ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 4, Issue 2, pp: (1-5), Month: April - June 2016, Available at: www.researchpublish.com

Smartapp: An Efficient and Effective App Based on Watermark for Smartphone

[1]Amitha Babu, [2]Veena Jacob, [3]Annie Chacko

[1]IVth year Btech student, Dept. of CSE, MG University, MBCCET Peermade, Idukki, Kerala, India
[2]IVth year Btech student, Dept. of CSE, MG University, MBCCET Peermade, Idukki, Kerala, India
[3]Asst. Professor, Dept. of CSE, MBCCET Peermade, Idukki, Kerala, India

***Abstract*: The increased popularity and use of smartphones, has lead to the increase in the number of social networking applications that leads to increase in threats to apps. Among threats to apps repackaging of apps is the common threat. To overcome this, we propose an mechanism that use watermark for the android apps. Watermark can used to determine ownership of code and prevent tampering of source code. We embed a picture based watermark into the app using some events.When we need to verify an originality of an app, we should verify the watermark of an app, for that we should first extract it from the app and check with that of the watermark of the original app that the author supplies. To do this, we need to run the app with the secret input that the author supplies. This is simple and efficient mechanism.**

***Keywords:* Repackaged app, watermark, events.**

I. INTRODUCTION

Smartphones and mobile devices have become explosively popular for personal or business use in recent years. A large number and a wide variety of mobile applications (apps) have been developed and installed to extend the capability and horizon of mobile devices. These apps in return foster an emerging app-centric business model and drive innovations across personal, social, and enterprise fields. In this way, smartphone apps also play an important role. At the same time, the wide use of mobile apps has introduced serious risks. Particularly, app repackaging has been considered as a major threat to both app developers and end. Through app repackaging, malicious users can be reach the revenue stream and intellectual property of original app authors, and plant malicious backdoors or payloads to infect unsuspecting mobile users.

Recent studies have shown that app repackaging is a real threat to both official and third-party Android markets and regarded as one of the most common mechanisms leveraged by Android malware to spread in the wild. Investigation has also shown that app re-packaging presents serious vulnerability to mobile banking apps. In particular, one kind of threat called repackaging could add malicious payloads or advertisements to legitimate apps. Attackers like to repackage apps since very little effort is needed in this kind of attack.Typically, attackers only need to add the code payloads without any need to implement the whole apps .Recent landmark research shows that 10%~45% apps in Android markets are repackaged ones. Seriously, people usually cannot tell the difference between original apps and repackaged ones,which means attackers have minimal risks when repackaging apps. Current techniques try to detect similar apps in Android markets. Some use graph-based approaches [3], and some use hash-based approaches[5] .However, the detection rate could be lowered by obfuscation techniques and DexGuard. Obfuscation transforms the control flow and data flow, which makes it hard for current approaches to detect repackaged apps. When facing the low cost of repackaging and high-cost of identifying repackaged apps, we need an effective and efficient approach to identify the illegal repackaged apps, which can not only protect the ownership of original authors but also bring the safeness of users.

Embedding watermarks in Android apps could solve this problem. The embedded watermark can be used to identify the owner of apps [4]. By extracting watermarks from apps and comparing them, we can identify the repackaged pairs. There are two kinds of watermarks: static watermarks and dynamic watermarks. Static watermark embeds data or variables into apps. However, semantic preserving trans-formation could break them. Once the watermarks are broken, we cannot extract the right one from the target apps. Dynamic watermark uses runtime information as the watermark such as running paths and memory status. AppInk[2] embeds a graph-based dynamic watermark into Android apps.

By constructing a special list of objects as the watermark and using the distance between the objects in the list to represent a number, AppInk could embed any number as the watermark into an app. However, the code for constructing the graph is easy to be identified.To embed a robust watermark into Android apps, we need watermark code. The watermark code is not easy to be tampered by sematic preserving transformation (e.g., obfuscation). We do not consider the transformation that breaks the original semantics of apps since attackers would not like to let the repackaged apps run abnormally. The watermark code should not be easily identified, which means the code is nothing special while compared with other code in the apps. Note that careful attackers could perform data dependence analysis. They could identify the (watermark) code which does not have any relationship with the original code. Most of previous approaches embed watermark by adding code. This will make the watermark code independent of the original code and prone to be found by this dependence analysis. The watermark itself (generated by the code) is not easy to be identified. Otherwise, it is not difficult for attackers to identify and tamper the watermark.

