Detecting Repackaged Android Applications Using Perceptual Hashing

Thanh Nguyen
Dept of Computer Science
School of Computing
University of South Alabama
tnn1001@jagmail.southalabama.edu

J. Todd McDonald
Dept of Computer Science
School of Computing
University of South Alabama
jtmcdonald@southalabama.edu

William Bradley Glisson
Cyber Forensics Intelligence Center
Dept of Computer Science
Sam Houston State University
glisson@shsu.edu

Todd R. Andel
Dept of Computer Science
School of Computing
University of South Alabama
jtmcdonald@southalabama.edu

Abstract

The last decade has shown a steady rate of Android device dominance in market share and the emergence of hundreds of thousands of apps available to the public. Because of the ease of reverse engineering Android applications, repackaged malicious apps that clone existing code have become a severe problem in the marketplace. This research proposes a novel repackaged detection system based on perceptual hashes of vetted Android apps and their associated dynamic user interface (UI) behavior. Results show that an average hash approach produces 88% accuracy (indicating low false negative and false positive rates) in a sample set of 4878 Android apps, including 2151 repackaged apps. The approach is the first dynamic method proposed in the research community using image-based hashing techniques with reasonable performance to other known dynamic approaches and the possibility for practical implementation at scale for new applications entering the Android market.

1. Introduction

The need for mobile security increases as consumers migrate toward mobile devices as their main form of communication. Research indicates that mobile devices are not only being introduced into legal context, but they are also being used to profile individual activities and as a proxy for cloud activities [1-5]. Coupling this information with market statistics indicating that the number of downloaded applications will reach 258.2 billion by 2022 [6] emphasizes corporate mobile device security concerns [7, 8].

Furthermore, as Figure 1 indicates, Android Operating System has been the dominant mobile OS globally standing at around 85% of the world's mobile market for the recent past and is expected to remain steady through 2023 [9]. Thus, attacking the Android OS is attractive and potentially lucrative for adversaries. Figure 2 illustrates trends in Android malware samples, reported in a 2019 Nokia Threat Intelligence Lab report [10]. The report indicates there are nearly 20,000,000 Android malware samples, increasing 31% from the previous year signifying that Android is the target platform of choice for malicious applications.

The emergence of repackaged applications has been an issue for several years and continues to be visible in the news [12, 13]. These applications are the source of both pirated software and malicious code because they utilize reverse engineered Android code of existing apps to form the basis of a new app, but with some newly inserted code. Both academicians and practitioners are voicing their concerns on this issue [14, 15].

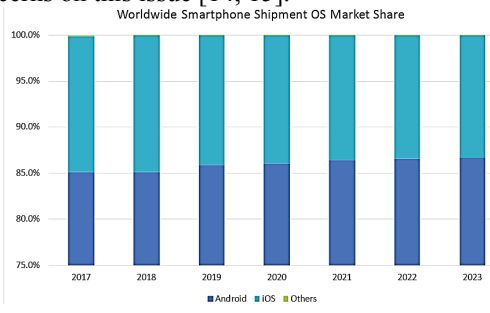

Figure 1: Smartphone Market Share [9]

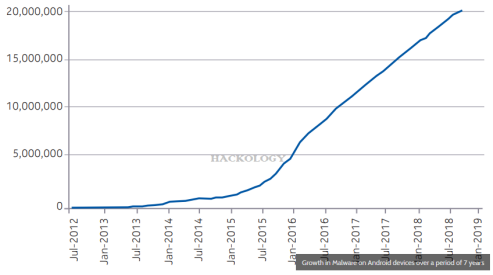

Figure 2: Android Malware Growth [11]

On the industry side, Trend Micro in 2014 issued a report highlighting several instances of cybercriminal activity targeting industries such as banking, gaming, and communications [14]. The study detailed: 1) South Korean banking apps with Trojanized versions used for phishing attacks, 2) the malicious repackaging of the classic Flappy Bird app (which was downloaded 50 million times), and 3) over 100,000 downloads of the repackaged version of Blackberry Messenger from the Google Playstore—one of the most popular instant messaging apps [14].

More recently, Nguyen et al. [15] demonstrated the complete ineffectiveness of commercial and open-source anti-virus Android apps in detecting a malicious repackaged application (Snapchatz). Li et al. [16] in a 2019 survey-based study also highlighted the shutdown by federal authorities of alternative Android market places because of app plagiarism (repackaging or cloning) [17]. The same survey [16] also recounted reports by Ustwo Games that only 5% of the Android installations for one of its more popular games were legitimate (as in paid for, and not repackaged or cloned).

Academic researchers have observed the emergence of repackaged applications into the market for some time, with a study by Zhou and Jiang [18] reporting that 80% of Android malware is actually repackaged. The predominance of repackaged apps and their effect on the Android user community has thus spurred considerable study, as researchers continue to look for better methods for detecting repackaged applications and techniques for code cloning or reuse detection. As Li et al. [16] report in a recent survey, a six-year window has produced 57 different proposed approaches, most of which involve the use of static techniques based on code similarity or heterogeneity. Static techniques are most negatively affected by code obfuscation and could be completely defeated by packing whereby the code base is encrypted until execution. Likewise, most of the studied techniques in Li et al.'s [16] survey do not scale to millions of apps or are evaluated on private datasets.

