Research Paper

DOI: 10.3969/j. issn. 1673-5188. 2016. 03. 008
http://www.cnki.net/kcms/detail/34.1294.TN.20160630.1000.002.html, published online June 30, 2016

An Efficient Scheme of Detecting Repackaged Android Applications

QIN Zhongyuan[1], PAN Wanpeng[2], XU Ying[2], FENG Kerong[1], and YANG Zhongyun[1]
(1. Southeast University, Nanjing 211100, China;
2. ZTE Corporation, Xi'an 710144, China)

Abstract

The increasing popularity of Android devices gives birth to a large amount of feature-rich applications (or apps) in various Android markets. Since adversaries can easily repackage malicious code into benign apps and spread them, it is urgent to detect the repackaged apps to maintain healthy Android markets. In this paper we propose an efficient detection scheme based on twice context triggered piecewise hash (T-CTPH), in which CTPH process is called twice so as to generate two fingerprints for each app to detect the repackaged Android applications. We also optimize the similarity calculation algorithm to improve the matching efficiency. Experimental results show that there are about 5% repackaged apps in pre-collected 6438 samples of 4 different types. The proposed scheme improves the detection accuracy of the repackaged apps and has positive and practical significance for the ecological system of the Android markets.

Keywords

Android; repackage; similarity; edit distance

1 Introduction

In the past few years, Android has developed strikingly and dominated the smartphone market with its market share exceeding Apple. The IDC study shows Android took up 82.8 percent of worldwide smartphone market in the second quarter of 2015 [1]. The popularity is also propelled by the large collection of feature-rich applications in various markets, including the official marketplaces (such as Google Play) or third-party ones (for example, Amazon Appstore or Wandoujia), from which Android users can download a wide variety of feature-rich apps about social networking, shopping, playing, etc. These apps in return foster an emerging app-centric business model and drive innovations across personal, social, and enterprise fields.

Android application developers usually release their apps in the Android market and get revenue by embedding advertisements or charging directly. However, a large number of so-called repackaged apps have been developed for saving development cost and making greater benefits. Their developers download an original application from the Android markets, disassemble it and modify the configuration file, or even inject malicious code and insert ads, then repackage it and release it to the Android markets again. Further manual analysis indicates that these repackaged apps are mainly used to replace or embed advertisements to steal or re-route ad revenues, get user location, phone number and other private data, even to control user's phone remotely [2]. It has a serious negative impact on the ecological security of the entire Android market. Recent studies have shown that app repackaging is a real threat to both official and third-party Android markets [3], and is regarded as one of the most common mechanisms leveraged by Android malware to spread in the wild [4].

Several schemes have been proposed to detect repackaged apps in Android markets [2], [5]–[9]. They can be divided into two categories: dynamic analysis and static analysis. The dynamic analysis usually uses system calls embedded in the kernel space at the low layer of the Android architecture. Ying-Dar Lin et al. [8] presented SCSdroid that captures the system call sequence of each thread when executing malicious repackaged applications and then extracts the common subsequences, which can be regarded as possibly malicious behavior of malicious repackaged applications. Since they may also exist in benign applications, the Bayes Theorem is adopted to filter these non-discriminating common subsequences and then find the common subsequences that indicate the truly malicious behavior. The static code-analysis-based detection is more efficient than the dynamic one. However, in practice, code obfuscations can be easily applied to evade static analysis-based detections. Huang et al. [2] proposed a framework capable of performing a set of obfuscation algorithms in various forms on the Dalvik bytecode to evaluate the obfuscation resilience of repackaging detection algorithms. DEXCD [5], developed by Ian Davis, extracts the opcodes from Java class in Dex file and tries to find a steam match of opcodes between different apps to detect the cloned apps. Crussell et al. [6] proposed DNADroid to detect android apps copying by comparing program dependency graphs between methods and can resist against several control flow obfuscations and noisy code insertion attacks that do not modify the data dependency. However, the side-effect free manipulation has the potential to evade the graph isomorphism algorithm based detection. DroidMOSS [7], presented by Wu et al., leverages specialized hashing technique, called fuzzy hashing, to measure the similarity between original and repackaged

