Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 78 (2016) 26 – 32

Procedia Computer Science

International Conference on Information Security & Privacy (ICISP2015), 11-12 December 2015, Nagpur, INDIA

Android Applications Repackaging Detection Techniques for Smartphone Devices

Sajal Rastogi[a], Kriti Bhushan[a], B. B. Gupta[a]*

[a]*Dept. of Computer Engg., National Institute of Technology Kurukshetra, Kurukshetra, 136119, India*

Abstract

The problem of malwares affecting Smartphones has been widely recognized by the researchers across the world. Majority of these malwares target Android OS. Studies have found that most of the Android malwares hide inside repackaged apps to get inside user devices. Repackaged apps are usually infected versions of popular apps. Adversaries download a popular Android app, and obtain the code using reverse engineering and then add their code (often malicious) to it and repackage and release the app. A number of techniques proposed in research and a number of commercial anti-virus products focus on detecting malwares. This is the traditional approach and requires a signature database. Zero day threats cannot be caught with such methods. There are many techniques which focus entirely on detecting repackaged apps. Since repackaged apps are in the majority among the infected Android apps, they can save the user from a large percentage of Android malwares. Detection and prevention of repackaging is also beneficial for original developer/publisher as they do not incur harm to revenue or reputation.
In this paper, we study in detail about some of the repackaging detection techniques. Mainly, there are two kinds of techniques - offline and online. They serve different purposes. An offline technique cannot be replaced by an online technique and vice versa. Offline techniques are for direct use of app market owner, whereas online techniques are for direct use of Android users. We study different offline and online techniques. These techniques use different features and metrics to detect similarity of apps and they are representatives of their category of techniques.

Keywords: Android; Smartphone security; repackaging; cloning; app similarity

* Corresponding author. *E-mail address:*gupta.brij@gmail.com

1. Introduction

Android is the most targeted smartphone OS. According to F-Secure, an incredible 97% of new mobile malware families are targeting Android[1]. In only the first quarter of 2014, 275 new Android threat families were identified by F-Secure[2]. The number of new threats identified for other smartphone OSs was ignorable compared to this figure. Studies[3,9]have made a very useful observation that most of the Android malwares, 86% of malwares as per[3], and 73% of malware families as per[9], use repackaged apps as the medium of propagation and installation. Repackaging an app with a malware is easy, and the popularity of original app helps the malware in infecting a lot of devices quickly. It has been found that many apps are repackaged to redirect the advertisement revenue from the original publisher to the adversary[12,17,20].

The existing techniques capable of detecting app repackaging can be classified as offline and online. Offline techniques are those that can be used for vetting app markets. Offline techniques detect repackaged apps among millions of apps from one or more market(s). Scalability becomes a more desirable trait for these techniques than accuracy. Online techniques are those that perform a significant part of their job on the user device. They usually detect whether an app is repackaged at the installation time. There may be some modifications that apps need to go through before installation for the online techniques to be effective. We discuss both kinds of techniques in this paper.

This paper is composed of the following sections. Section II introduces Android security, app repackaging, and the techniques to detect repackaging. In section III, we shed some light on Android OS, app repackaging, and app repackaging detection. Section IV discusses various techniques that claim to detect repackaging and highlights their unique features. Section V then presents the key takeaways from section IV. Finally, section VI conclude this paper and discuss some scope for future work.

2. Android app repackaging

During repackaging of apps, modifications can be made to the app by the adversary (plagiarist). These modifications performed may be one or more of the following: replacing of an API library with adversary owned library; redirecting the ad revenue of the app if the app uses some ads; adding some ads to the app; introducing malware code inside existing method(s); adding method/class specially for introducing malware code.

After the necessary modifications, the adversary can prepare a package (APK file) again. The adversary signs the app with her private key and the public key in the META-INF directory now corresponds to this private key. This app is now released on some unofficial market where the user fall prey to it.

Some repackaging detection/deterrence solutions assume that the adversary wants to exploit the popularity of the original app to infect a large number of users quickly. Thus, they work on the assumption that the metadata of the repackaged app is very similar to that of the original app. On the other hand, some solutions assume that the adversary is repackaging an existing app because she wants to save time/effort of creating a host app for the malware. In this case, the adversary can significantly change the metadata in her repackaged version. The only way to detect similarity in such cases is to compare the functionality/code of each and every pair of apps. The third possible case in which even the functionality is changed cannot be called repackaging.

3. Android app repackaging detection techniques

This section presents some of the better techniques that have been proposed by the researchers for detecting repackaged apps. An important thing to understand is that a technique does not have to be perfect. If a technique forces the adversary to apply many obfuscations/modifications, and makes the cost of repackaging high enough that the adversary makes no profit, then it is more than satisfactory.