This paper presents a novel method for detection of repackaged applications that can integrate seamlessly into current application screening approaches such as Google Bouncer or with third-party markets. The approach is dynamic and relies on visualizing component behavior of normal, vetted applications and comparing those to behaviors of submitted applications to a marketplace. The unique aspect of this approach relies on the emulation of Android apps so that their user interface interactions are exposed. During emulation, screenshot images are captured and then compared against known good applications. Novel use of visual (perceptual) hashing is incorporated into the framework whereby similarity of application screenshot images can be computed. A malicious, repackaged app that has UI images with too much similarity to known UI images becomes the basis for detection.

The remainder of this paper details further background about the state of the art in repackaged app and code cloning detection in Section 2. Section 3 describes the perceptual hashing methodology, of which three different versions of the algorithm are compared for effectiveness. Section 4 details the experimental approach and data sets used, while Section 5 discusses the experimental results. As a contribution, this research provides: 1) The first known method to leverage UI image analysis as a detection basis, completely avoiding the need for code or data examination; 2) A dynamic detection approach based on average hashing with comparable performance to other known dynamic techniques, producing 88% accuracy on a publically available data set; and 3) A scalable method and framework which could be easily integrated into marketplace vetting approaches, providing an alternative approach to exiting static triage techniques.

2. Background

The influx of mobile devices and applications is pressing the need for mobile security research. Android applications are deployed in an Android Package Kit (APK) format, which relies on traditional ZIP compression [19]. Repackaging is a predominant threat because malicious reverse engineering of APK files is relatively easy given standard open source tools that are readily available [16]. In the traditional attack model, repackaged apps are used primarily to insert ads that redirect advertisement revenue [20, 21] or to insert malicious code on top of benign code that will be spread by unsuspecting end-users [22].

Furthermore, researchers have investigated the use of code cloning as a means to identify malware families in repackaged apps by detecting similar

functions or methods used by previous malware generations [16]. Both areas of research (repackaged and code cloning detection) are briefly summarized in the following sections. Malware detection approaches are broadly categorized as static and dynamic, whereby static approaches use parts of the APK file itself without running the application and dynamic approaches require some type of sandbox or emulation environment to execute the Android app for collection of data [15].

2.1. Repackaged Application Detection

Repackaged applications remain a common source for Android malware distribution because of the ability of the malware to hide dormant in the background attached to fully functional applications [16]. This provides stealth for the malware as there is no indicator of malware being installed for the end-user. Hence, researchers are investigating methods for detecting repackaged applications as revealed by a 2019 study [16]. This study documents a systematic literature review of repackaged detection research. Table 1 provides a summary from the reported results of the study, which covered 57 papers from 2012-2017 related to Android cloning, plagiarism, reusing, piggybacking, camouflaging, and repackaging.

Table 1: Research Summary [16]

Approach Category	Papers	Dynamic
Similarity computation	42	4
Runtime monitoring	5	5
Supervised learning	5	1
Unsupervised learning	4	0
Symptom discovery	1	0
Total	57	10

As Table 1 indicates, Li et al. [16] divide repackaged detection approaches into five categories. By far, the majority of research has focused on static similarity analysis of bytecode or resource files included in an Android APK. The proposed dynamic technique described in our paper is unique from all prior approaches studied in Li et al.'s [16] review because it uses runtime emulation to generate images of the Android app during dynamic execution. Of the published studies using similarity computation and supervised learning [16], five used dynamic techniques that included analysis of layout group graphs, HTTP distance, user interfaces [23], system call sequences, and runtime API invocations. Of the runtime monitoring techniques (which are all dynamic), the analysis covered execution traces, virtualized protection, package naming, and watermarking [24, 25]. Watermark approaches [24, 25] differ in that they introduce prior known data either in the manifest file or in the code itself that can be recovered by execution or real-time examination of the application: the proposed perceptual hashing technique described in this paper does not require alteration of original APKs.

User interface analysis proposed by Soh et al. [23] also differs from the perceptual hashing approach described in this paper because it translates screen activity to a corresponding XML format and then analyzes XML data. Soh et al. [23] also point out that obfuscation preserves the semantics of I/O behavior and thus user interface look-and-feel. The appeal of repackaged apps is likewise that they actually look like some original, well-known app [15]. A contribution of our perceptual hashing research also points to the feasibility of generating dynamic data based on input generation, which Soh et al. [23] observed was a hard problem to automate.

Other prior research points to the applicability of the perceptual hashing method described in this paper as a potential triage approach for realistic marketplace environments. For example, Lindorfer et al. [26] developed AndRadar, a framework for crawling third-party markets and detecting malicious applications. The researchers exhaustively crawled 16 third party markets between the time frame of June and November 2013. Findings indicated that the majority of third party market applications are ad-aggressive applications and contained a fair number of malicious applications. Out of 20,000 crawled applications, 1,500 were found to be malicious in behavior.