This work was supported by ZTE Industry-Academia-Research Cooperation Funds

D:\EMAG\2016-08-52/VOL12\RP2.VFT——7PPS/P1

applications. DroidMOSS calculates fuzzy hash of all appsand compares the similarity of fingerprints of two apps. It can effi ciently identify those code pieces that were not touched by the repackager and works well when code manipulation was only performed at a few points, e.g., hard coded URLs. However, in DriodMOSS, the fuzzy hash is based on Spamsum [10], proposed by Andrew, in which any random 32-bit binary data is compressed to a 6-bit printable character, i.e., every 2^{26} values will be mapped to the same value. This has certain influence on the accuracy of similarity comparison [11], thereby reducing the accuracy of detection. Besides, as DroidMOSS needs to calculate similarity scores for all the apps, its highly time and memory consuming nature makes it unrealistic in deployment.

In this paper, we proposed an improved context triggered piecewise hash (CTPH) [12] based on DroidMOSS, which uses two small primes to perform twice CTPH (T-CTPH) process and generates two fingerprints for each app to detect the repackaged Android applications.

The main contributions of this paper are as follows:

1) We propose an improved Android application fingerprint generating algorithm T-CTPH, in which two small primes are used as the trigger values to increase the randomness against possible attacks and improve the accuracy. It can be further used to filter out unnecessary matching.
2) An improved algorithm is proposed to speed up the calculation of the fingerprints similarity. Besides, memory overhead is also greatly reduced.
3) We have realized our system to detect repackaged apps and found about 5% repackaged apps in pre-collected 6438 samples of 4 different types.

Portions of this work have previously appeared as an extended abstract [13]. We revise the paper a lot and add more technical details. Specifically, we discuss the choice of trigger value and methodology proof of T-CTPH in detail. We also redo the whole experiments and present the performance comparison of different methods and manual analysis. Moreover, a concrete study of one repackaged app is presented.

The remainder of this paper is organized as follows. In section 2, we introduce our approach, including feature extraction, fingerprint generation and similarity matching. In section 3, our approach is evaluated based on 6438 real applications from several Android markets. Section 4 gives the conclusion.

2 Scheme Design

To accurately detect repackaged apps in Android markets, we propose an improved scheme based on CTPH. With the help of a filtering method and the optimized similarity calculating algorithm, the similarity calculation is speeded up to detect the repackaged apps efficiently.

2.1 System Architecture

Android Package (APK) contains all resources that the application needs to run. The .apk files are actually compressed packages with ZIP format. The signature information stored in META-INF directory ensures the integrity of APK and the system security. Resource files like images are stored in RES directory. .apk files also include a manifest XML that specifies a number of aspects about the application, including its name, version information, permissions required to perform, referenced library files and other important information. Android applications are primarily developed in Java. The Java source code is first compiled to Java bytecode and then converted into the Dalvik executable (DEX) format. This paper mainly analyzes the DEX bytecode.

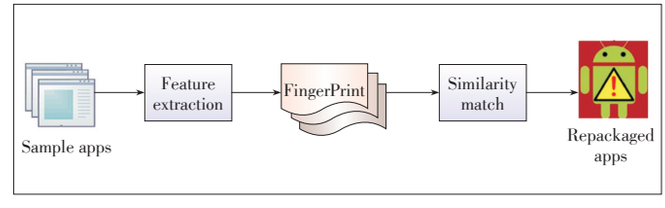
The overall architecture of our system is shown in **Fig. 1**. It is divided into three parts: Feature extraction, fingerprint generation, and similarity matching. Since malicious codes and advertisements are always injected to the repackaged apps, the types of repackaged application and the original one are always the same. Based on this observation, we first download the apps by category from the Android market and store them in their corresponding app databases. For each application in app databases, we extract its author information and application instructions. Next twice CTPH process is executed on the extracted instructions for fingerprints generation. Finally, similarity comparison algorithm is used to find the matching pairs (suspicious repackaged pairs). As in DroidMOSS, we assume that the signing keys from app developers are not leaked. Therefore, APKs with the same signature must be generated by the same author, and they can be ignored because the matched pairs with high similarity and the same signature are always the different versions of the same application.

2.2 Feature Extraction

For each app, feature extraction includes the steps of uncompressing, extracting the author information, disassembling DEX file, extracting the app instructions, and doing T-CTPH. Finally, the fingerprint of the APK is generated.

