Nexo Revista Científica
Vol. 33, No. 01, pp. 190-199/Junio 2020

ISSN-E 1995-9516
Universidad Nacional de Ingeniería

http://revistas.uni.edu.ni/index.php/Nexo
https://doi.org/10.5377/nexo.v33i01.10058

REPACKED ANDROID APPLICATION DETECTION USING IMAGE SIMILARITY

DETECCIÓN DE LA APLICACIÓN REEMPAQUETADA DE ANDROID USANDO SIMILARIDAD DE IMÁGENES

M.A. Rahim Khan*, R.C. Tripathi, Ajit Kumar

Lingaya's Vidyapeeth, Faridabad, Haryana 121002. India
*khan_rahim@rediffmail.com

(recibido/received: 07-enero-2020; aceptado/accepted: 15-marzo-2020)

ABSTRACT

The popularity of Android brings many functionalities to its users but it also brings many threats. Repacked Android application is one such threat which is the root of many other threats such as malware, phishing, adware, and economical loss. Earlier many techniques have been proposed for the detection of repacked application but they have their limitations and bottlenecks. In this work, we proposed an image similarity based repacked application detection technique. The proposed work utilized the main idea behind the repacking of application that is "the attacker wants to create fake application looking visually similar to the original". We convert each APK file into a grayscale image and then use perceptual hashing for creating a hash of each image. The string distance algorithms like Hamming distance was used to calculate the distance and searching for the repacked application. The proposed work also used distance calculation on binary features extracted from the app. The proposed work is very powerful in terms of detection accuracy and scanning speed and we achieved 96% accuracy.

Keywords: Repacked Android application; Image Representation; Perceptual Hashing String Distance.

RESUMEN

La popularidad de Android trae muchas funcionalidades a sus usuarios, pero también trae muchas amenazas. La aplicación de Android reempaquetada es una de esas amenazas, que es la raíz de muchas otras amenazas, como malware, phishing, adware y pérdidas económicas. Anteriormente se han propuesto muchas técnicas para la detección de aplicaciones reempaquetadas, pero tienen sus limitaciones y cuellos de botella. En este trabajo, propusimos una técnica de detección de aplicaciones reempaquetadas basada en similitud de imagen. El trabajo propuesto utilizó la idea principal detrás del reempaquetado de la aplicación que es "el atacante quiere crear una aplicación falsa que se vea visualmente similar al original". Convertimos cada archivo APK en una imagen en escala de grises y luego usamos el hash perceptual para crear un hash de cada uno. imagen. Los algoritmos de distancia de cadena como la distancia de Hamming se usaron para calcular la distancia y buscar la aplicación reempaquetada. El trabajo propuesto también utilizó el cálculo de distancia en las características binarias extraídas de la aplicación. El trabajo propuesto es muy poderoso en términos de precisión de detección y escaneo velocidad y logramos 96% de precisión.

Palabras clave: Aplicación de Android reempaquetada; Representación de imagen; Distancia porcentual de cadena de Hashing.

1. INTRODUCTIÓN

In recent years, the realization of the need for ubiquitous computing has boosted the development of the mobile application which can be noticed by the presence on millions of apps and their download counts in various official and alternative major Android app stores. The growth in mobile apps development created two parallel revenue models where one is controlled by benign business entities including developers and others by malicious attackers (Chien, 2011). The sale of different kind of services and advertisement is two main benign methods for revenue whereas attackers exploit these apps for various other monetary gains such as Premium SMS, malware-as-a-service, Pay-per-install, etc. The repacked app is a most used method for hijacking benign profits and committing other malicious act to achieve either financial or non-financial objectives. Repacked Android application is a major challenge to the Android ecosystem, according to Zhou and Jiang (2012), 86% of the detected malware is repackaged apps. According to Gibler et al., (2013), repacked apps have a share of 14% in the advertising revenue and 10% in user base. Recently, the Federal Trade Commission (FTC) releases Alert on Fake Apps for Mobile Devices and warns users regarding different type of Fake or repacked apps.