3.1. AnDarwin

Crussell et al.[4] present AnDarwin, an offline tool. Scalability is a pre-requisite of any offline tool. Scalability is, indeed, the primary focus of the creators of AnDarwin. AnDarwin boasts of a sub-quadratic time complexity by using Locality Sensitive Hashing (LSH)[5] and Min-wise independent permutations locality sensitive hashing (MinHash)[6]. These hashing techniques make it possible to detect similar apps without actually comparing every pair of apps.

From methods in the source code of the app, AnDarwin constructs Program Dependence Graphs (PDGs) using only the data dependencies in the code. The data dependencies are much harder (and expensive) to obfuscate than the control dependencies. After PDG construction, corresponding to each connected component of each PDG, a semantic vector is constructed which captures information such as the type and frequency of different programming constructs present. Then, LSH uses many hashing functions to obtain clusters of semantic vectors which are near-neighbors. So the task of the later stages of AnDarwin is just to find similar apps inside a cluster, i.e., there is no need to compare apps belonging to different clusters.

3.2. AppInk

Zhou et al.[7] propose AppInk to embed a watermark in Android apps so that if an app does not carry a watermark or the watermark on it is not authentic then it can be found that it is a repackaged app. They point out that it is not easy to embed watermark in a Java code, and that too in an Android app which may have multiple entry points. They involve the developer in the process as the developer understands the semantic and syntactic structure of the code and she can choose the right places to insert the watermark in the app code. AppInk does not directly embed the watermark value into the source of the app. It is designed to convert this watermark value (string, number, etc.) into a non-trivial data structure (specifically graph) which is, in turn, transformed into Java code, called watermark code. Executing this code produces the instance of the data structure which corresponds to the watermark value. The authors point out that the recognition part of the watermarking scheme should be automated too. The recognizer part of AppInk extends Dalvik virtual machine (DVM) so that all the object reference relationships can be scanned (and logged) when the app under review runs, with manifest app providing the input events to the app. The logged files are searched for reference relationship patterns that can possibly correspond to a watermarking graph. The graph is then decoded to obtain the corresponding value and it can be verified whether it is the same as developer's watermark value.

3.3. APKLancet

Yang et al.[8] propose APKLancet which relies on DroidMoss for identifying malicious payload in the app. APKLancet does not maintain a signature database, nor does it identify the malicious payload itself. It uses AndroGuard for these tasks. After identification, it removes malicious payload. APKLancet makes an impractical assumption that malicious payload is always quite independent in the APK. There are more assumptions. The authors assume that, upon execution, the payload runs in a separate workflow. They also assume that the integration of malware and app code is reversible. It is not specified how does APKLancet decide whether an ad library or a plug-in is inserted by the original developer or the plagiarist. APKLancet is purifying the APK and re-packing it, but it is not specified how does it procures the developer's private key. If APKLancet uses a new key then the developer/publisher of the app would not be able to update the application (also any assumed sharing of resources with apps from the same developer would fail).

3.4. AppIntegrity

Vidas et al.[9] study a few tens of thousands of apps to reinforce the knowledge that most Android malwares use repackaging as the propagation medium. They use different tactics to ensure the diversity among the collection of apps they study and scan using VirusTotal[10]. For ensuring diversity in apps downloaded from unofficial markets, some of the tactics they use are: selecting the markets based on how popular they are on search engines instead of manual selection; searching for every string (mainly app names encountered in markets) on search engines in many languages; downloading even a non-APK file if decompressing it can give an APK file. For the official market, they: simulated valid *market sessions* to obtain AssetID automatically; used different values of the parameters which can differ with session, user-defined filters, network carrier, geographic region, device hardware, etc.; used backoff mechanism in automatic downloading because official market does not allow too many downloads in a given time for a given session. The market sessions are mandatory for downloading apps from the official market. The authors reverse-engineered the official market client on a smartphone (Play Store) to obtain the session parameters − username, password, device identifier, and SDK version. The parameters are then used in the simulated session so that official market cannot distinguish it from a valid session. The AssetIDs obtained in these sessions are then independently used to download the apps.

Authors also propose a technique to detect whether an app is repackaged or not but there is a challenge of distributing the original public key corresponding to app's private key. They propose to make it available on the web address corresponding to the package name of the app. This mechanism will fail because the plagiarist can simply change the package name and provide her own public key at the corresponding web address.

