Detection of Repackaged Smartphone Applications On Android

Ankush Kumar Singh
E & C dept. IIT Roorkee
01332-286454
ancsiuec@iitr.ernet.in

Shree Garg
E & C dept. IIT Roorkee
01332-286401
shreedec@iitr.ernet.in

ABSTRACT
We are living in the post-PC era, where smartphones, tablets and other mobile devices have grabbed the market. These mobile devices accompany huge number of useful and feature-rich applications. Mobile devices contain more sensitive information than computer systems like GPS location, contact list, owner’s credential etc. Besides this mobile devices are capable of voice calling, messaging, internet connectivity etc. Apps optimize the usage of these devices. We have apps to entertain, connect with our friends, search and browse anything on the move. It attracts apps repackagers to exploit the mobile features and information contained in these devices and apps are the way to accomplish this. In this paper, we have given a systematic procedure to check whether a given app is repackaged or not. For this we have used context trigger piecewise hashing and edit distance measurements. Our prototype not only reveals whether the app is repackaged or not but also pinpoints the modified segments. Once the modified segments are recognized, Jaccard coefficient is calculated to find the similar segments. Then reverse engineering is performed only on these obtained similar segments not on the entire app. This reduces the time and efforts and improves the efficiency of reverse engineering.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features – Java, C++, Tool: ApkTool, Dex2Jar, JD-GUI and Android SDK.

General Terms
Security.

Keywords
Android, apps, repackaging, automated reverse engineering, context triggered piecewise hashing.

1. INTRODUCTION
Smartphones are getting popularity day by day. In this growing market of smartphones, Android, an open source platform of Google has overtaken others to become top mobile platform. According to report by Gartner [1] android accounted for 72.4% of the smartphone operating system sales. Google reported [2] that number of applications on its play store crossed 0.7 million. Due to various functionality provided by these apps, it has brought a revolution for smartphones users. This has made smartphones as hand-held mini-computer. As the usage of apps on smartphones increased, the malicious activity also increased. During the course of time, people started to repackage the legitimate app for their own benefit. They download apps from the official Android Market, repackage it and re-distribute to third-party marketplaces using reverse engineering tools (ApkTool, Dex2Jar, JD-GUI and Android SDK).

2. PROPOSED FRAMEWORK
Design consists of two parts. In first part, algorithms are used to detect repackaged smartphones applications while in second part, reverse engineering is done on detected apps in order to understand its effect and working. Figure 1 shows our proposed framework. In essence, our framework has four key steps. Android applications is a single .apk file which is written in java programming language [4]. We can decompile it using apktool using apktool d app_name.apk command. After decompilation a folder is generated consisting of four components i.e Res folder, Android Manifest File, Smali folder and YML file.

2.1 Fingerprint generation of the app
Once we have decompiled the app we open the smali folder to navigate the main .smali file. Calculating hash of complete file will check that either two files are same or not but it cannot express the similarity. We can apply similarity measurement like Jaccard [5] or edit distance [6] but calculation will be very expensive in terms of resources and time. In order to reduce the cost we are using context triggered piecewise hashing. We have used “Roll Hash” [7] to determine the boundary of each piece (segment). Then we have applied MD5 hashing on each segment. By concatenating the hashes of each segment we have generated the fingerprint. The algorithm used for fingerprint generation is given in Algorithm1.

Algorithm1: Fingerprint generation of the app
Input: Instruction sequence of the app
Output: fingerprint
Description: reset point (of segment), piecewise_hash , keyword(for reset point)
for all byte from sequence do
pos = rabin_karp ( keyword, sequence);
segment = text from start to pos of sequence;
sequence = sequence - text from start to pos;
piecewise_hash = md5(segment);
fingerprint = concatenate ( fingerprint, piecewise_hash);
end for
return fingerprint

Figure 1: An overview of our prototype

The approach for fingerprint generation used by DroidMoss [3] has also used piecewise hashing but the procedure is not clear. How they have taken the reset point and which algorithm is used to calculate hash value of each segment.

