CloneSpot: Fast detection of Android Repackages

Ignacio Martín[a,*], José Alberto Hernández[a]

[a]*Universidad Carlos III de Madrid, Avda de la Universidad 30, Leganés, Madrid, Spain*

Abstract

Repackaging of applications is one of the key attack vectors for mobile malware. This is particularly easy and popular in Android Markets, where applications can be downloaded, decompiled, modified and re-uploaded at a very low cost. Detecting clones and victims is often a hard task, especially in markets with several million of applications to analyze, such as Google Play Store. This work proposes CloneSpot, a novel methodology to efficiently detect Repackaged versions of Android apps using Min-Hashing techniques applied to applications' meta-data publicly available at Google Play. We validate our approach by analyzing 1.3 Million of applications collected from Google Play in September 2017, from which around 420K are detected as potential repackaged or victim versions of other applications.

Keywords: Android Malware; Repackaging; Min-Hashing; Meta-information

1. Introduction and Motivation

By design, Android applications' code, based on Java, can be easily decompiled, modified and uploaded to application markets again, enabling easy and fast creation of duplicated versions with additional code chunks that modify in some way the original application behavior. This process is typically referred to as *repackaging* of Android applications. Repackaging has become the most extended method for disguising and distributing malware applications in Android markets, including Google Play [23].

Many Android markets allow *repackagers* to easily upload and disguise their modified versions into them, often imitating as much as possible the victims' *meta-data* to increase their chances of luring users to get it installed. This *meta-data* or *meta-information* refers to those descriptive elements regarding an application available for users prior to download that give a description of the application functionality and requirements. Meta-data includes items like title, description, snapshots of the application, etc.

Hence, the ability of cloning any application from an Android market is within reach of anyone. In spite of its importance, the addition of malware, such

*Corresponding author
Email address: ignmarti@it.uc3m.es (José Alberto Hernández)

as botnets, APTs or phishing, is not the only target for developers to repackage any application; repackaging can also be used to introduce advertisements in an ad-free application or even swap the advertisement account of a legitimate developer by that of the repackager aiming at monetary gain.

Consequently, there has been an explosion on the development of repackaged applications or application clones, involving a large number of developers, areas and potential victims, which has become an issue for the Android Platform; for instance, 80% out of the top 50 most downloaded applications in the market have a repackaged version online[1]. While many researchers have typically addressed this problem by analyzing and inspecting application code, in this work, we consider a different approach targeting meta-information only.

Generally, repackagers' ability to alter meta-information with respect to that of the victim is very limited, since many repackaging applications or clones solely rely on being almost identical to their victims. This way, many users will be tricked to install a clone under the belief that they are installing the legitimate application. For instance, many repackaged applications are uploaded into markets with a very tiny change in the title (i.e. Capitalization changes, tildes or even tailing dots).

As a result, many repackaged applications are near-exact imitations of legitimate applications sometimes capable of fooling even advanced users. Consequently, comparing application's meta-information could unveil potential cloned applications, as their meta-data will be largely alike. Besides, this approach might discover any type of plagiarism as well, since applications with similar meta-data may be legitimate but copycats from a conceptual viewpoint.

This work proposes *CloneSpot*, a repackaging detection approach that targets applications' meta-information. Concisely, we use the Min-Hashing algorithm to aggregate similar applications together within the meta-data of 1.3 Million applications downloaded from Google Play in September 2017. Accordingly, applications will be ranked in terms of their local similarity to enhance application analysis by targeting first similar applications. Finally, the applicability of this approach will be demonstrated through a Proof of Concept (PoC) consistent on a lightweight real-time web-service capable of retrieving similar applications from the 1.3 million applications of the Google Play collection.

2. Previous Work

Many researchers have targeted repackaging detection from different flanks. One of the main tracks has been the search for similarities in application code: the authors in [2] compute the Control Flow Graphs of each application's code and cluster them to find apps with similar execution flows; *Andarwin* [5] computes features out of applications' code through Local Sensitive Hashing and

[1]See `https://blog.trendmicro.com/trendlabs-security-intelligence/a-look-into-repackaged-apps-and-its-role-in-the-mobile-threat-landscape/`, last access: October 2018

groups the application through Min-Hashing on a subset of such features.

At the interactive level, User Interface (UI) code and displays have also been targeted. Chen et al [3] develop a large-scale system which compares UI code to detect similar structures and intersections. The authors in [20] propose a novel approach based on analyzing the UI birthmarks of each application, clustering them based on Local Sensitive Hashing (LSH) and detect the application pairs with the highest similarity. In addition, RepDroid [24] automatically detects duplicates by extracting their Layout Group Graph (LGG) from UI traces.

Execution patterns of applications have been explored too. In [9], the authors propose MIGDrodid, a system based on the comparison of *Method Invocation Graphs* of each app that assign to each of them a *threat score*. Guan et al [7] propose a system to detect similarities in the input/output's symbolic representations of each app. Furthremore, the authors in [6] propose to look for repackaging indicators by detecting anomalies introduced by *Smali* decompilers in the data section of the dex code.

At market scale, many authors have addressed the problem by proposing scalable solutions, such as Andradar [16], which monitors several markets on real-time and tracks application deletions and other changes. Indeed, many of the proposed solutions are scalable [3, 8], able to cope with the daily-increasing pace of malware generation. In addition, the author in [12] performs large-scale analysis of $200,000$ android apps to determine which ones have been cloned.

