ProDroid: Resource-based Android Application Repackaging Detection

Zhuo Chen
Nanjing University
Nanjing, China
ouhznehc@outlook.com

Mujie Xu
Nanjing University
Nanjing, China
litrehinn@gmail.com

Feiyu Sun
Nanjing University
Nanjing, China
dawnfying@foxmail.com

ABSTRACT

This paper introduces ProDroid, an innovative approach for detecting Android application repackaging by dynamically capturing runtime resources. Utilizing a combination of randomized application testing via Monkey, and resource hooking through Xposed module, ProDroid efficiently generates application fingerprints. Employing algorithms like LSH and DHash, it processes hooked resources to create distinct software birthmarks. ProDroid demonstrates a high efficiency in detection rates, achieving a 97% recall and 90.3% precision while maintaining a direct proportionality between runtime and the number of APKs analyzed.

1 INTRODUCTION

In the era of mobile internet, Android smartphones have become an indispensable part of people's daily lives and activities. The Android ecosystem, known for its openness and a vast user base, has also attracted a significant number of individuals with malicious intent. These actors exploit a technique known as Android application repackaging, a process in which they decompile existing Android application packages (APK files), make subtle modifications, and then repackage them. The resulting applications may outwardly appear unchanged, but beneath the surface, they can conceal malicious code, including unwanted advertisements, cryptocurrency miners, malware, or fraudulent mechanisms. This poses substantial threats to both software developers and end-users.

Traditional approaches to addressing this issue involve the use of code-based software fingerprints. These techniques rely on the analysis of code syntax, semantics, and various features. However, these methods often fall short when confronted with code encryption and obfuscation. Even dynamic code analysis approaches struggle to adapt to changes stemming from the replacement of Android APIs.

Another avenue of research involves extracting software fingerprints based on similarities in the graphical user interface (GUI) structure of repackaged applications. These methods have shown promise, particularly for applications built on the Android framework. However, they encounter limitations when dealing with other applications, and the effectiveness of such approaches can vary significantly.

Resource-based software fingerprints present an alternative solution. These techniques leverage the similarity of resource files within repackaged applications, comparing extracted features to identify alterations. However, the predominant resource analysis methods currently rely on static approaches to capture and summarize resource files, rendering them susceptible to resource obfuscation tactics employed by attackers.

The potential breakthrough in combating Android application repackaging lies in adopting dynamic methods. This innovative approach involves segmenting resource files accessed by applications through runtime API calls, rather than relying on traditional file-level hash digests. By doing so, the detection sensitivity is significantly enhanced, rendering resource obfuscation techniques ineffective against the analysis.

2 MOTIVATION

This section discuss the limitations of recent resource-based Android application repackaging detection techniques when used in production environments, and motivates the need for a new approach.

FSquaDRA[1] is a well-known tool that employs the comparison of SHA256 resource hash values recorded in the manifest files of two APKs to rapidly calculate their Jaccard similarity coefficient. However, FSquaDRA exhibits significant limitations that motivate the need for our research.

Firstly, FSquaDRA is constrained in its applicability as it can only detect repackaged applications that employ the SHA256 hashing algorithm for packaging. This limitation restricts its effectiveness, especially considering the diversity of hashing algorithms used in Android app packaging.

Secondly, repackagers can easily circumvent FSquaDRA's detection by modifying information within the manifest files or making slight alterations to resource files. These modifications can result in significant changes in the hash values of resources, rendering FSquaDRA ineffective in identifying such repackaged applications. Our research seeks to enhance the robustness of repackaging detection by considering a wider range of factors beyond resource hashes.

Lastly, despite the speed of FSquaDRA's detection process, its computational complexity remains $O(n^2)$ with respect to the number of APKs(n) being analyzed. Given the exponential growth of the APK application market, the computational overhead for detecting repackaged application pairs remains substantial. Our research aims to optimize the detection process to make it more scalable and efficient, thus addressing the challenges posed by the ever-expanding app market.

In summary, the limitations of FSquaDRA, including its reliance on a specific hashing algorithm, susceptibility to evasion, and computational inefficiency, underscore the necessity for our research. We aim to develop a more adaptable and robust repackaging detection methodology that can effectively address these shortcomings, providing a valuable contribution to the field of Android application security.