The App repackaging is one of most broadly utilized assault method of Android malware (Jiang & Zhou, 2013). The repackaging is a multi step successive process which can be either done manually or can be automated with various available tools. The steps involved in repackaging are 1) select and download a popular app from official or third party store 2)unpack and disassemble 3)inject malicious payloads 4)re-assemble and pack 5)sign repacked app with false details and 6)submit to targeted distribution point. Jung et al. (2013) has given a detail explanation of App building and how repacking exploit the building process for malicious usages.

Due to availability of various reverse-engineering tools, it becomes very easy to repack an Android application and self signing feature further boost the chances of repacking. The high monterey benefits and ubiquitous presence of mobile devices compelled attacker to create new mobile malware or port desktop malware for mobile platform. The Spitmo and ZitMo are example of ported versions of nefarious PC malware, i.e.,SpyEye and Zeus (Jiang & Zhou, 2013).

The proposed work utilized the main idea behind repacking of application that is “the attacker want to create fake application looking visually similar to the original". We convert each apk file into grayscale image and then use perceptual hashing for creating hash of each image. The string distance algorithms like Hamming distance was used to calculate the distance and searching the repacked application. The proposed work also used distance calculation on binary features extracted from the app. The proposed work is very effective in terms of detection accuracy and scanning speed and we achieved 96% accuracy.

1.1 Main Contributions

• We are first to incorporate image-based hashing for repacked application detection.
• Hashing along with other static meta-data based features improves accuracy of detection.
• Hashing technique is best in searching with O(1) time complexity hence our proposed approach is scalable.
• We evaluated our proposed work with different type of experiments to create a real world application market scenario and we found that our technique is giving a high accuracy and highly scalable.

Further section 2 gives an overview of literature available in the domain of Repacked App detection. The Section 3 discuss and present the proposed work. The section 4 provides details of experimental setup and dataset and related sub-process. At the end section 5 presents the results of various experiments and section 6 concluded the proposed work.

1.2 Related Works

The detection of repacked android app is known problem and there have been many works for detecting and preventing the widespread of repacked app. For example, the DroidEagle, is a methodology which uses the layout resources within an app to detect apps which are "visually similar*"* (Sun et al., 2015). *The DroidEagle has two sub-systems RepoEagle and HostE*agle which is use to scan and detect visually similar applications in apps repositories and host machine respectively.

Ali-Gombe et al. (2015) have proposed OpSeq that calculate similarity scores based on normalized opcode sequences and app permission requests. According to authors combination of structural and behavioral features creates a distinctive fingerprint for a given Android application and improve overall recall rate of OpSeq. It is a signature-based method works against obfuscation techniques but can only detect known malware.

DroidMOSS is a system to measure similarity by using fuzzy hashing technique to effectively localize and detect repacked applications. Authors (Zhou et al., 2012), performed a systematic study of six popular third party Android app market and found out that 5% to 13% of apps hosted on these third party marketplaces are repackaged to achieved various purposes such as stealing or re-routing ad revenues and injecting different kind of malware (Zhou et al., 2012).

In Linares-Vásquez et al., (2016), authors have proposed CLANdroid which uses Information Retrieval(IR) techniques and five semantic anchors: identifiers, Android APIs, intents, permissions, and sensors to detect similar apps. CLANdroid is mainly focused on detection of similar app which need not to be a repacked app, for example similarity is, searching similar apps for car booking service and repacking is distributing same game apps by changing developer's details or replacing advertisement channel code etc.

In Al-Subaihin et al., (2016), authors have proposed technique to measure app similarity based on claimed behavior. The Raw features were extracted using information retrieval method and then augmented with ontological analysis and used as attributes to characterize apps. The Agglomerative hierarchical clustering method were used to cluster the apps. The experiments were carried out on 17,877 apps mined from BlackBerry and Google app stores. The proposed method improves the existing categorization quality from 0.02 to 0.41 and from 0.03 to 0.21 for Blackberry and Google stores respectively. Zhongyuan et al. (2019) have used Context triggered piecewise hash (CTPH), which used twice (T-CTPH) to generate two fingerprint of each application to detect repacked application. Authors also optimize the hash similarity calculation algorithm which optimize the efficiency of process.

Apart from aforementioned literature works, Rastogi et al. (2016) have presented review on important techniques for repacked app detection. Table 1 summarized the related works on repacked Android detection techniques and presents observations.

