Detecting Clones in Android Applications through Analyzing User Interfaces

Charlie Soh, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, Lipo Wang
School of Electrical and Electronic Engineering
Block S2, Nanyang Technological University
Nanyang Avenue, Singapore 639798
{csoh004, ibktan, yauhen, elpwang}@ntu.edu.sg

***Abstract*—The blooming mobile smart phone device industry has attracted a large number of application developers. However, due to the availability of reverse engineering tools for Android applications, it also caught the attention of plagiarists and malware writers. In recent years, application cloning has become a serious threat to the Android market. In previous work, mobile application clone detection mainly focuses on code-based analysis. Such an approach lacks resilient to advanced obfuscation techniques. Their efficiency is also questionable, as billions of opcodes need to be processed for cross-market clone detection. In this paper, we propose a novel technique of detecting Android application clones based on the analysis of user interface (UI) information collected at runtime. By leveraging on the multiple entry points feature of Android applications, the UI information can be collected easily without the need to generate relevant inputs and execute the entire application. Another advantage of our technique is obfuscation resilient since semantics preserving obfuscation technique do not affect runtime behaviors. We evaluated our approach on a set of real-world dataset and it has a low false positive rate and false negative rate. Furthermore, the results also show that our approach is effective in detecting different types of repackaging attacks.**

Keywords—Android; Repackaging; Clone detection; User interface; Obfuscation resilient

I. Introduction

Android dominates the worldwide smartphone OS market share, holding close to 85% of the market share [1]. At the time of writing this paper, Google's official android application (app) market, Google Play, hosts up to 1.3 million apps [2]. One of the main reasons for Android to attract a large amount of developers could be due to the ease of creating an Android app, as the tutorials and tools for android app development are readily available. Moreover, mobile app developers are encouraged by the potential profits from creating apps as they can monetize through various monetization models such as paid downloads, in-app purchases, subscriptions, in-app ads, sponsorships and crowd funding [3].

Despite the fact that source code is usually unavailable for published Android apps, it is usually straightforward to reverse engineer Android apps due to the availability of reverse engineering tools such as apktool [4], dex2jar [5] and Dare [6]. Plagiarists can easily make use of these tools to disassemble the Android package (an APK file) downloaded from any Android market, modify its content, repackage it and sign it as their own app. This attack is known as repackaging or cloning. The increase in popularity together with the ease of repackaging Android apps draws the interest of plagiarists and malware writers.

A repackaging of the app by redirecting advertisement revenues and adding malicious payloads could potentially cause the hard working legitimate developers to lose their revenues and even reputations. A recent study by Gibler et al. [7] reported that with the assumption that the user who downloaded the clones would have downloaded the original app, legitimate developer lose about 10% of the user base to the clones. Zhou et al. [8] found that 86% of the malware samples are clones of the legitimate apps. In addition, apart from Google Play, there are numerous third-party Android markets available for the legitimate developers and plagiarists to upload their apps. Most of these third-party Android markets do not ensure for the quality of the apps that are published on their market. A study by Zhou et al. [9] found that out of 6 third-party markets, 5-13% of the apps hosted are clones. Based on the current Android market growth rate, we envision that these figures are likely to increase in the future.

From the above, we observed that app cloning is a severe problem that does not only affect the developers, but also destroy the health of the Android app market, hence proving the importance of detecting app clones.

Previous studies to perform Android app clone detections are mostly based on static code analysis [9-12]. However, these approaches may have some weaknesses. The practice of code reuse and the usage of the more sophisticated obfuscation technique may affect the detection of such approaches [13]. Zhou et al. [8] reported that malware writers tend to employ obfuscation to avoid detection. Furthermore, since source code is generally not publicly available, the analysis is usually conducted on opcodes. A study [14] on 30,000 Android apps shows that the apps with median size of 754KB have 20,555 median opcodes. With the need for analysis to be conducted across multiple markets, the number of opcodes that need to be analyzed would increase greatly, resulting in the "billion opcodes" problem [11].

In this paper, we propose a novel approach for cross-market Android app clone detection based on robust dynamic software birthmarks. The birthmarks are generated by leveraging on the view hierarchy information of the user interface (UI). The view hierarchy can be extracted in XML format when the activity is in the foreground of the system (displayed on the screen) where user can interact with it. The intuition behind our approach is based on the following observations:

(a) Obfuscation techniques that preserve semantics do not affect runtime behaviors. Therefore, dynamic software birthmarks have a much better obfuscation resistance compared to static software birthmarks.

(b) The plagiarists would want to keep the look and feel of the UI similar to that of the original app so as to leverage on the popularity of the original app. Furthermore, it may be easy to modify the position and size of the UI, but modifying the functionality of the UI requires more effort and understanding of the app code. Thus, app clones would likely have UI similar to the original apps.

(c) Unlike traditional programs, Android apps have a unique feature of having multiple entry points. We can leverage on this feature to access and extract information from different components of the app directly.

Our approach provides an alternative to the traditional code-based static analysis detection approaches and does not inherit all disadvantages of dynamic analysis approaches. The proposed approach has the following advantages:

(a) **Obfuscation resilient**: The UI information used in our approach for generating birthmarks does not depend on static analysis. On the contrary, the UI information is obtained during runtime, which is not affected by semantics preserving code obfuscation techniques. Thus, the birthmarks will be resilient to code obfuscation techniques.

(b) **No input generation required**: By leveraging on the Android app multiple entry points feature, we extract a list of activities from the app's package, and start each of them explicitly to obtain their runtime UI information. Hence, despite being based on dynamic analysis, our approach does not require the generation of inputs to navigate through the complex UI, which is a process that is hard to automate.

Research question: We would like to answer the following research questions in our study:

RQ1. Can our proposed approach effectively detect different types of application clones across multiple Android markets?

