A Novel Technique for Detection of Repacked Android Application Using Constant Key Point Selection Based Hashing and Limited Binary Pattern Texture Feature Extraction

MA Rahim Khan [1†] **and** *Manoj Kumar Jain* [2††],
khan_rahim@rediffmail.com manojjain@lingayasuniversity.edu.in
Department of Computer Science & Engineering,
Lingaya's Vidyapeeth, Faridabad, Haryana 121002, India

Abstract
Repacked mobile apps constitute about 78% of all malware of Android, and it greatly affects the technical ecosystem of Android. Although many methods exist for repacked app detection, most of them suffer from performance issues. In this manuscript, a novel method using the Constant Key Point Selection and Limited Binary Pattern (CKPS: LBP) Feature extraction-based Hashing is proposed for the identification of repacked android applications through the visual similarity, which is a notable feature of repacked applications. The results from the experiment prove that the proposed method can effectively detect the apps that are similar visually even that are even under the double fold content manipulations. From the experimental analysis, it proved that the proposed CKPS: LBP method has a better efficiency of detecting 1354 similar applications from a repository of 95124 applications and also the computational time was 0.91 seconds within which a user could get the decision of whether the app repacked. The overall efficiency of the proposed algorithm is 41% greater than the average of other methods, and the time complexity is found to have been reduced by 31%. The collision probability of the Hashes was 41% better than the average value of the other state of the art methods.

Keywords:
App Repacking, Android Malware, Obfuscation, KeyPoint Selection, Limited Binary Pattern, Collision probability.

1. Introduction

Smartphones and other portable devices have become essential in ones' life. Due to the affordability and the huge functionalities, the Android devices have occupied 78.2% of the total share in 2019 [1] . Mobile-based applications or in short "Mobile Apps" holds a significant part in the Android functionality. At present, there exist around 1 million applications [2] in the Google Play store. Unlike the 'Apple's applications that can be distributed only through their stores, Android apps can be distributes through many third-party websites and stores that do not have a comprehensive review and pre-checks. This makes the hackers and the malicious app developers to easily breed the true applications by repacking with additional coded and functionality. As per the recent cyber threat report, Android counts for almost 97% of all threats happening in mobile devices, which can cause a severe damage to the technological ecosystem of Android developers and users.

Android malwares typically consist of Trojans, Adware, and Fake apps. Almost 85% of the malwares are made by the repacking technique, a process where the official apps are disassembled, and malicious codes or additional functionality is injected so as to gain money from ads and also to steal personal data [3]. By making use of the repackaging, the hackers perform many unethical activities such as data theft, unwanted ads posting and cracking of payment functionality and can replace the original 'developer's I.D. Also, the malware developers can make a replica of bank apps which can cause serious financial loss to the customers. In summary, in order to secure the Android being exploited, the repackaging needs more attention from the researchers for an efficient technique to address the same.

The image processing techniques are adopted mainly in many domains and data protection has become a serious concern [4]. Image Hashing is a technique which can extract the content-oriented signature from the image. Various image hashing schemes have been developed for the restriction of images that carry important messages being tampered. It has a wide range of applications, including digital forensic, authentications, and repacking detection of android apps. Generally, the hashing technique should ensure the following parameters [5].

One-way hashing, which means that the images converted to hash quickly, whereas facing issues when the Hash values are being converted to the original image accurately.

Compact: The size of the hash values should always be very less when compared to the original image.

Perceptually Robust: it means that the difference that exists between a couple of image hash values generated through different perceptions should be kept small.

Manuscript received September 5, 2023
Manuscript revised September 20, 2023
https://doi.org/10.22937/IJCSNS.2023.23.9.18

The resistance of the collision in hashes should be different for different images, and the unique images should be identified through the hash distance.

The hash values should be unpredictable without the knowledge of the secret keys.

A range of techniques are available for the detection of mobile apps that are repacked [6], but these methods often suffer from various hindrances and are more complex and time-consuming. Further, the Hash formed of an image must be resistant over the content tampering. The techniques also should be susceptible to any update done on the content. In keeping in mind of these issues, this paper proposes a novel technique named [KPS: LPB], where the core intention came from the very fact that most of the repacked android apps aim for modification in a logic of the original apps, but for the users to get a false intention, the appearance often needs to be similar to that of original apps. Most of the repacked android apps must have a replica of the look as that of the genuine app. Hence, a comparison-based technique for visual similarity can efficiently and quickly determine the repackaged apps. The proposed method extracts the Key points that are more stable and the texture-based features from the given image for generating the 'app's image. The hash distance and the update distance methods are used for the similarity check between repacked and original apps.

