Accepted Manuscript

SaaS: A situational awareness and analysis system for massive android malware detection

Yaocheng Zhang, Wei Ren, Tianqing Zhu, Yi Ren

PII: S0167-739X(18)30942-7
DOI: https://doi.org/10.1016/j.future.2018.12.028
Reference: FUTURE 4649

To appear in: *Future Generation Computer Systems*

Received date : 20 April 2018
Revised date : 22 August 2018
Accepted date : 12 December 2018

SaaS: A Situational Awareness and Analysis System for Massive Android Malware Detection

Yaocheng Zhang[a], Wei Ren[a,b,d,*], Tianqing Zhu[c], Yi Ren[e]

[a] *School of Computer Science, China University of Geosciences, Wuhan, P.R. China*
[b] *Hubei Key Laboratory of Intelligent Geo-Information Processing China University of Geosciences (Wuhan), Wuhan, P.R. China*
[c] *School of Software, University of Technology Sydney, Ultimo, NSW 2007, Australia*
[d] *Guizhou Provincial Key Laboratory of Public Big Data GuiZhou University, Guizhou, P.R. China*
[e] *School of Computing Science, University of East Anglia, Norwich, UK*

Abstract

A large amount of mobile applications (Apps) are uploaded, distributed and updated in various Android markets, e.g., Google Play and Huawei AppGallery every day. One of the ongoing challenges is to detect malicious Apps (also known as malware) among those massive newcomers accurately and efficiently in the daily security management of Android App markets. Customers rely on those detection results in the selection of Apps upon downloading, and undetected malware may result in great damages. In this paper, we propose a cloud-based malware detection system called SaaS by leveraging and marrying multiple approaches from diverse domains such as natural language processing (n-gram), image processing (GLCM), cryptography (fuzzy hash), machine learning (random forest) and complex networks. We firstly extract n-gram features and GLCM features from an App's smali code and DEX file, respectively. We next feed those features into training data set, to create a machine learning detect model. The model is further enhanced by fuzzy hash to detect whether inspected App is repackaged or not. Extensive experiments (involving [illegible]495 samples) demonstrates that the detecting accuracy is more than 98.5%, and support a large-scale detecting and monitoring. Besides, our proposed system can be deployed as a service in clouds and customers can access cloud services on demand.

*weirencs@cug.edu.cn

Keywords: N-GRAM; Machine Learning; Fuzzy Hash; GLCM; Cloud

1. Introduction

In recent years, smart phones have become increasingly popular. In Android market, a large number of Apps are uploaded or updated by hundreds or thousands of individual developers for App distribution everyday. A recent report shows that the number of Apps in Google play has increased nearly 30% since 2017 [1]. While various Apps bring convenience and entertainment to our daily life, Apps with malevolent intentions (e.g. malicious deductions) also inflict troubles and risks to customers. Indeed, the growing amount of malware has become an urgent problem. According to a report released by the QIHU 360 security center, the number of malware samples in the Android platform had surged to nearly 18.74 million by December 2015, which was 27.9 times and 5.7 times higher than that in 2013 and 2014, respectively [2]. The report also points out that over 370 million devices were infected. The above results give us an intuitive emergence on the severity of malware rampant on the Android platform.

Recently, detecting Android malware has been intensively studied [3, 4, 5, 6, 7, 8, 9], which are divided into two major categories: dynamic analysis [8, 9] and static analysis [3, 4, 5, 6, 7]. The former refers to obtain dynamic behavior features of Apps when executing Apps in real devices over sandbox environment. However, it usually time costly to find malicious behaviors, and may lose some harmful behaviors in a limited scope. Thus, dynamic analysis is not not suitable for detecting malware among massive Apps. In other words, the desired solution should be able to detect malware automatically, efficiently and accurately. In contrast, static analysis affords higher efficiency, fast processing, and full code coverage without relying on the compiler or execution environment, thus it is more scalable for massive malware detection. Nevertheless, static analysis may not be able to detect harmful dynamic behaviors, and possibly results in relatively lower accuracy when extracted features are not sufficient.

To tackle these limitations, we propose to obtain dynamic behavior features by using some methods that can be conducted automatically and scalably, e.g., n-gram sequences, GLCM features, and so on, to extract sufficient features in detection to improve the accuracy.

Our design goal is to build a malware detection system for processing massive Apps, with high processing throughput and high accuracy. This paper makes following contributions: We propose a machine learning based Android malware detection system. The system can automatically crawl new samples from App markets, which guarantees the training set is fresh and realistic. Even the most recently upcoming malware can thus be detected. To further improve detection accuracy, we employ comprehensive methods including n-gram, fuzzy hash, GLCM (Gray-level Co-occurrence Matrix) and complex networks. Furthermore, we conduct extensive experiments to evaluate system performance. The experimental results show that the system can achieve 98.5% detection accuracy. For repackaged Apps, our system achieves 96% detection accuracy.

The rest of the paper is organized as follows. Previous works are reviewed in Section 2. We present the pre-processing methods in Section 3 and propose the scheme design in Section 4. Evaluation is conducted in Section 5, and the paper is concluded in Section 6.

2. Related Work

Android App malware detection methods fall into two categories: dynamic analysis [8, 9], and static analysis [3, 4, 5, 6, 7]. As intensive computation resources are required, some detection systems are deployed in clouds [10, 11, 12, 13, 14, 15], in which both static analysis and dynamic analysis methods are used.

The basic idea of dynamic analysis methods [8, 9] is to obtain runtime features of Apps and to rely those features in detection. M. Apel et al.[8] proposed a dynamic analysis scheme to optimize distance measurement for grouping malware samples. Their scheme can gain satisfactory results, but its long analyzing time (over 2 minutes) may not be acceptable for a large scale malware analysis. L. Zeng et al. [9] proposed to encode a matrix with a low rank into a watermark graph and to embed the graph statements into smali code.

Static analysis methods [3, 4, 5, 6, 7] leverage specific information from inspected App, such as information from AndroiManifest.xml file or some special API calls. The syntactic approach can be used for detecting malware. M. Karim et al. [3] investigated the frequency of n-gram from the Opcode of instruction in the binary code, which can distinguish standard vector based distance. The n-perm are utilized as features to differentiate two malware samples, which, however, is unavailable due to the existence of many morphing techniques beyond instruction permutation. Similarly, based on Kolmogorov Complexity of malware, S. Wehner [4] leverages normalized compression distance (NCD) to assess the similarity of malware samples, where the complexity is approximated by the compressibility of malware samples. Nevertheless, such clustering approach is vulnerable to the morphing techniques due to its syntactic nature.