In this research question, we want to find out whether our proposed approach can effectively detect all kinds of clone attacks. Furthermore, due to the large number of apps across multiple third-party Android markets, we need to determine whether our approach is scalable, for it to be of practical use.

RQ2. What are the possible limitations of our approach and how can we overcome them?

In this research question, we seek to identify, from our experimental results, the possible weaknesses and limitations of our approach and what are the possible solutions to overcome them.

In summary, the main contributions of our work are as follows:

- We propose an approach to detect Android application clones based on the birthmarks generated from runtime UI information. To the best of our knowledge, we are the first to use runtime UI birthmarks for Android app clone detection.
- Our approach leverages on the multiple entry points feature of the Android system. Therefore, it does not require the generation of relevant inputs for execution of the entire app.
- We evaluate our approach on a set of real-world data collected from 4 different Android markets. Experimental results show that our approach has low false positive and false negative rates.
- We also evaluate the effectiveness of our approach on a collection of clone set data from another research group. The results show that our approach can effectively detect different types of clone attacks.

This paper is organized as follows: Section II presents the background. Section III describes the methodology of our approach in detail. Section IV covers the evaluation of our technique. Section V presents the discussion of our results. Section VI discusses related previous work. Section VII concludes the paper.

II. Background

A. Android Application Structure

An Android app is built from a combination of different components such as activity, service, content provider and broadcast receiver. An activity component provides a screen with UIs, while a service component performs operations in the background. The content provider manages the data for a set of shared apps and the broadcast receiver component provides response to broadcast announcements [15]. These components together with various resources are compiled and packaged into an APK file.

The components communicate with each other through intents. The intents can be thought of as a messenger that can be used to make requests to another component. There are 2 types of intents, explicit intent and implicit intent [16]. An explicit intent specifically states the fully qualified class name of the component, while an implicit intent states a general action to perform, without naming a specific class.

In this paper, we focus on the activity components. An Android app usually consists of a number of activities that are not closely related to each other. For example, an email app might consist of 3 activities, each with a different purpose, such as compose, read and list emails. If permitted, any one of these activities can be started by a different app [15]. Hence, unlike traditional programs that only possess a single entry point, Android apps possess multiple entry points.

B. User Interface

The UI components in an activity structure as a hierarchy of views (e.g., "*Button*", "*CheckBox*", "*TextView*") and view groups (e.g., "*FrameLayout*", "*LinearLayout*", "*RelativeLayout*"). Each of these views or view groups is tasked to manage a particular rectangular space within the activity's window. A view group may be a parent of multiple views and view groups and it provides a layout of the interface for its child views [17].

C. Uiautomator Tool

We use the *uiautomator* tool from the Android SDK to extract the UI information for analysis. *Uiautomator* is a testing framework that allows the developers to test the UI of their apps efficiently [18]. The tool provides a function that can be used to dump the view hierarchy of the current activity as an XML file. The generated XML file contains runtime information on all view objects in the activity. Each view and view group is represented as a node in the XML. In addition, each node has the exact same 17 distinct attributes with different possible values. However, if the *uiautomator* does not have access to a particular node, then the node will have an additional NAF attribute. Figure 1 shows an example of an activity's screenshot taken from an app and part of the corresponding XML file generated by the *uiautomator* tool for the activity.

D. Types of Application Repackaging

Generally, to repackage an app, the plagiarist will first obtain the APK of the target app. The plagiarist will then make certain modifications to the original app, such as redirecting advertisement revenue or injecting malicious payload. To avoid detection, the plagiarist is also likely to apply automatic code obfuscation techniques to the repackaged apps before signing it with a private key and publishing it on one or more Android market(s). In a recent paper, Zhang et al. [19] suggested that repackaging attacks can be classified into the following 3 categories:

Lazy Attack: A clone in the lazy attack category can have simple changes such as different author name or different advertisement. Automatic code obfuscation tools may also be applied without changing its functionalities.

Amateur Attack: In addition to automatic code obfuscation, an amateur attack also has changes to a small part of the app's functionality. This type of attack requires more effort and knowledge from the plagiarists.

Malware: In this attack, a malicious payload is added to the popular legitimate app to create a malicious app. The malicious app masquerades the original app by keeping the functionality and UI similar to the original app to leverage on its popularity.

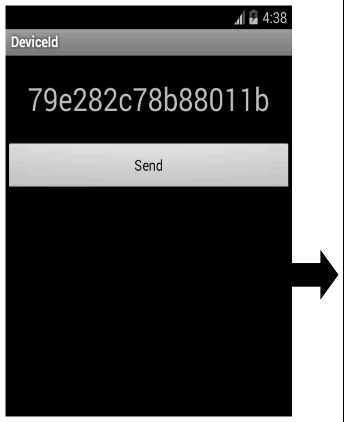
We will evaluate the effectiveness of our approach in detecting app clones from these 3 different forms of clone attacks, in Section V.

III. METHODOLOGY

In this section, we present in detail the methodology of our proposed approach. Firstly, given the APKs of the Android apps, we use an Android emulator to extract the names of the activities and execute them to gather their UI information in XML format. Secondly, we use 2 filters to exclude unnecessary information from the XML files and the remaining information are used to generate birthmarks. Thirdly, we employ locality-sensitive hashing (LSH) to find the near neighbors for the birthmarks. If the near neighbor of a birthmark contains more than 1 activity from another app, we apply the Hungarian algorithm to find the maximum similarity. Lastly, we cluster the apps into groups of clone sets. Figure 2 shows the overview of our approach.