It can be observed from the literature review that use of perceptual hashing for repacked app detection is novel and have not used earlier. It can also be observed that earlier methods have limitations and some are not specific for repacked app detection. The proposed work has address these issues and provides a solution for detecting repacked app with the help of image representation of apk file and applying perceptual hashing. The similarity is calculated using string distance methods such as Hamming distance.

Table 1. Observations on Related works

Works	Techniques	Observations
DroidEagle (Sun et al., 2015)	layout resources tree	Static method, Tree comparisons is costly
OpSeq (Ali-Gombe et al., 2015)	sequence of opcode and permissions requests	Static method, work only for known malware
DroidMOSS (Zhou et al., 2012)	Fuzzy hashing	Can't handle Obfuscation
CLANdroid (Linares-Vásquez et al., 2016)	Information Retrieval	Similar Apps
Al-Subaihin et al., (2016).	Clustering Method	App categorization

1.3 Proposed work

We have proposed a technique to detect repacked Android applications by the help of the ImageHash technique also termed as perceptual hashing. The app will be converted to its equivalent image representation and then a hash will be calculated for the query app. The hash value will be compared to the previously calculated hash of other apps. The searching with hash value is very fast and has only $O(1)$ time complexity but comparing one to many i.e. one query hash to many hashes many to many hash became a string comparison problem which makes the searching complex. We used the tree-based searching method to optimized our hash comparison. Figure 3 shows the block diagram of the proposed architecture. In this section, each component of the proposed architecture is explained in detail.

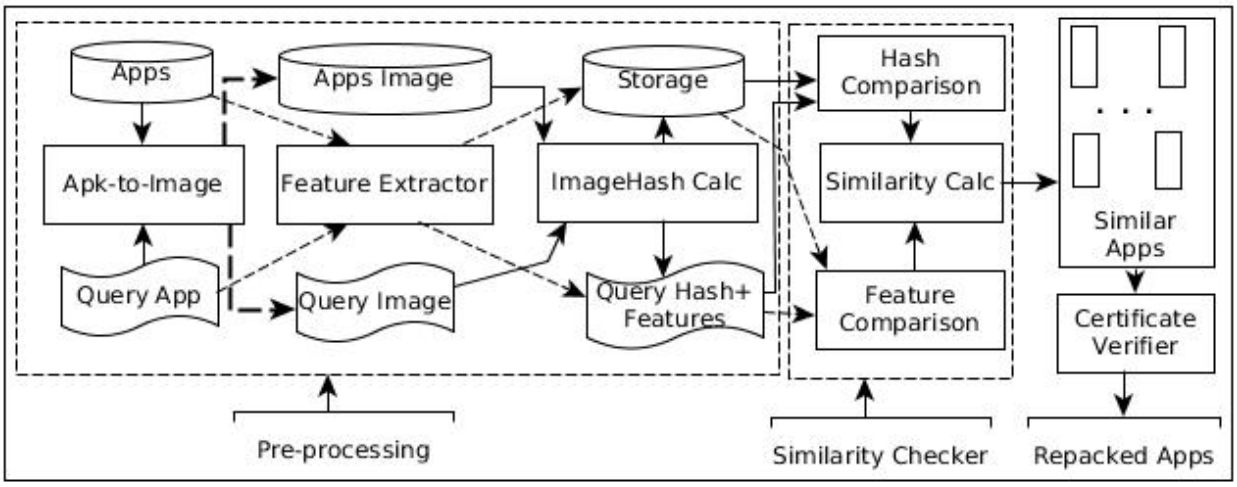
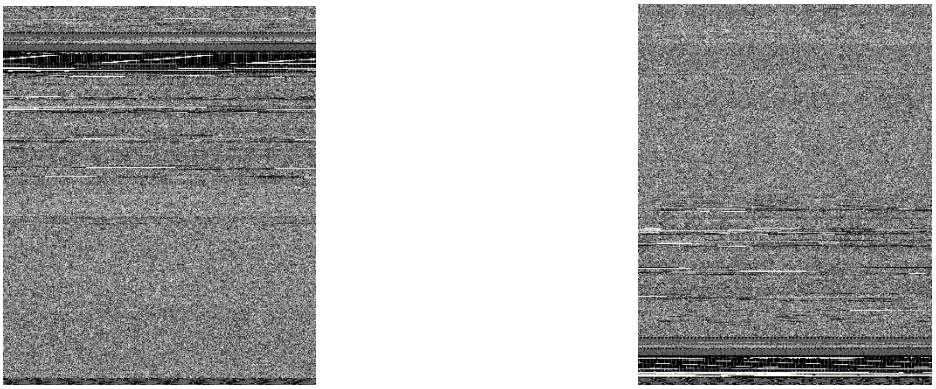