Zhou et al. [27] performed a systematic study on six Android third-party marketplaces, and their findings indicate that it is common to find repackaging of legitimate applications. Based on this, the researchers proposed DroidMOSS, an application similarity system utilizing fuzzy hashing techniques to produce a similarity score. Fingerprints, in this case, were based off feature extraction techniques that utilize meta-data inside of the Android application. Results indicate that 5 to 13% of apps hosted on these third-party websites were repackaged [27]. Furthermore, manual investigation indicated that the majority of these applications were changed with ads to redirect end-users to targeted sites that generate revenue for the adversary. The other major threat, of course, is that additional code can include other implanted malicious code, backdoors, or ransomware.

Static detection techniques also include utilizing APK application resource files [28] that require pairwise application comparison. Researchers found that the Google Play store has application similarity of around 10.31% and third-party market such as androiddrawer have similarity percentage of around

16.16% based off 550,000 application analyzed [28]. AnDarwin [29], a scalable framework that analyzes Android applications for plagiarism, was used to analyze 265,359 applications in various markets and identified 36,106 as repackaged or rebranded applications with 88 new variants of malware discovered.

2.2 Code Cloning and Reuse Detection

As Tian et al. [30] point out, similarity-based detection for repackaged malware faces quadratic complexity for the number of apps analyzed, making such techniques less appealing for large-scale screening. Past published research on general-purpose static Android malware detection has included techniques that analyze permissions, code hashes, API dependencies, control flow patterns, Android intents and activities, resources, and even code entropy [16]. Tian et al. [30] observe the reason many techniques are not fully effective is that analyses are performed on the entire app, which is some mixture (normally 80% or more) of the original code with some additional malicious code or manipulated advertising. Presence of benign code in repackaged apps can thus dilute features generated by malicious code, which results in high false negatives or missed detection, and researchers have found that a majority of false negatives are caused by repackaged malware [31].

Code cloning and reuse detection provide an alternate static means to identify repackaging, and relevant work is highlighted here for completeness, all of which are referenced in the study by Li et al. [16]. Bari et al. [32] define code cloning as the coping and modifying of a code block. Su et al. [33] denote the challenges in detecting code clones based on behavior analysis. Ideally, we want to be able to detect not only static code cloning but also code blocks that do not match but operate in the same manner. In its current state, behavioral code clone detection utilizes functional equivalence of inputs and outputs to classify code clones. The researchers [33] argue that advances in dynamic code clone detection can increase general behavioral code clone detection: thus this research provides a new technique with potential for future study that can enhance ongoing research in clone detection.

For mobile code cloning, Juxtapp [34] provides indicators of buggy code, evidence of significant code reuse, or code blocks, which are instances of known malware. Results from 58,000 Android apps indicate there were 463 instances of confirmed buggy code reuse and 34 instances of known malware instances and pirated variants of paid apps. NiCad [22] is a near-miss clone detector that functions by extracting the Java source code from apps to create code signatures. Results show that NiClad can detect 95% of previously known malware clones and pinpoint them to certain malware family based on clone detection. ViewDroid [35] profiles interactions between users and apps (called view graph) to deal with the problem of obfuscated code in repackaged applications. View graphs thus capture user navigation behavior and generate birthmarks that could then be compared against candidate apps: the proposed method in this research essentially represents the analysis of birthmarks as well, but the birthmark is fully captured in real use interface images. Tian et al. [30] focus on detection using code heterogeneity features to improve the performance of traditional methods and achieved a low false negative rate of 0.35% when evaluating malicious apps and a false positive rate of 2.96% when evaluating benign apps.

3. Perceptual Hashing Methodology

Dynamic detection avoids many of the issues with static analysis. However, all dynamic techniques require the execution of the target code. Dynamic approaches, like the perceptual hashing approach evaluated in this research, overcomes the difficulty of obfuscation, encrypted files, and virtualization. This research specifically focuses on the usefulness of the approach to detect repackaged apps, and so is not evaluated against all categories of Android malware.

Due to the extensive data set of images inside an APK, it is difficult to compare and fingerprint each image. Cryptographic hash functions can be used for integrity, checking the exact duplicates of an image [36]. However, a slight change in a bit in the data can cause major variation in the hash. This is called the avalanche effect, where a change in a small set of data causes a dramatic change in the hash [36]. Thus, it is not possible to distinguish images similar to one another. For example, changes in size, rotation, pixel modification, etc. will cause the hash to change completely.

A perceptual hash is a fingerprint for multimedia to derive various features of its content [37, 38]. Unlike cryptographic hash functions which rely heavily on slight changes in the media to produce an entirely different hash, perceptual hashes produce “close” hashes depending on the amount of change involved. Thus, a cryptographic hash function is used to measure whether two images completely match, whereas a perceptual hash generates a fingerprint that can be compared using hamming distance to measure image similarity. Figure 3 illustrates a perceptual

hash on a traditional image (3-a) as well as a screenshot of an Android mobile app (3-b). In both cases, an original image is reduced in size to some standard dimension (8 x 8 pixels for example) and then recolored typically to grayscale. Further calculations are done on these reduced images to produce perceptual hash fingerprints.