After uncompressing the app, we use the keytool [14] to extract certificate information from META-INF directory by command *keytool -printcert -V -file "XXX.RSA"*. Then we leverage Dalvik disassembler baksmali [15] to disassemble the classes.dex bytecode file as *java -jar baksmali-1.2.4.jar -o classout/classes.dex*.

All the disassembled smali files and folders are stored in the classout directory by the class hierarchical relationships. The app instructions extraction is done according to the following



▲**Figure 1. System architecture.**

D:\EMAG\2016-08-52/VOL12\RP2.VFT——7PPS/P2

rules: 1) Depth traversal with the alphabetical order of generated smali files and folders; 2) Since some class names may be modified before releasing, we ignore the confusing names of class so as to reduce the error of instruction extraction; 3) Extracting methods of different classes.

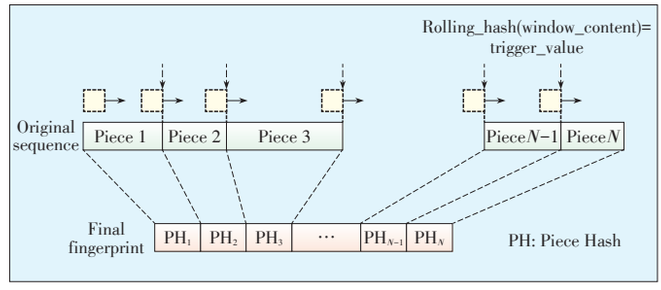
2.3 Fingerprint Generation

For an actual .apk file, the extracted instruction sequences in section 2.2 are extremely long. To generate the fingerprint of apps, the common way is to use a hash operation to compress the long sequence. Although hashing can determine whether two apps are the same, it is not helpful to know the similarity measurement of two apps. The reason is that one minor modification will greatly change the hashing value. Furthermore, calculating the similarity of two apps directly will be particularly expensive. So DroidMOSS [7] adopts fuzzy hashing to solve the above problems. Specifically, it first divides the long sequences into some short pieces with a fixed trigger value, calculates the hash value of each short piece, and then maps each 32-bit binary hash data to a 6-bit binary printable character based on Base64 [16], and concatenates piece hash results as the final fingerprint of one application at last. The fuzzy hashing can effectively localize the changes possibly made in repackaged apps and the similarity between the generated fingerprints represents how similar their corresponding apps are.

Typically, the fuzzy hashing algorithm consists of a weak hash algorithm with a trigger value for the piece, a strong hash algorithm for calculating the piece hash, a compression algorithm for mapping each piece hash to a shorter value and a similarity comparison algorithm used to calculate the similarity of two fuzzy hash values. DroidMOSS first divides all the long instruction sequences with a fixed trigger value. It randomly selects 200 samples from 6 Android markets to compare with 68,187 official applications. However, it is highly time-consuming to calculate 81,824,400 similarity scores for apps. Secondly, DroidMOSS maps any 32-bit binary hash data to a 6-bit binary printable character based on Base64, which means that every 2^{26} value will be mapped the same value. If the file is large enough, some different piece hashes will be mapped to the same value. Finally, in terms of similarity calculation method, DroidMOSS uses a two dimensional array to calculate the edit distance between two fingerprints. However, it is memory-consuming if the fingerprints are very long, especially when facing 81,824,400 pairs.

Based on the above analysis, we remove the compression mapping algorithm of fuzzy hash and execute CTPH process twice with two small primes as trigger values so as to generate fingerprint of long instruction sequences efficiently. Specifically, for an instruction sequence, we use two small primes to do twice CTPH processes respectively. Besides, only the fingerprints with the same triggered prime will be compared, which improves the overall efficiency of detecting repackaged apps.

Fig. 2 shows the CTPH algorithm. The original sequence in the figure is the input of the process and a trigger value (*tv*) is selected for dividing pieces. All the piece hashes are then calculated and concatenated as the final fingerprint. This process is presented in **Algorithm 1**.



▲**Figure 2. Context triggered piecewise hash.**

Algorithm 1. CTPH

Input: Data stream *sequence*
Output: Trigger value *tv*, fingerprint *signature*
Description:
rh − rolling hash, *ph* − piece hash, *ws* − size of sliding window
1: *set window_size(ws)*
2: *tv = compute trigger value(sequence)*
3: *initialize roll_hash(rh)*
4: *initialize piece_hash(ph)*
5: *initialize signature(signature)*
6: **for** each *byte b in sequence* **do**
7: *update roll_hash(rh, b)*
9: *update piece_hash(ph, b)*
10: **if** *rh mod tv = tv-1* **then**
11: *signature+=ph*
12: *initialize piece_hash(ph)*
13: **end if**
14: **end for**
15: **return** (*tv, signature*)

In our scheme, we present an improved approach to effectively detect repackaged applications in Android market. The concrete process is presented in **Algorithm 2** and visually summarized in **Fig. 3**.