Lately, there has been some focus on meta-data. The authors in [22] inspect such features and provide some insights on suspicious general-type malware apps, providing some regular patterns specially focused on permissions. In addition, the authors in [8] develop a system which checks incoming APKs with a whitelist database of applications and compares application icons in case no match is obtained. In [13] the authors propose computing pairwise metrics of different meta-information fields to detect application copies, even though it is unclear what the computational complexity of the approach is.

Actually, many authors have studied the risk different sensitive applications have to become victims of repackaging, such as bank applications [10], messaging applications [17] or even Android Home devices [4]. Nevertheless, some authors have attempted to use repackaging to enhance security or audit applications, like in [1], where the authors propose including a privacy reporter component in applications to audit the use of personal data or in [21] that proposes injecting a user-level sandbox into applications through repackaging.

Other countermeasures have been proposed to prevent repackaging: The authors in [19] propose an anti-plagiarism mechanism based on Copyright Watermarking that embeds applications with a way to verify authorship; Zhou et al propose in [26] a system to reproduce and watermark an Android applications based on dynamic graph mechanisms. Similarly, the authors in [11] propose to give applications *self-protection* capabilities by introducing detection codes into bytecode of apps. In a very recent approach [25], malware-style *Logic Bombs* are introduced to protect applications by corrupting their code upon repackaging detection.

A recent survey on repackaging detection methods can be found in [18]. Interestingly, there are not many works analyzing repackaged applications and their origins, except from the work from Li et al [15] where the authors perform an extensive analysis of *piggybacked* apps to understand their basic characteristics.

CloneSpot is an innovative approach targeting meta-information of applications extracted from Android application markets instead of actual application's code or execution patterns. Thus, CloneSpot yields better time performance at market scale as compared to code inspection approaches [2, 5, 9]. CloneSpot outputs sets of potentially repackaged applications that can be feed other sandboxing applications to further conclude whether there is indeed a real clone or not; ultimately, a malware analyst should manually verify whether or not suspicious applications are truly clones.

The rest of this paper is structured as follows: Section 3 details the crawling process of Google Play applications' meta-data; Section 4 extensively explains how clones are grouped and ranked. Section 5 offers some hints to validate our approach and Section 6 summarizes the conclusions and findings of this work.

3. Data Collection and Inspection

Google Play is the official Android market and has one of the largest collections of apps. It is managed by Google and contains over a million applications. This market is accessible through Android devices natively and also via web[2]. In this web, each application has its own page where meta-data, related applications, information about the developer and installation options for each user's devices are displayed.

We have developed a two-stage crawler with a twofold objective: (1) quickly elaborate a list containing the URL links of a large majority of applications in Google Play through recursively navigating links and (2) download all the meta-information (not the apks) contained in application URL. Firstly, our system starts by downloading Google Play's home page to identify and recover all application links there; then this process is recursively repeated throughout all the new links obtained until no new links for applications are obtained. Usually, most links appearing at each page correspond to customized suggestions or other apps from the developer. After some hours, we obtain a long list of apks URLs. In the second step, we have developed a multi-machine parallel crawler which downloads, parses, extracts and stores the meta-data of each URL obtained in the first step, collecting nearly 1.3 Million app's meta-information in a few days (Intel Xeon E5-2630 server with 24 cores 190 GB RAM memory). After removing different versions of applications, the size of the collection is reduced to exactly $1,288,643$ applications.

For each application, we have collected the following meta-data:

- **Title**: The application title.

[2]See: `https://play.google.com`, last access: October 2018

- **Description**: A textual description of the application and its features. It can be written in different languages and has a variable extension.
- **Category**: The category of the application, selected among all Google Play predefined categories.
- **Developer Name**: The name of the developer account.
- **Identifier**: The Google Play unique identifier for each application.
- **Ratings**: Google Play allows to rate applications between 1 and 5 stars. For each application, we download the number of votes for each star.

4. Fast Detection of Application Duplicates

4.1. Min-Hashing for Application Clustering

Essentially, most repackagers are reluctant to undertake major modifications to application's meta-information, as it reduces their chances to be mistaken for the real application. Indeed, many cloners stick to minor changes to one or more of these meta-data fields, such as changing some pixels in logos or subtly modifying titles or descriptions in a way that clones are almost identical, typically relying on users passing over them and believing they are installing the *victim application*.

Thus, the most straightforward and näive procedure to detect clones would be a brute-force approach to compute the pairwise similarity of serialized meta-data entries and rank them accordingly for manual inspection. However, this solution is computationally costly in order to perform operations between every pair in a set of 1.3M applications.

In this light, our approach, *CloneSpot*, relies on a previous stage involving similarity clustering to aggregate similar meta-data applications together into *app-sets* containing potentially similar applications and reduce this way the effective number of pairwise comparisons to compute. To perform similarity clustering, there are number of techniques that can be used to identify repackaged applications based on similar meta-data, namely:

- Text retrieval based approaches: *TF-IDF* ranks documents' words according to the inner and outer frequencies of their words and enables clustering according to them. *Concept Mining* follows a similar approach of generating vector-like definitions for each text following concept-mapping and moderated text translation approaches.
- Alternatively, modern hashing techniques, like *BitShred* and *Min-hashing* rely on computing hashes from an input text and select some of them as indexes or *buckets* into which items are assigned. The key of these approaches is that similar enough items are forced to collide and fall into the same bucket, being instantly grouped by similarity. In general, Min-hashing is known to provide more adjusted similarity estimations than BitShred.