Apart from the syntactic-based approach, P. Faruki et al. [5] proposed a malware detection system based on improbable feature signature database of known malicious Apps. Regardless of their given positive results, their scheme may not be preferable for the large scale data analysis, and may not be able to find out newest malware. The malware detection system proposed by Y. Zhang et al. is based on the vetting permission in Apps [6], and their scheme could effectively examine the internal sensitive behaviors of Apps by monitoring permission behaviors. K. Rieck et al. developed an automatic classification system for malware samples, where classifier labels samples by using anti-virus products [7]. In the scheme, samples unknown to the anti-virus products are classified as unknown. On the other hand, it also renders their scheme to be applied for categorizing malwares. V. Kelesj et al. proposed a method for authorship attribution based on character-level "n-gram" author profiles [16]. Their method is based on byte-level "n-gram" and thus the generated author profiles are subjected to size limitation. The internal connection are lost in their scheme and thus it may result in failing to detect malwares. The study proposed by Patodkar Vaibhavi et al. uses information from Twitter as a corpus for sentiment analysis [17]. The "n-gram" is also used to analyze the messages together with some classifiers to sort out the message type.

S. Yerima et al [12] employed Bayesian classification to characterize App's type with 58 features. The training set included 1000 malware samples from 49 families and 1000 benign samples. They further improved their work by using static method with ensemble machine learning [13]. They extracted 179 features from APKs which include API calls, commands, and permissions. They tested 6863 applications (2925 malware and 3938 benign samples) with multiple methods such as naive Bayes, simple logistic, and random tree. The experiment results showed a detection rate up to 97-99%.

F. Marudin et al. used public dataset and private dataset to evaluate malware detection with machine learning classifier [14]. Based on the evaluation results, Bayes network and random forest classifier both have more accuracy readings with a 99.97% true-positive rate, and multi-layer perception with only 93.03% on MalGenome dataset. Besides this, they found that k-nearest neighbor classifier efficiently detected the latest Android malware with 84.57% true positive rate, which is higher than other classifiers.

Overall, above schemes [8, 9, 3, 4, 5, 6, 7] suffer from some problems in processing massive Apps with high accuracy, so in this paper, we use comprehensive static analysis methods together with machine learning to detecting malware. Besides, we deploy the system in clouds to accelerate the speed of processing massive data.

3. Preliminaries

In our system, inspected App is pre-processed by three algorithms in the preparation stage:

- Fuzzy hash algorithm: We use fuzzy hash algorithm to distinguish whether the evaluated App is repackaged.
- N-gram: We extract App's n-gram features from App's smali code and feed features to train models to detect App's characteristics.
- GLCM: We extract App's GLCM-based features from the graph created from App's Dex file as model to detect App's characteristics.

3.1. Fuzzy Hash Algorithm

Fuzzy hash algorithm, also known as Context Triggered Piecewise Hashing (CTPH), firstly are used as a weak hash algorithm to calculate content, and the hash value of each piece is calculated by a strong hash algorithm again. Afterwards, the pieces of hash values are combined together to form a fuzzy hash string. The similarity comparison algorithms can be used to assess the similarity of two objects, i.e., documents, by comparing the fuzzy hash values. We employ it to evaluate the similarity of files (e.g. the differences among files with content addition or content deletion). In our system, we compare the data extracted from related Apps to evaluate their similarity for determining whether those Apps are repackaged.

3.2. Smali and N-gram

Smali is a tool for studying bytecodes in Dalvik Virtual Machine (DVM). Note that although Smali language is not an official standard, almost all statements in Apps follow this syntax specifications. As there are over 200 types of instructions in Dalvik Opcode, we need to classify and streamline the instructions. Thus, we remove the non-essential instructions. There are only 7 core instructions (i.e., M, R, G, I, T, P, V) left and they represent the operations of "move", "return", "goto", "if", "get data", "put data" and "invoke", respectively.

Table 1: Different n-gram features from an assembly file in Smali format

Smali Format	Instruction Classify and Describe
iput-object p1,p0...	P(input-object)
Invoke-direct {p0}...	V(invoke-direct)
Return-void....	R(return-void)
iget-object V0,P0....	T(iget-object)
Invoke-static{V0}...	V(invoke-static)
Return-void...	R(return-void)

Opcode 1-gram	Opcode 2-gram	Opcode 3-gram	Opcode 4-gram
P	PV	PVR	PVRT
V	VR	VRT	VRTV
R	RT	RTV	RTVR
T	TV	TVR	
V	VR		
R			

N-gram is used in natural language processing and it assumes that the probability of a word showing only relies on its previous $n-1$ words. This probability can be obtained by a sufficient amount of sentences in a corpus. For example, the word of "apple" or "pizza" is more likely to appear after "eating" than the word of "road". We could perceive that n-gram remains some linguistic features. Therefore, n-gram can be exploited for analyzing malicious code [18], whose method was based on the bytecodes. But, it is supposed that Opcode-based method was better than bytecode-based method [19]. We incorporate the Opcode-based method in our scheme. The Opcode n-gram can be extracted from instruction Opcode and n can be assigned as 2, 3 or 4. Tab. 1 gives an example of Opcode n-gram from an assembly file.

In the system, we extract features from DVM Opcode to constitute a training set, relying which machine learning is conducted to create a detection model for a large scale malware detection.

3.3. Gray-scale image and GLCM

For a binary file, each byte is ranged from 00~FF, it corresponds to gray values from 0 to 255 (0 represents black pixel and 255 denotes white pixel).

We can convert a binary file into a matrix, whose elements are corresponded with bytecodes in the file and the size can be adjusted accordingly. The matrix can then be easily transformed into a gray-scale image composed by pixels.

Gray-scale image of an App can show features on code execution, which can be used to explore code similarity and related patterns. Fig. 1 shows two gray-scale images in the same malware family. Both are created from DEX file, and we can see the similarity in image pattern by vision intuitively. Certainly, diverse image processing methods can be applied for further image analysis for similarity and pattern recognition.

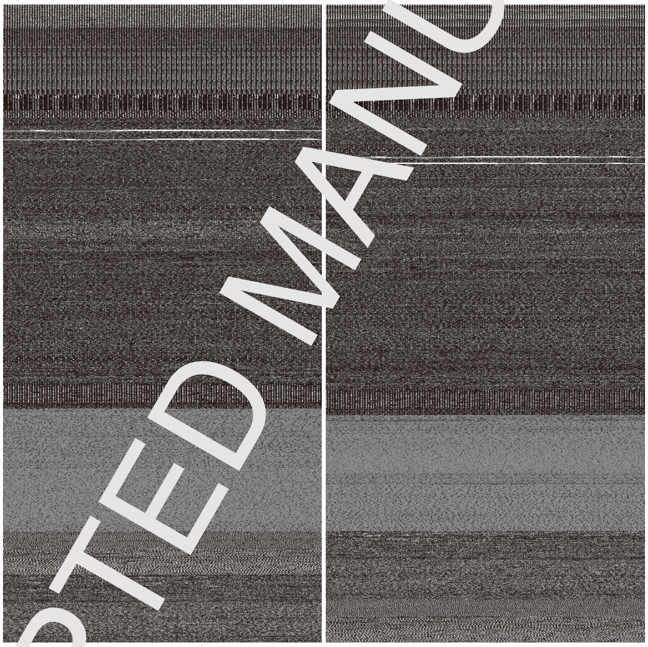

Fig. 1: Comparison of two gray-scale images in the same malware family. Some similarity in image patterns can be observed.

