RegionDroid: A Tool for Detecting Android Application Repackaging Based on Runtime UI Region Features

Shengtao Yue†‡, Qingwei Sun†‡, Jun Ma†‡, Xianping Tao†‡, Chang Xu† and Jian Lu†‡
†State Key Laboratory for Novel Software Technology, Nanjing University
‡Department of Computer Science and Technology, Nanjing University

Abstract—With the rapid development of mobile devices, Android applications (apps) are universally used. However, attackers repackaging Android apps and release them to the markets for illegal purposes, which brings great threats to the Android ecosystem. To leverage the popularity of original apps, they keep similar software behaviors to confuse app users. Furthermore, repackaged apps can be obfuscated or encrypted to avoid being detected. Besides, hybrid mobile apps, built by combining web technology and native elements, are becoming a preferred choice for developers. The structure of hybrid apps differs a lot from that of native apps which would raise great challenges to repackaging detection. Existing works still have some limitations in detecting repackaging from obfuscated and encrypted apps. Besides, few of them can deal with hybrid apps. In this paper, we proposed an approach based on the app UI regions extracted from app’s runtime UI traces. We also implement a tool named RegionDroid based on the approach. We apply RegionDroid to tree datasets with totally 369 apps. It successfully finds all the 98 RegionDroid based on the approach. We apply RegionDroid to detect apps from Android app markets, decompile these apps by reverse engineer tools (e.g., Apktool [2], dex2jar [3], and Soot [4]). After modification (e.g. adding some malicious payloads, changing advertisement account, etc.) and rebuilding, they repackaged apps and release them to Android markets. Moreover, another reason for the dramatic growth of app repackaging is because of the lack of effective supervision in the unofficial and third-party markets. It is shown that 5% to 13% of apps were plagiarisms in Android markets [5] and 1083 of the analyzed 1260 malware samples were repackaged versions of legitimate apps with malicious payloads [6].

To detect Android repackaging, there have been several approaches proposed. Exacting software birthmarks [7] from apps and comparing birthmarks to measure apps similarities is a common way for repackaging detection. Birthmarks are unique and native characteristics to identity the apps [7]. According to the way of birthmark generation, current works can be divided into two types, static and dynamic. Static birthmarks [5], [8]–[15] are generated by analyzing static code and resources. However, since apps are often obfuscated or encrypted, the features extracted from these apps with static analysis will be heavily affected. In such cases, static birthmarks may hardly work [16], [17]. To deal with app obfuscation and encryption, dynamic birthmarks [18]–[20] have been proposed. The work of [18] executes apps with the testing tool Monkey [21] and generates birthmarks from runtime API sequences during the executions. The work of [19] executes apps by starting all of activities and analyze UI information collected at runtime. Its 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. Repdroid [20] generates layout-group graph through runtime UI traces during apps execution as the birthmark. It traverses the apps by simulating user interactions with a heuristic strategy. Although these approaches improve their resistance to obfuscation, there are still some limitations. The first two works can be heavily affected by some semantic-preserving obfuscation and encryption [20]. Repdroid tries to traverse the whole app. However, when it traverse into a deep part of layout, it may be trapped into it. If Repdroid runs several times for the same app, it may lead to different birthmarks. Besides, since it relies on the type of views, and view types in the webviews (core elements in hybrid apps) are mostly identical, Repdroid could be inaccurate when dealing with hybrid apps.

To cover these shortages, in this paper, we propose an approach for app repackaging detection based on a dynamic birthmark, region-group graph (RGG). Like most existing works, our approach is based on the observation that repackaged apps should behave similar with original ones to leverage the their popularity. Although app obfuscation and encryption may change its static features but will not affect its runtime UI features. As a result, without knowing the source code or professional knowledge, app users can figure out whether two apps are repackaged easily by comparing the UI features between them. Another observations is that, intuitively speak-
when users try to tell whether two apps are a repackaged pair, they often check the first adequate layouts, rather than comparing apps in every corner. We do not have to traverse the whole app which can reduce the time consumption of birthmark generation without affecting accuracy (introduced in Section IV by the experiment). To model the UI features, we propose the RGG as the dynamic birthmark of an app. The graph is generated from app runtime UI traces. Besides, the RGG is based on extracting features from skeletons (defined as regions) of layouts instead of view types, which can still work when it comes to hybrid apps.

In summary, the main contributions of this paper are:

- We propose a dynamic app birthmark extracted from the skeletons of layouts called region-group graph (RGG). RGG is generated by partially traversing the app with a limited depth.
- We implemented a tool RegionDroid based on our approach by which we can extract the birthmark RGG and detect app repackaging.
- We conducted experiments to evaluate the effectiveness of RegionDroid on 3 datasets with totally 369 apps. RegionDroid successfully finds all the 98 obfuscated or encrypted repackaged pairs in dataset $S_1$ and has good credibility in dataset $S_2$ with 114 commercial apps. In dataset $S_3$, it also compares 157 hybrid apps pairwise and the false positive rate is 0.016%.