The remaining of the paper is structured as follows, section 2 provides the essential pieces of literature on the domain, section 3 gives the problem analysis, and section 4 explains in detail of the proposed method. Section 5 describes the experimental point of view in the similarity checking phase, and section 6 discusses the results obtained. The last section provides the conclusion and future scopes.

2. Related Study

In the literature, many works have been proposed for the detection of repacked mobile applications based on the app's visual similarity. The context of website similarity is different from that of mobile applications, and the conventional methods do not yield efficient output [7]. As far as the Security in smartphones is concerned, there are several pieces of research that focus on the detection of repacked apps. Most of them are based on the Dalvik bytecode disassembling. DriodMOSS [8] was introduced for the purpose, and it adopts a fuzzy-based logic for Hashing for disassembling the code for repacked apps. However, this approach is dependent on the pieces of instructions on a given sequence and hence it was easy for the malware writers to bypass the detection. DNADriod [9] was proposed with the concept of program dependency-based graphs and used the concept of sub-graphs isomorphism technique for the detection of apps similarity. However, this method suffered from scalability issues as the number of graphs to be generated for a large scale of apps in the repository was tedious. AnDrawin[10] used a direction oriented method for the categorization of statements in the disassembled source code into semantics and constructed vectors. It addressed the issue of the local sensitive hash algorithm and speeds up the detection. ViewDroid [11] used the graph mining method for detecting repacked apps. The graphs are called feature views and are based on the relationship between the [UI] dependant function callings. This method did not address the local leakage issue in privacy. In [12], a program decouple based graph generation was proposed for the detection of “piggybacked” applications.

Juxtapp [13] made use of the ML methods for clustering the similar kind of apps on the disassembled source codes. Here, the techniques are largely dependent on the feature selection techniques which cannot be applied for the Hash-based detection. A wavelet-based method for feature selection is introduced for the selection of feature points. In [14], a rotational invariant matrix for the feature is used for the generation of image hash. In [15], the three-tier tensor is constructed from the source image for generating the 'images' Hash. In [16], an original image hashing method is introduced by the incorporation of global level features and local level features. The former and the latter were based on the Xernike moments of the luminant components, which helped in deriving the key points very easily. In [17] again, the concept of global and local features were used for the generation of Hash. The global features are obtained through the partition of the ring and are projected into a non-negative confusion matrix for the extraction of salient features in an image.

There were only a few methods available in the literature that can resist the combined manipulations. In [18], an image hashing method is proposed, which are Gabor filter-based and lattice oriented vectors quantization. The proposed method scales up to robustness in a couple of geometrical rotation and cropping of the source image. In [19], a unique hashing method using the non-negative matrices that will restrict rotational operations. In [20], the combined ring segmentation is used for the image hash generation. This method was found to be robust for the rotational and content manipulations. This technique can only validate the combinational repacking and not robust to other repacking techniques. Although there exist a few pieces of literature on the detection of repacked apps, there still exists issues and challenges that need attention.

3. Problem Analysis

The repacked mobile applications constitute the major part of the Android malwares, and these are basically detected using a couple of mechanisms [21]. As the repackaged apps are from the original apps with some modifications, the major of the functions and look would be like that of the genuine app. Hence, one of the detection approaches is based on the "sequence of instructions" in the disassembled source codes. The detection systems under these categories are DriodMOSS and DroidAnlaysis that makes use of fuzzy logic for the hashing. These struggle from hindrance, such as the inability to handle code obfuscation. The next approach is based on the "Semantics" of the code that are disassembled. The detection systems that come under this are DNADroid, where the semantic graphs are generated based on the code that are disassembled. These suffer from scalability issues.

As it is understood, both the methods need to first disassemble the concerned app prior analysis, and this can result in large time complexity when a greater number of applications are considered. The existing tools for the disassembling are being easily bypassed by the malware writers. In summary, the existing approaches primarily focus on the codes that are disassembled and are not scalable. Further, they can easily be bypassed by the obfuscation of code. Although image hashing methods are introduced to overcome this, many of them cannot handle combined manipulation of the app.

4. Proposed Methodology

The proposed methodology is shown in Figure 1 The proposed methodology consists of four important phases namely the Pre-proceedings phase, Similarity Checker and the Repacked app detector. The following section explains the phases in detail.