The issue with text retrieval approaches is that they are conceived as a text to number mapping approaches that need to be clustered with standard approaches that do require defining number of clusters beforehand. On the contrary, hashing techniques provide a very fast single-step text-similarity clustering, making them very suitable for our purposes.

Therefore, we select Min-hashing for application clustering. The *hashing trick* or *Min-Hashing* is a very fast algorithm for estimating how similar (in terms of Jaccard similarity) two sets are. Min-Hashing relies on splitting strings into several chunks of the same length k called *shingles* and computing a unique-output function (i.e. common hash functions like MD5 or SHA1) over each chunk. Consequently, each signature produces a set of numbers (from the hexadecimal representations of hashes), of which the minimum is selected as the *Min-Hash.* Then, the Min-Hash value of each application serves as the index or bucket identifier for each different group of applications or cluster.

Min-Hashing relies on set groups theory, and specifically, it has been demonstrated that the probability of two signatures falling in the same group is approximately equal to the Jaccard distance between them. The Jaccard distance between two sets of shingles A and B is defined as follows:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

that is, the amount of shingles present in both sets (intersection) divided by all shingles appearing in any of the two sets (union). Consequently, Jaccard distance is bounded between 0 and 1 and the more similar two items are, the larger their Jaccard distance and therefore, the more likely they will output the same Min-Hash falling into the same bucket. A detailed explanation of Min-Hashing, Jaccard distance as well as the specific details and demonstrations may be found in [14].

Thus, we compute a *serialized* field for each application which consists on a joining its *title*, *category*, *description* and *developer name* fields using blank spaces as a separator. Min-Hashing of this serialized field is performed using a shingling key of $k = 7$ and applications obtaining the same Min-Hash value are directly aggregated together.

Single-application groups are removed since no application similar enough has been found within the dataset, which indicates low repackaging risk. Furthermore, groups containing applications from the same developer are also removed for two reasons: (1) it makes no sense for developers to wrongfully repackage their own applications and (2) many developers publish new versions of the same application, which usually share most of the meta-data from their previous versions.

4.2. Market-scale Detection of Application Duplicates

A common problem in malware analysis is the overwhelming amount of work required to detect, identify and obtain a signature from each sample, which is becoming even worse due to the latest advancements of malware development.

In the case of repackaging it is even worse, since the potential victim applications are accounted by millions and the detection and traceback of each sample to their victim is almost impossible in such a vast ecosystem.

Following the CloneSpot methodology, potential victims and perpetrators are aggregated together inside the same Min-Hash bucket forming an *app-set*. Hence, CloneSpot not only procures victim's traceback to analysts, but also potentially cloned and victim applications in groups that can be ranked to improve the time and effort management as the analysts just need to confirm or reject the relations inside an app-set.

In fact, the number of app-sets containing less than 10 applications is $44,729$, around 79% of the total. On the other side, there are $1,273$ groups containing more than 100 applications. The number of apps in those groups suggest they are not capturing any relevant similarity. Therefore, we remove app-sets of sizes larger than 100, as their Min-Hashes are based on common shingles (i.e. "the appl", "applica", etc) that have low risk of repackaging.

Consequently, we obtain a total of $54,944$ app-sets containing an overall amount of $419,830$ applications, which directly filters out 68% of the dataset that fell into single-sized app-sets and are not suspects of repackaging. App-sets provide similar application groups that could be potential cloners or victims, but these groups have to be manually inspected to verify these hypotheses. Still, nearly 55K groups are quite many applications for one (or many) analysts to easily deal with manually.

At this point, group-wide pairwise operations can be considered thanks to the potential reduction of the comparison space introduced by Min-Hashing based clustering. Indeed, groups contain an average 18.9 members, although the median value is 2 and the third percentile (75%) is 8, indicating a generally small app-set size that potentially contains a cloner and a victim.

In this case, performing all within app-set comparisons requires roughly 1.5 million operations (assuming 8 apps per app-set: $\binom{8}{2} \times 54,944$), which is a very manageable number as compared to the 1.6 trillion operations required just to compute the pairwise similarities of just two applications with all the rest in the raw dataset.

Although some degree of plagiarism in meta-data could be tolerated, it is very unlikely that nearly-identical applications' meta-information elements occur by chance. Therefore, we compute pairwise similarities over app-sets following the assumption that higher similarity scores are more likely to be clones and consequently should be analyzed before.

Precisely, we compute two different metrics for two items from each application locally: Title and description. Next, *edit distance* and *cosine similarity* are introduced as the similarity scores for title and description respectively and combined into a local scoring scheme to rank application groups.

4.2.1. Edit Distance of Application Titles

Many repackaged applications attempt to make very subtle changes to their victims' titles to avoid detection. Those typically range from changes in the capitalization of some letters to the addition of tiny or unnoticeable characters at the end, like dots. Usually, these changes are unnoticeable to the user eye unless she is prevented.

Hence, the observation of any two applications with almost the same title and some small changes is a pointer to potentially cloned or fraudulent applications. In order to quantify these changes, we compute the *edit distance* between the titles of applications in the same app-set.

The *Edit distance* between two strings a and b accounts the number of modifications to be made to a until it becomes b. For instance, the edit distance between *Hello, World* and *Hello, World!* is 1, since only one change is required to reach the second version: Adding an exclamation sign at the end of the first version.

Edit Distance itself allows detecting the applications with most similar titles within each app set and provides an insightful measurement of the changes made to any potential victim's title. By averaging all the edit distances in the group, we obtain a hint on how all titles in each group are related.