The rest of this paper is organized as follows: Section II introduces the background of Android and related works. Section III presents the methodology of our approach. Section IV evaluates our tool, along with Repdroid, by applying them to three different data sets. Section V discusses RegionDroid and shows its limitations and our future works. At last, we conclude the paper in Section VI.

II. BACKGROUND

A. Android Applications

Most Android apps are usually developed with Java programming languages. The Java program code is compiled into a .dex file by Android SDK tools. The .dex file along with related data and resource files are finally packaged into an .apk file. Each app project must have an AndroidManifest.xml file which describes the essential information about the app, including package name, launcher activity, permissions requested during app runtime, etc [22].

Android app UI is drawn into a container window. A window is often full-screen, but can also be floated on other windows or embedded inside of another window. UI elements drawn in a window consists of several different kinds of views, which are rectangular spaces and some of them can be interacted by users [23]. The views are organized in a tree. For instance, Fig. 1a shows a screen capture of an example login window. The views and their structure are shown in Fig. 1b. Besides the visible views (two EditTexts and a Button), there are another two views named LinearLayout. LinearLayout is used to arrange its child views horizontally or vertically. More specifically, those views which can contain children (e.g. LinearLayout) are called view groups. We use the single word layout to define the visual structure for a UI [24].

B. Hybrid Applications

Nowadays, Android fragmentation becomes a serious problem for Android app developers [25]. Developers should design their apps carefully so that apps can be compatible with different versions of Android systems and devices, which bring developers great burdens. However, web technologies, such as HTML, css, or Javascript, have good cross-platform features. Developers only need to change the native shells, which are used to present web contents, for specific platforms, rather than redesign the whole projects. Hybrid apps, which combine native elements with web technologies, have become a trend for mobile development. A survey of more than 13,000 community developers shows that 55.5% of their apps were hybrid and 64.4% of them expected to develop hybrid apps in the future [26]. There have been many hybrid app frameworks which are used to help develop hybrid apps efficiently, e.g. ionic [27], phonegap [28], framework7 [29], appcelerator [30], etc. Typical hybrid Android apps utilize the key view WebView, which can be treated as a browser embed in the native apps. Running on Android with API level higher than 17, the content of webview can be dumped like native views. An example is shown in Fig. 2. The left picture of the figure shows a screen capture of a layout in a hybrid app. The right part of the figure shows the dumped layout structure of this browser. The node named ‘android.webkit.WebView’ is the main element of this layout, inside which most of the elements (e.g. the ‘LOGIN’ and ‘SIGN UP’ buttons) are described as ‘view’s, rather than varied kinds of view types in native apps (‘android.widget.Button’ will be used in most cases for these two buttons). In summary, the views in webviews are weakly typed.

C. Related Work

1) Static Birthmarks: DroidMOSS [5] applies a fuzzy hashing as the birthmark to localize and detect the changes from app-repackaging behavior. It calculates the similarity of birthmarks by edit distance. DNADroid [8] compares program de-
pendsence graphs (PDGs) between methods in candidate apps and measure the graphs by VF2 subgraph isomorphism [31]. AnDarwin [9] computes an undirected PDG of each method in the app to extract semantic vectors. It identifies similar code by local sensitive hashing on the semantic vectors. Besides, AnDarwin does not compare apps pairwise, which increases its scalability. Juxtapp [10] uses k-grams of opcode sequences extracted from compiled applications and feature hashing to efficiently tackle the problem at large-scale. PiggyApp [11] divides an apps code into primary and non-primary modules. It extracts various semantic features from primary modules and converts them into feature vectors for comparison. Centroid [12] extracts methods from apps and encode a projection from of 3D control flow graphs (3d-CFGs) to get the “centroid” geometry characteristic. It uses the centroid-based comparison to measure the similarity between apps. Wukong [32] identifies suspicious apps by comparing light-weight static semantic features, and a ne-grained phase to compare more detailed features for only those apps found in the rst phase. Viewdroid [13] captures users’ navigation behavior across app views and generate view graphs. By analyzing the API method invoking, it collects information about the activity transitions. Viewdroid also uses VF2 subgraph isomorphism to calculate the similarity between graphs. Work [33] applies machine-learning-based detection relied on semantic features of apps to achieve better efficiency in detection. HookRanker [34] identifies the two types of hooks that trigger the execution of rider code and thereby ungrafting the malicious rider code. It outputs two recommended hook lists which can be applied for other implications such as malware detection.

Static birthmarks are usually credible, however, they may not resist code obfuscation and could even hardly work when it comes to app encryption. To compute similarities, they first use local sensitive hashing to find the near neighbors for all activities. Then they apply the Hungarian algorithm to find the pairs of activities that will result in the highest similarity score.