A. Data Extraction

An Android app consists of multiple components. Each of these components is a unique point whereby the system can access the app. We make use of this feature to start each activity explicitly using explicit intents. By doing so, we avoid the hassle of generating inputs for the execution of each path in the app's control flow graph. However, a drawback of this method is that the components might not be actual entry points and some of them are not independent. These components cannot be accessed directly by using explicit intents.



```
-<node index="1" text="" resource-id="android:id/content" class="android.widget.FrameLayout"
package="com.langlearner.deviceid" content-desc="" checkable="false" checked="false" clickable="false"
enabled="true" focusable="false" focused="false" scrollable="false" long-clickable="false" password="false"
selected="false" bounds="[0,106][720,1184]">

    -<node index="0" text="" resource-id="" class="android.widget.LinearLayout"
    package="com.langlearner.deviceid" content-desc="" checkable="false" checked="false" clickable="false"
    enabled="true" focusable="false" focused="false" scrollable="false" long-clickable="false" password="false"
    selected="false" bounds="[0,106][720,1184]">

        -<node index="0" text="79e282c78b88011b" resource-id="com.langlearner.deviceid:id/textView1"
        class="android.widget.TextView" package="com.langlearner.deviceid" content-desc=""
        checkable="false" checked="false" clickable="false" enabled="true" focusable="false" focused="false"
        scrollable="false" long-clickable="false" password="false" selected="false" bounds="[0,106][720,288]"
        />

        -<node index="1" text="Send" resource-id="com.langlearner.deviceid:id/btnSend"
        class="android.widget.Button" package="com.langlearner.deviceid" content-desc="" checkable="false"
        checked="false" clickable="true" enabled="true" focusable="true" focused="false" scrollable="false"
        long-clickable="false" password="false" selected="false" bounds="[0,288][720,396]" />
```

Fig.1. Screenshot of an activity with partial corresponding XML

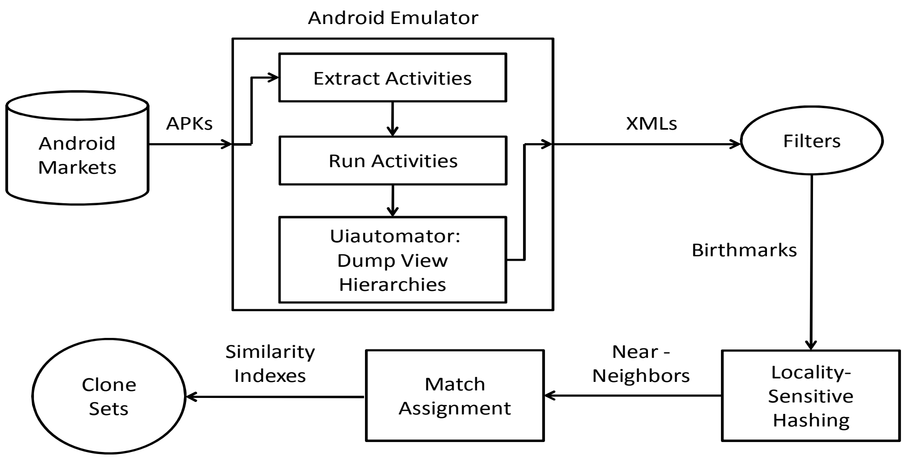


Fig. 2. Overview of our approach

Nevertheless, the same components that are inaccessible in the original app will most likely also be inaccessible in the clone. Thus, it may not affect the similarity between this pair of apps, or at least not significantly.

Even though this method poses a risk of losing information when activities fail to start, the complexity and required time is much lesser as compared to generating inputs for the execution of the entire app. The data extraction process is automated using Java programming language. In addition, we employ Android Debug Bridge (adb) and API libraries from the Android SDK [20] to interact with the APK files and Android virtual device. The details of the extraction process can be broken down into 2 main steps:

1. Extract activity names: All the activities in the app must be declared in the *AndroidManifest.xml*. The Android system will not see any activities that are not declared and therefore cannot execute these activities [21]. For each app, we extract the list of activity names from the APK file. The extracted names are the fully qualified class names of the activities and these can be used to start the activities explicitly. The activities can be classified into 2 general groups: app's activities and third-party libraries' activities. App's activities are the activities that are created by the developer of the app and are unique to the app. On the other hand, third-party libraries' activities are the activities that are included by third-party libraries. We do not consider third-party libraries' activities, as they can be easily changed and may induce false positives. For example, an advertisement library will usually include advertising activities and changing the advertisement library will also cause a change in the activities.

2. Obtain XML file: In order to dump the view hierarchy of the UI using *uiautomator* tool, the downloaded APKs first have to be installed no an Android environment. For this purpose, we use an Android emulator created from the Android virtual device (AVD) Manager. We use adb commands to install the app and explicitly start the extracted activities using their fully qualified class names. When the activity is started, we use *uiautomator* to dump the view hierarchy into an XML file. Thereafter, we stop the current activity and the process is repeated until we reach the end of the activities list.

B. Birthmark Generation

Software birthmark is a unique characteristic that the software possesses and serves as its identity. Software birthmark can be further classified as static birthmark or dynamic birthmark. In this paper, we extract the dynamic software birthmark of the app and use it for clone detection.

We implemented an XML parser in the Java programming language to generate the birthmarks. The birthmark of each activity is generated from their individual XML obtained from the previous process (Section III-A). As mentioned in Section II-C, each node in the XML has the exact same 17 distinct attributes with different possible values. An important attribute of each node is the "class" attribute, since the value of the "class" attribute denotes the type of view (e.g., "*Button*", "*TextView*", etc.) the node represents. Each birthmark is a vector where each element in the vector represents the frequency count of a unique combination of the view class, selected attribute and value of the selected attribute. If all 17 attributes were used to generate the birthmarks, each vector would contain a large amount of elements. However, we found that not all attributes are useful in detecting app clones. Therefore, to reduce the amount of computations, we apply 2 filters to exclude unnecessary information.