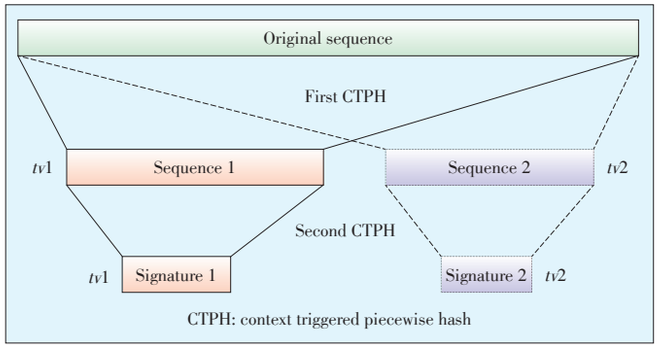
For the input *Original sequence* extracted in section 2.2, we first use a trigger value *tv1* to process the first CTPH with Algorithm 1 and *Sequence1* is generated. Then *tv1* is used again for the second CTPH process with *Sequence1*, and the final result *Signature1* is generated (see left with a solid line). For the right part in Fig. 3 (the dotted line), we also make the same process with trigger value *tv2*, and generate the second result *Signature2*.

In Algorithm 2, for the function CTPH(), i.e., *signature1= CTPH(tv1, sequence1)*. *signature1* is the result of Algorithm 1 with input sequence1 and trigger value *tv1*. The final finger-

D:\EMAG\2016-08-52/VOL12\RP2.VFT——7PPS/P3

Algorithm 2. T-CTPH for fingerprint generation

Input: App instruction sequence *Original Sequence*
Output: Trigger values *tv*1 *and tv*2, fingerprint *signature1*, *signature*2
Description:
*sequence*1, *sequence*2 – results of CTPH, *ws* – window size
1: *set window_size(ws)*
2: (*tv*1, *tv*2) = *trigger value(Original sequence)*
3: *initialize sequence(sequence*1, *sequence*2*)*
4: *initialize signature(signature*1, *signature*2*)*
5: *sequence*1 =*CTPH(tv*1, *Original sequence)*
6: *sequence*2 =*CTPH(tv*2, *Original sequence)*
7: *signature*1 =*CTPH(tv*1, *sequence*1*)*
8: *signature*2 =*CTPH(tv*2, *sequence*2*)*
9: **return** (*tv*1, *tv*2, *signature*1, *signature*2)



▲**Figure 3. T-CTPH for fingerprint generation.**

print of an app is:

$$\text{sign}\,ature = (tv1, \text{sign}\,ature1) \| (tv2, \text{sign}\,ature2)\,. \quad (1)$$

2.4 Similarity Matching

After the above steps, each app has its own fingerprint. Next we calculate the similarity between the fingerprints generated in section 2.3. There are several ways to obtain the similarity, such as bitmap algorithm and computing the longest common subsequence between strings. In this paper, we use edit distance to calculate the similarity between two fingerprints. The edit distance is the minimum edit operation to turn one fingerprint into another, including insertion, deletion and substitution of a single character. In order to calculate the edit distance of the two strings, a conventional two-dimensional matrix is used to represent the distance between two strings and fill the matrix circularly. Finally we get *matrix[len1,len2]* which is the edit distance between two strings with length *len1* and *len2* respectively.

It should be noted that if the two strings are relatively long, *len1*×*len2* size of memory needed for calculating will be quite large, thus reducing the speed for similarity matching. In order to speed up the calculation for long strings, we presented an improved calculation algorithm of edit distance. Specifically, we use three one-dimensional arrays *array1*, *array2*, *array3* (with sizes of *len1*, *len2*, *len3*, respectively) to calculate the edit distance. *array1* denotes the first column of the conventional two-dimensional array, *array2* and *array3* denote two adjacent rows of that two-dimensional array. We fill *array2* and *array3* circularly in an iterative method, which exchanges *array2* and *array3* continuously to denote two adjacent rows. In the end, if *len1* is odd, the edit distance is *array2[len2-1]*, otherwise the edit distance is *array3[len2-1]*. The process is shown in **Algorithm 3**. In this approach, only *len1*+ *len2*×2 memory is needed, which greatly saves memory and speeds up the process.

