Android Malware Identification Through Visual Exploration of Disassembly Files

Yong-liang Zhao[1] and Quan Qian[2,1,3]
(Corresponding author: Quan Qian)

School of Computer Engineering & Science, Shanghai University[1]
Shanghai Institute for Advanced Communication and Data Science, Shanghai University[2]
99 Shangda Road, Shanghai, China
(Email: qqian@shu.edu.cn)
Materials Genome Institute of Shanghai University[3]
No. 99, Shangda Road, Shanghai, China
(Received Apr. 28,2017; revised and accepted Aug. 20, 2017)

Abstract

Android malwares are the most serious threats for the current mobile Internet. In this paper, we propose a static analysis approach which does not need to understand the source code of the android applications. The main idea is as most of the malware variants are created using automatic tools. And there are special fingerprint features for each malware family. Depending on decompiling the android APK, we innovatively map the Opcodes, API packages and high level risky API functions into an integrated three channels of a RGB image respectively. And then use convolutional neural network to identify each family's features. The experimental results show that the proposed method successfully identified the entire 14 malware datasets with accuracy 90.67%, precision 93.36%, recall rate 93.95% and F1 93.56% on average.

Keywords: Android Malware; Deep Learning; Malware Identification; Visual Analysis

1 Introduction

With the rapid development of mobile internet, mobile devices, especially smartphones, are not only as important tools for people to communicate with the outside world, but also as personal digital assistants or enterprise digital assistants to plan or organize their users' work and also private life. So, mobile security has become increasingly important in mobile computing. According to the statistical analysis of AliMobileSecurity, although the number of mobile malwares and infections in 2015 showed a certain degree of decline,still 18% devices were infected with different kinds of virus or trojans. Moreover, the professional virus attacks over the year have been significantly upgraded, such as the abuse of system vulnerabilities, security reinforcement technology, especially the social engineering. All of these make the traditional mobile virus interception technology being encountered unprecedented challenges.

Although the number of malwares is increasing every year,most of the variants are modified or generated based on the original malicious code [6]. In most cases, hackers use automated or module reuse tools to generate malwares automatically. These variants often share the same ancestor's work, resulting in the high degree of similarity among variants. So, how to recognize the fingerprints among different variants is the main task we should pay more attention to.

About the mobile operating systems, there are Android, iOS, Symbian OS and Windows phone. Considering the open source and widespread applications, in this paper we mainly focus on Android system. We introduce a new method to classify the android malwares. For each malware sample, we extract the Opcode feature and map it into R channel of RGB image, OS API feature into G channel and risky API feature into B channel, then we merge three channels into one combined feature image and use the deep learning algorithm for further classification. The rest of the paper is organized as follows: Section 2 gives a brief introduction of the related work. Our main contributions, the image based feature integration and deep learning based classification are described in Sections 3 and 4. The experimental results are shown in Section 5. Section 6 summarizes the whole paper and give some directions for future work.

2 Related Work

According to code state for analysis, malware detection methods can be typically divided into two categories: static analysis and dynamic analysis. In static analysis, the codes of the sample are examined comprehensively

by disassembling or decompiling the malware binary files without executing it, which can prevent operating system from malicious damages. The advantages of static analysis are that we can get a complete view of what a given malware does. However, in most cases, static analysis is not a trivial task since hackers use code obfuscation, such as binary packers, encryptions to evade detection. Furthermore, static analysis does not allow a high degree of automation, since in most cases, it is done by hand and sometimes very time-consuming. Considering about the dynamic analysis, it can analyse the behavior of the malware during executing it in a debugger. Currently, sandbox-based dynamic analysis is one of the most popular solutions. A sandbox executes a malware sample in a controlled environment which can monitor and record information of system calls and behaviors dynamically. The main limitations of dynamic analysis, especially the sandbox-based solutions, are the overwhelming detailed reports for human analysts to face. How to make the malwares behavior more easier accessible. How to guarantee the high degree of execution path coverage are two critical factors. Furthermore, in contrast to static analysis, dynamic analysis can be automated to a high degree, though it has high computation complexity.

2.1 Android Malware Detection Using Static Analysis

Static analysis involves extracting information from the application's manifest of the Android application's bytecode. The features often used in static analysis include API calls, requested permissions, used permissions, control flows, data flows, hardware components, application components, intents, network addresses, *etc.*

Sanz *et al.* [15] presented PUMA, a system used the permission application requests upon installation to detect whether the application is malicious or not. Machine learning models including simple logistic regression, Naive Bayes, Bayes Net, SMO, IBK, J48, Random Tree (RT) and Random Forests (RF) are evaluated on a dataset consisting of 357 benign and 249 malicious applications. The best overall accuracy reaches by Random Forests 86%.

Lee *et al.* [17] presented a detection mechanism using runtime semantic signatures, which showed high family classification accuracy. They used three sets of elements to construct the signatures. The first set is binary patterns of malicious API calls instructions, runtime semantics of control and data flow. The second set is the malware family characteristics including family common strings, constants, methods, and classes. The third set is weights that each behavior belongs to a family. Experiments on 1,759 Android malwares including 79 variants of 4 malware families show that the proposed method can obtain 99.89% accuracy on detecting the malware family of a particular variant.

Arp *et al.* [1] proposed DREBIN, which was a similar approach to the method proposed by Peng *et al.* [8]. Eight different static feature sets are extracted including hardware components, requested permissions, application components, filtered intents, restricted API calls, used permission, suspicious API calls, and network addresses. Experiments on evaluation of 123,453 benign applications and 5,560 malware samples, DREBIN can detect 94% of the malware.

Zhang *et al.* [12] implemented DroidSIFT. They extracted a weighted contextual API dependency graph as program semantics to construct feature sets. Graph similarity metrics are introduced to uncover homogeneous application behaviors. Experiments on 2,200 malware samples and 13,500 benign samples are performed using Naive Bayes. The results show that DroidSIFT can detect 93% of malware instances.