Filter 1 (*F1*): There are attributes that do not provide suitable information for clone detections. Instead, these attributes would introduce noise into our birthmarks and cause our detection to be less accurate. There are also attributes with string type values that can be easily manipulated and costly to compare. We analyze these attributes as follows:

- The "*index*" attribute simply represents the position of the node in the view hierarchy. The positions can be easily switched within or even out of the view group. By doing so, it would result in a different index value being assigned to the node.
- The "*text*" attribute represents the text that is displayed on screen. Changing the *strings.xml* file that can be found in the resource folder can easily change this text. For example, the plagiarists can change the text from '*username'* to '*login id'* or from '*email'* to '*E-mail*'.
- The value of the "*resource-id*" attribute may be empty when no resource is required for that particular view. There are 2 ways to access a resource, in code or in XML [22]. In any case, it can be easily modified at all places where the *resource-id* is expected.
- The "*package*" attributes represent the package name of the app. It is common for plagiarists to modify the package name to avoid detections. Another reason to modify the package name could be that some Android markets do not allow 2 apps with the same package name to be hosted at the same time on their market.
- The "*content-desc*" attribute is similar to the text attribute. Modifying the *strings.xml* can also easily change it.
- The "*bounds*" attribute represents the position and the area of the rectangular space controlled by the views and view groups. This can also be easily manipulated by modifying the layout of the corresponding XML found in the layout folder.

In summary, to reduce the amount of computations, *F1* excludes the following attributes from the birthmark: "*package*", "*index*", "*bounds*", "*text*", "*resource-id*" and "*content-desc*". The rest of the attributes represent either the state or the functionality of the view. Firstly, attributes that represent the state of the views are as follows: "*checked"*, "*focused"* and "*selected"*. Secondly, examples of attributes that represent the functionality are as follows: "*clickable"*, "*checkable"*, and "*scrollable"*.

Filter 2 (*F2*): Due to the nature of the class, the value of certain attributes in the particular class will always remain the same across activities and apps. For instance, a node with "*Button*" class will never have the value of the password attribute equals to true. Such attributes do not provide any useful information for the app clone detection. Therefore, to further reduce the amount of computations, *F2* excludes these attributes that provide zero information gain for the generation of our birthmarks.

C. Similarity between Applications

From the previous step (Section III-B), we have a set of unordered birthmarks generated from multiple sets of unordered activities. In this step, we evaluate the similarities between apps. Apart from Google Play that hosts more than 1.3 million apps, there are also numerous third party markets available for user to download apps. Based on these evidences, the intuitive pairwise comparison across these multiple Android markets is impractical. To design an effective and efficient solution, there are 2 challenges that we need to overcome:

Challenge 1 (*C1*): Due to the large number of apps across multiple Android markets, we need a scalable and accurate method to compare the apps similarity.

We overcome *C1* by using Locality-sensitive hashing (LSH) algorithm. LSH is a primitive algorithm frequently employed in high dimension data processing for solving approximate or exact near neighbor problems. In our approach, we view the comparison of one vector to another as a near neighbor problem. For this purpose, we use the E2LSH [23] tool. The E2LSH tool's algorithm is based on the LSH algorithm presented by Datar et al. [24]. The tool solves the following problem:

Given a set of points $P \subset R^d$ *and a radius* $R > 0$*, for a query point q, find all points* $p \in P$ *with a probability of at least* $1 - \delta$ *such that* $||q - p||_2 \leq R$*, where* $||q - p||_2$ *is the Euclidean distance between point q and point p.*

We use the E2LSH tool to determine the near neighbor activities within a certain radius for each activity. The radius is calculated based on the Euclidean distance between the 2 vectors. Instead of using theoretical formulas the E2LSH tool empirically estimates and optimizes parameters as a function of P. This is because theoretical formulas focus on worst-case point sets, thus less suitable for real datasets. It was mentioned in the E2LSH user manual that since the parameters are estimated, it might not be optimal in all cases. However, in our case we found that the estimated parameters provide the best result. Therefore, we keep the estimated parameters.

Note that by varying the radius we can affect the false positive and false negative rates of our approach. Radius is a threshold that is set based on the desired acceptance of the difference between the UIs of the activities. However, it should also be balanced against the number of false positives. Based on the typical distance of corresponding activity from clones, we found radius, $\rho = 6.5$ to be a reliable value in the detection of app clones.

Challenge 2 (*C2*): Since the sets of activities are unordered and we do not know which activity from app *A* should be compared with which activity of app *B*. Further, each activity from one app should only be compared to one other activity from the other app. To ensure that the similarities between the apps are found, we must compare the activities in a way that results in the highest similarity scores. For 2 apps that are similar (as in app clones), their true similarity index can only be found if the activities are correctly compared with the corresponding activities in the other app. A wrong match in the comparison of activities will result in low similarity, despite the fact that the apps are similar. On the other hand, if the apps are independently developed, their highest similarity score will still be low. Note that after matching of near neighbor (solution to *C1*), *C2* is yet to be resolved in some cases. For example, app *A* has 2 activities, *A1* and *A2*. App *B* is a clone of app *A*, with 2 activities as well, where *B1* is similar to *A1* and *B2* is similar to *A2*. If *A1* is similar to *A2*, then the near neighbor of *A1* will possibly include *B1* and *B2*. This is known as assignment problem, a fundamental problem in the field of optimization or operation research in mathematics. Notably, this is not a balanced assignment problem. Firstly, when the pair of apps has a different number of activities the problem becomes unbalanced. Secondly, not all activities from one app will be in the near neighbor of another app. In this case, there will be invalid assignments.

