and family identification of Android malware. ACM Trans. Softw. Eng. Methodol. (TOSEM) **26**(3), 1–29 (2018)

12. Gennissen, J., Cavallaro, L., Moonsamy, V., Batina, L.: Gamut: sifting through images to detect Android malware (2017)
13. Grace, M., Zhou, Y., Zhang, Q., Zou, S., Jiang, X.: Riskranker: scalable and accurate zero-day Android malware detection. In: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, pp. 281–294 (2012)
14. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
15. Hsiao, S.W., Sun, Y.S., Chen, M.C.: Behavior grouping of android malware family. In: 2016 IEEE International Conference on Communications (ICC), pp. 1–6. IEEE (2016)
16. Jung, J., Choi, J., Cho, S.J., Han, S., Park, M., Hwang, Y.: Android malware detection using convolutional neural networks and data section images. In: Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems, pp. 149–153 (2018)
17. Kancherla, K., Mukkamala, S.: Image visualization based malware detection. In: 2013 IEEE Symposium on Computational Intelligence in Cyber Security (CICS), pp. 40–44 (2013)
18. Lin, C.M., Lin, J.H., Dow, C.R., Wen, C.M.: Benchmark dalvik and native code for Android system. In: 2011 Second International Conference on Innovations in Bio-inspired Computing and Applications, pp. 320–323. IEEE (2011)
19. Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B.: Malware images: visualization and automatic classification. In: Proceedings of the 8th International Symposium on Visualization for Cyber Security, pp. 1–7 (2011)
20. Pektaş, A., Acarman, T.: Learning to detect android malware via opcodesequences. Neurocomputing (2019)
21. Qiu, J., et al.: Data-driven Android malware intelligence: a survey. In: Chen, X., Huang, X., Zhang, J. (eds.) ML4CS 2019. LNCS, vol. 11806, pp. 183–202. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30619-9_14
22. Sebastián, M., Rivera, R., Kotzias, P., Caballero, J.: AVclass: a tool for massive malware labeling. In: Monrose, F., Dacier, M., Blanc, G., Garcia-Alfaro, J. (eds.) RAID 2016. LNCS, vol. 9854, pp. 230–253. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45719-2_11
23. Suarez-Tangil, G., Tapiador, J.E., Peris-Lopez, P., Blasco, J.: Dendroid: a text mining approach to analyzing and classifying code structures in Android malware families. Expert Syst. Appl. **41**(4), 1104–1117 (2014)
24. Wang, W., Zhao, M., Wang, J.: Effective Android malware detection with a hybrid model based on deep autoencoder and convolutional neural network. J. Ambient Intell. Hum. Comput. **10**(8), 3035–3043 (2019)
25. Wu, W.C., Hung, S.H.: Droiddolphin: a dynamic android malware detection framework using big data and machine learning. In: Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems, pp. 247–252 (2014)
26. Xiao, X., Zhang, S., Mercaldo, F., Hu, G., Sangaiah, A.K.: Android malware detection based on system call sequences and LSTM. Multimedia Tools Appl. **78**(4), 3979–3999 (2019)
27. Zhang, J., Qin, Z., Yin, H., Ou, L., Hu, Y.: IRMD: malware variant detection using opcode image recognition. In: 2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS), pp. 1175–1180 (2016)
28. Zhang, K., Zuo, W., Chen, Y., Meng, D., Zhang, L.: Beyond a Gaussian denoiser: residual learning of deep CNN for image denoising. IEEE Trans. Image Process. **26**(7), 3142–3155 (2017)

29. Zhou, Y., Jiang, X.: Android malware genome project. http://www.malgenomeproject.org/. Accessed 4 June 2018
30. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy, pp. 95–109. IEEE (2012)