Yang *et al.* [2] developed DroidMiner, which used static analysis to automatically learn the malicious program logic from known Android malwares. A two-tiered behavior graph is constructed in DroidMiner. The upper tier is a component dependency graph (CDG) in which each node represents an activity, a service or a broadcast receiver. The lower tier uses component behavior graphs (CBG) to present each component's lifetime behavior functionalities. From the behavior graph, different malicious patterns, named modalities, can be mined. In particular, function modality representing an ordered sequence of API functions, and resource modality representing a set of sensitive resources, are extracted and converted to a modality vector by DroidMiner. The vectors are then fed into several machine learning classifiers including Naive Bayes, SVM, decision trees, and random forests for malware detection. The best algorithm of DroidMiner can achieve a 95.3% detection rate on a dataset of 2,466 malwares. It can also reach 92% for classifying malwares into their proper families.

2.2 Android Malware Detection Using Dynamic Analysis

Dynamic analysis records the execution of an application and tries to identify malicious behavior. It is well known for being resilient to obfuscation techniques. However, dynamic analysis introduces more overhead because it requires running the application first and then deciding whether it is malicious based on run-time behavior. As a consequence, it is mostly applicable for offline malware detections. Besides that, the other deficiency of dynamic analysis is code execution path coverage. Since some malicious behaviors are triggered by special conditions, dynamic analysis will not record an application's malicious behavior if the conditions are not matched.

Ham *et al.* [20], proposed a method that was very similar to CrowDroid. They also aggregated real-time system calls to create a histogram using Linux strace tool. They discovered that some system call patterns only occurred in malicious applications and some only in benign ones. Different from the K-means used in CrowDroid, Ham *et al.*, applied a discrimination algorithm based on Euclidean distance on 1,260 malware samples published

by Genome [19]. But no classification accuracy was reported yet.

Tchakounté *et al.* [18] scrutinized system call invocations initiated by the malicious code at the moment the user runs it using the Linux strace tool. With their tool they discovered new scenarios of how the users are lured to aid the malicious developer.

Wei *et al.* [21] recorded system call invocations by manually installing and executing each application on a real Android phone. N-gram vectors are generated from the system call invocations and fed into a SVM and a naive Bayes for classification. Experiments on 96 benign applications and 92 digital book malware samples show their methods can reach 94% accuracy.

Dimjasevic *et al.* [3] proposed MALINE, which also records system call invocations for Android malware and converts them into two representations. One is histogram and the other is a variant of the Markov Chain representation. Experiments on 4,289 malwares and 12,789 benign applications show that they can achieve 93% detection accuracy.

So, there have been some relative work on Android malwares using static or dynamic methods. But different from the existing work, our contributions are as follows:

1) Proposing a new method that recognize the malware without understanding its source code and execution behavior.

2) Using visualization techniques to transform the Opcodes to family images.

3) Integrate the Opcodes sequence features, API calls features and high risky API calls features to three different RGB channels.

4) Using CNN network to train the malware feature images and get an excellent identification results.

3 Visual Representation for Android Malwares

Nataraj *et al.* [13], proposed a method for visualizing and classifying malwares using image processing techniques, which transform windows platform malware binaries to gray-scale images. In 2015, Little Boat *et al.* won the championship of Kaggle, a famous Microsoft Malware Classification Challenge. It is interesting that the championship team with three people did not engage in security, and the methods used are very different from our common methods. They use gray-images, n-gram and the PE-header features, and use machine to learning classify the malwares. Without understanding the malware's source code, it shows great potential that image based methods for malware detection.

Based on this idea, we propose a method mapping Android applications to RGB images. We extract three features: Opcode, sensitive API calls, and risky API requests. And integrate them into one RGB color image.

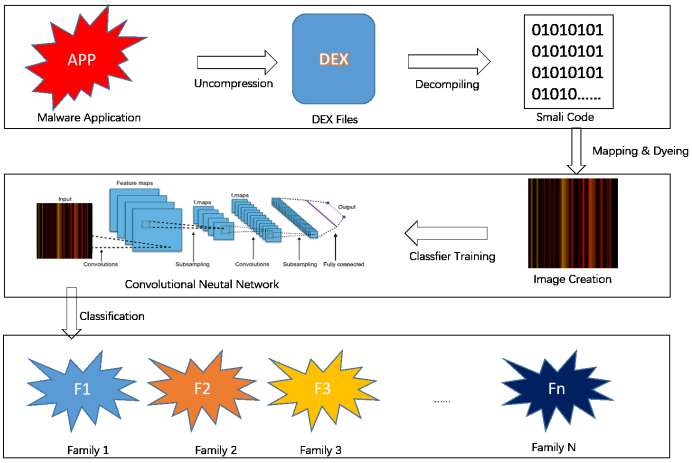


Figure 1: The whole procedure of the malware family identification

3.1 System Architecture

Figure 1 shows the working flow of our method. Firstly, we decompile the application and extract the Opcodes; Secondly, mapping the different Opcodes to pixels in R channel of the RGB image, and then colouring some sensitive API packages in G channel, highlighting the risky API functions in B channel. Thirdly, merging the three R, G, B channels to generate the feature image. Finally, using machine learning to model the features and identify different family fingerprints automatically.

3.2 Android Malware Decompilation

An Android application is commonly written in Java and compiled to Dalvik bytecode which contained in a .dex file. This file can be just-in-time compiled by the Dalvik virtual machine or compiled once into a system-dependent binary by ART on the Android paltform. Table 1 is the structure of the .apk file that the Android applications are packaged in. It contains the Dalvik executable .dex file. The Androidmanifest.xml used to describes the content of the package, including the permissions information. The native code (optional) in form of executables or libraries is usually called from the .dex file. Also, it contains the digital certification for authentication and the resources that the app uses, for instance, image files, sounds, *etc.*

The .dex file is a binary container for the bytecode and the data within the classes. The structure of a .dex file is showed in Table 2. This file is partitioned into a number of sections. After the header and before the data section, it contains the actual code. There are several identifier lists that contain offsets pointing to the corresponding entries in the data section. Due to the data section is the section that contains the actual code, and other sections are just for complimentary descriptions. So, for malwares family identification, we extract the fingerprint features from the data section to reduce the interferences of other sections.