Algorithm 3. Calculates the edit distance between two apps

Input: Two fingerprints *fp*1and *fp*2
Output: Edit distance between *fp*1and *fp*2
1: *len*1←*strlen*(*fp*1), *len*2 ←*strlen*(*fp*2)
2: *initialize array*1(*len*1)
3: *initialize array*2(*len*2)
4: **for** *i* = 1→*len*1 **do**
5: **for** *j* = 1→*len*2 **do**
6: *cost*=*fp*1[i]=*fp*2[j] ? 0:1
7: **if** (*i mod* 2=0) **then**
8: *array*2[0]=*array*1[*i*]
9: *array*2[*j*]=*min*(*array*2[*j*-1], *array*3[*j*], *array*3[*j*-1])+*cost*
10: **else**
11: *array*3[0]=*array*1[i]
12: *array*3[*j*]=*min*(*array*3[*j*-1], *array*2[*j*], *array*2[*j*-1])+*cost*
13: **end if**
14: **end for**
15: **end for**
16: **if** (*len*1 *mod* 2=0) **then**
17: *edit_dist*= *array*3[*len*2-1]
18: **else**
19: *edit_dist*= *array*2[*len*2-1]

After the edit distance between two fingerprints is calculated, equ. (2) is used to measure the similarity between the two fingerprints [4]:

$$Sim\_Score = [1 - \frac{edit\_dist}{\max(len1, len2)}] * 100\,. \quad (2)$$

If two apps are signed with different developer keys and the similarity score between two apps exceeds a certain threshold, we treat them as repackaged matching pairs. Note that the choice of threshold greatly affects the false positive and false negative rates, thus influencing the accuracy of our test results. In our experiments, we apply the threshold 70 empirically, and it shows a good balance between false positive and false nega-

D:\EMAG\2016-08-52/VOL12\RP2.VFT——7PPS/P4

tive rates.

3 Evaluation

3.1 Sample Collection and Classification

To perform a concrete study on the repackaged apps and measure the effectiveness of our scheme, we developed an APK crawler and collected 6438 apps from various Android markets, including social networking, game, system tool, and shopping. We store them in different databases. The exact numbers of different types of the collected apps are shown in **Table 1**.

3.2 Comparison of DroidMOSS and T-CTPH

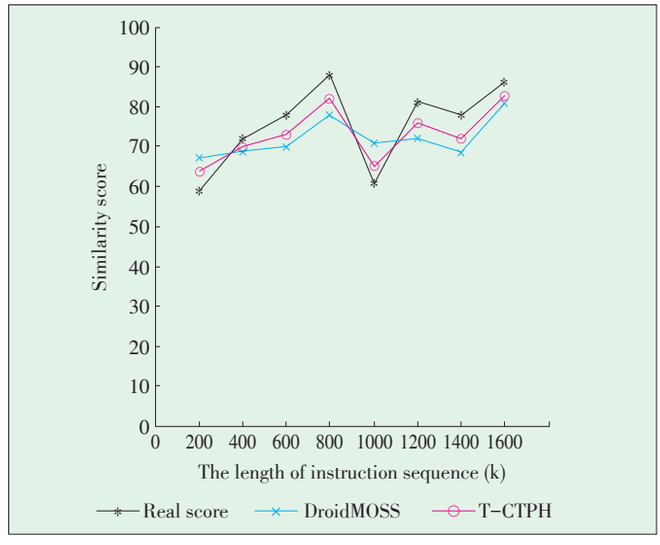
The use of compression mapping algorithm based on Base64 has a certain impact on the accuracy of the generated fingerprint and the false positive (negative) rate. In most case, larger trigger value produces shorter fingerprint, and poor accuracy, and vice versa. Therefore, the selection of trigger value has an important effect on the results. For example, for a web chatting tool QQ3.0 of size 6.37 MB, we get a trigger value 6385 with the method in [11] while the trigger value is (223, 227) in our approach. Next, we collect the same application from different sources and make a similarity comparison of the fingerprints generated by the different instruction sequences length. **Fig. 4** shows the similarity of apps using different length of instruction sequences. The real score is sequence similarity calculated by edit distance with Algorithm 3. The x axis represents the length of the sequence, and the y axis represents the similarity score of two apps. As in DroidMOSS most of the experimental parameters are not given, e.g., the string length. We select different string length and make a comparison between DroidMOSS and T-CTPH in Fig. 4.