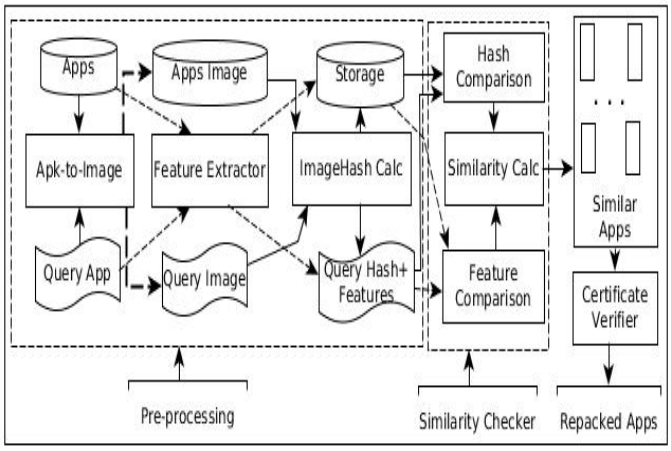

Fig. 1. Proposed Methodology

4.1 Pre-Processing Phase

4.1.1 APK to Image Conversion

The first phase of the proposed methodology is the pre-processing, wherein the Android A.P.K. files are converted into images. Any digital file on the memory device is store as a stream of bit of "0" and "1". The reverse engineering option available in the selendroid tool 20. All the readings are carried out on every apk file as binary stream and group each 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. Using the aforementioned method, an apk is converted to a grayscale image which is the first requirement to create an image hash. Due to varying apk size, the image dimensions will also be different; in our experiment, we have fixed the width of the image to 1024, and length varies accordingly. The apk-to-image component takes target and query app as input and convert all to a greyscale image file and store them to be used by further components. The aforementioned image conversion technique is adopted where all the binary content in the source code that is in the numerical form represented in a discrete fashion that considers all the byte value into an 8 bit ranging from 0-255 and then represented as a 2-D greyscale image.

4.1.2 Filtering and Normalization

Prior extraction of the key feature points of the app image, the pre-processing is done on the original file. Initially, in order to ensure that all the images are with similar Hash lengths. The concerned image is first put under normalization to get a constant size by making use of the linear interpretation. Then a specific filter, usually a low pass, is applied for the process of normalizing the image. The essentiality of filtering lies in eliminates the unwanted Information and to extract only the essential features. The filtering ensures the robustness of the hashing.

4.1.3 Feature Extraction

The feature extraction is carried out by the Limited Binary pattern (LBP) operator who is preferably a descriptor of the text and assigns binary labels to all the pixels of an 'app's image. During the computation of the label for every single pixel. The present pixel will be assigned as the center. The corresponding grey value of that center is then compared with the grey value in the 3*3 nearest pixel in a clock direction. If the nearest value is less than the center, the bit value will be 0. Else, the concerned bit value is set to 1. Once the gray values of the center and the nearest pixels are compared, the binary value of the center is then obtained and are converted to a decimal value.

Hence, there exists a total of 256 chances for the adjacent (includes 3*3 pixels), and the LBP vector range can be from 0-255. The histogram obtained will finally describe the texture.

The pseudo-code for the LBP extraction is given Algorithm 1. Every local sector is represented as a circle surrounding the key points where the key point is made the center with radius represented as r with a static value. In the given local sector. Later, the LBP (8 Bit vector) belonging to every pixel is then converted as a decimal (0-255). If the given pixel in the local sector is outside the boundary in an image, the concerned LBP value is then set to 0. The interval 0-255 is then subdivided into subintervals like [0-7], [8-15] ⋯ [247-255]. Hence for a local sector, the total count of the LBP in a given interval is represented as $\{N_1, N_2, N_3, \ldots N_m\}$ and the feature representing the texture in the local sector Ri is represented as Tex_i = $\{ T_1, T_2, \ldots T_n \}$. At last, the sequence in term of the binary representation of the local sector R is given as

$$T_{ij} = \begin{cases} 1 & N_i > \frac{m}{32} \\ 0 & otherwise \end{cases} \quad (1)$$

Where the range can take up the value between 1 and 32, and m is the total pixels in the local sector. Hence, the 'textures' feature for N number of key points represented as T = {Tex1, Tex2, Tex3 , . . . TexN }

Algorithm 1.: LBP Feature extraction

Input: Geometric Data on Apk image

Output: Textures features of the Key Points

STEP1: START

STEP2: FOR i=1 to k DO

STEP3: IDENTIFY THE LOCAL REGION RN FOR THE KEYPOINT K

STEP4: CALCULATE LBP END FOR

STEP5: FOR q=1 to m

DO: CONVERT m Vector to Decimal

STEP6: SEGMENT 0 to 255 in Equal Intervals

[0-7] , [8-15] , … [247-255]

STEP7: CALCULATE THE NUMBER OF VERCORS AS {N1, N2, N3,… Nm}

STEP8: FOR j = 1 to 32

STEP9: IF Ni > m/32 then Tij = 1 ELSE 0

STEP10: END IF

STEP11: Texi = { T1,T2,…Tn } END FOR

STEP12: RETURN T = { Tex1 , Tex2 , Tex3 , . . . TexN}

STEP13: STOP

4.2 Image Hashing

The process of image hashing has two sub-processes namely the $i.$ Key point extraction, ii. Hash construction. These are explained in the following sections.