4.2.2. Cosine Similarity of Application Description

Some repackaged samples tend to paraphrase as much as possible, when not directly copy, their victim's description. Certainly, the most similar two descriptions are, the more suspicious those apps are. Repackaging is just one possibility, but plagiarism or even unfair competition are other options for this undesired behavior. In some repackaging cases, developers attempt to modify their clone description somehow, but the information contained in the message has to be practically the same anyway.

The main issue regarding descriptions is that they can be large sequences of free text, including many words in different languages which are not trivial to analyze. To overcome this, we simplify descriptions by transforming them into their *Bag of Words* representations. Then, *cosine similarity* is computed over the *Bag of Words* of different applications' descriptions to obtain a simple and language-independent measure of message similarity.

The *Bag of Words* (BoW) representation of each description is obtained through *tokenizing* the texts into words (using blank spaces and removing trailing characters, such as commas or dots) and accounting for the frequency of appearance of each word within the description. The *Cosine Similarity* is computed over BoWs by homogenizing them into a common space where all terms appearing in each BoW is considered a dimension and all BoW are converted to the same dimension space, that is, include all dimensions (adding zeros when not present). The value for each term dimension is the number of times such word appears and the cosine similarity is computed as the angle each pair of vectors forms.

In this case, Cosine Similarity is bounded between 0 and 1 and produces a broadly accepted estimation of the *proximity* of the two BoWs, effectively measuring how similar two descriptions are (without considering word collocation).

Cosine similarity of two applications $\vec{a}$ and $\vec{b}$ is computed as follows:

$$\vec{CS}(\vec{a},\vec{b}) = \frac{\vec{a}\cdot\vec{b}}{\|\vec{a}\| \times \left\|\vec{b}\right\|}$$

where $\vec{a}$ and $\vec{b}$ are vectors of size i, which is the total amount of unique terms in both BoWs. Scoring a zero is equivalent to being completely different and scoring a one is complete identity, including the appearance of the exact same terms. In practice, any two applications obtaining a high cosine similarity score indicates that their description's messages are very strongly related.

4.2.3. Application Ranking

We have indicated above how to measure pairwise similarity of titles and descriptions as sharper indicators of repackaged or plagiarized applications. In order to rank app-sets for their inspection, both metrics need to be merged into a single comprehensive scoring scheme that summarizes the insights provided by both.

In this light, we define the *Application Similarity Index* (ASI) between apps a and b as the division between description of median cosine similarity and average title edit distance:

$$ASI(a,b) = \frac{median(\vec{CS}(a,b))}{\varepsilon(\vec{ED}(a,b))}$$

where $\vec{CS}$ represents each pairwise cosine similarity within each app set and $\vec{ED}$ does the same for each pairwise edit distance. This metric modulates the similarity of descriptions by their title similarity: if two titles are not very similar and, thus, have a large edit distance, their description's similarities will have less importance.

This way, the scoring forces that the more similar in both ways two applications are, the topmost they will appear in the ranking, following the intuition that two almost-identical applications are more likely to be clones than two less similar applications. Recall that this score is locally computed for each app-set, i.e. only applications within app-sets are required, reducing computation complexity and improving computation efficiency by means of, for instance, parallelization.

As a result, app-sets can be sorted to conform a list of *potentially cloned applications* dependent on local pairwise similarities once they have been scored. The ASI metric allows CloneSpot provides a ranked list of potentially repackaged app-sets, ordered by their inner-similarites, helping to set near identical applications to be inspected first. Malware analysts may therefore proceed in order to confirm or reject whether any app-set contains repackaged applications.

Fig. 1 visualizes the scoring for all the applications in the dataset by means of a sorted barplot and its distribution boxplot. In general terms, there are very few applications scoring high in the ranking, as the scoring scheme is very sensitive to large modifications on titles.

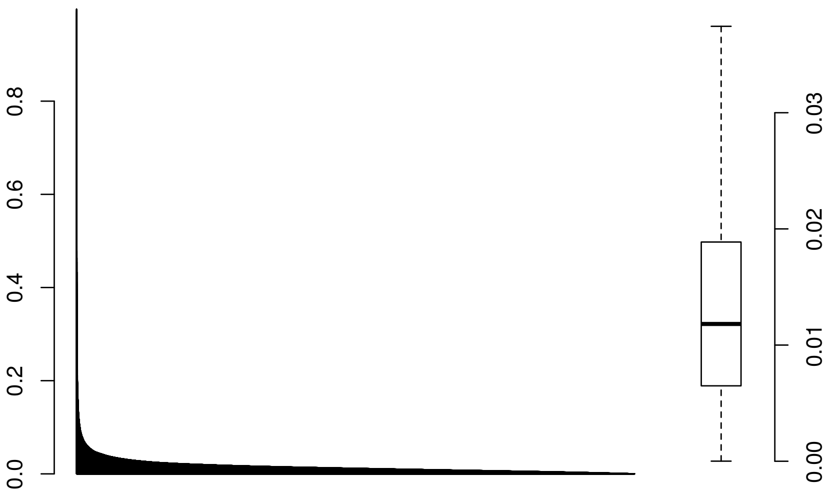
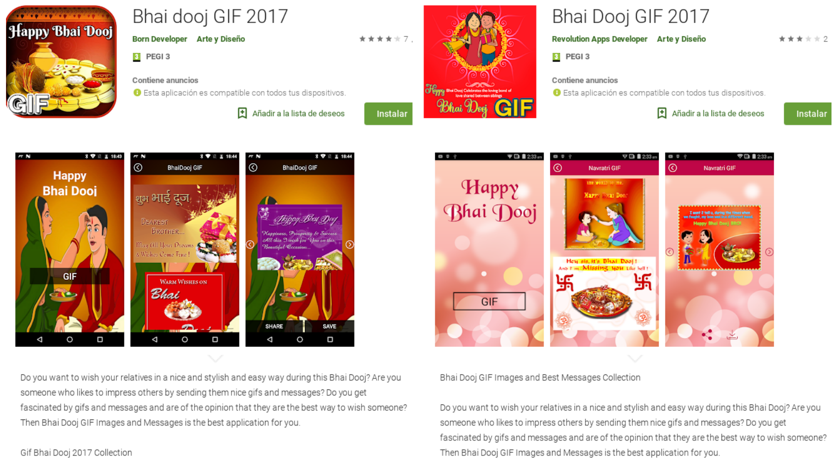