As seen from Figs. 3 and 4, the similarities of the sequences are closer to the real score in our approach, that is, our scheme has improved the similarity accuracy between the fingerprints, thus reducing the false positive (negative) rate. The results of detecting repackaged apps in pre-collected data are shown in **Table 2**. The total consumed time of DroidMOSS and T-CTPH are shown in **Table 3**.

Table 2 shows that the proposed scheme (T-CTPH) improves the accuracy of detecting repackaged apps, and its results are closer to the results of manual analysis. In addition, the apps of social networking have the highest repackaged rate, and the lowest is of system tool. It is because the repackaged apps are often used for stealing user's internet traffic and phone bill by injecting malicious code to existing apps or implanting Trojan to remotely control user's phone, which will inevitably require internet service. Besides, game apps are likely to be repackaged. The reason is that developers can re-route or steal ads revenues by replacing or embedding ads to games.

▼**Table 1. Distribution of collected apps**

Category	Number
Social	2557
Game	2396
System tool	838
Shopping	647
Total	6438



▲**Figure 4. Similarity score of different methods.**

▼**Table 2. Results of different detecting repackaged apps methods**

Category	DroidMOSS	T-CTPH	Manual analysis
Social	157	156	155
Game	140	140	138
System tool	39	38	38
Shopping	34	34	33

▼**Table 3. Total consumed time of DroidMOSS and T-CTPH**

Category	Number	DroidMOSS (hour)	T-CTPH (hour)
Social	2557	22.51	7.12
Game	2396	19.85	6.12
System tool	838	2.58	0.84
Shopping	647	1.52	0.61

3.3 Optimized Edit Distance Method

To calculate the pair-wise similarity scores of the fingerprints quickly, we optimize the edit distance method by using three one-dimensional arrays to replace a two-dimensional array, which not only saves a lot of memory resources, but also enhance the speed of the similarity calculation process. We test the optimized algorithm on a computer with Linux system (Ubuntu 10. 04). The CPU is Intel (R) Pentium 4 running at

D:\EMAG\2016-08-52/VOL12\RP2.VFT——7PPS/P5

2.93 GHz and the size of RAM is 2 GB. Table 3 shows a comparison of time consumption for some sequences in different length ranges.

Table 4 shows that the optimized algorithm improves the speed of similarity calculation significantly, and when the sequence is longer, it is more obvious that the consumed time for calculating similarity between sequences is reduced.

3.4 Case Analysis

To perform a concrete study of the repackaged apps and reveal how one app is repackaged, we show the analysis of repackaged apps detected by our scheme. For example, through manual analysis, we find a repackaged app QQ that is stealing the user´s private information. The permission to read phone state READ_PHONE_STATE is first inserted to AndroidManifest.xml file as shown in **Fig. 5**. Then the repackaged QQ gets the IMEI, and phone number, by calling *getDeviceId* and *getLine1Number* respectively (**Figs. 6** and **7**).

Usually, advertising Software Development Kit (SDK) needs to add publisher identifier to AndroidManifest.xml, and then modify the layout description and the program bytecodes to show ads. The following is a detected example of repackaging a normal app (com. racingstudio. racingmoto) by including AdMob [17] SDK in the app. We find that the signed keys are different, but they are the similarity matching pairs. Further, a manual analysis shows that ads always pop up in the bottom on the game interface of the repackaged app (right) as shown in **Fig. 8**.