4.2.1 Key Points Extraction

The Constant Keypoint selection is used for extracting the key points of the grey images of the android apps. Information about the feature point has the geometrical data and the descriptions. The set of all the geometric data on the feature points are denoted using the vector $V = \{v_1, v_2, \ldots, v_n\}$ where the $Vi = \{ x_m , y_j , \sigma_i , \hat{O} \}$ and n represents the total count of the feature points. Each of the keypoint has all the Information on the location and the scale with direction. The descriptor set denoted as a vector $D = \{d_1, d_2, \ldots d_n\}$. Every D can have up to 128 representations in the gradient that distributed across the feature points. Generically, the total count of the feature points will be large. However, few of these will not be robust when taken for consideration on the detection of content manipulations. It gives the opinion that the feature points which are not stable lost in different content manipulations, and these points are not stable then removed while the stable points retained. The algorithm for the keypoint extraction given under Algo.2.

Algo. 2. Keypoint extraction

Input: Pre-processed Apk images

Output: Key points set K

STEP1: START

STEP2: FOR i=1 to k DO

STEP3: GENERATE DOG OF IMAGES

STEP4: DETERMINE THE CANDIDATE FEATURE POINT VECTOR $V = \{v_1, v_2, \ldots, v_n\}$

STEP5: DETERMINE THE VECTOR DESCRIPTION $D = \{d_1, d_2, \ldots d_n\}$

STEP6: APPLY FOURIER TRANSFORMATION TO THE IMAGES

STEP7: GET THE TOP K KEYPOINT SET $K = \{ k_1, k_2, k_3, \ldots k_n\}$

STEP8: OBTAIN GEOMETRIC INFORMATION OF THE {k}

STEP9: IF NUMBER OF KEY POINTS < K

STEP10: SET FEATURE POINT VECTORS $V \equiv \{0\}$

STEP11: RETURN THE FEATURE POINT SET K

STEP12: END IF

STEP13: END FOR

STEP13: STOP

For the process of enhancing the robustness in the proposed method of image hashing, the critical regions detected for extracting the spectral residue, which can eliminate all the feature points that are not stable. The criteria for the same given in Eq. (2) which preferred as a filtering process.

$$\text{Reverse Loc } if\ \mathrm{S_n} > E(S) \quad (2)$$

Where, $E(S) = (\frac{1}{n} * \sum_i^n S_i)$ represents the mean intensity of the n keypoints. If the S_i length of the ith point is less when compared to the mean intensity, the particular feature point is removed else the same will be reverse.

4.2.2 Hash construction

Once the key points generated, the features of the textures that correspond to the local sector are obtained. In order to ensure storage convenience, all the Information pertaining to the keypoints are compressed and then converted into binary values. The Hashing algorithm proposed, in the initial state, the position, scales and the orientation of all the keypoints are then used for the formulation of position $Pos_i = \{ x_i , y_i , \partial_i \}$. The Information on the geometry of the K is then denoted by the vector $P = \{ P_1, P_2, \dots P_n\}$

where, each P is transformed into a binary sequence, the initial descriptor V on all the keypoints that has 128 dimension are then compressed into 32 dimensions, and also Vi is then normalized as $|V_i|_2 = 1$ where, i={1, 2, 3, ⋯ ,n}. Every element in the V_i is then subjected to quantization with either 0 or 1 based on the rule given in Eq. (3).

$$qV_{i_j} = \begin{Bmatrix} 1 \text{ , if } \left(V_{i_j} > \alpha\right) \\ 0\ otherwise \end{Bmatrix} \quad (3)$$

$where\ V_{i_j}$ is the jth element of V_i and j is range one up to 32, α is threshold.

Finally, the end position P, Descriptions Q, and the feature T forms the hash $H = \{P, Q, T\}$. The normal 'image's size is kept as 256*256. As every value in the set $Possition\ P = \{ P_1, P_2, \dots P_n\}$ shall then described in an 8-Bit length sequence; the total 'P's length will be of 8 Bits as 32 bits will be the $Q = qV_1, qV_2 \dots qV_n$. Similarly, the texture T will have a length 32*k bits. Hence, the full 'Hash's length will be of 96k bits. Table 1 portrays the hash elements.