Figure 1. Architecture of Proposed work

1.4 Apk to Image conversion

Any digital file on the memory device is store as a stream of a bit of '0' and '1'. We read every apk file as binary stream and group every eight bits and store them to a new file with the image file extension. A group of eight bits will produce values between $0-255$ which is equal to the grayscale image's pixel value range. In this way automatically a apk is converted to a grayscale image which is first required to create an image hash. Due to vary apk size, the image dimensions will also be different, in our experiment we have fixed the width of image to 1024 and length vary accordingly. Figure 2 shows the converted grayscale image of an original (Ref. Figure 2a) and its repacked variant (Ref. Figure 2b). apk-to-image component takes target and query app as input and convert all to the grayscale image file and store them to use by further components.

The aforementioned approach is the adoption of Natraj et al. (2013) work, where each binary content of the executable is first numerically represented as a discrete one-dimensional signal by considering every byte value as an 8-bit number in the range $0 - 255$ then it "reshapedâ€• to a two-dimensional grayscale image (Natraj et al., 2013).



(a)Original (b)Repacked

Figure 2. Image of original and same repacked apk

1.5 Feature Extraction

In the domain of content-based information retrieval, image-hash is very accurate for image-based content search or many other applications. In proposed work, image is different than the normal image so we also considered static features of app as the filter and used for similarity calculation. These static features of the app is extracted by feature extractor (Ref. Figure 1) component of the proposed work. Feature extractor components also take target and query app as input and extract pre-defined features and store them to storage location along with image hash. Selected static features are the number of request permissions, file size, total resource count, layout strings, number of classes, number of methods and total API calls. All aforementioned features can static in nature i.e. these are extracted without running the apps. Each app will have a set of these static features and can be used as a filter to triage similar apps. A comparison of these feature sets would be one-to-many (in with respect of query app) or many-to-many (if the comparison is done on all samples to find similarities among them). A comparison of these sets is explained in further sub-section 3.6
We use the apktool1 tool for the unzipping of the apk files(apk files are set of Zipped DEX files). APK files are unzipped by using the apktool1 tool, that help to disclose the compiled Android app.

1.6 Image hashing

In this aspect, the perceptual hash algorithms are suitable to generates hashes for the comparative study of the data. Similarity in data represent The hashes have the folowing merits:

• Memory economy (saving hashes is cheaper than keeping the initial images)
• The economy of computing power (we can organize hashes into different structures such as KD-Tree, MVP-Tree, and others, and then perform the operations with them with complexity O(N), a linear one)

1.7 Hash comparison

Comparing images by the value of each pixel is quite time consuming, technically complicated and in general, doesnâ€™t make any sense. The estimation of complexity for such a comparison constitutes O(N2) Assume you have N images, firstly scan them and compute their fingerprints. If

you take 100% similarity as duplicate (which suggests duplications have identical fingerprints), we can find all duplications in O(N) time. If you define duplication as a similarity threshold (such as 90%), finding all duplications requires O(N*N) time.

1.8 Hash Distance Calculation

JaroWinkler distance is a measure of similarity between two strings. It is a variant of the Jaro distance metric which is a type of string edit distance. The higher the Jaroâ€“Winkler distance for two strings is, the more similar the strings are. The normalized score 0 equates to no similarity and 1 is an exact match.

$$d_j = \begin{cases} 0, & m = 0 \\ \frac{1}{3}\left(\frac{m}{|h_1|} + \frac{m}{|h_2|} + \frac{m-t}{m}\right), & otherwise \end{cases} \quad (1)$$

where, m is the number of matching characters. t is half the number of transposition.