To overcome *C2*, we employ the Hungarian algorithm [25] that is frequently used to solve assignment problems. The Hungarian algorithm is an algorithm that finds the optimal minimum match. We use the algorithm to find the optimal pairs of most similar screens between 2 apps so that the overall Euclidean distance is minimized. The Hungarian algorithm is based on the following principle: if a constant is added to or subtracted from every element on any row or column of the cost matrix for a given assignment problem, then the optimal solution for the resulting cost matrix will have the same optimal solution as the original cost matrix. To employ the Hungarian algorithm for each pair of apps with the assignment problem, we first construct the cost matrix for the apps pair. The cost will be the Euclidean distance for each activities pair between the apps. Secondly, if the total number of activities between the pair of apps is not equal, we add dummy rows and/or columns with zero values to make the cost matrix a square matrix. Lastly, if there are assignments that are invalid, such as when the pair of activities between the apps are not near neighbors, we assign these activities pair a large positive number as the cost value.

Finally, after overcoming *C1* and *C2*, we compute the similarity index (*SI*) between each pair of potential clones. The *SI* is computed based on the ratio between the number of similar activities matched and the maximum number of activities between the pair:

$$SI = \frac{s_A \cap s_B}{\max(s_A, s_B)} \tag{1}$$

In summary, to compute similarities, we first use E2LSH to find the near neighbors within a fixed radius for all activities. Then we apply the Hungarian algorithm to find the pairs of activities within the nearest neighbor that will result in the highest similarity score. Lastly, we compute similarity index based on (1).

D. Clone Clustering

By clustering the apps into groups of clone sets we can determine how the clones are distributed across the Android markets. Furthermore, the clone sets can be used for further analysis of the relationships between the clones and to understand the behavior of the plagiarists. If the *SI* between 2 apps are above or equal to the pre-defined threshold ($SI \geq \sigma$) then they will be considered as clones. The clone set clustering is based on the following algorithm: The output is a clustering S for the apps, in which all apps in a cluster are with similarity index *SI* greater than or equal to σ, i.e., $SI \geq \sigma$. Each clone set will contain at least 2 apps. σ can be set based on the desired number of false positive versus false negative ratio. Generally, as the value of σ increases, it would decrease the number of apps in a set and increase the probability of incurring false negatives. On the other hand, as the value of σ decreases, the number of apps in the set and the probability of incurring false positives would increase. We empirically found that the similarity threshold, σ = 0.76 is a reliable value in Android app clone detection. Figure 3 shows a distribution of the number of false positives and false negatives with different thresholds.

IV. Evaluation

We implemented a prototype of our approach in Java programming language and shell script. The experiments are conducted on Linux with 3.2GHz Intel Xeon CPU and 8 GB of RAM.

A. Dataset

In the official Android market, Google Play, close to 85% of the apps belongs to the free category (requires no cost to download) [26]. Similarly, in third-party markets we also observed that the proportion of free apps is significantly larger than paid apps. Therefore, we limit our dataset to only free apps. The dataset was collected from Google Play and 3 other different third-party Android markets, *Anruan* [27], *Appsapk* [28] and *Pandaapp* [29]. The apps are usually categorized differently in different markets. We believe that from the perspective of the plagiarists, popular apps have more repackaging values. From each market we downloaded their top free popular apps. The number of apps from each market is presented in Table I.

To evaluate the false positive and false negative rates, we would need a set of benchmark apps that are labeled accordingly. Therefore, in order to evaluate our approach, we manually checked the set of real world apps, and labeled them as clones or unique, accordingly. With a dataset of 521 apps we have 135,460 pairs of apps. Since comparing the 135,460 pairs of apps pairwise is almost impossible, we apply a more efficient approach as follows:

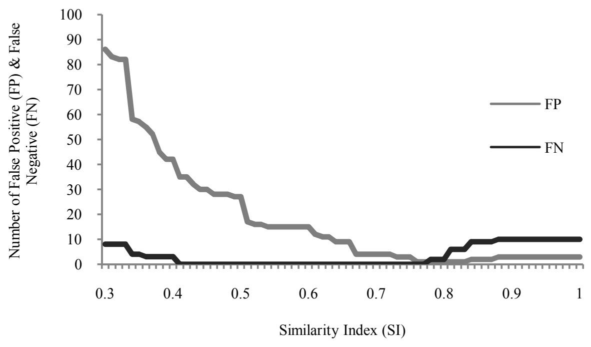


Fig. 3. Number of false positive and false negative with various similarity indexes

TABLE I. NUMBER OF APPLICATIONS FROM EACH MARKET

Market	Number of apps from each market
Google Play	105
Anruan	100
Appsapk	167
Pandaapp	149

1. Categorization: To greatly reduce the number of comparisons, we first divide the apps into various categories. We manually installed and launched each app to briefly grasp an idea of the app's main functionality. Next, we categorize each app to a category based on what we observed. To reduce the number of apps in each category, the categories we chose are more specific. For example, for an app that has calculator as main functionality, instead of a general category such as education we assign it under the category named calculator.

2. Clones within category: In this step, we determine the clone sets among the apps within each category (with >1 app) from 2 aspects: 1) We manually navigated through the apps to check for similarities in the functionalities among apps. 2) With the aid of apktool, we disassembled the APKs to check the smali code and resource. *Apktool* is able to decode the app's resources back to nearly its original form. If the apps are similar in both aspects, we include them in the same clone set. At the end of this process, we found 14 sets of clones with 33 apps in total.

B. False Positive

To measure the false positive rate (FPR), we execute our prototype implementation on the same dataset which we manually validated for clones. Any apps that were detected as clone by our approach, but not labeled as the clone is considered to be false positive. With radius $\rho = 6.5$ and threshold $\sigma = 0.76$, we found 15 sets of clones with 35 apps in total. Out of these 15 sets of clones, a clone set of 2 apps is found to be false positive, the remaining 14 sets were correctly detected. Therefore, with radius $\rho = 6.5$ and threshold $\sigma = 0.76$, our false positive rate, FPR = 0.4%.

We investigated the 2 false positive apps and found that there are only 2 activities with 1 view in each of these apps. The number of apps found from each market is shown in Table II. Figure 4 shows the distribution of the similarity index among the apps pairs.