Table 1: The structure of apk file

File or directory	Function
META-INF/MANIFEST.MF	The Manifest file
META-INF/CERT.RSA	The certificate of the application
META-INF/CERT.SF	The list of resources and SHA-1 digest of the corresponding lines in the MANIFEST.MF file
lib/	The directory containing the compiled code that is specific to a software layer of a processor
res/	The directory containing resources not compiled into resources.arsc (see below).
assets/	A directory containing applications assets, which can be retrieved by AssetManager.
AndroidManifest.xml	An additional Android manifest file, describing the name, version, access rights, referenced library files for the application.
classes.dex	The classes compiled in the dex file format understandable by the Dalvik virtual machine.
resources.arsc	A file containing precompiled resources, such as binary XML for example.

Table 2: The structure of dex file

Name	Format	Description
header	header_item	The header
string_ids	string_id_item[]	String identifiers list. These are identifiers for all the strings used by this file, either for internal naming (e.g., type descriptors) or as constant objects referred to by code.
type_ids	type_id_item[]	Type identifiers list. These are identifiers for all types (classes, arrays, or primitive types) referred to by this file, whether defined in the file or not.
proto_ids	proto_id_item[]	Method prototype identifiers list. These are identifiers for all prototypes referred to by this file.
field_ids	field_id_item[]	Field identifiers list. These are identifiers for all fields referred to by this file, whether defined in the file or not.
method_ids	method_id_item[]	Method identifiers list. These are identifiers for all methods referred to by this file, whether defined in the file or not.
class_defs	class_def_item[]	Class definitions list. The classes must be ordered such that a given class's superclass and implemented interfaces appear in the list earlier than the referring class. Furthermore, it is invalid for a definition for the same-named class to appear more than once in the list.
call_site_ids	call_site_id_item[]	Call site identifiers list. These are identifiers for all call sites referred to by this file, whether defined in the file or not.
method_handles	method_handle_item[]	Method handles list. A list of all method handles referred to by this file, whether defined in the file or not.
data	ubyte[]	The Data area, containing all the support data for the tables listed above. Different items have different alignment requirements, and padding bytes are inserted before each item if necessary to achieve proper alignment.
link_data	ubyte[]	Data used in statically linked files. The format of the data in this section is left unspecified by this document. This section is empty in unlinked files, and runtime implementations may use it as they see fit.

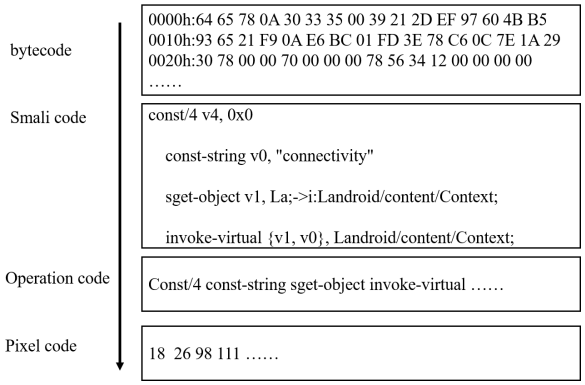


Figure 2: An example of mapping the Opcodes to pixel values in R channel of an image

3.3 Opcode Feature Extraction

Feature extraction is the basis for malware family identification. In this paper, we try to use RBG image to describe the Android malware features. The motivation is, for RGB image, there are three channels and we can use these three channels to represent different malware features integrally and simultaneously. Besides that, once we got the featured images, we can use some image processing techniques, *i.e.*, deep learning, to do feature modelling.

First of all, we should map the Opcode into R channel of RGB image. Android OS has a total of 255 Opcodes, coding from the 0x00 to 0xFF according to different functions. Here, we adopt a method similar to Nataraj *et al.* [13], mapping the Opcode to pixels by converting its hex value (encoded in Android OS) to decimal value. An example is given in Figure 2.

3.4 API Colouring

Application program interface (API) is a set of procedures, protocols, and tools for building software applications. API calls are applied in Android application development in order to implement functionalities conveniently. For example, if we want to get the phone number, we should call: $android.telephony.TelephonyManager \rightarrow getLine1Number$. API calls are also an important clue for malware identification. Wu *et al.*, [4] proposed DroidMat, which detects malware by characterizing applications using the manifest file, API call tracing and it reaches 97.87% classification accuracy. Lee *et al.*, [17] presented a detection method using runtime semantic signatures from malicious API call instructions, control and data flow, the family common string, constants, methods and classes. It reaches the 99.89% accuracy. So, the API calls are very important for malware identification.

In Android system, the API calls are usually given in parameters of a function call instruction as shown in Figure 3.

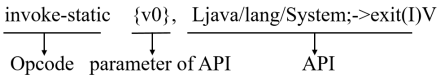


Figure 3: An API call example of a Android instruction

In order to extract the API call features, we divide them into 58 classes by their packages, as shown in Table 3. Among them, 18 classes are related to high level securities needed to focus on as shown in Table 4. They involve user privacy or device hardwares, such as camera, microphone, *etc.*

Table 3: Android APIs classification

Package Name (first level)	Package Name (second level)
android	Account animation app appwidget bluetooth content databse drm gesture graphics hardware inputmethodservice location media mpt net nfc opengl os preference provider renderscript sax security service speech support telephony text util view webkit bytecode system
java	Beans io awt lang math net nio security sql text util
javax	Crypto net security sql xml
junit	Framework runner
org.apache	Http
org	Json w3c xml
dalvik	System

3.5 Highlight the Risky APIs

In previous section, we summed up security related 18 classes from 58 Android OS classes. But the granularity is too coarse. In order to highlight some high level risky behavior, we should refine the granularity to concrete methods or functions. Therefore, 41 high level risky methods from Android OS API are extracted as shown in Table 5. And these features will be shown in B channel of the RGB image.