Gray-level co-occurrence matrix (GLCM) is defined as the tabulation of occurring times for different combinations of pixel brightness values (grey levels) in an image. The GLCM is usually used for a series of "second order" texture calculations. First order texture measures are statistics and calculated from original images. Second order measures consider the relationship between groups of two (usually neighboring) pixels in original images.

GLCM-6 represents the six largest eigenvalues of characteristics in GLCM, i.e., Contrast, Homogeneity, Correlation, Dissimilarity, ASM, and Entropy. We can extract the data from DEX file to form a gray-scale image, from which GLCM-6 values can be extracted as features to building a training set.

4. Proposed Scheme - SaaS

The proposed scheme consists of three major functions: network data capture and feature extraction, repackage detection, and code classification. The input process output (IPO) model of the system is depicted in Fig. 2.

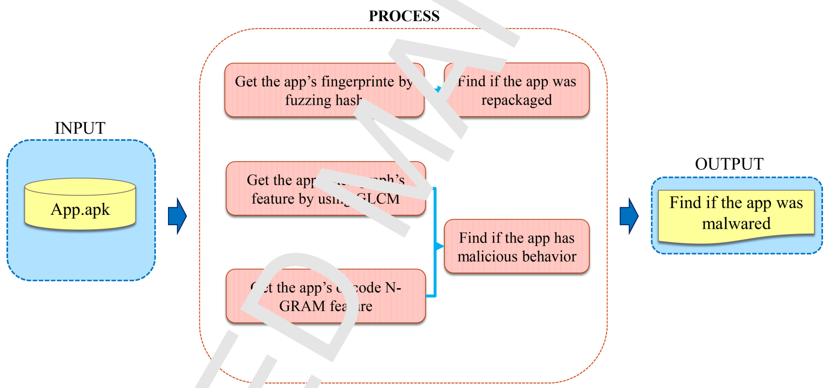

Fig. 2: The IPO (input process output) model of the system.

4.1. App Capture and Feature Extraction

4.1.1. App capture

We custom tailor crawling codes via python for a large scale App crawling from Android application markets. The crawled Apps will further be decompiled to obtain their n-gram sequences and GLCM information. We prefer to collect more samples in this procedure to establish a better training set (i.e. more features of Apps can be learned), which can improve the accuracy in future machine learning procedure.

4.1.2. App fingerprint recording

App fingerprint is recorded by following major steps: Extract the instruction sequences of DEX files; Attain a sequence of simplified instructions; Process sequence via fuzzy hash algorithm to record App fingerprint. Fuzzy hash algorithm outputs the hash value of each section related to sequences, which will not be influenced by most modification operations such as adding or deleting instructions. The specific tool can be selected for providing fuzzy hashing function, e.g., SSDEEP, which can compare similarity strength between candidate files.

4.1.3. N-gram extraction

We use Baksmali to process APK file to output corresponding smali source code. All smali files will firstly be examined and then seven critical instructions, i.e., M, R, G, I, T, P, V, are selected and extracted. We code a Python program to slice the list to produce the corresponding n-gram sequences. In our system, we assign $N = 3$ as the length of feature sequence. Some samples of 3-gram are illustrated in Tab. 1.

More specifically, extracting n-gram characteristics mainly presents following functions: Decompile APK file of an APP; Create an ALLsmali file which encloses the contents of all smali files in each folder (those folders are all come from one APK); Create file named F.smali (here F is the index of the order, which is identical with App order in decompiling) to extract the simplified instructions: Generate file named FSEQ.txt by converting instructions into instruction codes; Create file named n-gram.txt that contains n-gram features of designated App by extracting n-gram from instruction codes.

4.1.4. Feature extraction from gray-scale image

As we have mentioned previously that a binary file can be easily converted into a gray-scale image, we convert the data extracted from DEX file to a gray-scale image. MATLAB's GLCM funtion in Java environment will be invoked to compute GLCM-6 values from the gray-scale image.

The GLCM-6 values describe following six features for a given gray-scale image [20].

- **Contrast** reflects intensity difference between a pixel and its neighbors over the whole image, which is defined as

$$Con = \sum_i \sum_j (i-j)^2 P(i,j) \tag{1}$$

where i and j represent gray value of pixels in an image and $P(i,j)$ is the probability that both pixel i and j are at specific position.

- **Homogeneity** reflects the closeness of element distribution in GLCM to GLCM diagonal, which is defined as

$$Hon = \sum_i \sum_j \frac{P(i,j)}{1+|i-j|} \tag{2}$$

- **Correlation** reflects the statistical measure on how a pixel is correlated to its neighbors over whole image, which is defined as

$$Corr = \sum_i \sum_j \frac{ijP(i,j) - \mu_1\mu_2}{\sigma_1\sigma_2} \tag{3}$$

where $\mu_1 = \sum_i \sum_j iP(i,j)$, $\mu_2 = \sum_i \sum_j jP(i,j)$, $\sigma_1 = \sum_i (i-\mu_1)^2 \sum_j P(i,j)$, and $\sigma_2 = \sum_j (j-\mu_2)^2 \sum_i P(i,j)$.

- **Dissimilarity** reflects the dissimilarity between two pixels, which is defined as

$$Dis = \sum_i \sum_j |i-j|P(i,j) \tag{4}$$

- **Angular Second Moment (ASM)** reflects the summation of squared elements in GLCM, which is defined as

$$Asm = \sum_i \sum_j P(i,j)^2 \tag{5}$$

- **Entropy** reflects the complexity and the inhomogeneous degree of an image, which is defined as

$$Ent = \sum_i \sum_j P(i,j) \log P(i,j) \tag{6}$$

A training set is constructed by combining the data from n-gram and GLCM-6. Relevant procedure is illustrated in Fig. 3.

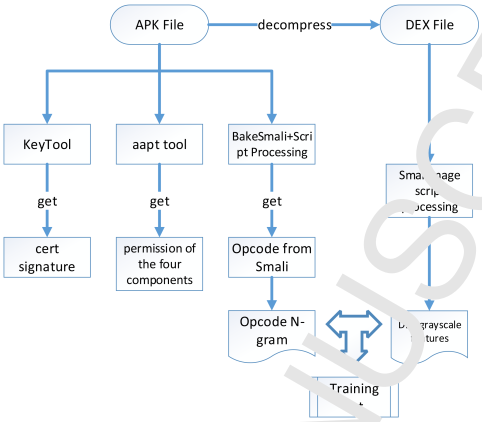

Fig. 3: The flow chart of forming the train set.

4.2. Repackage Detection

Repackage detection is employed in our system, in which two folders are considered: the similarity of App fingerprint between inspected two Apps, and the certificate of App. Although repackaged App has different certificates from original App, most functions in it remain similarity.

Our system collects many certificated App's fingerprint. When a raw App sample from markets is crawled, App fingerprint will be computed and stored. The fingerprint will then be compared with the other fingerprints which are stored before. If there exists a fingerprint which is highly similar to the detected fingerprint and the certificate is distinct with the detected one, the detected App is very likely to be repackaged one and will be assigned a score denoted as $score_{rp}$. The fingerprint of the repackaged App will be removed.