Table 1. Portrays hash elements with length

Description	***Element***	***Length***
Position (P)	$P = \{ P1 , P2 , \dots Pn \}$	32*k bits
Quantized descriptor(Q)	$V = \{v1, v2, \dots , v_n\}$	32*k bits
Texture Feature (T)	T $= \{ \text{Tex1} , \text{Tex2} , \dots \text{Tex}_n\}$	32*k bits

4.3 Similarity checking phase

The similarity checking phase is the final phase of the proposed method and comprises the following parts.

Hash comparison is made on the Hamming Distance Calculation.

The feature comparison is done through the layout tree formation method, and finally, the similarity checking is done through the Layout Edit Distance method.

4.4 Hash distance calculation

The Normalized Hash Distance (NHD) is used here as a metric for the evaluation of the similarity between the given pair of image hashes and is obtained as

$$NHD(Hi , Hj) = \frac{1}{N} \sum_i^l | H(i) - h_2 (i)| \quad (4)$$

Where i denotes the length of the Hash, and the $Hj\ and\ H(i)$ represents the elements in the Hash. The H takes the value between 0 and 1. In the case of two images being similar, the value of H will be tending to 0. For two different images, the H will be much greater than 0. A predefined threshold value used for the current image for deciding whether the image is similar or not when compared to that of the source image. The threshold is set to values between 0.1 and 0.4. The Hash distance is then calculated through

$$HD(TEXi , TEXj) = \sum |TEXi(0) - TEXj(0)| \quad (5)$$

Where the $TEXi$ and $TEXj$ are the Nth texture feature and ith feature of the image hash.

4.5 Layout Tree Formation

The SDK of Android gives a range of objects for viewing, such as the Buttons, Text, and scrolls. Presents the sample U.I. and the concerned Layout file. The following hypothesis is taken for the construction of the layout tree.

A layout can be considered as the Tree data structure where, the given node of the tree describes the element in the lay file. The nodes in the tree reflect the elements of the particular t tree. The name of the node corresponds to the name of the element and the values contain all the attributes concerned with the element. The relationship between the nodes of a layout is same as that of the file. For instance, the VIEW object has other sub-objects like the VIEW-GROUP, then the latter is considered as the parent of VIEW. Fig. 2 shows the layout tree for the U.I. interface considered. The other variations of the tree are also generated by changing each feature of the lay out screen and finally the features are compared with that of the original app and the similar features are removed. This tree can be used for the detection of apps that are similar. Initially, the 'layout's structure defines the visual hierarchy of the U.I. Next, the repacked applications have a similar Tree as they have to rely on the original appearance to cheat the users. Last, the nodes present in the layout graph with attributes exactly represent the elements on the screen.

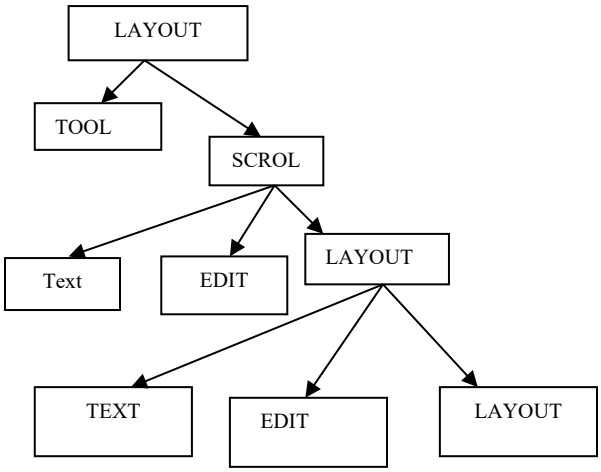

Fig. 2. Layout Tree for the UI

4.6 Similarity Calculation

The calculation of the similarity is done through the Update Distance methods. The layout's updated distances defined as one of the metrics for measuring the similarity of the two-layer trees. For a given pair of mobile apps, initially, their layout trees are obtained from the corresponding files. The update distance is the minimal count of the operations that required for the transformation of one layout to the other layout tree. Node inserting and substitution are the two operations performed.