<Diagram: An overview of our prototype>
2.1 Similarity Score calculation and comparison

First we will calculate the edit distance of fingerprint of the downloaded app from the apps in database. For this a threshold, \( \theta \) (say) is defined. Edit distance of each pair (downloaded app and one app from database) is matched with this threshold. The pair for which edit distance is less than the threshold, that app is marked for further analysis. If the edit distance does not lie under threshold then the app is not a repackaged and it is new app.

2.2 Signature and permission matching

Now the signature of the marked pair is compared. There are only two possibilities either the signature of both app will match or not. If both app have the same signature then the downloaded app is the updated version of matched app from the database. Accordingly the database is updated with this new app. Otherwise, downloaded app can be re-packaged. In order to get surety we further compare the permissions used by the apps. If the permissions are altered then we conclude the app is re-packaged.

2.3 Analysis through reverse engineering

Reverse engineering is a challenging task because to understand the working of whole application, one has to check implementation of all the function s used manually. In order to automate the reverse engineering, fingerprint of an app is calculated as a concatenation of the fingerprints of all the segments. As we have mentioned above, the fingerprints of each segment is calculated by MD5 which gives a constant size (32 byte) output irrespective of the input size. A list is maintained for all the segments along with their fingerprints for both the apps. Now the fingerprint of each segment is matched against the fingerprints of all segments of genuine app. The matched segments are striked out from the list as described in algorithm 2.

Algorithm 2: Identification of unmatched segment

**Input:** Lists of fingerprints, A and B (of each segment) for both the app

**Output:** Unmatched segments

**Description:** size\((A) = n\), size\((B) = m\); list \(A\) and \(B\) contains \(SA[i], SB[j]*\); \(i(1)n; j(1)m\);

for \(i = 1\) to \(n\)

for \(j = 1\) to \(m\)

if \(SA[i] == SB[j]\) then Delete \(SA[i]\); Delete \(SB[j]\);

end if

end for

end for

return: list \(A\) and \(B\).

Now we are left with the unmatched segments. There are three possibilities: number of segments in re-packaged app is more, less or equal to the number of segments in genuine app. If the number of segments is more, less or equal than the number of segments in genuine app, it is concluded that some code has been appended, some code has been deleted or code is altered respectively. To identify the similar segments Jaccard coefficient (proximity) is calculated for each segment against the all segments of genuine app as given in algorithm 3.

Algorithm 3: Identification of similar segment

**Input:** Lists of segments, \(A\) and \(B\) for both the app

**Output:** list \(C\) of similar segments; \(C\) will have two columns one for segments of \(A\) and their corresponding segment having maximum proximity from \(B\).

**Description:** size\((A) = p\), size\((B) = q\); list \(A\) and \(B\) contains \(SA[i], SB[j]*\); \(i(1)p; j(1)q\);

for \(i = 1\) to \(p\)

max = Jaccard_coefficient\((SA[0], SB[0])\);

for \(j = 1\) to \(q\)

temp = Jaccard_coefficient\((SA[i], SB[j])\);

if \(temp > max\) then max = temp;

insert corresponding segments of max to \(C\)

end if

end for

end for

return: list \(C\).

This will reduce a lot of reverse engineering efforts, as we can directly jump to the pair of similar segments contained in list \(C\) and left segments in list \(B\) for reverse engineering.

3. CONCLUSION

In post-PC era smart phones and tablets have seize the market and it’s main attraction are apps. All apps are not genuine, some may be re-packaged of genuine apps and it may perform malicious operations. In this paper we have given a prototype to detect the re-packaged app and its malicious behavior. Our prototype uses piecewise hashing to localize the modified section of the app. Our prototype is also able to tell the amplitude of the modification of the detected segments by using Jaccard coefficient. This helps to automate the reverse engineering process. This will be useful for apps developers as well as app security practitioner.

4. REFERENCES