Thus, perceptual hashes can be used as a metric to determine image similarity. The primary three perceptual hashes that are widely used are average hashing, perception hashing, and difference hashing. **Average hashing** [37, 38] is the simplest form of a perceptual hash. The method involves taking a picture and changing it to grayscale and reducing the picture to an 8x8 pixel size. Each pixel is then used to generate an average pixel value. Lastly, the hash is generated by comparing each pixel value to the average to generate a 64-bit hash. **Difference hashing** is similar to average hashing, but it relies on gradient change rather than average pixel greyscale for generating the hash [39]. The algorithm starts by converting the image to grayscale. Next, the image is downsized to an 8x8 square of gray values. The row hash is calculated for each row from left to right. An output of 1 is generated if the next grayscale value is greater than or equal to the previous one. Otherwise, a 0 bit is calculated if the value is less than. Finally, the bits are concatenated to generate the final hash.

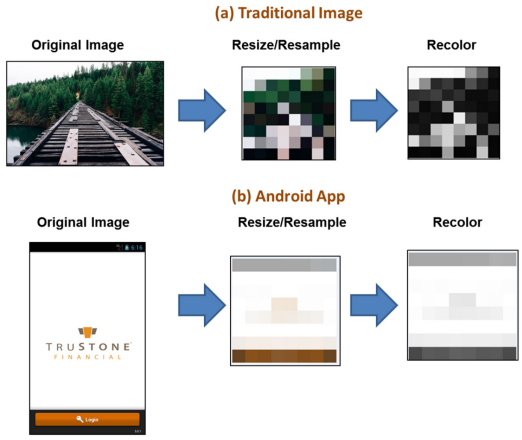

Figure 3: Example Perceptual Hash Images

Perception hashing relies on a discrete cosine transform (DCT) instead of the average greyscale [39]. The algorithm operates in the following manner:

1. Reduce Image Size: The image is reduced to a 32x32.
2. Reduce Color: The image is converted to grayscale to simplify computation.
3. Compute the DCT: The DCT separates the image into a collection of frequencies and scalars.
4. Computing Average Value: The DCT mean is calculated.
5. Further reduce DCT: Each bit of the 64-bit hash is set to 0 or 1 depending on whether each of the 64 DCT value is above or below the average value.
6. Construct Hash: The bits are concatenated to make a 64-bit hash.

4. Experimental Data and Approach

The approach in this research will apply perceptual hashes to dynamic screenshots of Android apps that are extracted via a custom processing engine. The goal of the experiments is to validate the effectiveness of perceptual hash (pHash) algorithms in identifying fake repackaged apps when the original application has been previously recorded [38]. A dataset of 576 trusted applications from the Google Play Store was selected and verified that they are virus-free using VirusTotal. Once the application set is crawled from the Play Store, it is processed using an Android VM Sandbox that supports dynamic screenshot captures of its UI components. Furthermore, the certificate and version info are extracted from the APK. This app collection forms the baseline images for further evaluation. Figure 4 depicts the experimental environment created for this research.

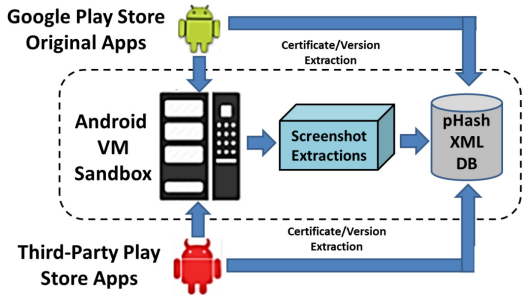

Figure 4: Image Collection Environment

Each trusted (benign) application is processed through the sandbox environment for analysis, which exercises the application and creates one or more screenshots. Captured screenshots are processed through the perceptual hash function and stored in a JSON database along with the application's APK, version info, and certificate. These screenshots of the UI components act as the dataset for repackaged and clone detection.

The test dataset consists of pairs of 2,151 repackaged apps obtained from AndroZoo [40]. AndroZoo is a growing collection of Android APK collected from various sources that contain pairwise original apps and a repackaged app version. The

original app becomes part of the trained benign data set. The repackaged app is used to test the effectiveness of different versions of the perceptual hash algorithm.

4.1 Architectural Framework

Dynamic analysis is performed by utilizing a sandbox environment. Specifically, the research leverages the capabilities of DroidBox [41] to fit in architecture for conducting perceptual hash experiments. DroidBox was developed to offer dynamic analysis of Android application and offers a wide range of capabilities, including hashes for analyzed packages, network traffic, and file read/write operations among others. This research leverages the sandbox part of the DroidBox emulator image that automates the installation and startup process for Android applications. Other emulators could have been chosen, such as Android's own AnroidStudio [19] or more popular PC-based applications such as Archon, Bliss, Bluestacks, or Droid4x [42], to name a few. DroidBox was chosen for its ease of setup in Linux environments, ease for automating dynamic launching of APKs, ability to connect via virtual networking, and its rich set of analysis tools geared for capturing network traffic, file read and write operations, information leaks, and circumvented permissions [41]. The native tools provided by DroidBox were envisioned to provide additional data extraction capabilities for correlating detection of repackaged apps, but future work will focus on use of other emulation possibilities as DroidBox is no longer being updated.

DroidBox required patching to allow for screenshot capabilities and execution of the pHash functions to create signatures of the application's layout. Furthermore, Droidbox only allows for one application to be run at a time. To extend this capability, a Docker container was created to run the patched DroidBox. This Docker container, in conjunction with a batch script, allowed the automation of the dynamic analysis for Android apps from the test set of benign and infected apps.