4.3. Code Classification

The code classification process intends to identify Apps that contain malicious codes. A suspected strength in percentage that indicates the possibility of an App to be malware is assigned to each evaluated App. It will be automatically labeled as "normal" or "malware" by the classifier according to presetting threshold. Two thresholds are assigned in our system based on our empirical results from experiments on code classification. A specific App is regarded to be malware, if its strength percentage is larger than α; An App is probably to be malware, if that percentage is larger than β but lower than α.

The classification method is based on random forest. In our experiments, the test of binary classification reports an accuracy 99.5987% (Correctly Classified Instance 1489, Incorrectly Classified Instance 6, Kappa statistic 0.992, Mean absolute error 0.0293, Root mean squared error 0.0775, Relative absolute error 5.866%, Root relative squared error 15.5187%) After above detections, each App will be assigned a score denoted as $score_C = score_N + score_G$, where $score_N$ is a n-gram score, and $score_G$ is a gray-scale image score. If an App is labeled by classifier as "malware", $socre_N$ is set to a negative value, whose absolute value equals probability calculated from machine learning results. In contrast, if an App is labeled by classifier as "normal", $score_N$ will be set a positive value. $score_G$ is set similarly.

We hereby briefly give an example of classifier by n-gram. Firstly, all Smali files are obtained from an APK by using Baksmali, and they are merged into a new file named AllSmali. The system then retrieves all Opcodes orderly from AllSmali and these Opcodes will be simplified. The n-gram method is employed to count the amount of n-gram sequences, which will be dumped if the amount is larger than 300. The system then obtains the n-gram features of the APP as a file. We further create a test model that learns from n-gram features of other Apps, using random forest technique to classify the App ("malware" or "normal"). $score_N$ of the App will be assigned according to the results of classifier. Features of analyzed App will be included into the test model for model upgrading.

4.4. Enhancement Method

To better analyze behaviors of an APP, we further propose an enhancement method based on complex networks to characterize features on function calling graph, and then combine the n-gram information of features with multiple metrics borrowing from complex networks, e.g., degree, average clustering coefficient, average path length, to contribute features set in classifier model.

4.4.1. Function calling graph

We use FlowDroid to create a graph about an App's function calling. FlowDroid is an open source static analysis tool for Android Apps, which can output a graph which starts at function named "dummyMain", and connects all invoked functions in the App. The file named graph.gexf is created by Flowdroid, which is a graph containing nodes and edges. Nodes present API names and function names. Edges present source node information and target node information.

An App may call some safe SDK (Software Development Kit) to simplify the coding process, same development time, and reduce bugs. However, it increases the difficulties in analyzing internal behaviors of Apps. Because certain SDK libraries may call sensitive APIs, false positive may increase due to auditing those sensitive APIs. Thus, we need to reduce the false alert from SDK libraries, such as Alipay SDK, BaiduMap SDK, and so on. Besides, we also need to remove common advertisement libraries to increase the accuracy of the detection. In our experiments, we remove 75 common advertisement libraries, such as com.google.android.gms.ads, net.cavas.show, com.adsmogo.adview, net.youmi.android, et al.

The specific method to remove some safe SDK libraries and common advertisement libraries is show in Alg. 1. It takes as input 3 files - graph.gexf, node_sdk.dot and node_sensiti.dot. Here graph.gexf file is created by using FlowDroid, node_sdk.dot lists the names of safe SDK libraries and advertisement libraries, and node_sensiti.dot contains names of sensitive APIs. In graph.gexf the names of safe SDK libraries and advertisement libraries are shown in nodes and edges, so it is easy to remove nodes or edges which possess those names. By using Alg. 1, we obtain a simplified function calling graph.

4.4.2. Get sensitive APIs information

In this section, we define sensitive APIs that will be used in complex networks. We use TF-IDF (Term Frequency - Inverse Document Frequency) method to define sensitive APIs.

Definition 1 *Sensitive API. The API that occurs more in malware but less in normal Apps will be regarded as a sensitive API.*

In Android environment, developers need to write some permissions in AndroidManifest.xml file to call some specific APIs. Thus, we can comb sensitive permissions in AndroidManifest.xml to reveal sensitive APIs.

Definition 2. *Sensitive Permission. The permission that occurs more in malware but less in normal Apps will be regarded as sensitive permission.*

Algorithm 1 Remove some safe SDK libraries and common advertisement libraries

Input: $graph.gexf, node\_sdk.dot, node\_sensiti.dot$
Output: edge.dot

```
function RmWght(graph.gexf)
    for each node from edge in graph.gexf do
        if node contains node_sdk.dot then
            erase(node); // erase the node information from original file
            function SimlifyEdges(node)
                erase(edge);
                node ← node.target;
                SimlifyEdges(node);
            end function
        elsenode contains node_sensiti.dot
            edge.weight ← 2;
        end if
    end for
end function
```

We use APKtool to dig permissions from 757 malware and 346 normal Apps. Partial permissions and their percentages in two types are listed in Tab. 2. The percentage is calculated by the number of permission divide the number of Apps in normal or malware.

Table 2: Partial of permissions and their percentages in two types

Permission	Percent in malware (%)	Percent in normal (%)
ACCESS_WIFI_STATE	26.81	43.93
CHANGE_WIFI_STATE	12.29	27.17
BROADCAST_PACKAGE_REMOVED	2.38	0
CONTROL_LOCATION_UPDATES	1.45	0
DELETE_PACKAGES	17.97	0
DEVICE_POWER	1.98	0
INTERNAL_SYSTEM_WINDOW	2.77	0
UNINSTALL_SHORTCUT	7.13	0
WRITE_HISTORY_BOOKMARKS	7.79	0
BAIDU_LOCATION_SERVICE	0	2.02
BROADCAST_PACKAGE_CHANGED	0	2.31
BROADCAST_PACKAGE_REPLACED	0	2.31
INTERACT_ACROSS_USERS_FULL	0	4.34
SEND_DOWNLOAD_COMPLETED_INTENTS	0	1.16
SYSTEM_OVERLAY_WINDOW	0	2.02

By using TF-IDF we summarize some permissions with strong indications in Tab. 3. By analyzing those permissions with strong indications, we finally confirm 35 sensitive APIs, e.g., android.telephony.SmsManager.sendDataMessage, android.telephony.SmsManager.sendMultipartTextMessage, android.telephony.SmsManager.sendTextMessage, android.content.ContentResolver.query, et al. Those sensitive APIs will contribute to complex networks modeling.

Table 3: Permission with strong indications

Permission intends to malware	Permission intends to normal
UNINSTALL_SHORTCUT	INTERACT_ACROSS_USERS_FULL
WRITE_HISTORY_BOOKMARKS	BROADCAST_PACKAGE_REPLACED
INTERNAL_SYSTEM_WINDOW	BAIDU_LOCATION_SERVICE
CONTROL_LOCATION_UPDATES	SYSTEM_OVERLAY_WINDOW

Relying Alg. 1 and sensitive APIs, we further define a graph that contains function calling relations, removes safe SDK and advertisement nodes, edges that link one or both nodes related to sensitive APIs are labeled a specific weight, namely, 2. The output of Alg. 1 is a file named edge.dot that saves all edges and nodes. To create a complex network, we propose Alg. 2 that takes as input edge.dot to output a file on complex networks data. The algorithm denotes sensitive APIs as leaf nodes and inverses source nodes within 5 layers for complex networks. The sample is illustrated in Fig. 4 and Fig. 5.