We design a new kind of watermark called picture-based watermark, which meets the above three requirements. we do not use the static code or data as watermarks. Instead, we make use of the semantics of the code. In this way,no matter what obfuscation techniques are used, the watermark will not change as long as the semantics are preserved.We try to leverage the code and data structures that are in the original apps. We leverage a unique characteristic of pictures. That is, even if part of a picture is tampered, the picture could still be identified with high possibility. So we could combine some of the original code with the watermark code, which may change the picture a little. On one hand, this little change will impact the identification process of the picture-based watermark. On the other hand, the little change makes it difficult for dependence analysis. Note that the amount of the change here is controlled by us. It has no relationship with attackers. Attackers could also transform the code (e.g., obfuscation). However, they usually perform semantic-preserving transformations. No matter how much the code changed by these transformations, it will not impact our watermarks, and we do not use the commonly used formats for pictures (e.g., JPEG or BMP). Instead, we use ASCII to represent a picture. Then we store the picture in commonly used data structures, which makes it difficult to identify this kind of watermarks.

In a nutshell, after a developer provides a picture as a watermark, we transform the picture into ASCII characters. Then we embed some pieces of code into the app to generate the ASCII characters dynamically in memory. Once a watermark generated, it is ready for extracting. To extract the watermark, we run the app and locate the watermark using a *mark*. The mark is a random and secret number which is set up previously in the process of embedding the watermark. Attackers do not know anything about the mark. In this way, even if an app is repackaged (and obfuscated), we could still extract the watermark from the repackaged apps.To show the effectiveness of our approach, we implement a prototype called AppMark. It could automatically embed a picture into an Android app.

II. PROPOSED APPROACH FRAMEWORK AND DESIGN

There are several challenges when we embed pictures as special watermarks into Android apps. Pictures are easy to identify. They have their fixed file formats. For example, bitmap pictures use ”BM” as the first two bytes in the File headers. Attackers could easily scan the memory to find the pictures. Thus, we cannot directly store a picture into an app as a watermark. One could encrypt the pictures. However, this will make the code for encrypting pictures in the target Android apps look special, which is not difficult for attackers to identify. Note that, although it is necessary to make the pictures difficult to identify, we still need to extract the watermarks from apps to verify. And embedding a picture does not mean it is combined with original apps. We still need extra effort to combine the watermark code with the original code. We also need to combine the watermark objects in memory with the other objects of apps. We also need the format to be robust against tampering. Typically, we could use ASCII (i.e., letters, numbers .We design a system that overcomes the two above challenges. It consists of three main parts.

Watermark generation. Different from static watermarking which embeds a secret watermark object (e.g., a numerical value or a message string) into the code or data section of a target application, dynamic watermarking embeds a watermark object into special structures that present themselves only in the runtime of the target application. Different from other works, our watermark is a picture which is initially neither a sequence of numbers nor English letters. Since we do not want to use the original file formats of pictures, we need to transform the pictures. We first generate an ASCII sketch of a picture.

To make the watermark more robust, we divide the sketch into several parts. Different parts share redundant information. In this way, even if some parts of the sketch (watermark) are tampered, we can still know whether the watermark in an app is the same one as the one that original author supplies. Secondly, we automatically generate code that could draw the pictures in memory dynamically. Each part of the sketch is corresponding to a piece of code. The last step is to generate a piece of code for each part of a sketch. The generated code can create the sketch dynamically when the code gets executed.

we split a part of the picture to several pieces and use several arrays to represent those pieces. We also remove the characters for carriage return and line feed

in the array. In this way, it is difficult for attackers to rebuild the picture of the watermark without knowing the number of characters in a line. But for original authors, they know the number and can rebuild the picture to verify it.