3.6 RGB Image Creation

In previous section, we extract opcodes, API calls, risky API features, and map them to different pixel values. After that we should integrate all the three features into a integrated RGB image as combined features. For example, if an operation is as follows:

$$invoke\{v0, v1\}, Landroid/content/Context; \rightarrow$$
$$getSystemService(Ljava/lang/String); Ljava/lang/object$$

Table 4: High level security related API classes and pixel values for G channel

#	API Package Name	Description	Pixel value
1	android.account	Account Manager	6
2	android.app	Contains high-level classes encapsulating the overall Android application model	18
3	android.bluetooth	Provides classes that manage Bluetooth functionality, such as scanning for devices, connecting with devices, and managing data transfer between devices	30
4	android.graphics	Provides low level graphics tools such as canvases, color filters, points, and rectangles that let you handle drawing to the screen directly	42
5	android.hardware	Provides support for hardware features, such as the camera and other sensors	54
6	android.media	Used to play and,in some case,record media files	66
7	android.location	Define location-based and related services	78
8	android.nfc	Provides access to Near Field Communication (NFC) functionality, allowing applications to read NDEF message in NFC tags	90
9	android.telephone	Provides APIs for monitoring the basic phone information, such as the network type and connection state, plus utilities for manipulating phone number strings	102
10	android.content	Contains classes for accessing and publishing data on a device	114
11	android.database	Contains classes to explore data returned through a content provider	126
12	android.net	Classes that help with network access, beyond the normal java.net.* APIs	138
13	java.net	Net connection relative	150
14	android.os	Provides basic operating system services, message passing, and inter-process communication on the device	162
15	android.service	Notification relative	174
16	dalvik.system	Dynamic loading relative	200
17	java.lang	Classes loading relative	212
18	others	other package	0

Then, we define the *Opcode* is *invoke*, and the parameters are $Landroid/content/Context;\rightarrow\ getSystemService$. Algorithm 1 shows how to merge the three different sort of features.

Algorithm 1 Map Opcode to RGB image

```
Begin
Accept the data Opcode,parameters.
R_pix ⇐ Call getPixelValue(Opcode)
if parameters is SensitiveAPI/*given in Table 4*/ then
    G_pix ⇐ Call getGPixelValue(parameters)
else
    G_pix ⇐ 0
end if
if parameters is RiskAPI /*given in Table 5*/ then
    B_pix ⇐ Call getBPixelValue(parameters) /*get from Table 5*/
else
    B_pix ← 0
end if
Image ⇐ Call merage(R_pix,G_pix,B_pix)
End
```

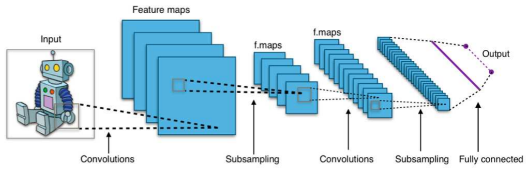
4 Features Modelling and Machine Learning

Traditional machine learning techniques were limited in their ability to process natural data in their raw form, for example, the pixel based image data. Deep learning allows computational models that are composed of multiple processing layers to learn representations of raw data with multiple levels of abstraction [11]. Deep learning techniques, powered by advanced computation ability and large datasets, have shown great performance in strategic games like Go [16], ImageNet competition [14], language translation and speech recognition. A very important paper published recently in Nature [5] also validates that deep constitutional neural network exhibits very high melanoma classification ability. In this paper, the author utilized a GoogleNet Inception v3 CNN architecture with well trained weights on 1.28 million ImageNet data. The final layer is removed and finely tuned to the author categorized dataset containing more than 13,000 images which was collected from a combination of open-access dermatology repositories. The experimental results show it has exceeded the common well trained dermatologists (72.1% vs 66.0%)

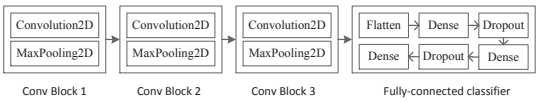
In this paper, we adopt Convolutional Neural Network (CNN) to identify the previous Android malware images. For CNN, Fukushima [7] proposed a calculation model for *CNN* in 1980 firstly based on local connections between neurons and hierarchical transformation. Based on this model, LeCun *et al.* [9] proposed the first real *CNN* multilayer network structure learning algorithm and used it for handwritting digit recognition. The model can automatically extract local features of image with strong adaptability. Parameters sharing makes it more similar to the

Table 5: High level risky API and its pixel values for B channel

API class	Method or function	Description	Pixel value
java.lang.Runtime	exec	execute script	220
android.content.Intent	startActivity	mail	210
android.app.PendingIntent	send	delayed trigger	200
android.app.AlarmManager	Set	delayed trigger	200
android.content.pm.PacakageManager	removePackageFromPrefe	uninstall application	190
android.database.sqlote.SQLiteDatabase	execSQL	database related	180
android.content.ContentResolver	delete	delete data	170
android.app.AcitivityManager	killBackgroudProcess	kill process	160
android.media.MediaRecorder	MediaRecorder	sound record	150
java.net.HttpURLConnection	connect	internet connection	140
java.net.URLConnection	connect	internet connection	140
org.apache.http.impl.client	DefaultHttpClient	internet connection	140
android.content.BroadcastReceiver	abortBroadcast	intercept SMS	63
android.telephony.PhoneStateListener	onCallStateChanged	monitor phone status	130
android.content.Intent	getAction	monitor broadcast	120
javax.crypto.Cipher	getInstance	encryption/decryption	110
javax.crypto.Cipher	Init	encryption/decryption	110
javax.crypto.Cipher	doFinal	encryption/decryption	110
android.telephony.TelephonyManager	getLine1Number	get phone number	100
android.content.pm.PacakageManager	getInstallerPackageName	get application information	90
android.content.pm.PacakageManager	getInstalledPackages	get application information	90
android.content.pm.PacakageManager	getInstalledApplications	get application information	90
android.location.LocationManager	getLastKnownLocation	get location information	80
android.telephony.TelephonyManager	getCellLocation	get location information	80
android.telephony.TelephonyManager	getSubscriberId	get IMSI	71
android.telephony.TelephonyManager	getDeviceId	get IMEI	70
android.telephony.SmsManager	sendTextMessage	send SMS	64
android.telephony.gsm.SmsManager	sendMultipartTextMessage	send multipart SMS	62
android.telephony.SmsManager	sendMultipartTextMessage	send multipart SMS	62
android.telephony.gsm.SmsManager	sendDataMessage	send multimedia message	61
android.telephony.gsm.SmsManager	sendTextMessage	send multimedia message	61
android.telephony.SmsManager	sendDataMessage	send multimedia message	61
android.telephony.gsm.SmsManager	getDisplayOriginatingAddress	read SMS	60
android.telephony.gsm.SmsManager	getDisplayMessageBody	read SMS	60
dalvik.system.DexClassLoader	loadClass	dynamic loading	50
dalvik.system.PathClassLoader	loadClass	dynamic loading	50
android.content.ContentResolver	update	tamper	40
android.content.ContentResolver	insert	insert	30
android.ContentResolver	query	traverse	20
android.content.Intent	setDataAndType	install	10