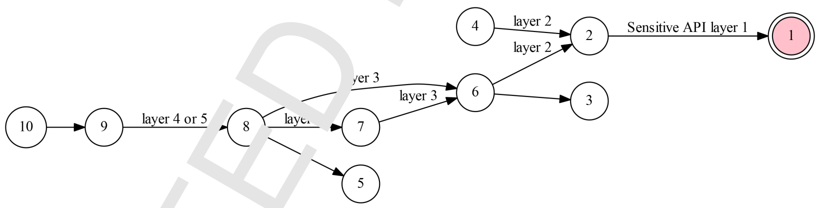

Fig. 4: original graph

The visual exchange is shown in Tab. 4. The left table on edge information matches Fig. 4 and right table matches Fig. 5.

Layer depth is 5 is due to following reasons: 1) retain the features about calling sensitive APIs in malware, and 2) reduce the combine probability with sensitive APIs in normal App.

The graph we created by using Alg. 1 and Alg. 2 is complex networks, because the graph matches the features of complex networks such as 1) short

Algorithm 2 Construction of complex networks

Input: *edge.dot*
Output: *cplx_ntw.dot*

if edge.weight==2 **then**
    $vector \leftarrow edge.targetinedge.dot$
end if
$i = 4$
function $ConstCN(vector)$
    for each node from vector **do**
        while $i > 0\&\&edge.source! = empty()$ **do**
            $new\_vector \leftarrow edge.source$; //new-built vector, different from the previous
            put The edge into cplx_ntw.dot;
            $ConstCN(vector)$;
            $i - -$;
        end while
    end for
    $i = 4$;
    $edge.source \leftarrow ori$;
    $eged.source \leftarrow new_vector$;
    put The edge into cplx_ntw.dot;
end function

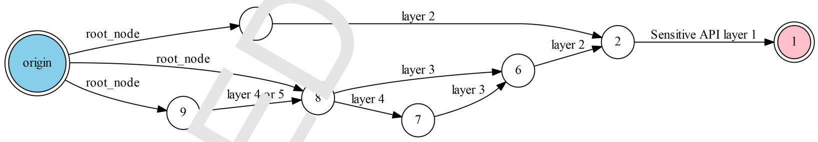

Fig. 5: complex graph

path lengths, 2) scale-free and 3) power-law degree distributions. Tab. 5 lists some sample data from original graph, simplified graph, and complex networks. The first column in Tab. 5 is sample name. The second and third columns are original node number (ONN) and original edge number (OEN). The fourth and fifth columns are simplified graph node number (SNN) and simplified graph edge number (SEN) by calling Alg. 1. It shows that Alg. 1 is useful to simplify the graph. The sixth and seventh columns are network

Table 4: From original graph to complex networks.

label	weight	source	targe
1	2	2	1
2	1	4	2
3	1	6	2
4	1	6	3
5	1	7	6
6	1	6	5
7	1	8	6
8	1	8	7
9	1	9	8
10	1	10	9

ID	label	weight	source	targe
1	1	2	2	1
2	2	1	4	2
3	4	[illegible]	[illegible]	2
4	5	1	7	6
5	6	1	8	6
6	7	1	8	7
7	9	1	9	8
8	0	1	origin	4
9	0	1	origin	8
10	0	1	origin	9

node number (NNN) and network edge number(NEN) by calling Alg. 2. It shows that node number and edge number decrease again. The last 3 columns in Tab. 5 are features of networks: average degree (AD), average clustering coefficient (ACC), and average path length (APL).

Table 5: Some simple's features about complex network

Sample label	ONN	OEN	SNN	SEN	NNN	NEN	AD	ACC	APL
1	204	418	192	390	41	58	1.415	0.039	3.021
2	387	811	373	753	82	139	1.695	0.027	3.259
3	394	[illegible]	378	731	104	156	1.500	0.031	3.252
4	425	968	[illegible]	827	65	93	1.431	0.026	3.144
5	656	1627	[illegible]	1399	126	188	1.492	0.021	3.287
6	711	[illegible]	[illegible]	1521	236	324	1.373	0.024	3.272
7	[illegible]	[illegible]	2035	5788	308	429	1.597	0.036	3.331
8	[illegible]	[illegible]	3148	6290	317	427	1.347	0.033	3.168

We observe that the average path length is much less than sample network edge number, and this matches the feature on short path length in complex networks. Fig. 6 shows that network degree distribution matches power-law degree distribution. Base on above observation, we claim that our created networks are complex networks, and we may apply features of complex networks to detect malware.

4.4.3. Sensitive API n-gram constructing and vector creating

This section explains how to obtain sensitive API n-gram from App complex networks. Firstly, we define what is sensitive API n-gram.

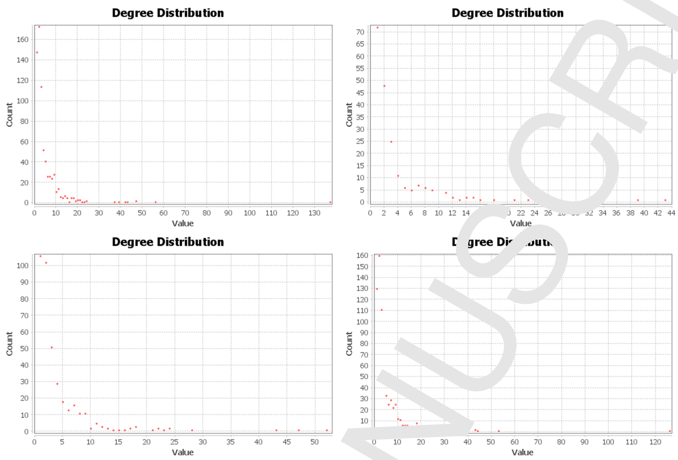

Fig. 6: Degree distribution of 4 samples.

Definition 3. *In complex networks, if there exist identical nodes among the paths from original node to distinct sensitive API nodes within the depth less than 5 layers, those different sensitive APIs construct a sensitive API's n-gram.*

Base on above definition and the file named cplx_ntw.dot, we propose Alg. 3 as follows:

Base on Alg. 3, if there exist some paths from original node to sensitive API nodes, and there exist identical nodes in those paths, we can collect those sensitive APIs into n-gram, where n represents the number of sensitive APIs. In the Fig. 7, there are 3 paths from original node to sensitive API nodes (node_1, node_2, node_3). Because branch_1 and branch_2 are two different nodes, and in the paths from original to node_2 and node_3 there exist same node - branch_2, we say that node_1 is sensitive API 1-gram or 1-gram, and node_2 and node_3 are called 2-gram.