Figure 1: App-sets ranking for the Google Play dataset

Nonetheless, this zipf-like pattern appears due to many small sized app-sets on the top of the ranking, most of them containing only two applications (in median and third percentile), which are very similar between them, which increases the obtained score. These should indeed be the first groups to inspect, as they include large similarities, even with respect to the rest of app-sets.


Bhai dooj GIF 2017

Born Developer Arte y Diseño

PEGI 3

Contiene anuncios

Esta aplicación es compatible con todos tus dispositivos.

Añadir a la lista de deseos Instalar

Do you want to wish your relatives in a nice and stylish and easy way during this Bhai Dooj? Are you someone who likes to impress others by sending them nice gifs and messages? Do you get fascinated by gifs and messages and are of the opinion that they are the best way to wish someone? Then Bhai Dooj GIF Images and Messages is the best application for you.

Gif Bhai Dooj 2017 Collection

Bhai Dooj GIF 2017

Revolution Apps Developer Arte y Diseño

PEGI 3

Contiene anuncios

Esta aplicación es compatible con todos tus dispositivos.

Añadir a la lista de deseos Instalar

Bhai Dooj GIF Images and Best Messages Collection

Do you want to wish your relatives in a nice and stylish and easy way during this Bhai Dooj? Are you someone who likes to impress others by sending them nice gifs and messages? Do you get fascinated by gifs and messages and are of the opinion that they are the best way to wish someone? Then Bhai Dooj GIF Images and Messages is the best application for you.

Figure 2: Top app-set's applications in the *ASI* ranking

As an example, consider app-set number one in the ranking: *Bhai dooj GIF*

2017 and *Bhai Dooj GIF 2017*, which scores 0.9979 in ASI, 1 in Edit Distance and 0.9979 in Cosine Similarity. Fig. 2 depicts the Google Play's home pages of both applications.

This is a very illustrative case of repackaging, not necessarily for malware introduction, but for plagiarism. The lone difference between titles is the capitalization of the initial letter in a middle word and the description is practically the same except for some minor details. Moreover, it can be observed that although images are not the same, the application UI structure is very similar. Despite both applications could pass by legitimate, since their appearance is different and they could be thematically the same, the degree of lookalike of title and description as well as the structural similarities in the UI points to a clear case of repackaging where the clone could be using the legitimate application for monetary gain through changing the original advertisement account.

CloneSpot is described in Alg. 1, which summarizes methodologically each of the steps performed to aggregate and rank the different app-sets from raw applications to the final result. In addition, this workflow represents the minimum steps needed to aggregate applications into app-sets for production.

```
Data: CloneSpot: Google Play meta-data from 1.3M applications
Result: Ranked List of app-sets
1. Min-Hashing Clustering;
forall the app in allApplications do
    Split in Shingles $k = 7$;
    Compute Min-Hash;
    Send app to Min-Hash Bucket;
end
2. Clean app-set list and Compute Local ASI;
forall the app-set in groups do
    if Single-sized app-set or Single developer app-set then
        Remove app-set;
    end
    else
        Compute Title Edit distance;
        Compute BoWs and Cosine Similarity;
        Compute ASI;
    end
end
3. Sort app-sets according to ASI;
```

Algorithm 1: Summary of the repackaging detection pipeline

Next section evaluates the ability of this approach to detect and rank clones within Google Play and presents a Proof of Concept(PoC) application to illustrate how this approach can be used in real implementations to improve security and help analysts, application markets or even developers.

5. Validation and Proof of Concept

After applying CloneSpot methodology to our 1.3M dataset, there are a total of 54,944 ranked app-sets containing potential victims or clones ordered according to their local pairwise similarity. In this section we aim to validate our approach through the following experiments:

- Number of applications no longer in the market by app-sets: We measure the number of app-sets containing at least an application that has been removed from market between September 2017 and May 2018.
- *CloneSpot*, a proof-of-concept web-service to demonstrate the applicability of this approach that is able to receive an application's meta-data and retrieve in real-time the most similar ones.

5.1. Apps removed from Google Play

Google Play applications are constantly revised and they may be removed from the market for different reasons, namely, developer withdrawal, non-compliance with market's policies or plagiarism. Removed applications are no longer downloadable within Google Play, being their application meta-information page unreachable.

In spite of their diversity, all these reasons involve some kind of shady or undesired behavior. In the case of policy noncompliance or plagiarism it is straightforward. Oppositely, it seems strange for a developer to be willing to remove their own applications from the market without a clear reason or incentive.