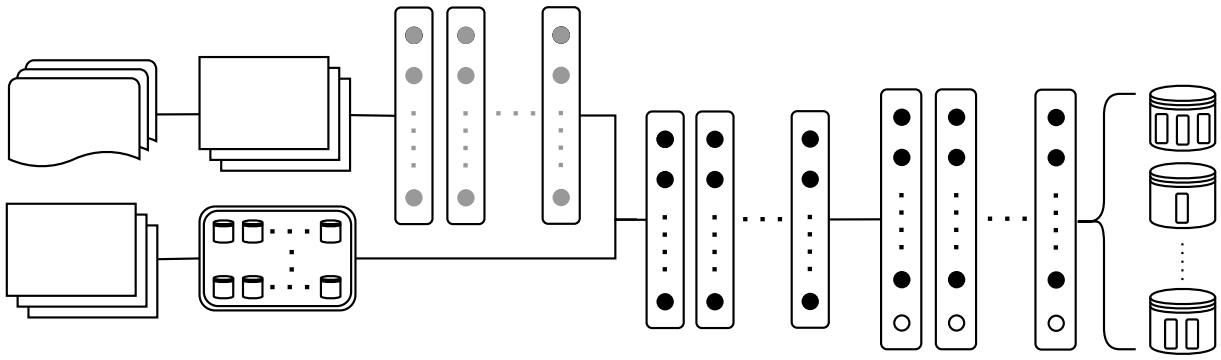

Figure 1: Overview of ProDroid Structure

3 OVERVIEW

The primary objective of ProDroid is to establish a more comprehensive and resilient method for detecting repackaged applications, a prevalent issue in the Android app ecosystem. This tool is designed to enhance the accuracy and efficiency of identifying unauthorized modifications or reproductions of original applications.

This paper contributes to the field in several significant ways:

- **Dynamic Resource Extraction:** ProDroid introduces an innovative dynamic hooking method to extract runtime resources that are actively used by the application. This approach effectively counters the strategies employed by repackagers who typically add redundant resources statically to evade detection.
- **Advanced Image Hashing Techniques:** ProDroid incorporates sophisticated image perception hashing algorithms. These algorithms are particularly adept at identifying subtle changes at the pixel level, preventing repackagers from making minor adjustments that could drastically alter traditional hash values.
- **Optimized Detection Algorithms:** ProDroid combines the efficiency of minihash with the effectiveness of Locality-Sensitive Hashing (LSH) algorithms. This fusion significantly streamlines the detection process, reducing the computational complexity to $O(n)$, thus accelerating the identification of repackaged app pairs.
- **Empirical Validation:** Extensive testing across various app markets, including Google Play, AppChina, and Anzhi, has demonstrated that ProDroid outperforms existing solutions like FSquaDRA. It not only identifies a higher number of repackaged app pairs but also does so more swiftly.

Figure 1 presents a comprehensive view of ProDroid's dual-phase architecture, encompassing both online resource capture and offline repackaging detection.

Online Phase: This phase employs Monkey[2] for randomized testing, enabling the capture of a diverse array of runtime resources. The use of an Xposed[3] module facilitates the interception and redirection of these resources. Xposed first assigns necessary permissions to the app, followed by intercepting resource calls via rewritten Android Java-layer APIs. The "am task lock" mechanism is applied to each app to ensure focused testing and maximal resource capture.

Offline Phase: The collected resources are initially processed using Dhash[4], generating a preliminary hash set, $\{Hash_{dhash}\}$. This set undergoes further refinement via the k-means[5] clustering algorithm, yielding $\{Hash_{k-means}\}$, which minimizes the impact of negligible image variations on the hash results. The minihash[6] algorithm then transforms $\{Hash_{k-means}\}$ into a tuple, $\{Hash_{minihash}\}$, collectively forming the software's unique birthmark.

Comparison Stage: The final stage involves utilizing the LSH[7] algorithm to process $\{Hash_{minihash}\}$, efficiently grouping similar application pairs. For each grouped pair, a pairwise comparison of their $\{Hash_{k-means}\}$ is conducted, facilitating the calculation of the Jaccard similarity coefficient. This metric is crucial for accurately identifying repackaged applications.

ProDroid's comprehensive approach, combining dynamic resource interception with advanced hashing and clustering techniques, sets a new benchmark in the detection of repackaged Android applications, offering an effective solution to a critical challenge in mobile application security.