Figure 5 provides a deployment diagram of the execution platform used to conduct experiments. A Dell Precision T5600 Workstation with 128 GB of memory and 2.30 GHz, Intel Xeon processor, was utilized for all experiments. Ubuntu 16.04 operating system installation was used as a Docker container that executed DroidBox and an Android emulator.

The architecture required five different steps to generate experimental data. First, DroidBox came prepackaged for the Android 4.1.1 release. Thus, to insert new functionality into the script, a custom Python script was developed to patch DroidBox to provide an experimental version for this research.

Step two of the process involved the creation of a Docker container that is used to execute DroidBox. This consisted of creating a DockerFile that specified the build instructions for the container which included: 1) Ubuntu 14.04 environment, 2) commands for *tinydb*, *pillow*, and *imagehash* libraries, 3) *Android SDK* and *Android 4.1.2* emulator, 4) SSH to forward logs and traffic to the emulator system, and 5) a VNC library to allow control of the emulator screen. The third step involved running the DroidBox Docker container to automate mass, dynamic analysis of APKs.

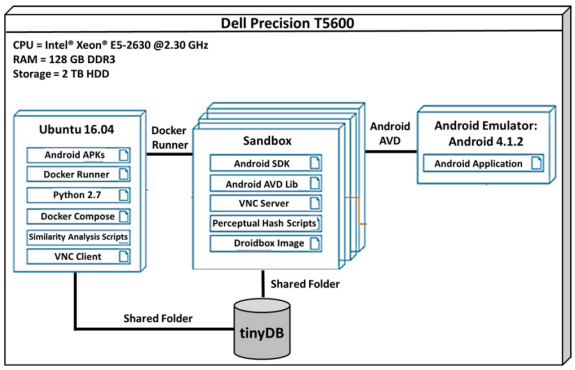

Figure 5: Experimental Architecture

In step four, generated perceptual hash data. During runtime of the sandbox, *adb tool* and *monkey tool* were utilized in DroidBox for getting status updates of the Android emulator. To begin image collection, the *monkey tool* instantiated the APK and timed screenshots were taken with the *adb* command. Images were stored on the host machine every five seconds. The ImageHash [43] library was utilized to generate the average hash, perception hash, and difference hash. The hashes were then inserted into a centralized *tinydb* database in JSON format.

As the fifth and last step before analysis of image hash data, the data were pre-processed to ensure reliability. Hashes were obtained for the main android display screen and the error screen if an application crashed. All three databases were then scanned for images matching the two-pre-processed hashes. This ensured that the dataset does not contain any error samples. Repackaged applications that were not processed in the benign data set were removed to ensure valid training pairs for an accurate prediction.

4.2 Evaluating Perceptual Hash Functions

To calculate the effectiveness of each perceptual hash function, the best threshold for the hamming distance is calculated to provide the highest accuracy

in detecting matching images in the image data set. First, the repackaged data set is processed against the trained benign dataset that has a matching corresponding APK. The accuracy of each hash algorithm is then generated with its corresponding hamming distance between 1 and 16. For purposes of the experiment, anything above a hamming distance of 16 was considered too big of a distance and indicated a mismatch. This process was repeated for the set of benign applications not in the training set (downloaded from Google Play store) so that a true negative assessment could occur. Table 2 provides a summary of terms, definitions, and calculations used in reporting accuracy measurements in Section 5 (TP, TN, FN, FP, TPR, FPR, TNR, ACC).

Table 2: Accuracy Measurements

TP	True positives: Repackaged app correctly classified
TN	True negative: App correctly classified as benign
FP	False positive: Benign app classified as malware
FN	False negative: Repackage app misclassified
TPR	TP Rate = **Sensitivity** = TP/(TP + FN)
FNR	FN Rate = FP/(FP + TN)
TNR	TN Rate = **Specificity** = TN/(TN + FP)
ACC	**Accuracy** = TP + TN/(TP + TN+ FP + FN)

5. Experimental Results

The experimental phase consisted of evaluating the three different proposed types of perceptual hashing algorithms: average hash, PHash (perception hash), and DHash (difference hash). Effectiveness in detecting repackaged apps relies on similarity scoring that assesses the closeness of dynamically visualized app layouts. In total, 2,151 benign apps paired with 2,151 repackaged apps along with 576 benign apps were analyzed. A set of screenshots for the first one minute of the application run-time were generated for each application. The evaluation criteria for measuring effectiveness includes calculation of the optimal hamming distance for the most accurate detection of repackaged applications while also having a low false-positive rate in benign applications.

5.1 Average Hash Results

Average hash produced the best results out of the three perceptual hashing algorithms. Figure 6 displays the results for the average hash with corresponding hamming distances. The results showed that TPR increases as hamming distance increased until a breakpoint of 10. Furthermore, a faster drop in TPR is seen with an increase in hamming distance of 14. TNR for benign apps showed its best results at a lower hamming distance. The results indicate that at a hamming distance of 1, there were 72 apps that matched 100% with a benign app in the data set. This can be due to an application taking a basic template or form layout from the Internet. As hamming distance increased, TNR steadily decreased as well, with more applications being falsely detected. From the analysis, a hamming distance of 10 provides the most accuracy for detecting repackaged applications using the average hash algorithm.