C. False Negative

Any apps that are labeled as clones from our manual inspection, but not detected by our approach are considered to be false negatives. As aforementioned, our approach detected all the clone sets that were manually validated. Since our approach detected all app clones, our approach has 0 false negative. Therefore, with radius $\rho = 6.5$ and threshold $\sigma = 0.76$, our false negative rate, FNR = 0.0%.

TABLE II. NUMBER OF APPLICATIONS DETECTED AS CLONES

Android App Markets	Number of Apps Detected as a clone	Percentage of Apps in our dataset from individual market detected as clone (%)
Google Play	8	7.6
Anruan	8	8
Appsapk	7	4.8
Pandaapp	12	7.1

D. Performance

We evaluate the performance of our approach from 2 aspects: 1) The time needed to obtain the UI hierarchy dumps from APKs. 2) The time needed to detect clone sets given the XML of UI hierarchy dumps.

The time needed to obtain the XML depends on the number of activities in the app. For the dataset of 521 apps we have approximately 16,000 activities in total. Figure 5 presents the relationship between the number of activities and the time taken to obtain the XML from 1 Android emulator. In Fig. 5, we can see that the time needed increase linearly with increasing number of activities. On average it takes less than 2 seconds per activity to dump its UI hierarchy. Figure 6 presents a histogram of the number of activity components within the apps in logarithmic scale. For better presentation we split the activity counts into bins of 50. In Fig. 6, we can observe that majority of the apps contain less than 50 activities.

After applying the filter *F1*, we are left with 10 attributes from each node in the XML for birthmark generation. After applying the Filter *F2*, the number of elements in the birthmark vectors is reduced by nearly 40%. For the dataset of 521 apps, the time from birthmarks generation to clone sets output takes about 55 seconds.

E. Comparison with Existing Approach

In this section, we evaluate our prototype implementation on a dataset provided by a research group [11]. The dataset includes 259 apps which are divided into 99 clone sets as detected by their algorithm, with each clone set consisting of 2 or more apps. Based on this dataset, we evaluate the effectiveness of our approach in detecting different types of clones. In addition, we also measured the false negative rate of our approach on this dataset.

One of their clone sets contains 3 service apps. These non-typical apps are not within our scope because unlike typical Android apps, these apps execute services in the background and do not require UI for user interactions. With radius $\rho = 6.5$ and threshold $\sigma = 0.76$, we detected 96 clone sets and some of which are not identical to clone sets they have detected. We manually investigated all cases where our clone sets do not coincide with their clone sets.

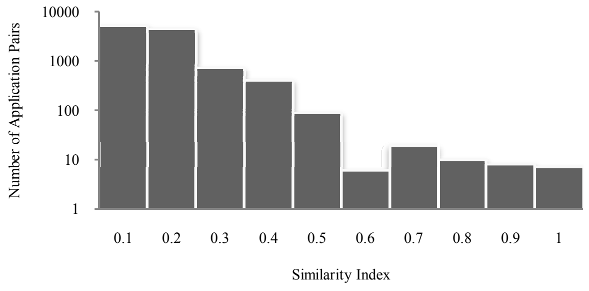


Fig. 4. Histogram of detected application pairs similarity index

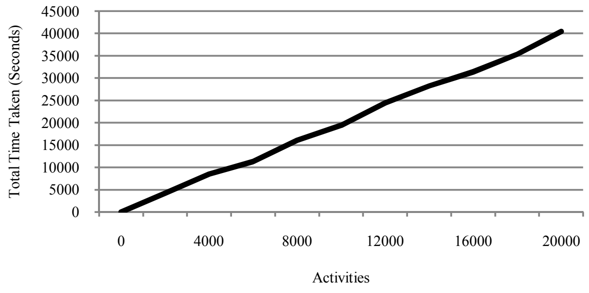


Fig. 5. Time taken to collect XMLs

Firstly, apps in 2 of their clone sets were divided into 4 different clone sets by our approach. When we check the apps in these 4 clone sets, we found that the apps within each clone set were indeed more similar. There were some similarities between these apps, but we do not perceive them as clones as their functionalities are different. Secondly, we found 3 sets of clones that were falsely included in their clone sets. The apps in each of these 3 sets were signed by the same developer key but with different functionalities. We believe that for these apps, the high similarity scores resulting from code base detection are due to developers reusing most of their codes in their other apps. Lastly, another mismatch we have is because both apps in the set consist of only a few activities and some activities are not similar. The differences in activities between these 2 apps are due to different third party libraries. The *SI* between these 2 apps is 66.6%. This shows that our approach is able to detect their similar activities. For this dataset, with radius $\rho = 6.5$ and threshold $\sigma = 0.76$, our false negative rate, FNR = 0.8%.

Based on the clone sets that we have detected from the 2 experiments, we evaluate the ability of our approach to detect different types of clones. Of all the clone sets we detected, we found 14 clone sets with at least a pair of apps belonging to amateur attacks. For example, we found clones with different social media functions. This partially affects the functionalities and UIs of the app. In addition, we also found some apps that are translated and signed with a different developer key. Further investigation is required to determine whether it is an actual clone case or a legal translation. All other apps in the remaining clone sets belong to lazy attacks.