3.5. Centroid based Detection

Chen et al.[11] complain that there are no existing solutions for detecting app repackaging which are both accurate and scalable. Both are desired properties for an offline technique for vetting app markets. The authors propose the concept of a centroid which, they claim, achieves both the goals. For scalability, they apply a filtering based on comparing centroids of Control Flow Graphs (CFGs) of methods. A centroid is a geometric property of a CFG. Since CFGs have no physical reality, they need to be projected to a three-dimensional space so that the concept of centroid can be applied. The time complexity of obtaining this projection is linear in the size of the CFG. Three dimensions are assigned to every node in CFG to give the "coordinates". First dimension (sequence number) in the coordinate is assigned on the basis of the order of execution of nodes. For nodes which are branches of the same conditional statement, the order may be different for different executions. A static criterion, such as relative number of statements in nodes or binary values of these nodes, is used for branches. *These static criteria can be easily obfuscated by plagiarists using additive or subtractive attack.* Second dimension is the number of edges outgoing from the node, and the third is the number of loops the node is involved in.

The greatest shortcoming of their approach is the assumption that if the plagiarist is using additive or subtractive obfuscation, even then the control flow of the methods in the repackaged app will be identical or very similar to that of original app. It is easy for plagiarists to obfuscate the control dependencies in the code than data dependencies.

3.6. DNADroid

The first step of DNADroid[12] is to cluster the apps that appear together on search engine results. Solr[13] search tool is used by DNADroid for this step, which takes the meta-information (name, market name, publisher, description, etc.) of one app and returns the apps with similar meta-information. Mutually similar apps are put in one cluster. False negatives can occur in the first step itself if similar apps do not appear in search engine results. This will not be considered as DNADroid's shortcoming as repackaging is a social engineering attack and it is obvious that the plagiarist is not conducting the attack effectively. The repackaged apps which do not appear with the original app in search engine results will probably not be downloaded often anyway.

In the second step, a re-engineering tool is used to obtain the JAR files of the apps and fed to WALA[14] which returns one PDG corresponding to each method of app (JAR file). Only pairs of apps belonging to the same cluster are considered as potential repackaging pairs, therefore, only their PDGs are compared. The number of pairs to be compared in a cluster is quadratic in the total number of PDGs of the apps in the cluster. Before comparing PDGs, DNADroid applies a couple of filters which eliminate pairs that are highly unlikely to be repackaging pairs. Specifically, one filter removes small PDGs as they are not rich enough to characterize any functionality of their app and may lead to false positives. The other filter removes pairs in which PDGs differ in the type and the frequency of each type of PDG nodes. *Both the filters can actually help plagiarists in evading detection by DNADroid.* All that needs to be done is refactoring the methods in the repackaged app into many smaller methods and this will change the frequency of node types in the new, smaller PDGs.

In the third step, similarity between PDGs is determined using VF2 algorithm[15] based on subgraph isomorphism. The similarity of A to B tells how much of A's code is found in B. The similarity of B to A tells how much of B's code is found in A. This metric of similarity is asymmetric and has a significant benefit. If the plagiarist adds a lot of her own code to the repackaged version then the percentage of the repackaged app that can be found to be similar to the original app would be small. However, the percentage of the original app that would match the repackaged app would still be large.

3.7. Droidmarking

Ren et al.[16] propose a watermarking technique called Droidmarking, which offers a way to insert watermark non-stealthily and still deter repackaging attack. Their technique requires decryption of app code at runtime and they provide a native library to do that. The authors underscore the greatest drawback of stealthy watermarking − the limitation that the watermark recognizer cannot be released to general public and must remain limited to use for reputed app markets and other authorized parties. The reason that their technique is non-stealthy (making it possible to publicly release the recognizer) is that a value generated at runtime during a normal execution of the program is used as key for encrypting the watermark code. Encryption ensures that the watermark recognizer does not have to store the "undisclosed locations" where watermark code segments are hidden since there is nothing undisclosed now. In fact, the beginning and end of the encrypted segments is marked explicitly for easy decryption and for ensuring the integrity of these segments.

The encryption adds a one-time overhead to the execution of the app. The execution overhead is that the every encrypted segment, which holds some original source code along with the watermarking code, has to be decrypted the first time it is encountered during execution on user device. It is not encrypted again. The encryption also requires a small tweak to the Dalvik virtual machine (DVM). DVM should run in non-optimization mode so that so that encrypted segments of the app are not modified during bytecode optimization. Droidmarking also needs to ensure that the native library it needs for decryption should not be itself compromised. It performs integrity checks for that. Also it ensures that the call to the right decrypting function should be made and that too at exactly the beginning of the encrypted segment.