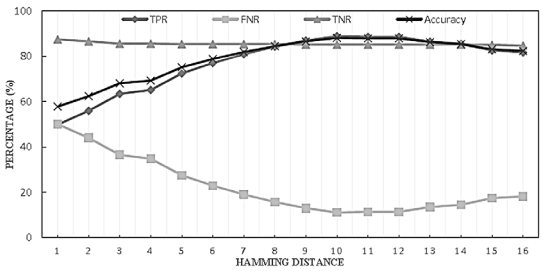

Figure 6: Average Hash Accuracy

5.2 Perception Hash

Results show that the perception hash produced the least accurate results for repackaged detection. Figure 7 displays the results for perception hash with its corresponding hamming distance. The analysis shows that perception hash follows the same trend as an average hash. The increase in hamming distance correlated with an increase in TPR until a critical point at a hamming distance of 11 and 12. TNR followed the same trend as in average hash. As hamming distance increases, TNR steadily decreases. The analysis shows that a hamming distance of 11 or 12 gives the best accuracy for TP and TN.

One potential reason for the poor performance of this algorithm may stem from the fact than an image that matches a wrongly identified benign sample pair is classified as a false negative. In addition, each screenshot contained the Android OS top and bottom taskbars. These may have potentially increased the similarity of each sample pair. Thus, this algorithm-generated many more pairs of applications that were similar to the repackaged APK, causing a higher false-negative rate.

5.3 Difference Hash

Figure 8 displays the results for the difference hash. Difference hash works very similar to the average hash but utilizes its neighbors' gradient for computation of its hash [37, 38]. Thus, it was

expected to show similar results to the average hash. However, the results were relatively close to the accuracy of perception hash. This is due to the accuracy of the algorithm. The algorithm is expected to find more similar benign apps that are not in its designated repackage pair; thus, this counts as a false negative and resulted in a lower TPR. Furthermore, the gradient of the android main interface is very static and dark in nature already. This may result in a higher similarity score for each image comparison. TNR follows the same trend of decreasing in relations to the increase of hamming distance.

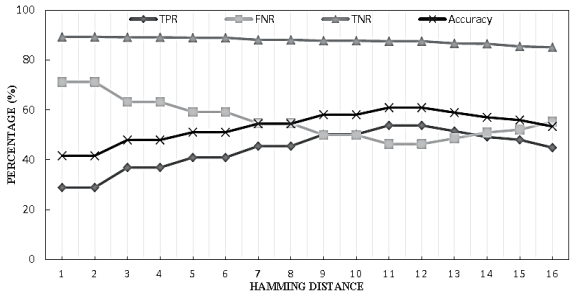

Figure 7: Perception Hash Accuracy

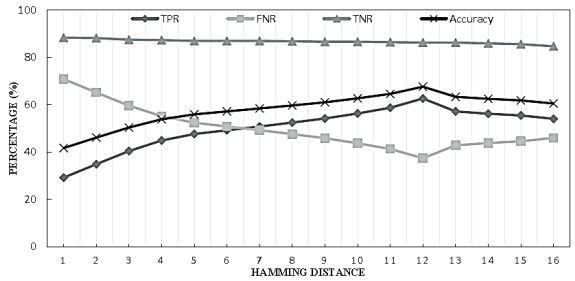

Figure 8: Difference Hash Accuracy

5.4 Comparative Summary and Evaluation

Based on the results, the average hash is the most effective perceptual hash algorithm for detecting repackaged app. Figure 9 summarizes the overall findings of the experiment. To increase the effectiveness and accuracy of future iterations, we will take into account cropping out the main UI. Furthermore, there is an application that utilizes a URL to load form layout, which tends to lower the TPR. A repackaged app can simply change the URL to a designated website and completely change the whole form layout. Future research using perceptual hashing of UI images will need to take this into account.

Figure 10 displays the receiver operating characteristic (ROC) curve for the three approaches, where transition points correspond to hamming distances of 1-16. As a whole, the average hash approach had the highest number of hamming distances with the highest TPR and lowest corresponding FPR. For all three approaches, hamming distances of 10-12 produced the highest accuracy, highest TPR, and highest TNR.

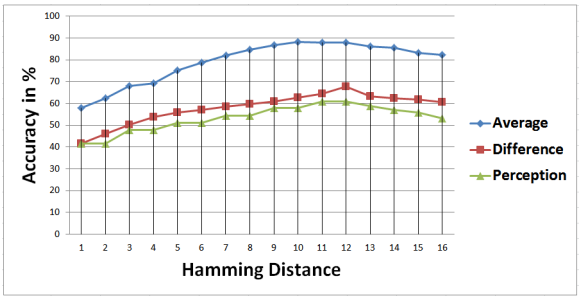

Figure 9: Algorithm Accuracy

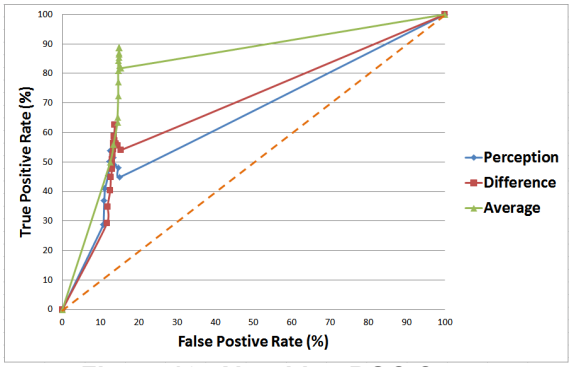

Figure 10: Algorithm ROC Curve

In comparing the perceptual hashing approach to other dynamic approaches, the user interface analysis proposed by Soh et al. [23] reported results on a dataset of 521 paired apps (benign and a repackaged version), whereas we examined 2,151 pairs and an additional 576 benign apps. Researchers reported an FNR as low as 0.8% [23] whereas the best perceptual hashing approach (average hash) generated an FNR as low as 11%.