Watermark Encryption. Encryption is the process of encoding messages or information in such a way that only authorized parties can read it. Encryption does not of itself prevent interception, but denies the message content to the interceptor. In an encryption scheme, the intended communication information or message, referred to as plaintext, is encrypted using an encryption algorithm, generating cipher text that can only be read if decrypted. For technical reasons, an encryption scheme usually uses a pseudo-random encryption key generated by an algorithm. It is in principle possible to decrypt the message without possessing the key, but, for a well-designed encryption scheme, large computational resources and skill are required. An authorized recipient can easily decrypt the message with the key provided by the originator to recipients, but not to unauthorized interceptors.

We need to embed the watermark code into original apps. In this step, we should guarantee all the watermark code to be executed. Otherwise, we cannot get the watermark in memory and cannot extract it for comparison. We design an event-based embedding to add the code into original apps, which is a two step approach. The basic idea is to get an execution path that is guaranteed to run. Then we insert the watermark code into the path.

Encryption Algorithm:

1. Read the input image (image).

2. Read the logo for water marking (im_logo).

3. Find out the size of im_logo.

4. Check if image is large enough to hold img_logo.

5. Convert the image into a single column vector.

6. Convert im_logo into a single column vector.

7. Apply bitand function with image.

8. Apply step 9-11 for for each bit of the eachpixel.

9. Calculating the linear index of the pixel to be changed.

10. Apply bitget function for each bit of pixel.

11. Apply bitset for each index.

12. Inserting a character cap (end of string).

13. For each bit of the character cap.

14. Calculating the index, Updating bits into template.

15. Reconstruct the watermarked image

Watermark decryption. When we need to verify a watermark of an app or when an app is suspicious to be a repackaged app, we should first extract it from the app and compare with one that original author suplies. To do this, we need to run the app with the secret input (i.e., a sequence of events) that the author supplies. Then we scan the memory and get the watermark. After extracting the watermark, we manually compare it with the one that the author supplies. Note that we do not require that the two watermarks are exactly the same. This is mainly due to the special characteristic of pictures (including ASCII pictures). That is, some different pixels in two pictures will not make the pictures different. In this way, even if some parts of a watermark are tampered, we could still identify the watermark, which protects the authorship of the original authors.

One challenge to extract watermark is to locate the array containing the watermark . If the app is not changed, we can identify and instrument the statement that operates on the watermark. However, since the app may be obfuscated, the statements in a method and names of variables might be changed. So we cannot directly locate the watermark.

Algorithm for decryption

1. Read watermarked image and key.

2. Convert the watermarked image into single column.

3. Apply randperm function for random variables.

4. Apply loop to find the character cap.

5. Using index find out the bit position using bitget function.

6. Extract the least significant bit.

7. Change combination of bit into character using and store into bit word.

8. Display the extracted image or text.

III. IMPLEMENTATION

We implement a prototype called SmartApp, which consists of three parts. The first two parts generate the watermark and embed it into Android apps automatically. The second part is an Android native library to extract the watermark from apps.It is implemented using java and mysqlLite codes.

Watermark generation. We do the sketch blurring by randomly adding ASCII bytes to a watermark. In the "blank" area in the watermark (the ASCII bytes are "0x20"), we could add more characters since they will not impact the original watermark when we compare two watermarks. In the area that is not blank, we add a small number of characters near the border of the picture. To generate a sketch part in the step of sketch dividing, we randomly mutate some characters which are not blank ("0x20") in the picture to blank. After repeating this process, we could get several sketch parts. Then we generate the code according to these sketch parts.

Watermark embedding. To get the execute trace of an Android app, we instrument every basic block by adding a statement in each basic block in an app. This statement only prints out a sequence number to mark the basic block. Then we run the app with a sequence of events as an input and get an execution trace. If the trace is too short, we randomly generate more events to the previous used input. In the trace, we remove the redundant basic blocks which execute more than one time. Then, we randomly add the watermark code to the basic blocks. After adding the code, we add the data dependence between watermark code and variables/constants by introducing some operations.