▼**Table 4. Consumed time before and after optimization**

Length of sequence (k)	Before optimization (ms)	After optimization (ms)
0.5	10.27	4.04
1	50.23	16.21
2	127.41	68.24
4	468.56	272.91
6	1028.37	604.49
8	1818.52	1032.50
10	2992.54	1443.57
12	18900.53	2432.13

```
<uses-permission
        android:name="android. permission. ACCESS_WIFI_STATE">
</uses-permission>
<uses-permission
        android:name="android. permission. READ_PHONE_STATE">
</uses-permission>
<uses-permission
        android:name="android. permission.
KILL_BACKGROUND_PROCESSES">
</uses-permission>
```

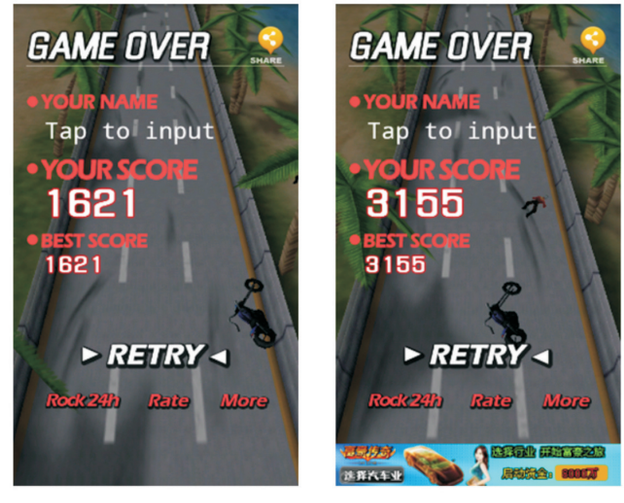
▲**Figure 5. Permissions in AndroidManifest.xml file.**

```
. method private readCell()V
    .registers 5
    ……
    const-string v1, "phone"
    invoke-virtual {v0, v1}, Landroid/content/Context;->
        getSystemService(Ljava/lang/String;)Ljava/lang/Object;
    ……
    invoke-virtual {v0}, Landroid/telephony/TelephonyManager;->
        getDeviceId()Ljava/lang/String;
    ……
. end method
```

▲**Figure 6. Get IMEI.**

```
. method private b()Ljava/lang/String;
    .registers 5
    ……
    const-string v0, "phone"
    invoke-virtual {p0, v0}, Lcom/tencent/mobileqq/activity/RegisterActivity;->
        getSystemService(Ljava/lang/String;)Ljava/lang/Object;
    ……
    invoke-virtual {v0}, Landroid/telephony/TelephonyManager;->
        getLine1Number()Ljava/lang/String;
    ……
. end method
```

▲**Figure 7. Get phone number.**



▲**Figure 8. Screenshots before and after repackaging.**

We also find that functions like *setVisibility*, *findViewById*, *loadAd* are inserted in onCreate function to display ads in the disassembled files, by which developers can steal the ads revenues (**Fig. 9**).

4 Conclusions

In this paper, we propose an improved repackaged application detection scheme based on T-CTPH. An improved finger-

D:\EMAG\2016-08-52/VOL12\RP2.VFT——7PPS/P6

```
. method public onCreate(Landroid/os/Bundle;)V
    .registers 7
    ……
    invoke-super {p0, p1}, Lcom/droidhen/game/racingengine/BaseActivity;->
        onCreate(Landroid/os/Bundle;)V
    ……
    const/4 v3, 0x4
    invoke-virtual {v2, v3}, Lcom/droidhen/api/scoreclient/widget/UsernameEdit;->
        setVisibility(I)V
    ……
    const v2, 0x7f060004
    invoke-virtual {p0, v2}, Lcom/droidhen/game/racingmoto/GameActivity;->
        findViewById(I)Landroid/view/View;
    ……
    check-cast v2, Lcom/google/ads/AdView;
    iput-object v2, p0, Lcom/droidhen/game/racingmoto/GameActivity;->
        _adView:Lcom/google/ads/AdView;
    ……
    invoke-static {p0}, Lcom/droidhen/game/racingmoto/AdController;->
        loadAd(Landroid/app/Activity;)V
. end method
```

▲**Figure 9. Inserted codes for displaying ads.**

print generating algorithm is presented by using two small primes as the trigger values for T-CTPH so as to increase the randomness against possible attacks and improve the accuracy. We then optimize the similarity calculation method and filter unnecessary matching processes to make the similarity matching more efficient. Our experimental results show that about 5% of the apps are repackaged in our pre-collected data. The proposed scheme improves the detection accuracy of the repackaged apps, and has positive and practical significance for the ecological system of the Android market.