(a) Structure of the LeNET network



(b) Structure of our used CNN network

Figure 4: Structure of the LeNET network and the used CNN network

biological neural network and reduces the complexity of the network model.

The CNN structure (Figure 4(b)) used in this paper is modified from Lenet-5 [10] as shown in Figure 4(a). Our architecture consists of three sets of convolutions, activation, and pooling layers, followed by a fully-connected layer, activation, another fully-connected, and finally a softmax classifier. The convolution layer will learn 20 convolution filters, where each filter is of size 5x5. The input dimensions of this value are the same width, height, and depth as our input images. If each input feature image is shown as x_i, learn-able weight value w_{ij}, then the output feature image is:

$$y_j = b_j + \sum_i w_{ij} * x_i \tag{1}$$

Where "*" is a convolution operator, and b is a learn-able bias parameter.

The purpose of convolution layer is to extract different features from the input layer. The first layer can only extract convolution low-level features such as edges, lines, angles, *etc.*, more layers of the network can extract more complex features from low-level features in iteration.

The output feature image adopts an activation function $R = h(y)$ for non-linear mapping. The original LeNet architecture used $Tanh$ activation functions rather than $ReLU$. But in this paper we use $ReLU$. The reason is that $ReLU$ tends to give much better classification accuracy. The comparison results will be discussed in Section V. The $ReLU$ is defined as:

$$f(y) = \max(0, y) \tag{2}$$

and the $Tanh$ is defined as:

$$R = \frac{e^y - e^{-y}}{e^y + e^{-y}} \tag{3}$$

Activation function can enhance the non-linear characteristic of the decision function and the whole neural network, but does not change the Convolution layer itself.

The $ReLU$ activation function followed by 2x2 max-pooling in both x and y directions with a stride of 2 to reduce training parameters. It divides the input image into several rectangular regions, and outputs the maximum value for each subregion. The max-pooling layer will constantly reduce the size of the data space, so the number of parameters and the amount of computation will drop. Also, it can control the overfitting during training to a certain extent. Typically, the CNN convolutional layer is periodically inserted into the pool layer.

After the third subsampling layer (S2), we flatten the output feature image to vector and link it to three fully connected layers whose dimensions are 512, 256, 13 (number of malware categories) respectively. And after the first two fully connected layers, there is a dropout layer whose probability is 0.5 to avoid overfitting. The first two fully connected layers adopt $ReLU$ as activation function and the last one (Loss Layer) uses $softmax$, which is defined as:

$$\sigma(Z)_j = \frac{e^{Z_j}}{\Sigma_{k=1}^{K}} for\ j = 1, ..., K. \tag{4}$$

The $softmax$ is used to determine how the training process "punishes" the difference between the predicted results and the actual results of the network.

5 Experiments and Analysis

The experimental dataset is downloaded from the Drebin Dataset of Techinische Universität Braunschweig [1]. The dataset contains 5,560 applications and we choose 14 representative families to do our experiments. The malware samples distribution of the experimental dataset is shown in Table 6. The experimental programs are written in Python, and the hardware environment is Intel Core i7-3370 and 12GB main memory.

Table 6: The experimental android malware samples

#	Malware family	Number of samples
1	FakeInstaller	925
2	DroidKungFu	666
3	Plankton	625
4	Opfake	612
5	BaseBridge	327
6	Iconosys	152
7	Kmin	147
8	FakeDoc	131
9	DroidDream	81
10	MobileTx	69
11	FakeRun	61
12	SendPay	59
13	Gappusin	58
14	Imlog	43

5.1 The Fingerprint Image of Malware Family

In order to train the model by CNN preferably, different malware feature images should zoom into the same size, here 64x64. As shown in Figure 5, images generated by malware variants from the same family have some specific similar textures in some area. Figure 6 shows an example of the difference between Opcode features and the features after combined three channels. It shows that the combined features can describe more details of each malware family.

Based on this feature image, the identification results of our method are shown in Table 7. To evaluate the results scientifically, we use the Accuracy, TPR (true positive rate, also call Recall), FPR (false positive rate), Precision, F1, and receiver operating characteristic curve (ROC). All the metrics have the following definitions:

$$
\begin{aligned}
Accuracy &= \frac{TP+TN}{P+N}, \\
TPR &= \frac{TP}{TP+FN} \\
FPR &= \frac{FP}{FP+TN}, \\
Precision &= \frac{TP}{TP+FP} \\
F1 &= \frac{2 \times Precision \times TPR}{Precision + TPR}
\end{aligned}
$$

Where, TP is the number of true positive predictions, FP is the number of false positive predictions, and FN is the number of false negative predictions. And the F1 score is as a weighted average of the precision and TPR. The F-measure or balanced F-score (F1 score) is the harmonic mean of precision and TPR. From the experimental results, we can see that our algorithm performs well at Opfake family at 96.91%. Since there are only 58 samples in *Gappusin* family, and the identification result is not as good as other families.

5.2 The Identification Accuracy of Different Feature Representations

Figure 7(a) shows a comparison of the identification accuracy between the Opcode feature and the combined features (Opcde feature, API feature and risky API feature). It shows that the combined features perform better than just Opcode feature on malware family identification. That is to say, the API call and the risky API functions can significantly improve the texture features on different families to some degree. Although for different families, the improvements are not the same, but the effect is positive.