Hence, application removal can be considered a good proxy to validate the approach. In most cases, application removal is a sign indicating something wrong about it and, if such application belongs to a top ranked app-set, then the CloneSpot workflow will comply with certain requirements to fight malware.

In this light, we attempt to download again the pages for the applications in the dataset that have been indexed by CloneSpot into a valid app-set. Whenever an application is not found in Google Play, an HTTP 404 (Not found) error code is served, so by comparing the correct (code 200) vs Not found responses from Google Play, we can easily infer which applications have been removed since September 2017.

As a result, out of 419,830 applications discovered though CloneSpot in September 2017, we observe that 78,164 have been removed from the market in May 2018, the other 341,666 potential clones were still present in Google Play in May 2018. By October 2018, the number of removed applications has risen to 218,621, leaving just 198,651 applications in total, which shows that a considerable majority of applications detected by CloneSpot have had issues inside Google Play and have thus been removed one year later. These results somehow confirm that CloneSpot indeed identify applications that are eventually removed by Google Play.

Fig. 3 depicts the boxplots for app-set size, that is, the number of applications initially detected per app-set and its evolution over time (May and October

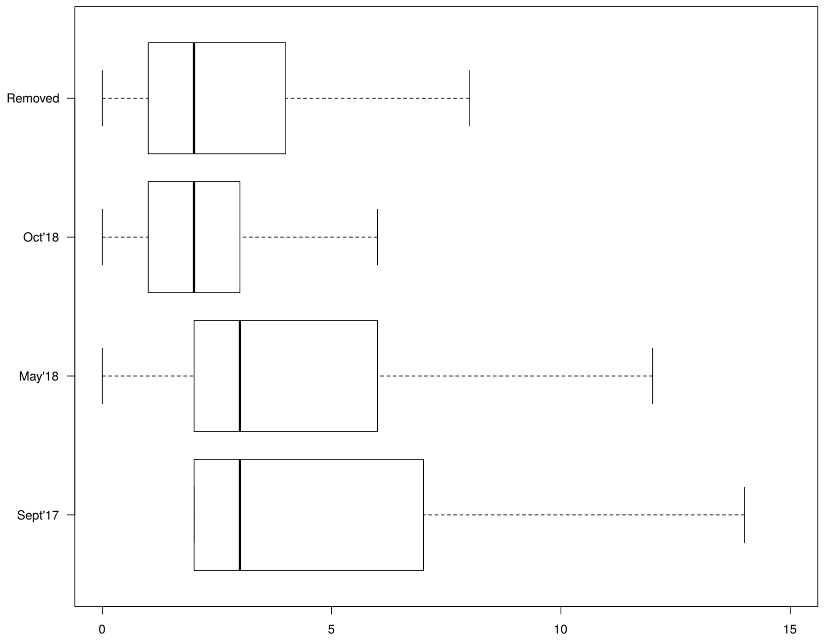


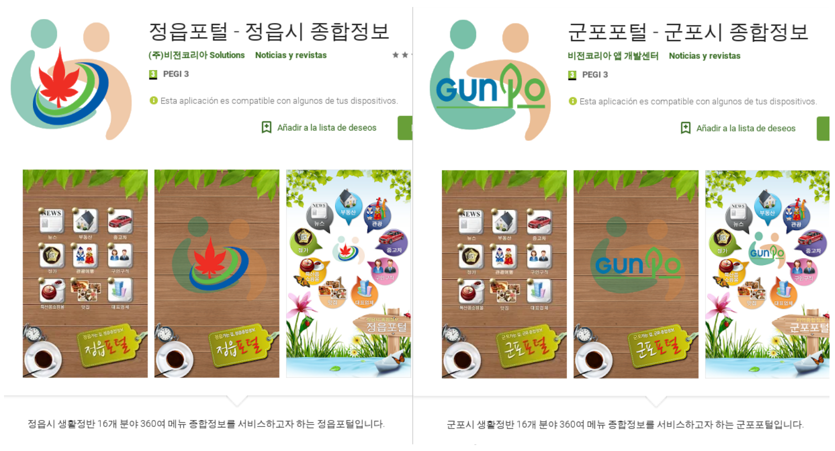
Figure 3: Distribution of app-set sizes over time

2018) as well as the distribution of removed apps per app-set. The figure reveals that the app-set sizes decrease over time (mainly percentiles 25, 50 and 75), showing that most app-sets contain only one, two or three applications by October 2018.

In addition, approximately 50% of the app-sets (in concrete 29,145 groups out of 54,944) have suffered application removal by May 2018 and even more, 47,887 by October 2018. These observations indicate that the trend in the survivor app-sets is to be left alone within their groups in such a way that app-sets would be reduced to the original legitimate app while potential clones or plagiarisms slowly disappear from the market. This somehow verifies that a large number of applications detected by CloneSpot are indeed conflictive applications, potential clones and victims together.

Moreover, when ranked according to our Application Similarity Index, most app-sets in the top of the rank contain at least one application that has been removed. In concrete, exactly 71 app-sets in the ASI top-100 have lost at least an application and, after manual/visual inspection of the other 29, we believe that 17 look very much as clones of others, due to similarities in User Interfaces, application structures, sizes and permissions. In other words, only 12 of the initial top-100 apps spotted by CloneSpot look like false positives according to our visual inspection.

Finally, Fig. 4 depicts another example of *"Potential Clones"*, in this case, it is clear that one of the two developers has copied entirely the application from


정읍포털 - 정읍시 종합정보

(주)비전코리아 Solutions Noticias y revistas

PEGI 3

Esta aplicación es compatible con algunos de tus dispositivos.