Two characters from h_1 and h_2 respectively are considered matching only if they are the same and not farther than equation 2.

$$\left\lfloor \frac{max(|h_1|,|h_2|)}{2} \right\rfloor - 1 \quad (2)$$

Several transpositions (t) is defined as the number of matching (but different sequence order) divided by 2.

The Levenshtein distance is used to measure the difference between two sequences and very commonly used as a string metric. The count of the minimum number of single-character edits required to change one string into the other is used as the Levenshtein distance between two given strings. The three main single-character edits are insertions, deletions or substitutions. This distance metric is suitable for short string size but can also be computed and used for longer strings. The cost to computation (which is roughly proportional to the product of the two string lengths) for the short string is more feasible than the longer string.

The Hamming distance is the number of positions at which the corresponding symbols are different in two given strings of equal length. So in simple terms, to measure the Hamming distance, one has to count the minimum number of substitutions which will change one given string into the other string. Sometimes it is also counted as the minimum number of errors that could have converted one given string into the other string.

The important point of difference between Hamming distance and earlier discussed **Levenshtein distance** is that in Hamming distance only the substitution is allowed and so it can be only used to calculate the distance of two equal-length strings. In the proposed work, we are going to compare the distance of two hash of equal size.

For example, the Hamming distance of two binary strings (will have only zeros and ones) a and b is the total number of ones persent in the outcome of XOR operation on a and b. For example, if a = "11100011" and b = "11011010" , then a XOR b will be the string "00111001". There are four ones present in the result so Hamming distance between a and b is 4.

1.9 Static Feature Comparison

To improve the detection, we used static features such as permissions (type and counts), intents, classes, and API methods call to calculate the similarity between apps. These static features are used in literature for malware detection as well as repacked app detection. The result of static features based similarity further filter out the result of image hashing.

1.10 Similarity Calculator

The Similarity calculator takes input from hash comparison and static feature comparison components and calculates the final similarity score for apps shortlisted by both. The Similarity calculator considered both dimensions i.e. visual and static features for filtering repacked apps which result in controlling the false positive rate. Adopting two-level filtering at later stage results in higher accuracy than any of these individual filtering.

1.11 Certificate Verifier

Certificate Verifier is the last component of the proposed architecture and it makes the final decision for a repacked application. It takes visually similar apps as input and extracts the developer's signature from each and matches with the query app signature. The decision is being made on the assumption that similar apps should have the same signature and if very similar apps have two different signature reflect that one or other is repacked app.

2. EXPERIMENTS

Because any app store used to have a large number of a different class of applications, detection of repacked applications in these stores create two types of scenarios:

1. One-to-Many: A query app is given and we have to find all applications from the app store which are repacked versions of the given query app. This is a usual scenario and occurs every time a new app is submitted to the store and the store's moderator wants to verify its originality within its store.

2. Many-to-Many: Assuming all applications present in an official app store original (say such as Google Play), we have to find all repacked apps from one or many third-party app stores. This scenario is seldom but does occur when either third party store's moderator wants to cross-reference it apps originality with an official store or vis-versa.

In this section, we have explained the experimental steps and procedures which were carried out to validate our proposed repacked apps detection techniques under the aforementioned listed scenarios. We have also explained the dataset we have collected and processed for these experiments.

2.1 Dataset

We have crafted two datasets to satisfy both detection scenarios explained earlier. For one-to-many detection (here onwards dataset-I), first, we have collected 10,000 apps from various third-party apps stores and then we mixed some of the known repacked applications which were shared by Sun et al., (2015), and 10 repacked apps samples which were repacked by us. For all known repacked apps we also have an original app that was used as a query app. To create a dataset for many-to-many detection (here onwards dataset-II), first, dataset-I were randomly divided into two equal parts having 5000 apps in each, this represents two third party apps stores. To represent an official app store, we downloaded the top 5000 free applications from Google play. We have downloaded top and free applications, making because popular and free applications are the main target of repacked attacks. These two datasets are created to validate our proposed detection techniques on two important metrics that is accuracy and scalability which fall in line with many previous works (Sun et al., 2015; Ali-Gombe et al., 2015; Zhou et al., 2012). We also used a sample from AndroZoo repacked dataset, which has a pair (15297) of original and repacked app (Allix et al., 2016).