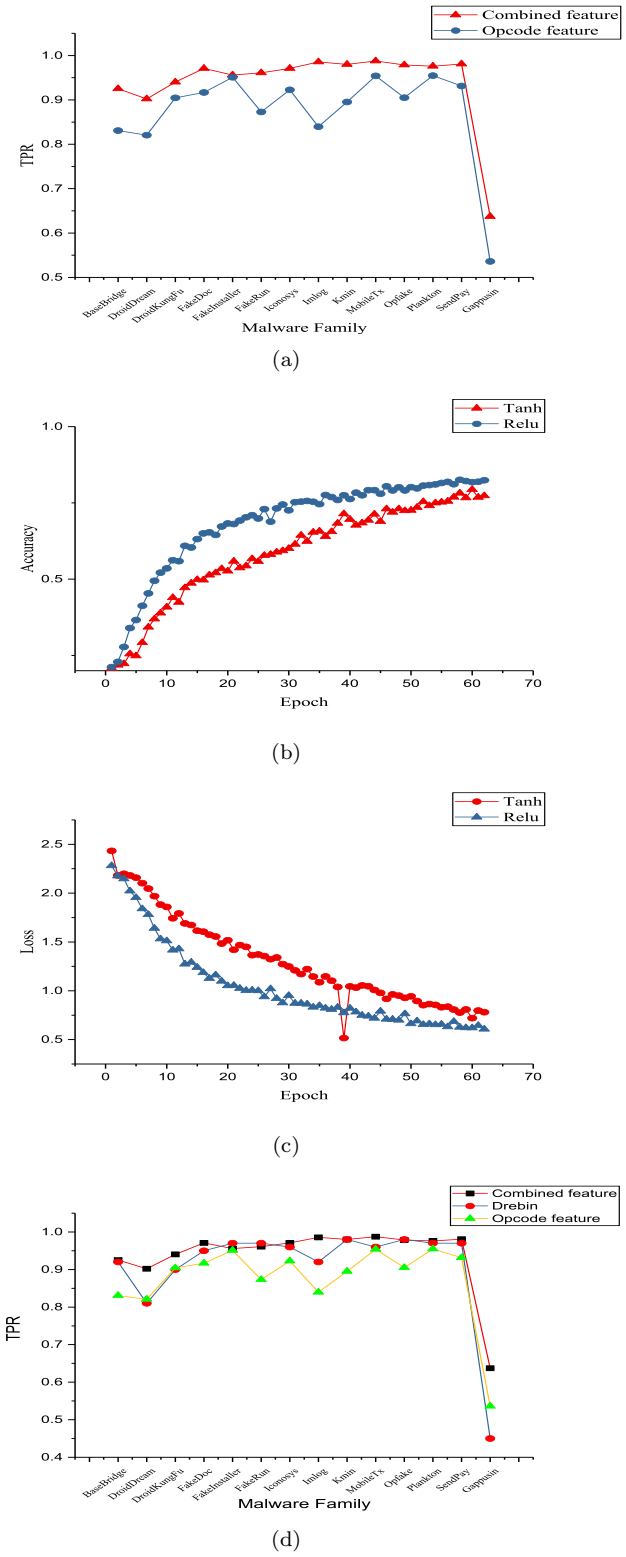
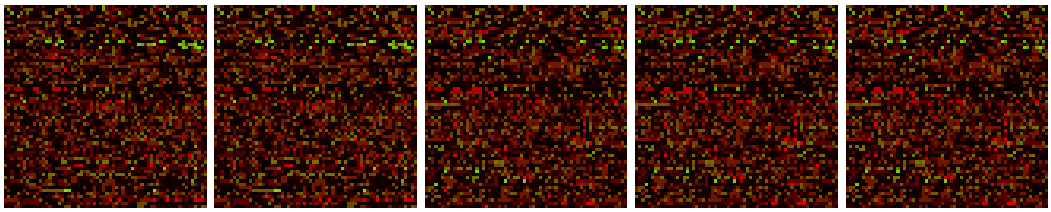


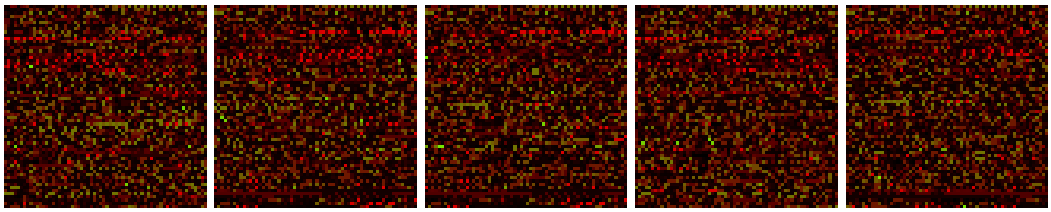
Figure 7: (a) Comparison of the TPR between the combined features and Opcode feature (only R_channel); (b) Comparison of different activation functions for identification accuracy; (c) Comparison of different activation functions for loss; (d) Comparison of TPR between Combined feature, Opcode feature and Drebin's method