Base on Alg. 3, we can obtain App sensitive n-gram features. We extract 757 malware and 356 normal App's sensitive n-gram features and use TF-IDF to get some n-gram sequences that have the greatest difference between those two types of Apps. In Tab. 6 there exist some functions in n-gram sequences, and each function represents more than one sensitive APIs. There are 22 functions and we can finally form 242 n-gram sequences from those

---

Algorithm 3 Extraction n-gram sequence

Input: *cplx_ntw.dot*
Output: *n_gram.dot*
if edge.weight==2 **then**
  vector ← edge.target in *cplx_ntw.dot*;
end if
for each $node_i$ in vector **do**
  $List_i\_value$ ←noeds on the road from origin to $node_i$ in the order by layer;
  $List_i\_key \leftarrow node_i$;
end for
for each $List_i$ **do**
  add $node_i$ to n-$gram_i$;
  for $eachList_j$ **do**
    if have common element between $List_i$ and $List_j$ **then**
      add $node_j$ to n-$gram_i$;
    end if
  end for
  put n-$gram_i$ into n-$gram.dot$;
end for

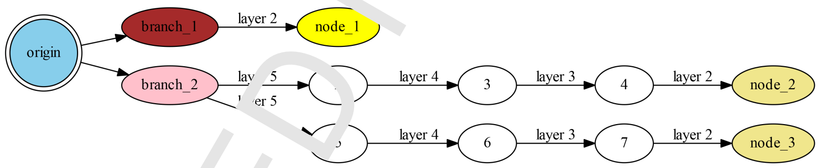

Fig. 7: Sensitive API n-gram:[(node_1)(node_2,node_3)]. Here (node_1) is 1-gram and (node_2,node_3) are 2-gram

function combinations, and those n-gram sequences are stored in the file named ngfr.txt.

After extracting n-gram from an App, we further extract features with respect to complex networks including average degree, average clustering coefficient, and average path length. The complex networks is create by Alg. 1 and Alg. 2 from the graph created by FlowDroid. After all required features are available, we create a vector containing those n-gram features and com-

Table 6: The functions in n-gram from complex networks

Label	Type	Argument 1	Argument 2	Argument 3	Argument 4
1	1-gram	delete function			
2		call telephone function			
3		send message			
4		capture broadcast			
5	2-gram	Send short message	read short message		
6			file access		
7			access address list		
8			receive broadcast		
9			get location information		
10		send by internet	read short message		
11			file access		
12			access address list		
13			receive broadcast		
14			get location information		
15			capture broadcast		
16		call telephone	access address list		
17		capture broadcast	send broadcast		
18	3-gram	send by internet	equipment's IMEI	equipment's IMSI	
19			receive restart broadcast	read short message	
20			receive restart broadcast	get location information	
21	4-gram	send by internet	read short message	access address list	call telephone
22			get location information	equipment's IMEI	equipment's IMSI

plex network features. That is, vector is $= (g_1, g_2, g_3, ...g_n, D, J, L, M/N)$, where g_i $(i = 1, 2, ..., n)$ are n-gram features; If the App has this feature, g_i will be set as 1; Otherwise, it is 0. D is average degree; J is average clustering coefficient; L is average path length; M represents "malware"; N represents "normal". In the enhancement experiments, vector information are feeded into WEKA to train detection model and accuracy results are evaluated.

4.4.4. Experiment Evaluation

We use WEKA to train model by vector information from 8364 malware and 5318 normal Apps. These vectors contain n-gram features, complex network features, and App type in terms of "M" or "N". In the experiments, we use K cross validation to obtain the average accuracy of the proposed method. Tab. 7 lists detection performance in terms of Time, True Positive (TP) rate, False Positive (FP) rate, Precision, Recall, and Receiver Operating Characteristic Curve (ROC) by evaluating 5 different machine learning methods with 10 cross validation in WEKA.

From Tab. 7, we observe that the accuracy of 5 machine learning methods are all accepted, since all TPRs are greater than 0.94 and all FPRs are lower than 0.06 (all ROCs approach 1). Among them, J48 and NavieBayes cost less time and Random Forest and Bagging cost more time. But, TPRs of

Table 7: The results of different machine learning methods

Algorithm	Time(s)	TP Rate	FP Rate	Precision	Recall	ROC
J48	2.49	0.961	0.048	0.961	0.961	0.974
RandomForest	18.74	0.963	0.038	0.963	0.963	0.992
SMO	14.45	0.945	0.052	0.946	0.945	0.946
NaiveBayes	0.23	0.942	0.06	0.942	0.942	0.98
Bagging	11.64	0.965	0.045	0.965	0.965	0.985

J48, RandomForest and Bagging are all greater than 0.96. Thus, J48 is the best method to this vector data set in WEKA.

To justify our method, we choose the same data as in the paper written by N. Peiravian et al. [21]. The data performs as a benchmark in comparisons, which are shown in Tab. 8. Perm represents the permission information in AndroidManifest.xml, API represents API calling graph features, and Com+ represents combinative features with both Perm and API.

Table 8: The benchmark data

Data Set	Algorithm	Precision	Recall
Perm	J48	0.898	0.866
API	J48	0.894	0.903
Com+	J48	0.906	0.928
Perm	Bagging	0.92	0.882
API	Bagging	0.936	0.907
Com+	Bagging	0.949	0.941

Comparison results with benchmark data are depicted in Fig. 8. It shows that our proposed method outperforms others in terms of accuracy in the detection of malware.

5. Experiment and Performance Evaluation

5.1. Module Evaluation

As an integral system with multiple modules, we prefer to evaluate the performance of individual component first and then evaluate the overall performance. The major function modules to be tested include crawling module, feature extraction module, classifier module, and repackage detection module.

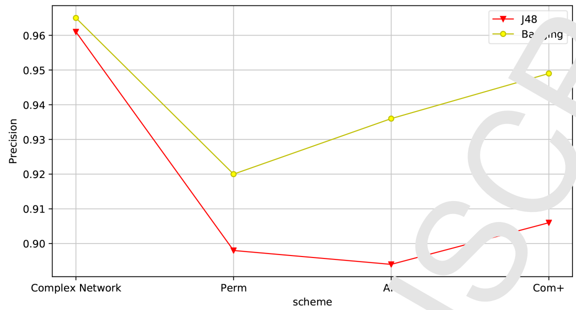

Fig. 8: Performance comparison in accuracy with benchmarks

5.1.1. App crawling module test

In this module, we examine whether the designed crawling program can download Apps so fast as to sense and monitor third-party App markets (Wandoujia market[1], Mumayi market[2], Anzhi market[3], Android market[4] and Huawei market[5]), which are 5 most popular Android markets in China. The speed of downloading under normal PC client is about 0.4 Mbps at first, which certainly is not suitable for large-scale App analysis. We next deploy our system at clouds, it then can reach nearly 1.6 Mbps downloading throughput, e.g., obtaining 3GB data in half an hour. The downloading speed in clouds is four times faster than that in PC end.

5.1.2. App feature extraction module test

In this module test, we mainly test n-gram feature extraction and GLCM-6 feature extraction. For n-gram test, we test the performance of two steps: Decompile APK files, and Get n-gram features.