6. Conclusions and Future Work

The primary goal of this research is to evaluate and implement a scalable and dynamic analysis technique for sandboxing Android applications. The primary goal of the sandbox environment supports generating and extracting live screenshots of Android apps, generating result outputs as perceptual hashes, and storing them in a database. To this end, the research successfully implemented a scalable Docker container. Our final dataset consisted of 2,151 benign app pairs, 2,151 repackaged app pairs, and 576 benign apps. Our findings indicate that average hashing produced the best accuracy rate compared to perception and difference hashing. The findings show

that with a hamming distance of 10, it is able to match the repackaged app to its benign pair with an accuracy rate of 88.16%.

The experimental framework and results show the practical possibility of implementing a perceptual hashing approach using available tools, software, and emulation environments. In particular, the use of image hashing as a basis for high accuracy classification appears to be a viable starting point for future triage techniques. Future work will focus on better pre-processing of screenshot images to eliminate universal similarities across all Android apps such as the main UI features and elimination of form layouts that can be configured by malicious repackaged apps. To keep up with rapidly changing Android AVD versions, future work will also consider more adaptable sandbox environments beyond DroidBox and the potential use of PC-based analysis tools and sandbox emulators for image generation and capture.

7. References

[1] Grispos, G., W.B. Glisson, J.H. Pardue, and M. Dickson, *Identifying User Behavior from Residual Data in Cloud-based Synchronized Apps.* Journal of Information Systems Applied Research, 2015. **8**(2): p. 4-14.

[2] Berman, K., W.B. Glisson, and L.M. Glisson. *Investigating the Impact of Global Positioning System (GPS) Evidence in Court Cases*. in *Hawaii International Conference on System Sciences (HICSS-48)*. 2015. Kauai, Hawaii IEEE

[3] Grispos, G., W.B. Glisson, and T. Storer, *Chapter 16 - Recovering residual forensic data from smartphone interactions with cloud storage providers*, in *The Cloud Security Ecosystem*, R.K.-K.R. Choo, Editor. 2015, Syngress: Boston. p. 347-382.

[4] Grispos, G., W.B. Glisson, and T. Storer. *Using Smartphones as a Proxy for Forensic Evidence contained in Cloud Storage Services*. in *Hawaii International Conference on System Sciences (HICSS)*. 2013.

[5] McMillan, J., W.B. Glisson, and M. Bromby. *Investigating the Increase in Mobile Phone Evidence in Criminal Activities*. in *Hawaii International Conference on System Sciences (HICSS-46)*. 2013. Wailea, Hawaii: IEEE.

[6] Statista. *Number of mobile app downloads worldwide in 2017, 2018 and 2022 (in billions)*. 2017. https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/. Date of Last Access: 09/03, 2019.

[7] Glisson, W.B. and T. Storer, *Investigating information security risks of mobile device use within organizations.* arXiv preprint arXiv:1309.0521, 2013.

[8] Glisson, W.B., T. Storer, G. Mayall, I. Moug, and G. Grispos, *Electronic retention: what does your mobile phone reveal about you?* International Journal of Information Security, 2011. **10**(6): p. 337.

[9] IDC. *Smartphone Market Share*. 2019. https://www.idc.com/promo/smartphone-market-share/os. Date of Last Access: 06/15, 2019.

[10] Hackology. *Cyber Threat Intelligence Report 2019* 2019. https://blog.drhack.net/threat-analysis-report-2019-android-malware-wins/. Date of Last Access: 06/15, 2019.

[11] Dogtiev, a. *App Download and Usage Statistics* 2018. http://www.businessofapps.com/data/app-statistics/. Date of Last Access: 06/15, 20109.

[12] Micro, T. *Malware in Apps' Clothing: A Look at Repackaged Apps*. 2014. https://www.trendmicro.com/vinfo/us/security/news/mobile-safety/malware-in-apps-clothing-a-look-at-repackaged-apps. Date of Last Access: 09/03, 2019.

[13] Lakshmanan, R. *'Agent Smith' malware replaces legit Android apps with fake ones on 25 million devices*. 2019. https://thenextweb.com/security/2019/07/10/agent-smith-malware-replaces-legit-android-apps-with-fake-ones-on-25-million-devices/. Date of Last Access: 09/03, 2019.

[14] Paper, A.T.M.R. *Fake Apps*. 2014. https://documents.trendmicro.com/assets/wp/wp-fake-apps.pdf. Date of Last Access: 09/03, 2019.