Table 7: Experimental results with different metrics

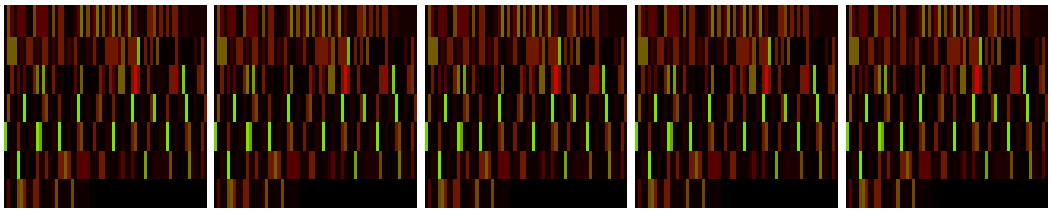
Malware Family	Accuracy	TPR	FPR	Precision	F1
BaseBridge	0.9338	0.9253	0.0020	0.9807	0.9520
DroidDream	0.8966	0.9021	0.0020	0.9492	0.9250
DroidKungFu	0.8855	0.9402	0.0460	0.9015	0.9200
FakeDoc	0.9362	0.9710	0.0010	0.9419	0.9560
FakeInstaller	0.9629	0.9560	0.0070	0.9911	0.9730
FakeRun	0.8977	0.9610	0	0.9180	0.9390
Iconosys	0.9495	0.9710	0.0500	0.9617	0.9660
Imlog	0.8710	0.9857	0.0	0.8604	0.9190
Kmin	0.9619	0.9800	0.0010	0.9727	0.9760
MobileTx	0.9293	0.9876	0.0	0.9192	0.9520
Opfake	0.9691	0.9787	0.0070	0.9795	0.9790
Plankton	0.9698	0.9760	0.0060	0.9821	0.9790
SendPay	0.9647	0.9809	0.0010	0.9880	0.9840
Gappusin	0.5663	0.6373	0.0	0.7245	0.6780
Average	0.9067	0.9395	0.0090	0.9336	0.9356



(a) BaseBridge family

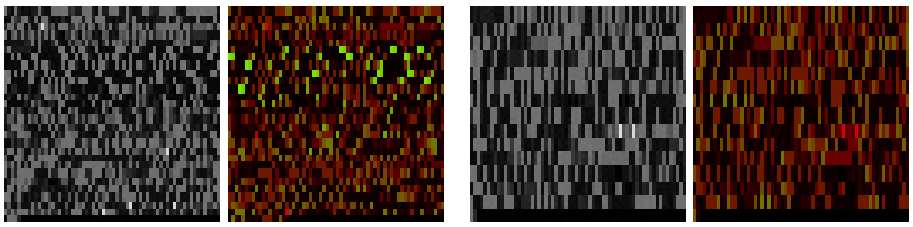


(b) DroidDream family



(c) Iconosys family

Figure 5: The fingerprint feature images of different malware families



(a) Iconosys family (b) FakeInstaller family

Figure 6: Examples of the difference between Opcode features and combined features of three channels

5.3 Comparison of Different Activation Functions on Accuracy

In neural networks, the *neuron* is a computational unit that takes as input x_1, x_2, x_3 , and outputs $h_{W,b}(x) = f(W^T x) = f(\sum_{i=1}^{3} W_i x_i + b)$, where $f : \Re \mapsto \Re$ is called the activation function. The activation function affects the training speed and final result of the whole model. It is very important for the generation of the whole model.

Tanh and *Relu* are the two typical activation functions, we use both of them in our experiment, and the effect of each function are compared in Figure 7(b) and Figure 7(c). From Figure 7(b), it shows that the *Relu* can get a better accuracy than that of *Tanh*, and the convergence rate of *Relu* is faster than that of *Tanh*. Figure 7(c) shows the effects of different activation functions on misclassification rate (loss) on the testing dataset, we can see that *Relu* is better on over-fitting than that of *Tanh*.

5.4 Comparison With the Drebin Method

Figure 7(d) shows the true positive rate (TPR) comparison among combined feature, Opcode (on R_channel) feature and Drebin's method. We can see that although Opcode feature shows the similar performance with Drebin in some family, while for the overall performance, the Opcode feature is still better than Drebin's. Furthermore, when considering the combined features, the overall detection rate has improved a lot. We can get that in most cases, the combined features are better than that of Drebin and Opcode only.

For *ROC* curve, in Figure 8, we can get similar conclusions that for different malware families, using combined features, the *ROC* curve is closer to the upper left corner of the coordinates. That is to say that using combined feature we can get better results in general.

6 Conclusions and Future Work

In this paper we propose a malware identification algorithm which combines malware visualization method and machine learning techniques. First of all, we extract the Opcode features, API calls features and high risky API function features. Then we adopt convolutional neural network to train the fingerprint images and identify the malware families. The experimental results show that the classification accuracy can be 96.91% at best and the average accuracy is higher than DREBIN [1] on the same malware dataset. Besides that, the experimental results show once again that Android malware variants in the same family have some common textures in feature image.

About the future work, we can consider the following directions:

1) Using parallelization techniques to accelerate classification and detection speed.

2) Study further to detect effectively when malwares using packing, encryption, anti-debugging, anti-dissembling techniques.

3) Integrate the proposed static method with dynamic analysis to extend the robustness and adaptability of the detection system.

Acknowledgments

This work is partially sponsored by National Key Research and Development Program of China (2016YFB0700504), Shanghai Municipal Science and Technology Commission (15DZ2260301), Natural Science Foundation of Shanghai (16ZR1411200). The authors gratefully appreciate the anonymous reviewers for their valuable comments.

References

[1] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in *Network and Distributed System Security Symposium*, pp. 1–12, 2014.

[2] Y. Chao, X. Zhaoyan, G. Guofei, V. Yegneswaran, and P. Porras, "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications," in *European Symposium on Research in Computer Security*, pp. 163–182, 2014.