3.8. DroidMoss

Zhou et al.[17] propose DroidMoss. They scanned nearly 23,000 apps. They found 5% to 13% of the apps to be repackaged. Estimation of how much the repackaging attack is used was the primary purpose of the study. Their study also focuses on finding the reasons why repackaging is used and finds that repackaging is used mainly to reroute ad revenues. They say that markets need a vetting process.

DroidMoss can be used for vetting in markets by comparing the fingerprint of a newly introduced app with the fingerprints in fingerprint database of all the known apps. DroidMoss decompiles the dex file of the app to Dalvik bytecode, which is made up of opcodes and operands. It then removes the operands and retains the opcodes because the operands can be easily obfuscated. So, removing the operands is likely to increase accuracy. It also removes ad libraries because they will imply false similarity between apps. Additionally, the entire apk files are not used for detecting similarity between apps. Only the dex file is used and rest of the portions such as resource files are ignored to achieve robustness. They argue that they do it because a simple modification, such as an inserting extra resource file, is capable of changing the hash of such portions easily. It is more difficult to change the hash of Java code − but not much difficult.

Scalability is handled by extracting characteristic features of the apps and generate app "fingerprints". The instruction sequences in the bytecode of two apps are not compared directly to find similarity in those apps. That would make going through millions of apps very slow. Instead, much shorter fingerprints of apps are obtained. To obtain a fingerprint from an instruction sequence, first the sequence is divided into many pieces. Each piece independently contributes to the final fingerprint. The effect is that when the app is repackaged, only the fingerprint of those pieces which change during repackaging changes. Thus, similarity between the original app and the repackaged one can still be detected because hashes of certain pieces of their fingerprints are still identical. This is the main strength of DroidMoss that it localizes any changes introduced by the plagiarist. The accuracy of the technique depends, to a good extent, on how large are these pieces on which hashing is applied.

3.9. ViewDroid

Zhang et al.[18] propose ViewDroid, which detects similarity between apps by comparing the user interfaces of the apps. The visible features such as images, screenshots or even layouts of different screens are not compared which are prone to obfuscation. Rather, views (or activities) and their interactions are extracted from the smali code (the intermediate code between source code and Dalvik executable) in the form of a view graph. The view graphs are then compared. The analysis of smali code obtained after disassembling the APK file is accomplished statically.

The greatest hurdle in adoption of ViewDroid is, possibly, that it is unable to detect repackaging of apps with a small number of views (less than 10 views). Such apps form the majority in Android markets. Even seemingly large and complex apps, such as gaming apps, typically have very few views. Similarly, it cannot be applied on apps which run in the background and have no views.

4. Key Takeaways

We present some of the key characteristics we observed in the works we have examined. Table 1 mentions the metrics used by the offline techniques for detecting similarity between apps. Table 2 lists the strengths of the works.

Table 1. The metrics used by the techniques for detecting similarity between apps.

Work	Metric Used for Detecting Similarity
AnDarwin	Similarity based on Locality Sensitive Hashing is used first to restrict the number of app pairs to be compared and then to detect similarity between sets of features and sets of apps
APKLancet	Malwares, ad libraries, and plug-ins are recognized directly using a signature database
Centroid-based Detection	Difference between centroids of Control Flow Graphs determines similarity among methods; Asymmetric coverage that tells the number of methods of one app covered by the other determines similarity among apps
DNADroid	Degree of sub-graph isomorphism between Program Dependence Graphs determined using VF2 algorithm
DroidMoss	Compares hashes of segments of filtered bytecode of apps
IR-AR	(i) Jaccard index between symbol tables (ii) Euclidean distance between feature vectors (iii) Mutual coverage between feature vectors
ViewDroid	Degree of sub-graph isomorphism between view graphs determined using VF2 algorithm

Table 2. The key strengths of the works.

Work	Key Strengths
AppIntegrity	The study conducted to confirm the prevalence of repackaging in the real markets is one of the broadest studies conducted on the topic.
AnDarwin	It achieves sub-quadratic time-complexity in the comparison of apps.
APKLancet	-
AppInk	The hidden watermark code with unknown watermark value considerably increases the repackaging cost for adversaries.
Centroid-based Detection	The concept of centroid itself is the greatest strength of the work. The accuracy and efficiency with which the centroid captures the characteristics of the code is impressive.
DNADroid	It uses search engine results to cluster apps and then detects any possible repackaging between apps belonging to the same cluster only. It is a very efficient way of improving scalability.
DroidMarking	It is a non-stealthy watermark technique, thus, it makes it possible to release the watermark recognizer to the public.
DroidMoss	It performs hash-based fingerprinting of segments of bytecode of the app, independently. This independent hashing ensure that if changes are made to one part (segment) of the code then hash of only that part changes as the hash of other part is being still calculated separately.
ViewDroid	User interfaces are at the highest level of semantics and, therefore, hardest to obfuscate. Their obfuscation imposes very high cost on adversaries.