References

[1] IDC. (2015). *Smartphone OS Market Share, 2015 Q2* [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp
[2] H. Huang, S. Zhu, P. Liu, and D. Wu, "A framework for evaluating mobile app repackaging detection algorithms," in *6th International Conference on Trust and Trustworthy Computing, TRUST 2013*, London, United kingdom, 2013, pp. 169–186. doi:10.1007/978-3-642-38908-5_13.
[3] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of 'piggybacked' mobile applications," in *3rd ACM Conference on Data and Application Security and Privacy*, San Antonio, TX, United states, 2013, pp. 185–195. doi:10.1145/2435349.2435377.
[4] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *33rd IEEE Symposium on Security and Privacy*, San Francisco, CA, United states, 2012, pp. 95– 109. doi:10.1109/SP.2012.16.
[5] I. Davis. (2012). *Dex clone detector* [Online]. Available: http://www.swag.uwaterloo.ca/dexcd/index.html
[6] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on Android markets," in *17th European Symposium on Research in Computer Security*, Pisa, Italy, 2012, pp. 37–54. doi:10.1007/978-3-642-33167-1_3.
[7] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, San Antonio, USA, 2012, pp. 317–326. doi:10.1145/2133601.2133640.
[8] Y.-D. Lin, Y.-C. Lai, C.-H. Chen, and H.-C. Tsai, "Identifying android malicious repackaged applications by thread-grained system call sequences," *Computers and Security*, vol. 39, pp. 340–350, 2013. doi: 10.1016/j.cose.2013.08.010.
[9] J.-H. Jung, J. Y. Kim, H.-C. Lee, and J. H. Yi, "Repackaging attack on android banking applications and its countermeasures," *Wireless Personal Communications*, vol. 73, pp. 1421–1437, 2013. doi: 10.1007/s11277-013-1258-x.
[10] T. Andrew. (2010). *Spamsum README* [Online]. Available: http://www.samba.org/ftp/unpacked/junkcode/spamsum
[11] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital Investigation*, vol. 3, pp. 91–97, 2006. doi: 10.1016/j.diin.2006.06.015.
[12] L. Chen and G. Wang, "An efficient piecewise hashing method for computer forensics," in *1st International Workshop on Knowledge Discovery and Data Mining*, Adelaide, Australia, 2008, pp. 635-638. doi:10.1109/WKDD.2008.80.
[13] Z. Qin, Z. Yang, Y. Di, et al., "Detecting repackaged android applications," in *3rd International Conference on Computer Engineering and Network, CENet 2013*, Shanghai, China, 2013, pp. 1099–1107. doi:10.1007/978-3-319-01766-2_125.
[14] SourceForge. (2013). *Keytool* [Online]. Available: https://sourceforge.net/projects/keytool/
[15] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, "Madam: a multi-level anomaly detector for android malware," in *Computer Network Security*, ed: Springer, 2012, pp. 240–253.
[16] J. Walker. (2007, June 10). *Base64—encode and decode base64 files* [Online]. Available: http://www.fourmilab.ch/webtools/base64
[17] Google. (2015). *Admob for Android Developers* [Online]. Available: http://support.google.com/admob/topic/1307236?hl=zh-Hans&ref_topic=1307209

Manuscript received: 2015-07-24

Biographies

QIN Zhongyuan (zyqin@seu.edu.cn) received the MS degree in computer science and the PhD degree in communication and information system from Xi'an Jiaotong University, China in 1999 and 2003, respectively. He is currently an associate professor in the School of Information Science and Engineering, Southeast University, China. His research interests include wireless network security and Android security. He has published more than 40 papers in refereed international journals and conference proceedings.

PAN Wanpeng (pan.wanpeng@zte.com.cn) received his MS degree in network and information security from Northwestern Polytechnical University, China in 2007. He is the chief security director of terminal business division at ZTE Corporation. His research interests include Android security and network security.

XU Ying (xu.ying6@zte.com.cn) received the BOM degree in Information Management and the MS degree in computer application from Zhengzhou University, China in 2005 and 2010. Now she is working with ZTE. Her research interests focus on software testing.

FENG Kerong (fengkerong@163.com) received the BE degree in communication engineering from China University of Petroleum in 2013. Now she is pursuing her MS degree at Southeast University, China. Her research interests focus on security in Android.

YANG Zhongyun (midcloud@foxmail.com) received the MS degree in information security from Southeast University, China in 2014. He is currently a software engineer at CoolPad Corporation. His research interests focus on security in Android.

D:\EMAG\2016-08-52/VOL12\RP2.VFT——7PPS/P7