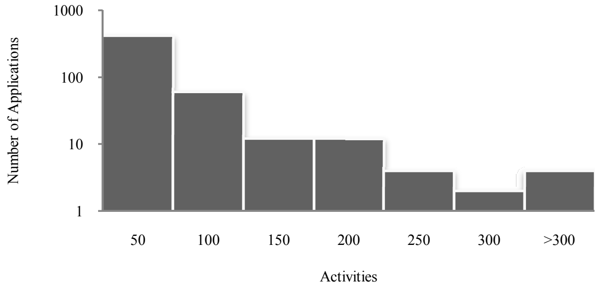


Fig. 6. Histogram of activities counts per application

To determine whether the apps are malware, we upload them to VirusTotal [30] for scanning. VirusTotal is a free online service that allows user to upload a suspicious file for analyzing to identify malware. About 67% of the clones detected from both experiments are reported as malware by at least one of the scanners from VirusTotal. Most of the malware are adware, Trojan horses and there are also a number of spyware. Note that malware attack can, at the same time, be classified as lazy attack or amateur attack. In other words, a lazy attack or an amateur attack with malicious payload attached will also be classified as a malware attack. In our detected clone sets, most of the malware attacks are primarily classified as lazy attacks.

V. Discussion

A. Accuracy and Efficiency

In Section II, we mentioned that there are 3 categories of repackaging attacks: lazy attack, amateur attack and malware attack. In this section, we discuss the effectiveness of our approach to detect the attack in each of these categories.

Lazy attack: In lazy attack the plagiarist makes only simple changes to the apps. Such an attack has little or no impact on the views in the activities. Therefore, our approach can effectively detect lazy attacks.

Amateur attack: An amateur attack requires more effort and knowledge from the plagiarist. They employ automatic code obfuscation and also make small changes to the functionality of the app. Code obfuscation will not affect the UIs of the activities. The small changes in the functionality of the app may or may not affect the UIs in some of the activities. Our approach detects similar activities and not identical activities. In other words, it can tolerate a certain degree of changes in the view hierarchy. Moreover, the small changes in the functionality may affect only some activities but not all activities. If there are many activities that are not affected by the changes, it will not affect our overall detection.

Malware: The clone with malicious payload attached often masquerade as the original app by keeping the functionality and user interface similar to the original app to leverage on its popularity. Since our approach focuses on detecting similar UI, we can effectively detect this type of attack. Furthermore, the additional malicious payload usually does not include an additional activity that has views visible to the user. Thus, it will not affect our detection.

RQ1. Can our proposed approach effectively detect different types of application clones across multiple markets?

The result shows that, our approach is effective in detecting lazy, amateur and malware attacks. Our approach may not be able to detect certain amateur attacks, depending on the extent of the changes to the apps. However, it is not common for the plagiarist to bother understanding its bytecodes to make extensive modifications to the app. We note that there is no general solution for detecting similar app with extensive changes. In addition, our approach certainly raises the bar for the plagiarist to repackage apps without being detected when uploaded. To avoid detection by our approach by modifying the UIs without proper planning will affect the user experience and in turn affect the popularity of their repackaged apps. Proper modification of the UI requires the plagiarists to put in extra effort to redesign the UI carefully.

The performance bottleneck of our approach is the speed of the emulator that limits the efficiency of dumping the UI hierarchies from APKs. However, this process can be parallelized for better efficiency. For example, we can use multiple computers with each computer running multiple emulators.

B. Limitations and Future Works

RQ2. What are the possible limitations of our approach and how can we overcome them?

Firstly, our approach is unable to detect clones in service apps since they just provide services in the background and usually do not contain UIs that are necessary for user interaction.

Secondly, apps with a small number of activities or with low view counts in the activities may increase the false positive rate. However, we noted that this is a common limitation for most birthmark or finger print clone detection approaches.

Thirdly, our approach may also be limited by apps with multiple data dependent activities. For example, apps that require login credential is one of them, this category of apps usually restricts their entry point to the main activity with login UI. Any attempt to enter into the app from other activity will fail or be redirected to the main login activity. The information that can be obtained from these apps is limited, thus affecting the accuracy of our approach.

Lastly, our evaluation is based only on free apps and the result may be different for paid apps, which requires the users to pay certain fees before the download is available to them. Cloning may be more prevail in paid apps, since users would be more likely to choose the similar but free apps. Conversely, the plagiarists may be less willing to purchase the app to repackage it.

For future work, we would like to conduct more in-depth studies on how different types of repackaging attacks can affect the UIs. We would also like to explore additional alternatives to extract more relevant information from the APKs, such as those data dependent activities. These would help us to design better software birthmarks that may further improve our accuracy and efficiency. Due to the complex nature Android apps, code or resource analysis alone is insufficient for clone detection. We believe that a hybrid static and dynamic analysis, which integrates both code and UI information for app clone detection, would be a very promising and interesting research direction.

V. RELATED WORK

A number of previous studies have already been conducted on Android clone detection. Most of the existing approaches only focus on code-based similarities.

Zhou et al. [9] proposed an app similarity measure system, DroidMOSS, to detect repackaged apps in third-party Android markets. DroidMOSS first compute the fuzzy hashes for each method within the app to generate a fingerprint for the app. They then compute a similarity score based on the edit distance between the 2 fingerprints.

DNADroid [12] first detect potential similar apps based on their meta information that is used to describe the app. In the second stage, DNADroid creates the program dependency graph (PDG) as a fingerprint for each app that is to be compared. Lastly, they apply a filter to prune unlikely clones, before comparing the rest of the PDG pairs that passed the filter using a subgraph isomorphism.

Androguard [10] provides a similarity measure tool that supports several standard similarity metrics. The similarity is computed by comparing similar methods in the dex code of the apps. Rather than detecting cross-market app clones, Androguard is meant for finding the difference between 2 apps on a small set of data.

Chen et al. [11] extracted the methods from the apps and construct a 3D-control flow graph (3D-CFG) to get the centroid. They then leverage the centroid to measure method level-similarity across multiple markets. Lastly, the method-level similarity result is used to group similar apps together.

Zhou et al. [31] focused on the problem of detecting "piggybacked' apps, which are clones with additional malicious payload attached. They first perform module-decoupling technique to split the code into primary and non-primary modules. They then extract a semantic feature fingerprint for each primary module and use a linearithmic search algorithm to detect similar apps.