Añadir a la lista de deseos

정읍시 생활정반 16개 분야 360여 메뉴 종합정보를 서비스하고자 하는 정읍포털입니다.

군포포털 - 군포시 종합정보

비전코리아 앱 개발센터 Noticias y revistas

PEGI 3

Esta aplicación es compatible con algunos de tus dispositivos.

Añadir a la lista de deseos

군포시 생활정반 16개 분야 360여 메뉴 종합정보를 서비스하고자 하는 군포포털입니다.

Figure 4: Example of two similar applications detected by CloneSpot

the other, just changing the app logo by another one and performing slight modifications to description and title (two characters are changed). Nevertheless, both applications are so similar that they have obtained the same Min-Hash value of $0x21a62ee1184c08b3d8a9efba058338808cedaac$.

The example above belongs to an app-set of size 4 of which the other two applications have already been removed from Google Play. Its *ASI* score is 0.223, its edit distance is 4 and its cosine similarity 0.892. This application is still in the top 100, at position 76.

At a glance, these applications look like clones, not withstanding that further manual inspection is required to completely determine how these two applications are related: The Min-Hashing approach is capable of accurately grouping these similar applications together for faster inspection. Additionally, this example illustrates that the language is not a problem for the Min-Hashing approach, being capable of joining two applications regardless of the language their meta-data is written in.

These results show that CloneSpot is able to quickly detect, rank and sort accurately potential clones for manual inspection enhancing the analysis process by reducing the time required to find potential victims or clones within a large sample of applications.

5.2. *CloneSpot: Fast Retrieval for Potential Clones*

Previous sections have focused on grouping and ranking a large collection of applications for market-scale analysis based on meta-data. Although fast, analyzing an entire market requires the recollection of a market-scale dataset in order to obtain meaningful results, not to consider the moderate requirements of time and computing resources to complete the operation.

However, the actual computation of the Min-Hash value for a single application requires a small number of operations over few kilobytes of data, so it is very fast. In addition, market-scale analysis can be carried out in an offline fashion over a very large collection of applications, that could be one or more markets at once, and stored in a database where Min-Hash values can serve as an index for fast retrieval.

As a result, CloneSpot enables the search of application duplicates by querying a database that can be updated and extended in parallel to that process. The resulting system could return the app-set of similar applications to any given tuple of title, description, category and developer in real-time.

In this light, we develop *CloneSpotPoC*: a real-time retrieval system of potential clones given any sample application's title, description, category and developer name. This PoC uses the Min-Hashing value obtained by the CloneSpot methodology as a database index to optimize storage and improve query time. Indeed, the application can be used by different stakeholders within the Android ecosystem to reduce the incidence of plagiarism and malware at different levels:

- From a market perspective, This PoC can be used as a direct implementation of the CloneSpot approach, so any time a new application is uploaded it can be tested for plagiarism at upload time. Furthermore, the PoC can be extended to return the ranked list of app-sets according to the ASI scoring scheme.
- For analysts, the CloneSpot PoC can serve as a fast recommendation engine for potential victims to inspect when analyzing any malware sample that returns similar apps and their similarities with the target application.
- From a developer perspective, CloneSpot PoC is an useful tool to keep track of other similar applications as well as monitoring plagiarism and repackaging at any time.

The application is designed as a Rest API service that has been implemented using Java Servlets and MongoDB. The Web Service serves as a proxy for querying the Min-Hash-indexed database containing all app-sets found in our Google Play collection. The system returns for each query the title and url of every single application that can be similar to the one provided following the CloneSpot approach. To obtain similarity app-sets the user can send a request to any of these two methods:

- *getDuplicates*: HTTP GET method that receives a Min-Hash value in the request url and returns all the potentially similar applications to that given, that is, the app-set corresponding to that Min-Hash.
- *analyze*: HTTP POST method that takes as input a JSON-encoded representation of application meta-data (title, description, category and developer name) and returns all the similar applications by computing the Min-Hash value of the fields and returning the corresponding app-set (if any).

In addition, the service provides an *info* method containing the user guide and usage considerations. The reader should note that this service is presented to demonstrate the capabilities of the CloneSpot approach and does not have full production capabilities. The service is publicly available at: `http://163.117.192.31:8080/CloneSpot/commands/info`

6. Summary and conclusions

In sum, this work has presented *CloneSpot*, a methodology to detect Android repackaged applications using the meta-information available in most application markets. In order to cluster applications according to their textual similarities, we leverage the well-known Min-Hashing algorithm. CloneSpot yields a ranked list of app-sets, groups of potentially repackaged applications, to be either analysed by sandboxing repackaging tools or manually inspected and confirmed by a malware analyst.

We validated this approach by analyzing what happens to applications falling into multiple size app-sets (more than one application), showing that half of the app-sets have experienced a reduction in applications from September 2017 to October 2018 in Google Play. In addition, we have developed *CloneSpot PoC*, a Proof of Concept service that returns the most similar app-set when given an application's title, description, category and developer name in real-time to demonstrate the potential applications of CloneSpot methodology[3].

Finally, this work has shown that it is possible to quickly find potential clones comparing application meta-information through a language independent approach that enables within-group pairwise comparison. Future work will address the extension of the collection of applications to other Android Markets to enable and analyze cross-market analysis of potential clones.