4 BIRTHMARK GENERATION

This section will demonstrate how ProDroid generates software birthmarks. Birthmark generation consists of three main components: randomized runtime testing, runtime resource collection, and offline birthmark generation.

4.1 Randomized Runtime Testing

To optimize the online execution efficiency while ensuring a thorough capture of runtime resources, ProDroid employs Monkey, an automated testing tool for Android applications. This tool is instrumental in performing randomized testing, a crucial process for simulating a broad spectrum of user interactions.

A key concern in our testing methodology is to prevent unintended navigation to external applications, which could disrupt the integrity of resource capture. To address this, we utilize the "am task lock" command, effectively locking the currently running software. This ensures that all generated events are contained within the scope of the target application, providing a controlled environment for resource extraction.

Furthermore, recognizing the potential interference of advertisement injections, which are prevalent in modern applications,

we implement a deliberate 5-second delay at the commencement of each testing session. This delay is designed to bypass initial ad presentations, thus mitigating their impact on the birthmark generation process.

Randomized testing is executed in a meticulously structured manner. We repeat the testing cycle ten times, each iteration consisting of 100,000 events. This approach is not arbitrary; it is carefully calibrated to ensure a comprehensive exploration of the application's event space. By diversifying the event sequences, we avoid the pitfall of testing getting concentrated in a specific branch of the application's logic. This strategy is critical for achieving a balanced and extensive coverage of the application's functionalities, thereby maximizing the variety and quantity of runtime resources captured.

The outcome of this rigorous testing regimen is a robust dataset of runtime resources, reflecting a wide array of user interactions and application states. This dataset forms the foundation for the subsequent phases of resource analysis and birthmark generation in ProDroid, setting the stage for effective repackaging detection.

4.2 Runtime Resource Collection

In the realm of Android applications, runtime resources primarily bifurcate into text-based (like XML and Strings) and image-based categories (such as PNG and JPG files). Given the intricate nature of text-based resources, which can significantly impact the user experience through variations in language or textual layout, ProDroid strategically focuses on image-based resources for a more reliable and efficient detection process.

Utilizing the versatility of Xposed modules, ProDroid adeptly captures these runtime resources. The process begins with Xposed granting the application all necessary permissions, thus ensuring a seamless and unrestricted resource access during the randomized testing phase. This step is pivotal in enabling comprehensive data capture without any permission-related hindrances.

In its pursuit of resource collection, ProDroid employs a targeted approach. For resources located within the "res" directory, the system hooks into the "$getResource()$" method, effectively capturing any resources requested by the application. Similarly, for resources within the "assets" directory, ProDroid intercepts calls to the "$open()$" method. This methodical interception ensures a thorough collection of all significant image resources used by the application during runtime.

Further enhancing its resource collection capabilities, ProDroid also integrates hooks into the "$inflate()$" method. This proactive measure facilitates an in-depth traversal of the XML tree structure, allowing for the identification and extraction of resource IDs linked to image files. This approach not only broadens the spectrum of collected resources but also deepens the insight into the application's resource utilization patterns.

Once the resources are intercepted, ProDroid instructs the application to transmit these resources securely via HTTPS to a designated remote server. This step is crucial for the subsequent offline processing phase, and is made possible due to the network permissions granted at the onset of the application's runtime.

Designed with flexibility in mind, ProDroid offers the capability to customize the specific method names for hooking. This feature is particularly advantageous for adapting the system to applications developed on unconventional Android frameworks, thus enhancing ProDroid's applicability and effectiveness in diverse application environments. The comprehensive and adaptable nature of this resource collection methodology underscores ProDroid's robustness in capturing a wide array of image-based resources, which is essential for the accurate detection of repackaged applications.

4.3 Offline Birthmark Generation

ProDroid addresses the inherent weaknesses of traditional hashing algorithms like MD5 and SHA256, especially when applied to image resources in the context of Android application repackaging detection. Repackagers often manipulate images by subtly altering pixels, which can significantly impact the hash values of an APK's resource set, evading detection. To counteract this, ProDroid implements a more robust approach using Dhash in combination with k-means clustering, effectively reducing the influence of minor image discrepancies on hash values.

Dhash, a perceptual image hashing algorithm, excels in resisting obfuscation tactics such as brightness adjustment, scaling, and minor pixel modifications. This resilience stems from its focus on capturing the essential visual features of images, thus maintaining hash consistency despite superficial changes. The integration of Dhash into ProDroid's framework marks a significant step towards accurately identifying repackaged apps through image analysis.