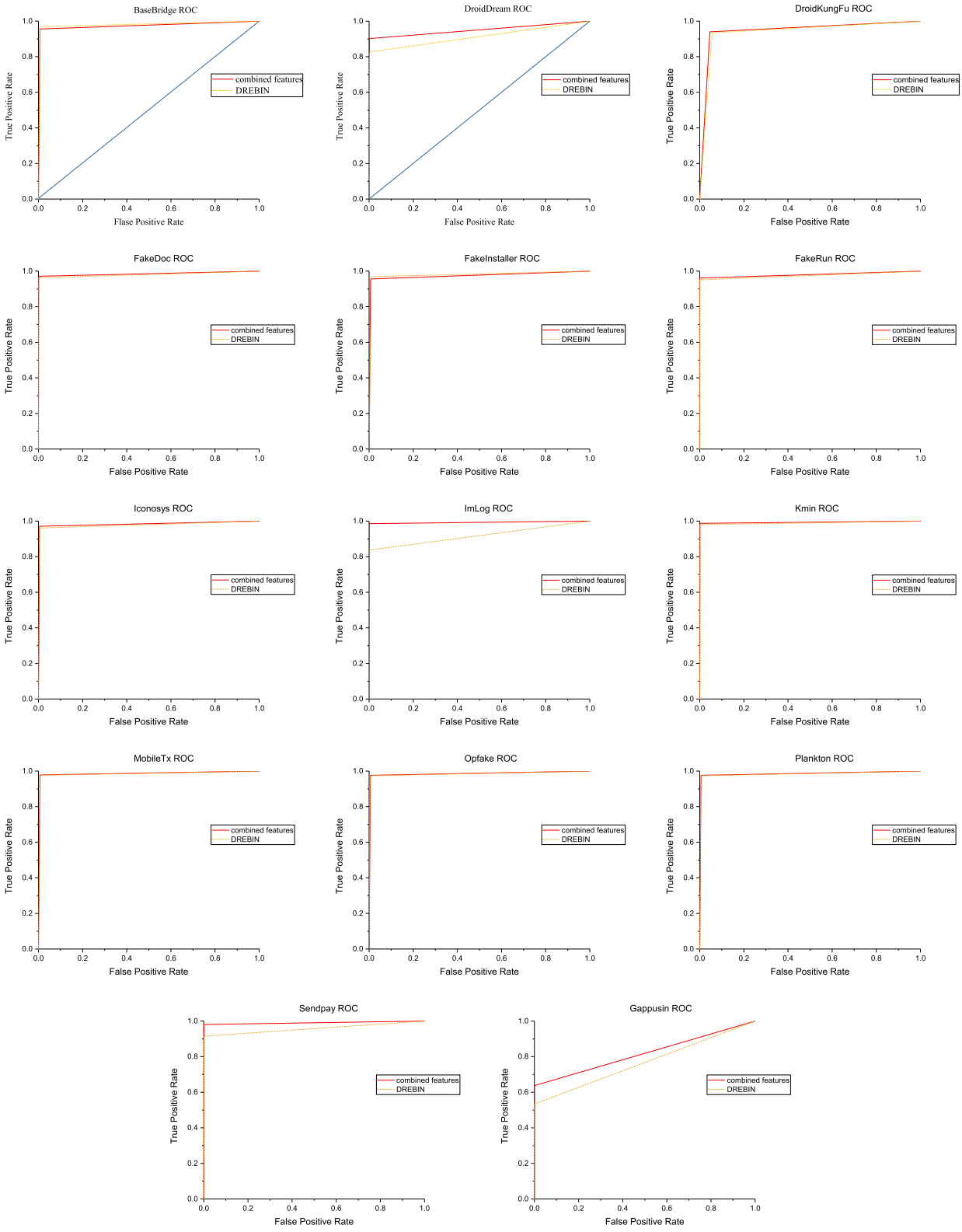


Figure 8: ROC curve of different malware families with different algorithms

[3] M. Dimjaševic, S. Atzeni, L. Ugrina, and Z.Rakamaric, "Android malware detection based on system calls," *University of Utah, Technical Report*, UUCS-15-003, 2015.

[4] W. Dong-Jie, M. Ching-Hao, W. Te-En, L. Hahn-Ming, and W. Kuo-Ping, "Droidmat: Android malware detection through manifest and API calls tracing," in *Proceeding Seventh Asia Joint Conference Information Security*, pp. 62–69, Aug. 2012.

[5] A. Esteva, B. Kuprel, R. Novoa, J. Ko, S. Swetter, H. Blau, and S. Thrun, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, no. 7639, pp. 115–118, 2017.

[6] M. Fossi, D. Turner, E. Johnson, T. Mack, T. Adams, J. Blackbird, S. Entwisle, B. Graveland, D. Mckinney, and J. Mulcahy, "2010 symantec internet security threat report," *Volume*, no. 5, pp. 277–278, 2011.

[7] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

[8] P. Hao, C. Gates, B. Sarma, L. Ninghui, Q. Yuan, R. Potharaju, C. Nita-Rotaru, and L. Molloy, "Using probabilistic generative models for ranking risks of android apps," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pp. 241–252, 2012.

[9] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[10] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[11] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[12] Z. Mu, D. Yue, Y. Heng, and Z. Zhiruo, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1105–1116, 2014.

[13] 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*, pp. 4, 2011.

[14] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[15] B. Sanz, L. Santos, C. Laorden, X. Ugarte-Pedreror, P. Bringas, and G. Álvarez, "Puma: Permission usage to detect malware in android," in *International Joint Conference CISIS´ 12-ICEUTE´ 12-SOCO´ 12 Special Sessions*, pp. 289–298, 2013.

[16] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. V. D. Driessche, J. Schrittwieser, L. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[17] L. Suyeon, L. Jehyun, and L. Heejo, "Screening smartphone applications using behavioral signatures," in *IFIP International Information Security Conference*, pp. 14–27, 2013.

[18] F. Tchakounté and P. Dayang, "System calls analysis of malwares on android," *International Journal of Science and Technology*, vol. 2, no. 9, pp. 669–674, 2013.

[19] Z. Yajin and J. Xuxian, "Dissecting android malware: Characterization and evolution," in *2012 IEEE Symposium on Security and Privacy (SP'12)*, pp. 95–109, 2012.

[20] H. You-Joung and L. Hyung-Woo, "Detection of malicious android mobile applications based on aggregated system call events," *International Journal of Computer and Communication Engineering*, vol. 3, no. 2, p. 149, 2014.

[21] W. Yu, Z. Hanlin, G. Linqiang, and R. Hardy, "On behavior-based detection of malware on android platform," in *Global Communications Conference (GLOBECOM'13)*, pp. 814–819, 2013.

Biography

Yong-liang Zhao is a master degree student in the school of computer science, Shanghai University. His research interests include cloud computing, big data analysis, computer and network security especially in android platform.

Quan Qian is a full Professor in Shanghai University, China. His main research interests concerns computer network and network security, especially in cloud computing, big data analysis and wide scale distributed network environments. He received his computer science Ph.D. degree from University of Science and Technology of China (USTC) in 2003 and conducted postdoc research in USTC from 2003 to 2005. After that, he joined Shanghai University and now he is the lab director of network and information security.