[15] Nguyen, T., J.T. McDonald, and W.B. Glisson. *Exploitation and Detection of a Malicious Mobile Application*. in *Proceedings of the 50th Hawaii International Conference on System Sciences*. 2017.

[16] Li, L., T.F. Bissyande, and J. Klein, *Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark.* IEEE Transactions on Software Engineering, 2019.

[17] Rashid, F.Y. *Feds Seize Alternative Android App Markets For App Piracy*. 2012. https://www.securityweek.com/feds-seize-alternative-android-app-markets-app-piracy. Date of Last Access: 09/03, 2019.

[18] Zhou, Y. and X. Jiang. *Dissecting android malware: Characterization and evolution*. in *2012 IEEE symposium on security and privacy*. 2012. IEEE.

[19] Guide, A.D. *Developer Guides*. https://developer.android.com/guide/. Date of Last Access: 09/03, 2019.

[20] Gibler, C., R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. *Adrob: Examining the landscape and impact of android application plagiarism*. in *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. 2013. ACM.

[21] Crussell, J., C. Gibler, and H. Chen. *Attack of the clones: Detecting cloned applications on android markets*. in *European Symposium on Research in Computer Security*. 2012. Springer.

[22] Chen, J., M.H. Alalfi, T.R. Dean, and Y. Zou, *Detecting android malware using clone detection.* Journal of Computer Science and Technology, 2015. **30**(5): p. 942-956.

[23] Soh, C., H.B.K. Tan, Y.L. Arnatovich, and L. Wang. *Detecting clones in android applications through analyzing user interfaces*. in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. 2015. IEEE Press.

[24] Ren, C., K. Chen, and P. Liu. *Droidmarking: resilient software watermarking for impeding android application repackaging*. in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 2014. ACM.

[25] Zhou, W., X. Zhang, and X. Jiang. *AppInk: watermarking android apps for repackaging deterrence*. in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. 2013.

[26] Lindorfer, M., S. Volanis, A. Sisto, M. Neugschwandtner, E. Athanasopoulos, F. Maggi, C. Platzer, S. Zanero, and S. Ioannidis. *AndRadar: fast discovery of android applications in alternative markets*. in *DIMVA*. 2014. Springer.

[27] Zhou, W., 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*. 2012. ACM.

[28] Zhauniarovich, Y., O. Gadyatskaya, B. Crispo, F. La Spina, and E. Moser. *Fsquadra: fast detection of repackaged applications*. in *IFIP Annual Conference on Data and Applications Security and Privacy*. 2014. Springer.

[29] Crussell, J., C. Gibler, and H. Chen. *Andarwin: Scalable detection of semantically similar android applications*. in *European Symposium on Research in Computer Security*. 2013. Springer.

[30] Tian, K., D.D. Yao, B.G. Ryder, G. Tan, and G. Peng, *Detection of repackaged android malware with code-heterogeneity features.* IEEE Transactions on Dependable and Secure Computing, 2017.

[31] Elish, K.O., X. Shu, D.D. Yao, B.G. Ryder, and X. Jiang, *Profiling user-trigger dependence for Android malware detection.* Computers & Security, 2015. **49**: p. 255-273.

[32] Bari, M.A. and D.S. Ahamad, *Code Cloning: The Analysis, Detection and Removal.* International Journal of Computer Applications, 2011. **20**(7): p. 34-38.

[33] Su, F.-H., J. Bell, and G. Kaiser. *Challenges in behavioral code clone detection*. in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2016. IEEE.

[34] Hanna, S., L. Huang, E. Wu, S. Li, C. Chen, and D. Song. *Juxtapp: A scalable system for detecting code reuse among android applications*. in *DIMVA* 2012. Springer.

[35] Zhang, F., H. Huang, S. Zhu, D. Wu, and P. Liu. *ViewDroid: Towards obfuscation-resilient mobile application repackaging detection*. in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. 2014. ACM.

[36] Stinson, D.R., *Some Observations on the Theory of Cryptographic Hash Functions.* Designs, Codes and Cryptography, 2006. **38**(2): p. 259-277.

[37] Bin, H., *A Study and Analysis on a Perceptual Image Hash Algorithm Based on Invariant Moments.* Sensors & Transducers, 2013. **159**(11): p. 337.

[38] pHash. *pHash: the Open Source perceptual hash library.* . http://www.phash.org/. Date of Last Access: 06/15, 2019.

[39] Hoyt, B. *Duplicate image detection with perceptual hashing in Python*. 2017. http://tech.jetsetter.com/2017/03/21/duplicate-image-detection/. Date of Last Access: 06/15, 2019.

[40] Allix, K., T.F. Bissyandé, J. Klein, and Y. Le Traon. *Androzoo: Collecting millions of android apps for the research community*. in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 2016. IEEE.

[41] Desnos, A. and P. Lantz, *Droidbox: An android application sandbox for dynamic analysis.* Lund Univ., Lund, Sweden, Tech. Rep, 2011.

[42] Authority, A. *14 best Android emulators for PC and Mac of 2019!* 2019. https://www.androidauthority.com/best-android-emulators-for-pc-655308/. Date of Last Access: 09/03, 2019.

[43] *Imagehash*. https://github.com/jenssegers/imagehash. Date of Last Access: 06/15, 2019.