In the similarity checking phase, the update distance metric is used to detect the visual similarity between the applications. This technique is intended to achieve minimum time and space complexity of $O(m^2n^2)$ and O(mn), where the variables m, n denotes the no of nodes. The similarity measure between a couple of apps directly corresponds to the Update distance. It is to be noted that the single app may have n number of layouts, and therefore, it is essential for all the pairs to compared with that of the original app. For example, if there are x number of layouts distance in the app A and y in app B, the score of the similarity is the minimal count of all the update distances that exist between (m, n) pair. In order to provide an optimized comparison, the layout trees that has the total nodes less that of the threshold, say n is omitted for the comparison. Next, the total count of the elements is used for the reduction in the pairs to be compared. For example, if the need to find the applications in a particular repository whose similarity measure is less than the threshold value M. In this case, prior to comparing the files, the comparator initially calculates the number of elements in the apps. In case their difference is larger than that of the threshold m, the system ignores the same for comparison. It is due to the fact that as per the update distance method, insertion is treated as a transformational operation. The deviations of the count of the elements in a layout pair file is that a one that needs at least an insert operation for the transformation. Using these optimization techniques, the total number of comparisons needed to be reduced.

5. Experiment Set-Up and Results

The experiments were conducted on the repository of 95124 apps that are acquired from the third-party store. Again, a total of about 456 unique apps were downloaded from the play store and were compared to that of the apps already in the repository. Table 2 gives the experimental results of the initial metrics of True positive [TN, True Negative [TN], False positive [FP] and False Negative [FN]. A total of 1354 similar apps were detected, and most of them were cracked games and adware. If the detection successfully identifies a repacked app, it is called as TP event. If the genuine application is reported as repacked, it is termed as FP If genuine app is identified as genuine, it is termed as TN, and if it is the vice versa, it is termed as FN The accuracy of the system is calculated based on the values obtained for these metrics. Fig. 3 shows the comparative results, which are self-explanatory.

Table 2. Comparison of {TP, TN, FP, FN}

METHODS	*TP*	*TN*	*FP*	*FN*
DroidMOSS	81.2	8.61	78.2	7.84
AnDarwin	79.6	9.52	74.2	10.25
ViewDroid	64.5	11.56	68.5	12.59
Juxtapp	86.2	6.21	82.1	7.81
KPS:LBP	94.8	1.26	96.2	1.27

For the experimental analysis, the single manipulated image set, and the combined manipulated set used. For both the hash value, the images were classified as per the hash values that are generated. The accuracy of the classification was obtained from the different values of thresholds. The Receiver Operating Character (ROC) then calculated. The ROC for the content manipulations were computed using the TP rate and FP rate which are obtained by

$$\text{FTP} = \frac{n_{SIM}}{N_{SIM}} \tag{6}$$

$$\text{FTP} = \frac{n_{dif}}{N_{dif}} \tag{7}$$

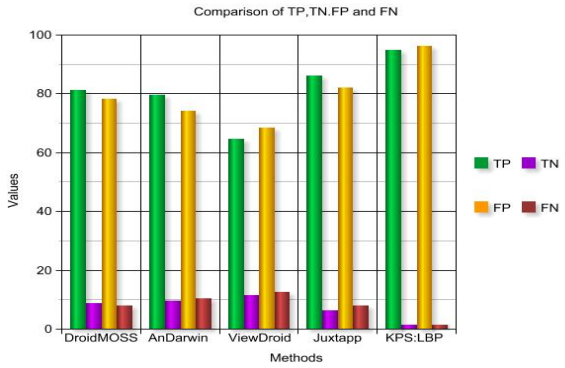

Fig. 3. Comparison of TP, TN, FP, FN

The Table 3 gives the comparative results of the values of the rate of TP and FP for the different methods taken for comparison. Fig. 4 shows the plot for the same. It is obvious from the results that the proposed method has more TPR. and less FPR. which ensures the efficiency.

METHODS	*TPR.*	*FPR.*
DroidMOSS	1.26	8.24
AnDarwin	2.54	5.62
ViewDroid	3.85	4.28
Juxtapp	4.86	3.58
KPS: LBP	8.21	1.51

Table 3. Rate of TP and FP

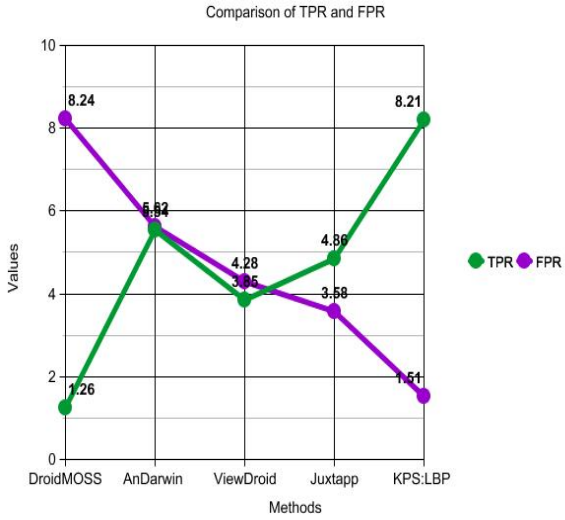

Fig. 4 TP and FP rate Plot

As the collision probability [22] takes up an important characteristic in the experiments, the CP of the hash techniques were calculated as

$$P(T) = \frac{1}{\sqrt{2\pi\rho}} \int_o^t \frac{x-\rho}{2\sigma}\, dx \tag{8}$$

where, T is the threshold assigned, σ is the Standard deviation, and ρ is the average mean.