Watermark extracting. Since Android apps use Java which does not support direct memory access, we use native code developed by Android NDK. The native code could read the memory in user space. We use it to find the special mark that the original author leaves in the app. After locating the mark, the code dumps the memory with a given size of the watermark (original author supplies). Then the watermark is ready for comparison.

IV. RELATED WORK

Watermark. Developers embed watermarks into software to identify its owner. Watermark are of two types static and dynamic. Static watermarks [7] make use of original code and data. However, they are usually vulnerable to obfuscation such as semantic-preserving trans-formations. Dynamic watermarking mechanisms, including thread based, graph based and path based watermarking [6], are proposed to solve the upper problem. AppInk [2] embeds a graph based dynamic watermark into Android apps. Compared with these mechanisms, our approach embeds a picture based watermark into Android apps.

Java software protection: Android apps are mainly written in Java. Due to its high-level expressiveness, Java code is relatively easier to be decompiled and reversed than native code. To protect software written in Java, various solutions are pursued since its inception. One popular solution is to apply different levels of obfuscation to Java code, such as code or data layout obfuscation, control flow obfuscation, and string encryption. Watermarking is also used to prove the ownership of Java code and to discourage Java software piracy. SandMark[7] is one popular research platform to study how well different obfuscation and watermarking mechanisms work in protecting Java software.

Repackaged apps detection. With the growth of the apps, attacks on these apps are also increasing. One serious attack adds malicious payloads or advertisements to legitimate apps. These modified apps, called repackaged apps, share similar functionalities with the original apps, which makes them easily spread. To mitigate this attack, we embed watermarks into Android apps. Current techniques try to detect similar apps in Android markets. Juxt app uses k-grams of opcode sequences of Android bytecode and feature hashing to detects repackaged Android apps. Androguard also uses a hash based approach to compare apps. Centroid-based approach is usedto detect app clones in Android markets. DroidMOSS uses a fuzzy hashing technique to measure the similarity between Android apps and use it to detect repackaged apps.DNADroid [3] detects repackaged apps by generating and comparing program dependency graphs (PDGs) for methods in Android apps. In the PDG comparison, isomorphic analysis is used. An Darwin tries to avoid isomorphic analysis of graphs. It extracts a vector for each PDG component and uses locality sensitive hashing to compare different vectors. PiggyApp partitions the code of an Android app into primary and non-primary modules. Then it extracts several features from primary modules such as intent types. By comparing these features, it could detect "piggybacked" apps. Our approach adds watermarks to Android apps. By identifying the watermarks, we could detect repackaged apps.

V. CONCLUSION

App repackaging is a serious threat to Android ecosystem including app developers, app store operators, and end users. To prevent the propagation of unauthorized repackaged apps, we propose to adopt a dynamic picture based watermarking mechanism.

REFERENCES

[1] YingjunZhang,Kai Chen ,"AppMark: A Picture-based Watermark for Android Apps",*Eighth International Conference on Software Security and Reliability*, 2014,pages 58-67, Beijing, China.

[2] W. Zhou, X. Zhang, and X. Jiang, "Appink: watermarking android app for repackaging deterrence," in *ASIA CCS*. ACM, 2013, pp. 1–12.

[3] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," *Computer Security–ESORICS 2012*, pp. 37–54, 2012.

[4] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *Software Engineering, IEEE Transactions on*, vol. 28, no. 8, pp. 735–746, 2002.

[5] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *CODASPY*. ACM, 2012, pp. 317–326.

[6] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn,and M. Stepp, "Dynamic path-based software watermarking," *ACM SIGPLAN Notices*, vol. 39, no. 6, pp. 107–118, 2004.

[7] J. Hamilton and S. Danicic, "A survey of static software watermarking," in *Internet Security (WorldCIS), 2011 World Congress on*. IEEE, 2011, pp. 100–107.

[8] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Reverse Engineering, 1995*. IEEE, 1995, pp.86–95.