5. Conclusion and Future Work

Online detection techniques require some extra information in the apps, or they require some changes in the Android application framework or Dalvik virtual machine. All of them put some processing overhead on user device. However, in lieu of any market vetting procedures, they are the only thing that can protect the user from threats.

Offline techniques have to be highly scalable as they are supposed to be used for vetting markets. Some studies have found that there are unofficial app markets which almost exclusively host repackaged apps. This raises many question such as: Should the app markets be rated based on the percentage of infected/repackaged apps they host? If yes, who should perform the rating? Which techniques would be the best to rate the markets? Would users of Android collectively pay for the cost of such an expensive procedure such as by paying a little more for apps hosted on rated markets? How would the ratings be communicated to the user? Perhaps the most important question is: How many Android users know the state of the unofficial app markets?

We think that Google and other stakeholders in Android should come together to ensure the security of Android users. At the very least, a security framework should be established which lets the user know what kind of risks are involved when they install a certain app from a certain market.

References

1. 5 Reasons 97% Of All New Mobile Malware Is Targeting Android. [cited 2014 September 12]. Available from: URL: http://safeandsavvy.f-secure.com/2014/03/13/5-reasons-97-of-all-new-mobile-malware-is-targeting-android/.
2. F-Secure. Mobile Threat Report Q1 2014. [cited 2014 September 12]. Available from: URL: https://www.f-secure.com/documents/996508/1030743/Mobile_ Threat_Report_Q1_2014_print .pdf.
3. Zhou, Y., and Jiang, X. Dissecting Android Malware: Characterization and Evolution in the proceedings of the 33rd IEEE Symposium on Security and Privacy; 2012, p. 95-109.
4. J. Crussel, C. Gibler, and H. Chen. AnDarwin: Scalable Detection of Semantically Similar Android Applications.in the proceedings of the European Symposium on Research in Computer Security, Springer; 2012, p. 182-199.
5. A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions" in the proceedings of 47th Annual IEEE Symposium on Foundations of Computer Science; 2006, p. 459-468.
6. A.Z. Broder, M. Charikar, A.M. Frieze, and M. Mitzenmacher. Min-wise independent permutations.in the proceedings of the thirtieth annual ACM symposium on Theory of Computing; 1998, p. 327-336.
7. Z. Wu, X. Zhang, and X. Jiang. AppInk: Watermarking Android Apps for Repackaging Deterrence. in the proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security; 2013, p. 1-12.
8. W. Yang, J. Li, Y. Zhang, Y. Li, J. Shu, D. Gu. APKLancet: Tumor Payload Diagnosis and Purification for Android Applications. in the proceedings of proceedings of the 9th ACM symposium on Information, computer and communications security; 2014, p. 483-494.
9. T. Vidas, and N. Christin. Sweetening Android Lemon Markets: Measuring and Combating Malware in Application Marketplaces. in the proceedings of the third ACM conference on Data and Application Security and Privacy; 2013, p. 197-207.
10. Virustotal. [cited 2014 November 5]. Available from: URL: http://www.virustotal.com.
11. K. Chen, P. Liu, and Y. Zhang. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets.in the proceedings of the 36th International Conference on Software Engineering; 2014, p. 175-186.
12. J. Crussell, C. Gibler, and H. Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. in the proceedings of the European Symposium on Research in Computer Security Computer Security, Springer; 2012, p. 37–54.
13. Solr - ApacheLucen. [cited 2014 November 10]. Available from: URL: http://lucene.apache.org/solr/.
14. T.J. Watson Libraries for Analysis (WALA). [cited 2014 November 10]. Available from: URL: http://wala.sourceforge.net/wiki/index.php/Main_Page.
15. L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence 2004; 26(10), 1367-1372.
16. C. Ren, K. Chen, and P. Liu. Droidmarking: Resilient Software Watermarking for Impeding Android Application Repackaging. in the proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering; 2014, p. 635-646.
17. W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. in the proceedings of the second ACM Conference on Data and Application Security and Privacy; 2012, p. 317-326.
18. F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. ViewDroid: Towards Obfuscation-Resilient Mobile Application Repackaging Detection. in the proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks; 2014, p. 25-36.