Complementing Dhash, ProDroid utilizes the k-means clustering algorithm, an iterative process renowned for its effectiveness in grouping data based on similarities. By training this algorithm on a curated dataset of images, ProDroid can identify central image patterns or centroids. These centroids represent the quintessential characteristics of the image set, further refining the analysis process.

In the birthmark generation phase, the application of Dhash to the gathered resources yields an initial set of hash values, denoted as $\{Hash_{dhash}\}$. This set undergoes a matching process with the predetermined image centroids from the k-means clustering, leading to a refined hash set $\{Hash_{k-means}\}$. This two-step procedure ensures the extraction of pivotal image features while minimizing the hash value discrepancies caused by insignificant image variations.

To prepare for the subsequent repackaging detection, ProDroid employs the minihash algorithm on $\{Hash_{k-means}\}$. This algorithm executes k iterations, each mapping the hash set and selecting the minimal value. The collation of these minimal values forms a k-tuple, $\{Hash_{minihash}\}$. This k-tuple is instrumental in the ensuing detection phase, as it encapsulates the core characteristics of the app's resources in a compact and efficient format.

The final step in the birthmark generation involves combining $\{Hash_{minihash}\}$ and $\{Hash_{k-means}\}$, culminating in a comprehensive and distinctive birthmark for the software. This birthmark is not just a mere collection of hash values; it embodies the unique resource footprint of the application, enhancing ProDroid's capability to discern repackaged applications with heightened precision. By leveraging these advanced image processing and hashing techniques, ProDroid sets a new standard in the detection of repackaged Android applications, offering a potent solution to a pervasive challenge in app security.

5 REPACKAGING DETECTION

This section delves into the intricacies of how ProDroid executes repackaging detection for applications, leveraging its uniquely generated birthmarks. The methodology is divided into two distinct stages: coarse-grained detection for initial filtering and fine-grained detection for detailed analysis.

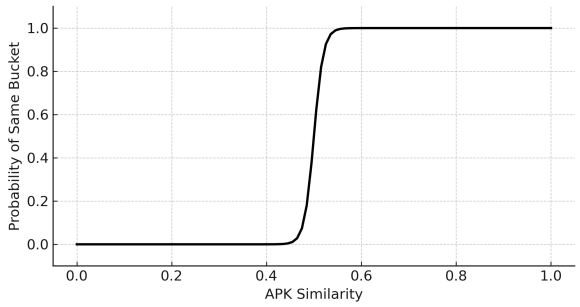

Figure 2: LSH Bucketing Probability

5.1 Coarse-Grained Detection

In ProDroid, the initial phase of repackaging detection is the coarse-grained detection, which fundamentally employs the Locality-Sensitive Hashing (LSH) algorithm. This phase utilizes LSH on the $\{Hash_{minihash}\}$ component of the birthmarks to categorize applications into "buckets" based on their resource feature similarities. The process boasts an efficient time complexity of $O(n)$, where 'n' is the total number of APKs analyzed.

LSH plays a critical role in this phase by transforming high-dimensional data into a lower-dimensional hash space, while maintaining the similarity structure among data points. As depicted in Figure 4(a), the LSH algorithm maps applications with similar $\{Hash_{minihash}\}$ values to the same bucket with a high probability. This mapping is illustrated by a smooth curve, where the probability of being placed in the same bucket escalates as the similarity exceeds a certain threshold.

This efficient grouping by LSH significantly narrows the scope of the subsequent analysis to a more manageable subset of data, considerably enhancing the detection process's efficiency, especially in large-scale datasets. The shift from a broad initial analysis to a focused examination is key in optimizing resource utilization and improving the accuracy and speed of identifying potential repackaged applications within ProDroid.

5.2 Fine-Grained Detection

Following the coarse-grained detection, ProDroid proceeds to the fine-grained detection stage. This phase involves a meticulous pairwise comparison of the $\{Hash_{k-means}\}$ from each application's birthmark. The goal is to calculate the Jaccard similarity coefficient, which serves as a quantitative measure of similarity between pairs of applications. This coefficient is instrumental in the generation of a comprehensive repackaging report. It also plays a critical role in filtering out false positives, i.e., pairs of applications that might have been coincidentally mapped to the same bucket but are not repackaged versions of one another.