It takes 37 minutes to decompile 100 sample APK files firstly. It seems not to be efficient. After analyzing the reason, we observe that the decompile speed is related to APK size. If APKs that are larger than 100M are removed, the speed of decompiling increases from 2.7 APKs/min to 4~5 APKs/min.

[1] https://www.wandoujia.com/
[2] url=http://www.mumayi.com/
[3] url=http://www.anzhi.com/
[4] url=http://apk.hiapk.com/
[5] url=http://app.hicloud.com/

After decompiling the APK file, we further test n-gram feature extraction performance. Our experiments spend 1500s to process 1000 malware samples for extracting n-gram features. In experimental results we find there exist some n-gram feature files with size less than 1k, which indicates that feature extraction is unsuccessful. We further analyze the reason - heads of some APKs are damaged in decompiling procedure in Windows.

Some features we obtained (e.g., 18 3-gram features from one n-gram file) in this module test are shown in Tab. 9.

Table 9: 3-gram features extracted from normal Apps

MGR	GRG	RGT	GIP	IPT	PTV
TVP	VPP	PPM	PMT	MTP	TPM
PMV	MVM	VMI	MII	IIG	IGI

5.1.3. App classifier module test

To build training set, we choose 1495 App samples including 754 malicious Apps and 741 normal ones to extract n-gram features and GLCM-6 values. We label them with “Normal” or “Malware”, and write them into CSV format file. After that, the data in training set are processed by WEKA, in which a classifier can be created finally. Above procedure takes about 270 seconds in the experiments. It shows that after ten-fold cross validation, TP rate of the model is 0.989 and FP rate is 0.054, it justifies that the classifier model has high accuracy for detecting malware.

To classify unknown APKs, we build a testing set by including 200 malware and normal samples collected from online BBS. The establishment of the testing set is similar to the training set, except that the former excludes the attribute label (“Normal” and “Malware”). We can classify the testing set by using classifier model, whose results will be compared with BBS declaration manually to evaluate the accuracy of the classifier. The experimental results are given in Fig. 9, in which there are 5 columns - The first column displays sample sequences; The second column indicates actual class of sample (because we exclude attribution labels in the test set, all they are 1:?, where 1 presents default label and ? presents actual label); The third column outputs predicted results by using classifier model (“Normal” or “Malware”), The fourth line shows whether there are some errors (nothing showed) or not; The fifth column presents the probability of predict results that is in the range of 0˜1.

By checking the results manually, our system can achieve an accuracy with nearly 98.5% in detecting malware.

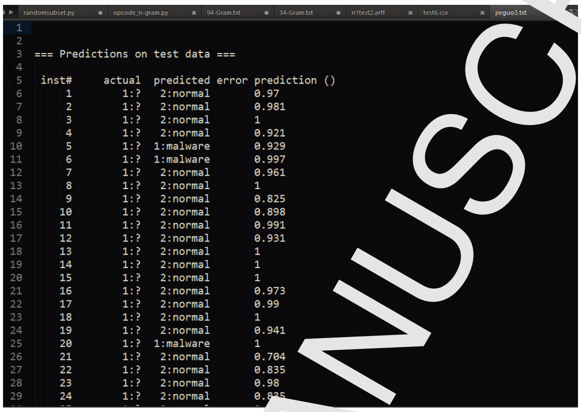

Fig. 9: The results of classification by classifier model.

5.1.4. Repackaging detection module test

In this module test, SSDEEP is selected to provide fuzzy hash algorithm to get an App's fingerprint, which is shown in Tab. 10. The first line shows the brief information of the results, which are blocksize, hash, hash, and file name.

Table 10: The fingerprint of App by using ssdeep

ssdeep,1.1--blocksize:hash:hash,filename
196608: [illegible]ym3WGPKh/kOiCb6Mm05Y0YjM81F+RAaCbLm7w2BV: sWC[illegible]6RKY0Y71F+0DmLiw2BV,/Users/idFTPClienttestnew.zip

Some comparison results by using SSDEEP's command "-m" and judgement on App certification are given in Tab. 11 and Tab. 12. In the former table, there exist multiple Apps possessing identical certificates, and the similarity is 100%. Those Apps are the same from one publisher, and are compiled for many times. In the latter table, there exist multiple Apps that hold low similarity (lower than 50%) but possess the same certificate. The reason is due to the different versions of the same App. Sometimes there exist some Apps that match with others with similarity more than 95%, although they do not hold the same certificate. Such situation can be reasoned from two aspects as follows: 1) Either or both is (are) repackaged; 2) A number of same third-party libraries are called in source code of both Apps.

Table 11: Some Apps have same certificates with others

Number of App	Similarity with others	If have same cert
No.0		
	matches No.2 49%	
	matches No.11 100%	same cert
	matches No.21 50%	
	matches No.25 49%	
	matches No.30 100%	same cert
	matches No.181 100%	same cert

Table 12: Some App is lower similarity with other App but have same cert with other App

Number of App	Similarity with others	If have same cert
N0.7		
	matches No.25[illegible] 54%	
	matches No.287 38%	same cert
	matches No.290 49%	same cert
	matches No.291 54%	same cert

In the experiments, 340 malware samples include 50 repackage samples are included. The repackage module detects 48 repackaged samples among 50, which justifies the repackage detection method is sound.

5.2. Integral Evaluation

We conduct the cloud-based real-time monitoring on large-scale Apps in this section. Malware situational awareness curve will be created, shown, updated for App markets with multiple applications, e.g., massive filtering, supervision, risk management, trend alter and so on. It can also be provided as a third party service for network governance. Fig. 10 depicts malware trends in 3 mainstream markets for a given period. It shows that in those 3 markets from April 23 to April 29, 2017, there exist some new malware that are detected by our proposed system but not aware by App markets.

A risk assessment on App markets are also sorted for five major markets. The metrics is based on the proportion of malware and repackaged applications in the market, which is normalized into a score range in [0,100]. The

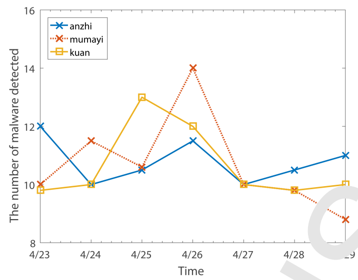

Fig. 10: Malicious code trends.

results (Fig. 11) shows that all scores are not high. It means that in those markets there exist contain malware or repackaged applications that are not detected.

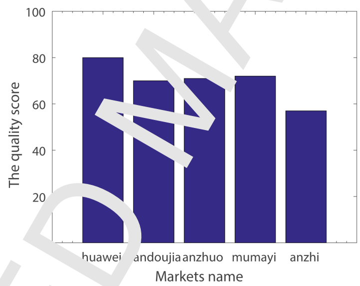

Fig. 11: The risk assessment for major App markets.