Hanna et al. [14] proposed Juxtapp, a tool to detect code reuse among Android apps. Juxtapp uses k-grams of the opcode sequences and apply the feature hashing to extract the feature of the apps. Juxtapp can identify vulnerable code reuse, instance of known malware and pirated copies of the original apps.

All these approaches focus on static code-based detections that are vulnerable to advance obfuscation techniques. On the other hand, there are a few existing works that detect Android app clones without relying on code similarities.

Differing from code similarity based approaches, FSquaDRA [32] detects Android app clones based on the comparison of the resource files that are necessary for creating the APK. They leverage on the hashes that were computed and stored in the package during the process of app signing. This approach is resilient to code obfuscation, but some small changes in the resources will affect the similarity.

Viewdroid [19] proposed a user interface based approach to detect app repackaging. Our approach is similar to Viewdroid in the sense that both our approaches leverage on UI information. However, they construct view graph based on static analysis of the control flow relationship between the views within the app. Our approach differs from them in that our birthmark information is collected from runtime and we generate vectors from this information.

VI. CONCLUSION

In this paper, we presented a novel approach to detect Android app clones based on birthmarks generated from runtime UI information. Our approach uses locality sensitive hashing to find a near neighbor for similar birthmarks and apply the Hungarian algorithm to find the optimal activities pairs with the overall highest similarity. The result shows that our approach can effectively detect different types of repackaging attacks such as, lazy, amateur and malware attacks with low false positive (FPR=0.4%) and false negative (FNR=0.8%) rates. Many of our detected apps (67%) belong to malware attacks, this call for a more rigorous vetting across all Android markets. Our paper helps the reader to understand the UI of Android apps and how this information can be applied to clone detection. We believe that our study supports a new research direction or can be used to complement existing code-based approaches in mobile app clone detection.

VI. REFERENCES

[1] IDC Corporate USA. (2014, 2 dec). *Smartphone OS Market Share, Q3 2014*. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[2] AppBrain. (16 Sep). *Number of Android applications*. Available: http://www.appbrain.com/stats/number-of-android-apps

[3] Business and Revenue Working Group. (3 Dec). *Monetization: Picking the Path to App Profitability*. Available: http://www.appdevelopersalliance.org/app-monetization/

[4] (18 Sep). *android-apktool: A tool for reverse engineering Android apk files*. Available: code.google.com/p/android-apktool/

[5] (28 SEP). *dex2jar: Tools to work with android .dex and java .class files*. Available: https://code.google.com/p/dex2jar/

[6] D. Octeau, S. Jha, and P. McDaniel, "Retargeting Android applications to Java bytecode," presented at the Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, Cary, North Carolina, 2012.

[7] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi, "Adrob: Examining the landscape and impact of android application plagiarism," in *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, 2013, pp. 431-444.

[8] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (SP), 2012 IEEE Symposium on*, 2012, pp. 95-109.

[9] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, 2012, pp. 317-326.

[10] (24 Sep). *Androguard: Reverse engineering, Malware and goodware analysis of Android applications*. Available: https://code.google.com/p/androguard/

[11] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *ICSE*, 2014, pp. 175-186.

[12] J. Crussell, C. Gibler, and H. Chen, "Attack of the Clones: Detecting Cloned Applications on Android Markets," in *Computer Security – ESORICS 2012*. vol. 7459, S. Foresti, M. Yung, and F. Martinelli, Eds., ed: Springer Berlin Heidelberg, 2012, pp. 37-54.

[13] H. Huang, S. Zhu, P. Liu, and D. Wu, "A Framework for Evaluating Mobile App Repackaging Detection Algorithms," in *Trust and Trustworthy Computing*, ed: Springer, 2013, pp. 169-186.

[14] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, ed: Springer, 2013, pp. 62-81.

[15] Google Inc. (10 Dec). *Application Fundamentals*. Available: http://developer.android.com/guide/components/fundamentals.html

[16] Google Inc. (25 Sep). *Intents and Intent Filters*. Available: http://developer.android.com/guide/components/intents-filters.html

[17] Google Inc. (10). *Activities*. Available: http://developer.android.com/guide/components/activities.html

[18] Google Inc. (27 Sep). *uiautomator*. Available: http://developer.android.com/tools/help/uiautomator/index.html

[19] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "ViewDroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2014). ACM*, 2014.

[20] Google Inc. (4 Dec). *Tools Help*. Available: http://developer.android.com/tools/help/index.html

[21] Google Inc. (25 SEP). *<activity>*. Available: http://developer.android.com/guide/topics/manifest/activity-element.html

[22] Google Inc. (03 DEC). *Accessing Resources*. Available: http://developer.android.com/guide/topics/resources/accessing-resources.html

[23] *LSH Algorithm and Implementation (E2LSH)*. Available: http://www.mit.edu/~andoni/LSH/

[24] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*, 2004, pp. 253-262.

[25] H. W. Kuhn, "The Hungarian method for the assignment problem," *Naval research logistics quarterly,* vol. 2, pp. 83-97, 1955.

[26] AppBrain. (1 Dec). *Free vs. paid Android apps*. Available: http://www.appbrain.com/stats/free-and-paid-android-applications

[27] (05 SEP). *Anruan*. Available: http://www.anruan.com/

[28] (05 SEP). *Appsapk*. Available: http://www.appsapk.com/

[29] (05 SEP). *Pandaapp download center*. Available: http://download.pandaapp.com

[30] (03 JAN). *VirusTotal*. Available: https://www.virustotal.com/en/

[31] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of piggybacked mobile applications," in *Proceedings of the third ACM conference on Data and application security and privacy*, 2013, pp. 185-196.

[32] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. La Spina, and E. Moser, "FSquaDRA: Fast Detection of Repackaged Applications," in *Data and Applications Security and Privacy XXVIII*, ed: Springer, 2014, pp. 130-145.