Despite the theoretical time complexity of $O(n^2)$ for this stage, the actual computational load is typically much lower in practice. This efficiency arises from the fact that repackaged applications constitute only a small fraction of the total applications in the market. Consequently, the number of APKs that advance to the same bucket after the coarse-grained filtering is relatively minimal. This aspect ensures that the fine-grained analysis, though thorough, does not impose a significant computational burden, thus maintaining the overall expediency of ProDroid's detection mechanism.

6 EVALUATION

This section presents a comprehensive evaluation of ProDroid, focusing on assessing its runtime overhead, offline analysis efficiency, and effectiveness in repackaging detection.

6.1 Methodology

Our evaluation methodology involved a series of experiments conducted on robust hardware setups and a diverse dataset of APKs. The offline analysis was executed on a high-performance computing system equipped with a 4.4GHz 12-core Intel Core i7-12700K processor, 32GB of RAM, and an NVIDIA GeForce RTX 4060 graphics card. The operating system for this setup was Windows 11, which provided the necessary environment for intensive data processing tasks.

For the online testing phase, we utilized a Redmi Note 10 Pro, featuring MIUI 12.5.1 based on Android 11 and equipped with 8GB of RAM. This device was selected for its capability to represent a typical user experience in terms of processing power and memory availability.

The APK dataset was sourced from AndroZoo[8], encompassing a range from the years 2021 to 2023. A total of 2000 APKs were gathered from prominent application markets, including play.google.com, appchina, and anzhi. Out of these, 1000 APKs were designated as the test set for detecting repackaged application pairs, while the remaining 1000 were utilized for extracting images to train and determine the image centers essential for the LSH algorithm. Figure 3 provides a detailed breakdown of the APKs' origins and their respective roles in the evaluation process.

The following subsections will delve into the specifics of the runtime and offline analysis, along with a detailed examination of the detection effectiveness of ProDroid.

6.2 Static Analysis

We download 120 apps from Google Play Store, repackage and obfuscate 30 of them by modifying all images of packages a little. We then draw the histogram of the similarity of irrelevant package pairs and repackaged package pairs calculated with their birthmarks, respectively.

It can be seen that the similarity of irrelevant package pairs are all below 0.8 and the similarity of repackaged package pairs are all above 0.4, thus evidently distinguishing the repackaged package pairs.