6. Conclusion

In this paper, we propose a comprehensive system that can automatically crawl Android Apps and detect malware in a large-scale at real-time. The features of App are extracted by n-gram and GLCM-6 values. Fuzzy hash algorithm is utilized for detecting repackag. The model of complex networks are applied for extracting characteristics in calling function graph. The detection accuracy of our system is evaluated over a large amount of Apps crawled from top 5 popular App markets in China. The results validate the scalability of our system. Our system can detect malware in these markets unaware. Moreover, it can evaluate the risk of those markets in portion of malware.

7. Acknowledgment

The research was financially supported by the Major Scientific and Technological Special Project of Guizhou Province (20183001). Open Funding of Guizhou Provincial Key Laboratory of Public Big Data with No. 2017BD-KFJJ006, National Natural Science Foundation China 61502362, and Open Funding of Hubei Provincial Key Laboratory of Intelligent Geo-Information Processing with No. KLIGIP2016A05.

References

[1] S. News, Google play have an obvious growth in 2017, http://tech.sina.com.cn/it/2018-04-05/doc-ifysuuya8013472.shtml (Apri 2014).

[2] dqriot, 2016 android malware special report, http://blogs.360.cn/blog/review_android_malware_of_2016/ (feb 2017).

[3] M. E. Karim, A. Walenstein, A. Lakhotia, L. Parida, Malware phylogeny generation using permutations of code, Journal in Computer Virology, 1 (1-2) (2005) 13–23.

[4] S. Wehner, Analyzing worms and network traffic using compression, Journal of Computer Security, 15 (3) (2007) 303–320.

[5] P. Faruki, V. Laxmi, A. Bharmal, M. Gaur, V. Ganmoor, Androsimilar: Robust signature for detecting variants of android malware, Journal of Information Security and Applications, 22 (2015) 66–80.

[6] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, B. Zang, Vetting undesirable behaviors in android apps with permission use analysis, in: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, ACM, New York, NY, USA, 2013, pp. 611–622.

[7] K. Rieck, T. Holz, C. Willems, P. Düssel, P. Laskov, Learning and classification of malware behavior, in: Detection of Intrusions and Malware, and Vulnerability Assessment, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 108–125.

[8] M. Apel, C. Bockermann, M. Meier, Measuring similarity of malware behavior, in: 2009 IEEE 34th Conference on Local Computer Networks, IEEE, Zurich, Switzerland, 2009, pp. 891–898.

[9] L. Zeng, W. Ren, Y. Chen, M. Lei, Lmdgw: a novel matrix based dynamic graph watermark, Journal of Ambient Intelligence and Humanized Computing.

[10] X. Wang, Y. Yang, Y. Zeng, Accurate mobile malware detection and classification in the cloud, Springerplus 4 (1) (2015) 583.

[11] S. Jadhav, S. Dutia, K. Calangutkar, T. Oh, Y. H. Kim, J. N. Kim, Cloud-based android botnet malware detection system, in: 2015 17th International Conference on Advanced Communication Technology (ICACT), IEEE, Seoul, South Korea, 2015, pp. 347–352.

[12] S. Y. Yerima, S. Sezer, G. McWilliams, I. Muttik, A new android malware detection approach using bayesian classification, in: 2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA), IEEE, Barcelona, Spain, 2013, pp. 121–128.

[13] S. Y. Yerima, S. Sezer, I. Muttik, High accuracy android malware detection using ensemble learning, IET Information Security 9 (6) (2015) 313–320.

[14] F. A. Narudin, A. Feizollah, N. B. Anuar, A. Gani, Evaluation of machine learning classifiers for mobile malware detection, Soft Computing, 20 (1) (2016) 343–357.

[15] S. H. Hung, C. H. Tu, C. W. Yeh, A cloud-assisted malware detection framework for mobile devices, in: 2016 International Computer Symposium (ICS), IEEE, Chiayi, Taiwan, 2016, pp. 537–542.

[16] V. Kešelj, F. Peng, N. Cercone, C. Thomas, N-gram-based author profiles for authorship attribution, in: Proceedings of the conference pacific

association for computational linguistics, PACLING, Vol. 3, Halifax, Canada, 2003, pp. 255–264.

[17] V. N. Patodkar, I. R. Sheikh, Twitter as a corpus for sentiment analysis and opinion mining, in: Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10), ELRA, Valletta, Malta, 2010.

[18] T. Abou-Assaleh, N. Cercone, V. Keselj, R. Sweidan, N-gram-based detection of new malicious code, in: Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004(COMPSAC 2004), Vol. 2, IEEE Computer Society, Washington, DC, USA, 2004, pp. 41–42.

[19] R. Moskovitch, C. Feher, N. Tzachar, E. Berger, M. Gitelman, S. Dolev, Y. Elovici, Unknown malcode detection using opcode representation, in: Intelligence and Security Informatics, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 204–215.

[20] L. K. Soh, C. Tsatsoulis, Texture analysis of sar sea ice imagery using gray level co-occurrence matrices, IEEE Transactions on Geoscience and Remote Sensing, 37 (2) (1999) 780–795.

[21] N. Peiravian, X. Zhu, Machine learning for android malware detection using permission and api calls, in: 2013 IEEE 25th International Conference on Tools with Artificial Intelligence, IEEE, Herndon, VA, USA, 2013, pp. 300–305.

Author Biograph

Yaocheng Zhang is a graduate student at School of Computer Science, China University of Geosciences, Wuhan. His research interests include mobile security and information security。

Wei Ren currently is a Professor in School of Computer Science, China University of Geosciences (Wuhan), China. He was with Department of Electrical and Computer Engineering, Illinois Institute of Technology, USA in 2007 and 2008, School of Computer Science, University of Nevada Las Vegas, USA in 2006 and 2007, and Department of Computer Science, The Hong Kong University of Science and Technology, in 2004 and 2005. He obtained his Ph.D. degree in Computer Science from Huazhong University of Science and Technology, China. He published more than 70 refereed papers, 1 monograph, and 4 textbooks. He obtained 10 patents and 5 innovation awards. He is a senior member of China Computer Federation and a member of IEEE.

Tianqing Zhu received her B.Eng. and M.Eng. degrees from Wuhan University, China, in 2000 and 2004, respectively, and a Ph.D. degree from Deakin University in Computer Science, Australia, in 2014. Dr. Tianqing Zhu is currently a continuing Teaching Scholar in the School of Information Technology, Deakin University, Australia. Her research interests include privacy preserving, data mining and network security. She has won the best student paper award in PAKDD 2014.

Yi Ren obtained his Ph.D. in Information Communication and Technology from the University of Agder, Norway in 2012. He was with the Department of Computer Science, National Chiao Tung University (NCTU), Hsinchu, Taiwan, as a Postdoctoral Fellow and an Assistant Research Fellow from 2012 to 2017. He is currently a Lecturer in the School of Computer Science at University of East Anglia (UEA), Norwich, U.K. His current research interests include security and performance analysis in wireless sensor networks, ad hoc, and mesh networks, LTE, and e-health security. He received the Best Paper Award in IEEE MDM 2012.

Highlights

In this paper, we have some highlights, such as :

1） Employing comprehensive methods to cooperatively improve the accuracy of detection

2） Using machine learning method instead of artificial, increase detection efficiency.

3） Cloud-based, it can be used across platforms.