2.2 Experimental Setup

All the experiments were carried out in a system having Ubuntu 16.4, 64-bit OS running on Intel Core 2 Duo CPU E7400@2.80 GHz Ã— 2 processors with 8 GB primary memory and 1 TB secondary memory. All the tasks during experiments were implemented using Python programming language along with its various modules.

2.3 Implementation

The implementation of our proposed experiments having three main tasks, 1) crawling and downloading apps from the third party store and Google play store, 2) creating repacked applications for test set and 3) implementation of proposed techniques to detect repacked applications. Each of these tasks is explained in further subsections in detail.

Collecting apps: The app collection is a time-consuming task and so for the experiments, we have used publicly available datasets and have also written Python crawler to automate the app download from the app stores. AndroZoo have API for download so we obtained the api key from the maintainer and used to download the sample from their repository.

Repacking apps: Using a repackaging technique, source code can be added to the APK (Android Package) file. Resources can also be changed. The process is as follows:

1. Unpack an APK file with apktool1. It can decode resources and rebuild them after modifications.
2. Decompile the Java source code with JAD. JAD can extract source code from class files.
3. Modify the Java source and resources.
4. Rebuild the file using apktool.
5. Sign the code using jarsigner.

Repacked apps detections: The proposed repacked app detection is based on comparison of perceptual hashing. Each steps such as apk to image conversion, hash calculation, distance calculation and comparison of the proposed method were implemented as a python script.

3. RESULT AND DISCUSSION

To test the efficiency and effectiveness of the proposed work various experiments were carried out and this section present and summarize the results of the experiments.

3.1 One-to-Many Comparison

In this experiment, we have used dataset-I and perform the operations as explained earlier. We have converted the similarity value into a percentage (total string length and similarity value) so it can be used as a threshold. It was observed from the result that that higher threshold result in better accuracy for repacked app detection while lower threshold includes more app and hence decrease the repacked app detection. Table 2 summarizes the result of the one-to-many comparison with three different distance methods.

Table 2. One to Many Comparison with Different Threshold Value

Threshold	Hamming	Levenshtein	Jaroâ€“Winkler
and above	96.23	96.00	95.70
-95	94.57	94.23	93.00
and below	90.00	90.76	90.00

3.2 Many-to-Many Comparison

For performing many-to-many experiments, we have used dataset-II. Table 3 summarizes the result of the many-to-many comparison with three different distance methods. The time for comparison was larger than the one-to-many comparison (average 5 sec (1-m) 11 sec (m-m)).

Table3. Many to Many Comparison with Different Threshold Value

Threshold	Hamming	Levenshtein	Jaroâ€“Winkler
and above	94.80	94.00	93.60
-95	92.00	92.80	91.50
and below	86.00	87.60	86.0

4. CONCLUSIONS

In this proposed work a novel method i.e. perceptual hashing is used to detect the repacked android application. The proposed work converts each app to its equivalent image representation and calculates the hash of its image representation which is also known as perceptual hash and has been used for image retrieval. To detect an app is repacked or not string distance is used on the perceptual hash. With the help of two different experiments, it was verified that the proposed method is suitable for repacked app detection. The experiments achieved 96.00% and 94.00% detection accuracy for one-to-many and many-to-many comparison respectively. The time taken is acceptable but can be reduced by using other suitable comparison methods.

REFERENCIAS

Chien, E. (2011). Motivations of recent android malware. Symantec Security Response, Culver City, California.

Zhou, Y., & Jiang, X. (2012, May). Dissecting android malware: Characterization and evolution. In 2012 IEEE symposium on security and privacy (pp. 95-109). IEEE.

Gibler, C., Stevens, R., Crussell, J., Chen, H., Zang, H., & Choi, H. (2013, June). Adrob: Examining the landscape and impact of android application plagiarism. In Proceeding of the 11th annual international conference on Mobile systems, applications, and services (pp. 431-444).