Acknowledgments

The authors would like to acknowledge the support of the national project TEXEO (TEC2016-80339-R), funded by the Ministerio de Economía y Competitividad of SPAIN and the EU-funded project SMOOTH (Grant no. H2020-786741). In addition, Ignacio Martín would like to acknowledge the support of the Spanish Ministry of Education by means of the FPU grant he holds (FPU15/03518).

References

[1] Pascal Berthome, Thomas Fecherolle, Nicolas Guilloteau, and Jean-Francois Lalande. 2012. Repackaging android applications for auditing access to private data. In *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on.* IEEE, 388–396.

[3]Available at: `http://163.117.192.31:8080/CloneSpot/commands/info`

[2] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 175–186.

[3] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. 2015. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale.. In *USENIX Security Symposium*. 659–674.

[4] Taejoo Cho, Geonbae Na, Deokgyu Lee, and Jeong Hyun Yi. 2015. Account forgery and privilege escalation attacks on android home cloud devices. *Advanced Science Letters* 21, 3 (2015), 381–386.

[5] Jonathan Crussell, Clint Gibler, and Hao Chen. 2013. Andarwin: Scalable detection of semantically similar android applications. In *European Symposium on Research in Computer Security*. Springer, 182–199.

[6] Hugo Gonzalez, Andi A Kadir, Natalia Stakhanova, Abdullah J Alzahrani, and Ali A Ghorbani. 2015. Exploring reverse engineering symptoms in Android apps. In *Proceedings of the Eighth European Workshop on System Security*. ACM, 7.

[7] Quanlong Guan, Heqing Huang, Weiqi Luo, and Sencun Zhu. 2016. Semantics-based repackaging detection for mobile apps. In *International Symposium on Engineering Secure Software and Systems*. Springer, 89–105.

[8] Iakovos Gurulian, Konstantinos Markantonakis, Lorenzo Cavallaro, and Keith Mayes. 2016. You can't touch this: Consumer-centric android application repackaging detection. *Future Generation Computer Systems* 65 (2016), 1–9.

[9] Wenjun Hu, Jing Tao, Xiaobo Ma, Wenyu Zhou, Shuang Zhao, and Ting Han. 2014. Migdroid: Detecting app-repackaging android malware via method invocation graph. In *Computer Communication and Networks (ICCN), 2014 23rd International Conference on*. IEEE, 1–7.

[10] Jin-Hyuk Jung, Ju Young Kim, Hyeong-Chan Lee, and Jeong Hyun Yi. 2013. Repackaging attack on android banking applications and its countermeasures. *Wireless Personal Communications* 73, 4 (2013), 1421–1437.

[11] Fumihiro Kanei, Yuta Takata, Mitsuaki Akiyama, Takeshi Yagi, and Takeshi Yada. 2017. Poster: Protecting Android Apps from Repackaging by Self-Protection Code. (2017).

[12] Ayush Kohli. 2017. DecisionDroid: a supervised learning-based system to identify cloned Android applications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 1059–1061.

[13] Su Mon Kywe, Yingjiu Li, Robert H Deng, and Jason Hong. 2014. Detecting camouflaged applications on mobile application markets. In *International Conference on Information Security and Cryptology*. Springer, 241–254.

[14] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. 2014. *Mining of massive datasets*. Cambridge university press.

[15] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. 2017. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics and Security* 12, 6 (2017), 1269–1284.

[16] Martina Lindorfer, Stamatis Volanis, Alessandro Sisto, Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, and Sotiris Ioannidis. 2014. AndRadar: fast discovery of android applications in alternative markets. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 51–71.

[17] Su-Wan Park and Jeong Hyun Yi. 2014. Multiple Device Login Attacks and Countermeasures of Mobile VoIP Apps on Android. *J. Internet Serv. Inf. Secur.* 4, 4 (2014), 115–126.

[18] Sajal Rastogi, Kriti Bhushan, and BB Gupta. 2016. Android applications repackaging detection techniques for smartphone devices. *Procedia Computer Science* 78 (2016), 26–32.

[19] Chuangang Ren, Kai Chen, and Peng Liu. 2014. Droidmarking: resilient software watermarking for impeding android application repackaging. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 635–646.

[20] Charlie Soh, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, and Lipo Wang. 2015. Detecting clones in android applications through analyzing user interfaces. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. IEEE Press, 163–173.

[21] Jun Song, Mohan Zhang, Chunling Han, Kaixin Wang, and Huanguo Zhang. 2016. Towards fast repackaging and dynamic authority management on Android. *Wuhan University Journal of Natural Sciences* 21, 1 (2016), 1–9.

[22] Peter Teufl, Michaela Ferk, Andreas Fitzek, Daniel Hein, Stefan Kraxberger, and Clemens Orthacker. 2016. Malware detection by applying knowledge discovery processes to application metadata on the Android Market (Google Play). *Security and Communication Networks* 9, 5 (2016), 389–419.

[23] Timothy Vidas and Nicolas Christin. 2013. Sweetening android lemon markets: measuring and combating malware in application marketplaces. In *Proceedings of the third ACM conference on Data and application security and privacy*. ACM, 197–208.

[24] Shengtao Yue, Weizan Feng, Jun Ma, Yanyan Jiang, Xianping Tao, Chang Xu, and Jian Lu. 2017. RepDroid: an automated tool for Android application repackaging detection. In *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*. IEEE, 132–142.

[25] Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, and Zhoujun Li. 2018. Resilient Decentralized Android Application Repackaging Detection Using Logic Bombs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*. ACM, New York, NY, USA, 50–61. `https://doi.org/10.1145/3168820`

[26] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. 2013. AppInk: watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 1–12.