5.1 Time complexity

The time complexity is also to be treated as a crucial metric when Hashing techniques evaluated. In order to compare the Man computation time, various hashing methods are used for the generation of Hash values from N number of images, and the average time taken for the computation is calculated. For the experimental analysis, the number of key points is set to 10, and hence the hash length would be 960 Bits. The length seems to be large as the Hash here consists of much geographical Information

texture features and Key-points of the images. This Information helps a lot in the stability of the proposed method and ensures the detection of even small tampering. Table 4 shows the comparison of the ROC, Collision Probability (CP) and Time complexity (TC) of the proposed method to that of the other methods. It is seen that the proposed method outperforms the other state of the art methods taken for the comparison. Fig. 5 depicts the graphical plot of the same.

Table 4. Comparison of ROC, Collision Probability, and Time Complexity.

METHODS	*ROC*	*CP*	*TC*
DroidMOSS	0.26	0.18	0.84
AnDarwin	0.57	0.67	0.76
ViewDroid	0.48	0.45	0.52
Juxtapp	0.68	0.54	0.61
KPS:LBP	0.91	0.94	0.21

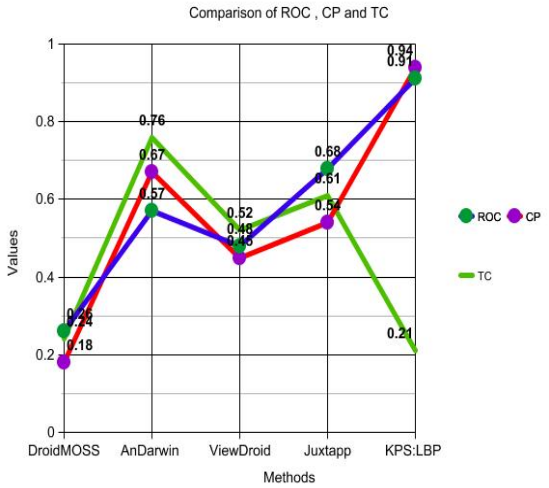

Fig. 5. Comparison of ROC, CP and TC

6. Conclusion

The android app repacking is the major issue that causes a severe threat to the technical ecology of Android. The available methods do not have high performance and also suffer from various issues. This manuscript presents a novel CKPS: LBP method for the detection of repacked Android apps by detecting the visual similarity of the apps. The Constant Keypoint selection used for the selection of key points, and the Limited Binary pattern used for the Feature selection. The experimental results produce high performance in terms of the TPR and FPR. The time complexity, ROC, and Collision probability were also found to be better than the other competitive methods. The future work - aiming at the increase in efficiency and to apply this technique for the detection of code obfuscation, which is also a severe threat to the Android devices.

References

[1] J. Li, X. Liu, H. Zhang, and D. Mu, "A Scalable Cloud-Based Android App Repackaging Detection Framework," Green, Pervasive, and Cloud Computing Lecture Notes in Computer Science, pp. 113–125, 2016. https://doi.org/10.1007/978-3-319-39077-2_8

[2] X. Sun, J. Han, H. Dai, and Q. Li, "An Active Android Application Repacking Detection Approach," 2018 10th International Conference on Communication Software and Networks (ICCSN), 2018. https://doi.org/10.1109/iccsn.2018.8488263

[3] Q. Zeng, L. Luo, Z. Qian, X. Du, and Z. Li, "Resilient decentralized Android application repackaging detection using logic bombs," Proceedings of the 2018 International Symposium on Code Generation and Optimization - CGO 2018, 2018. https://doi.org/10.1145/3168820

[4] K. Khanmohammadi, N. Ebrahimi, A. Hamou-Lhadj, and R. Khoury, "Empirical study of android repackaged applications," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3587–3629, 2019. https://doi.org/10.1007/s10664-019-09760-3

[5] M. O. F. K. Russel, S. S. M. M. Rahman, and T. Islam, "A Large-Scale Investigation to Identify the Pattern of App Component in Obfuscated Android Malwares," *Communications in Computer and Information Science Machine Learning, Image Processing, Network Security and Data Sciences*, pp. 513–526, 2020. https://doi.org/10.1007/978-981-15-6318-8_42