Jiang, X., & Zhou, Y. (2013). A survey of android malware. In Android Malware (pp. 3-20). Springer, New York, NY.

Jung, J. H., Kim, J. Y., Lee, H. C., & Yi, J. H. (2013). Repackaging attack on android banking applications and its countermeasures. Wireless Personal Communications, 73(4), 1421-1437.
Sun, M., Li, M., & Lui, J. C. (2015, June). DroidEagle: seamless detection of visually similar Android apps. In Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks (pp. 1-12).

Ali-Gombe, A., Ahmed, I., Richard III, G. G., & Roussev, V. (2015, December). Opseq: Android malware fingerprinting. In Proceedings of the 5th Program Protection and Reverse Engineering Workshop (pp. 1-12).

Zhou, W., Zhou, Y., Jiang, X., & Ning, P. (2012, February). Detecting repackaged smartphone applications in third-party android marketplaces. In Proceedings of the second ACM conference on Data and Application Security and Privacy (pp. 317-326).

Linares-Vásquez, M., Holtzhauer, A., & Poshyvanyk, D. (2016, May). On automatically detecting similar Android apps. In 2016 IEEE 24th International Conference on Program Comprehension (ICPC) (pp. 1-10). IEEE.

Al-Subaihin, A. A., Sarro, F., Black, S., Capra, L., Harman, M., Jia, Y., & Zhang, Y. (2016, September). Clustering mobile apps based on mined textual features. In Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement (pp. 1-10).

Zhongyuan, Q. I. N., Wanpeng, P. A. N., Ying, X. U., Kerong, F. E. N. G., & Zhongyun, Y. A. N. G. (2019). An Efficient Scheme of Detecting Repackaged Android Applications. ZTE Communications, 14(3), 60-66.
Rastogi, S., Bhushan, K., & Gupta, B. B. (2016). Android applications repackaging detection techniques for smartphone devices. Procedia Computer Science, 78(C), 26-32.

Nataraj, L., Kirat, D., Manjunath, B. S., & Vigna, G. (2013, December). Sarvam: Search and retrieval of malware. In Proceedings of the Annual Computer Security Conference (ACSAC) Worshop on Next Generation Malware Attacks and Defense (NGMAD).

Allix, K., Bissyandé, T. F., Klein, J., & Le Traon, Y. (2016, May). Androzoo: Collecting millions of android apps for the research community. In 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR) (pp. 468-471). IEEE.

Li, L., Li, D., Bissyande, T. F. D. A., Lo, D., Klein, J., & Le Traon, Y. (2016). Ungrafting malicious code from piggybacked android apps. SnT.

APPENDIX

A malware writer could upload a malicious repackaged app to a web page or send it by email. Android users installing the app onto their smartphone wonâ€™t notice that it is a repackaged app because it looks and behaves just like the original one. 2

We propose a new repacked app detection approach that is both accurate and scalable. As assumption made in [Zhou et~al.(2012)Zhou, Zhou, Jiang, and Ning], we also assume that the signing keys from app developers are not leaked. This assumption will help to filter repacked app among visually similar app (hashes which are below the threshold edit distance value) considering that similar app will not be signed by two different developer's key.

The state-of-the-art on cloned/repackaged/piggybacked app detection does comparisons between apps, either directly based on source code similarity, or based on the distance between ex- tracted feature vectors. Such approaches thus re- quire the presence of the carrier apps in the dataset to work. This requirement is however unrealistic, since the piggybacked app may be in a different market than the original app (e.g., Google Play vs AppChina) (Li et al., 2016).

In practice however, many piggybacked apps are likely to be uploaded on different markets than where the original app can be found.

The main limitation in the state-of-the-art of piggybacked app detection is the requirement im-posed by all approaches to have the original app in order to search for potential piggybacked apps which use it as a carrier.

The problem of piggybacked app detection is closely related to that of Software clone detection in general, and repackaged app detection in particular. Our work is also relevant to the field of malware detection in the wild.

DroidMoss or DNADroid have focused on per-forming pairwise comparisons between apps to compute their degree of similarity.

[2] https://www.virusbulletin.com/virusbulletin/2012/05/mobile-banking-vulnerability-android-repackaging-threat