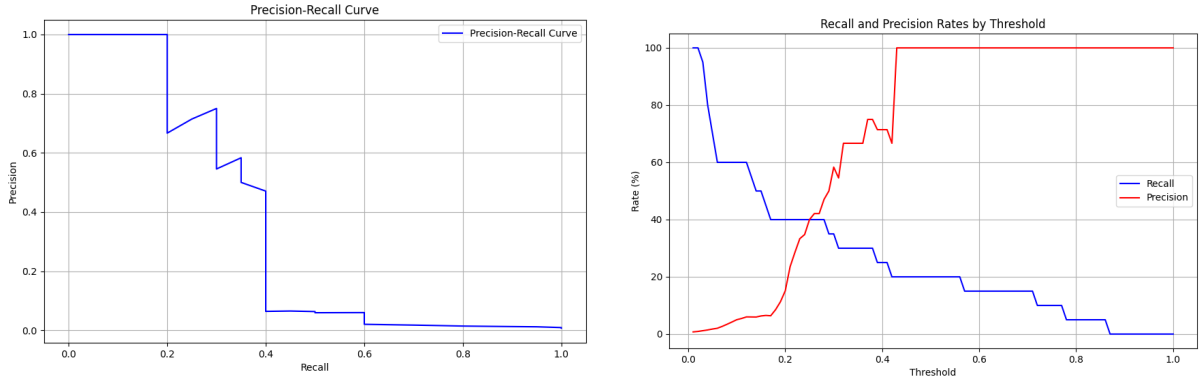

Figure 3: FsquaDRA's ROC and Recall-Precision Graph

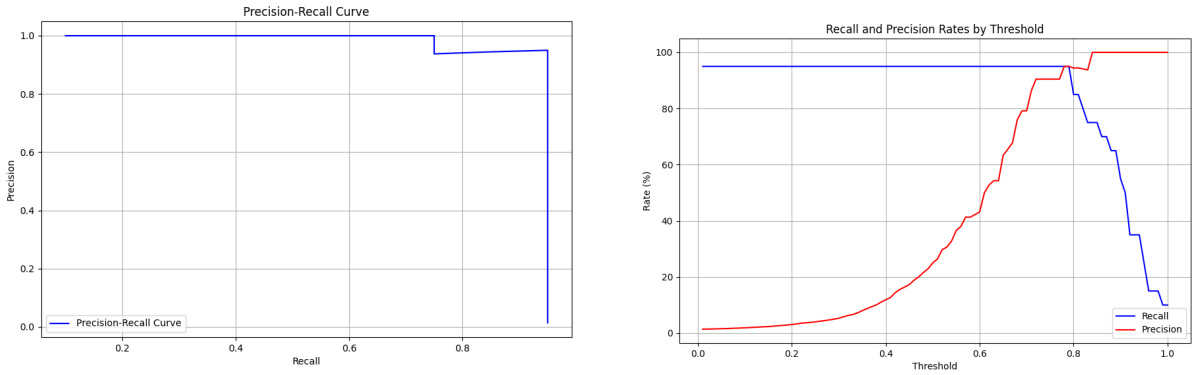

Figure 4: ProDroid's ROC and Recall-Precision Graph

6.3 Experiments

6.3.1 Dataset Description. Our experiment was conducted on a curated dataset consisting of 300 software packages. These packages were sourced from a variety of repositories to ensure a diverse range of applications. In addition to these, 20 packages were selected for repackaging, resulting in a total of 320 applications forming the test set. The repackaging process was conducted under controlled conditions to mimic common repackaging scenarios in the wild.

6.3.2 Experimental Design. The primary objective of the experiment was to compare the effectiveness of FsquaDRA and ProDroid in detecting repackaged software. To this end, each tool was independently used to analyze all 320 applications in the test set. The tools were configured according to their default settings, and each application was tested multiple times to ensure consistency in the results.

6.3.3 Performance Metrics. The performance of FsquaDRA and ProDroid was evaluated based on two key metrics:

- **ROC (Receiver Operating Characteristic) Curve:** This metric helped in understanding the true positive rate versus the false positive rate of both tools at various threshold settings.
- **Precision-Recall Statistics:** Precision and recall rates were calculated for each tool, providing insight into their accuracy and reliability in detecting repackaged software.

6.3.4 Data Analysis. The collected data from the experiments were subjected to rigorous analysis. ROC curves for each tool were plotted to visually compare their performance. The Area Under Curve (AUC) was calculated for both tools, with a higher AUC indicating superior performance. Furthermore, precision and recall rates were computed at different similarity threshold levels to determine the optimal setting for each tool.

6.3.5 Findings. Preliminary analysis indicated that ProDroid outperformed FsquaDRA in terms of AUC. Specifically, ProDroid achieved its best differentiation performance at a similarity threshold of 0.8, suggesting a more robust detection capability in this setting compared to FsquaDRA.

REFERENCES

[1] Yury Zhauniarovich, Olga Gadyatskaya, Bruno Crispo, Francesco La Spina, and Ermanno Moser. FSquaDRA: Fast Detection of Repackaged Applications. In

Proceedings of the 28th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy, DBSec '14, pages 131–146, 2014.

[2] UI/Application Exerciser Monkey | Android Studio | Android Developers. The Monkey is a program that runs on your emulator or device and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events.

[3] Jyoti Gajrani et al. Android Rooting: An Arms Race between Evasion and Detection. *Hindawi*, 2017. This paper discusses the Android rooting process and includes a detailed explanation of the Xposed framework's role in enabling method call hooking and custom code injection to manipulate application behaviors.

[4] Christoph Zauner. Implementation and Benchmarking of Perceptual Image Hash Functions. Technical report, Upper Austria University of Applied Sciences, Hagenberg Campus, July 2010.

[5] k-means clustering - Wikipedia. k-means clustering is a method of vector quantization, originally from signal processing, that is popular for cluster analysis in data mining.

[6] MinHash - Wikipedia. MinHash (or the min-wise independent permutations locality sensitive hashing scheme) is a technique for quickly estimating how similar two sets are.

[7] Locality-sensitive hashing - Wikipedia. In computer science, locality-sensitive hashing (LSH) is a fuzzy hashing technique that hashes similar input items into the same "buckets" with high probability.

[8] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.