[6] V. Rastogi, Y. Chen, and X. Jiang, "DroidChameleon," Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security - ASIA CCS 13, 2013. https://doi.org/10.1145/2484313.2484355

[7] X. Liao, Z. Geng, Y. Meng, Y. Yu, Y. Li, and D. Kang, "A Detection Method for Android Repackaged Applications with Malicious Features Similarity of Family Homology," *2017 International Conference on Computer Technology, Electronics and Communication (ICCTEC)*, 2017. https://doi.org/10.1109/ICOMSSC45026.2018.8941586

[8] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," *Proceedings of the second ACM conference*

on Data and Application Security and Privacy - CODASKY 12, 2012. https://doi.org/10.1145/2133601.2133640

[9] A. Gharib and A. Ghorbani, "DNA-Droid: A Real-Time Android Ransomware Detection Framework," *Network and System Security Lecture Notes in Computer Science*, pp. 184–198, 2017. https://doi.org/10.1007/978-3-319-64701-2_14

[10] J. Crussell, C. Gibler, and H. Chen, "AnDarwin: Scalable Detection of Android Application Clones Based on Semantics," *IEEE Transactions on Mobile Computing*, vol. 14, no. 10, pp. 2007–2019, 2015. https://doi.org/10.1109/TMC.2014.2381212

[11] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "ViewDroid," Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks - WiSec 14, 2014. https://doi.org/10.1109/TMC.2014.2381212

[12] Q. Chen, J. Wang, and Y. Wang, "An Online Approach for Detecting Repackaged Android Applications Based on Multi-user Collaboration," *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, 2015. https://doi.org/10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.66

[13] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A Scalable System for Detecting Code Reuse among Android Applications," *Detection of Intrusions and Malware, and Vulnerability Assessment Lecture Notes in Computer Science*, pp. 62–81, 2013. https://doi.org/10.1007/978-3-642-37300-8_4

[14] C. Yuan, S. Wei, C. Zhou, J. Guo, and H. Xiang, "Scalable and Obfuscation-Resilient Android App Repackaging Detection Based on Behavior Birthmark," *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, 2017. https://doi.org/10.1109/ APSEC.2017.54

[15] Z. Li, J. Sun, Q. Yan, W. Srisa-An, and Y. Tsutano, "Obfusifier: Obfuscation-Resistant Android Malware Detection System," Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering Security and Privacy in Communication Networks, pp. 214–234, 2019. https://doi.org/10.1007/978-3-030-37228-6_11

[16] L. Li, D. Li, T. F. Bissyande, J. Klein, Y. L. Traon, D. Lo, and L. Cavallaro, "Understanding Android App Piggybacking," *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017. https://doi.org/10.1109/ICSE-C.2017.109

[17] Z. Qin, Q. Zhang, X. Zhang, and Z. Yang, "An efficient method of detecting repackaged android applications," *International Conference on Cyberspace Technology (CCT 2014)*, 2014. https://doi.org/10.1049/cp.2014.1331

[18] H. Huang, S. Zhu, P. Liu, and D. Wu, "A Framework for Evaluating Mobile App Repackaging Detection Algorithms," *Trust and Trustworthy Computing Lecture Notes in Computer Science*, pp. 169–186, 2013. https://doi.org/10.1007/978-3-642-38908-5_13

[19] F. Sierra and A. Ramirez, "Defending Your Android App," *Proceedings of the 4th Annual ACM Conference on Research in Information Technology - RIIT 15*, 2015. https://doi.org/10.1145/2808062.2808067

[20] Q. Zeng, L. Luo, Z. Qian, X. Du, Z. Li, C.-T. Huang, and C. Farkas, "Resilient User-Side Android Application Repackaging and Tampering Detection Using Cryptographically Obfuscated Logic Bombs," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2019. https://doi.org/10.1109/TDSC.2019.2957787

[21] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild," *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering Security and Privacy in Communication Networks*, pp. 172–192, 2018. https://doi.org/10.1007/978-3-030-01701-9_10

[22] Y.-L. Chen, "An explicit and novel forward collision probability index," *2015 IEEE 10th Conference on Industrial Electronics and Applications (ICIEA)*, 2015. https://doi.org/10.1109/ICIEA.2015.7334399

M A Rahim Khan received the M.Tech. degrees in Information Technology from GGIP University New Delhi in 2008, he worked as a lecturer in Majmaah University Saudi Arabia. Currently Ph.d Scholar in Lingayas Vidyapeeth Fridabad India.

Dr. Manoj Kumar Jain (Associate Professor CSE & In charge Academics) Lingayas vidyapeeth Fridabad India. His research area in Neural Network, Information Security, Wireless Sensor Netwo