Repdroid [20] is an automated tool to detect Android app repackaging. Repdroid uses layout to name the view structure of a window. It proposed the layout-group graph (LGG) as dynamic birthmark of an app. LGG is constructed based on the runtime UI traces. The vertexes of LGG are layout groups (the sets where similar layouts are grouped) and the edges are the group transitions. The reason why LGG does not simply use layout as vertex of graph is that app users consider layouts with little different views are the same layouts. Repdroid models this observation and group similar layouts together. To generate LGG, Repdroid executes each app and collects runtime UI trace by traversing the app. The traverse process is in a loop that for each round, a view in current shown layout is selected to interact with, and an action (e.g. Click, ScrollBackward, Back, etc.) associated with the selected view is selected to trigger. After triggering the selected action on selected view, LGG will be updated by the UI state at that time. Besides, Repdroid describes that when app users try to traverse an app, they often prefer to interact with the views which can lead to the new layouts that they have not arrived at yet. So Repdroid assigns a new attribute weight to each view of layouts and updates their weights during runtime execution. The larger the view’s weight is, the more possibly it will be chosen.

Compared to static birthmarks, dynamic birthmarks focus on the characteristic at runtime, which can be an effective way to deal with app obfuscation. However, there are still some limitations. For example, the work of [19] still relies on some static features (activity lists by analyzing APKs), which can be affected by inserting unused activities. Some encrypted apps (e.g. encrypted by Ijiami [35]) partially run with the native binary instead of Android API. As a result, the work of [18] can not collect correct API traces for extracting birthmarks. Repdroid relies on view types in the layouts, while hybrid apps are weakly typed so that Repdroid can hardly deal with them. Besides, Repdroid tries to traverse the whole app, it may be trapped into a deep part of layout and affect the accuracy of the birthmarks.

III. METHODOLOGY OVERVIEW

Like most existing works, our methodology is based on the observation that the repackaged apps should share similar software behaviors (e.g. visual presentations, user interactions). Since the app obfuscation and encryption will heavily affect static analysis, we try to model dynamic UI as the birthmark for each app. Moreover, as the UI layouts of hybrid apps are weakly typed, we do not focus on the view types like Repdroid. Instead, we pay our attention to the view’s visual presentation itself. Views in the layouts are in the shape of a rectangles, in which different contents (e.g. colors, texts, or images) are drawn. Modifying these contents, such as
changing the color of buttons or the background image of a layout) can maintain the similar visual presentations. The skeleton of layouts (i.e. the size and position of view rectangles in each layout) are hardly affected by these modifications. As a result, we can extract birthmarks by analyzing these view rectangles of layouts.

The overview of our approach is shown in Fig. 3. Our approach is divided into two major parts, RGG generation and birthmark similarity calculation. RGG is used as the birthmark of an app (introduced in Section III-A). Given an APK, we generate RGG during app execution (Section III-C). After RGGs are generated, we construct a bipartite graph for each pair of RGGs which are to be compared. Then we use bipartite graph’s maximum weight matching algorithm to calculate the similarity score between RGG pairs (Section III-D), which can be used to detect app repackaging. Specifically, if the similarity score between RGG pairs (Section III-D), which can be calculated, is larger than a threshold $\delta$, we consider this pair of apps is repackaged.

### A. The Region-group Graph (RGG)

For a clear introduction to our approach, we define some key concepts as follows.

**Definition 1 (Region):** A layout contains several views and each view is in the shape of a rectangle. A region $r$ is defined as a rectangle set. Each of the rectangles is corresponding to a view in the layout. A region describes the skeleton area of a layout.

**Definition 2 (Region Group):** A region group $g$ is defined as a region set in which similar regions are grouped.

**Definition 3 (RGG):** A RGG is defined as a directed graph $RGG = (G, E)$ where $G$ is a set of region groups, edges $e \in E \subseteq G \times G$. $e = g_1 \rightarrow g_2$ means a transition from group $g_1$ to $g_2$.

An example of region group is shown in Fig. 4. Supposing we have a layout shown in the screen capture (Fig. 4a), the skeleton area of the layout is shown in Fig. 4b. Each rectangle in the region represents a layout in the view. Then we group the similar region into a region group (Fig. 4c).

### B. Region Similarity Calculation

A region of the app layout consists of several rectangles. We measure the similarity of regions by calculating the spatial overlapping of rectangles in each region.

Firstly, we define the rectangle similarity with Jaccard distance [36]. Given two rectangles $r_1$ and $r_2$, their corresponding sizes (width $\times$ height) are $size_1$ and $size_2$, and their overlapped size is $size_o$ (see Fig. 5). The similarity of these two rectangles is shown in Equation 1.

$$Sim_{r}(r_1, r_2) = \frac{r_1 \cap r_2}{r_1 \cup r_2} = \frac{size_o}{size_1 + size_2 - size_o}$$  (1)

We then construct a R-tree [37] from the rectangles of the region. Each region is corresponding to a R-tree. R-tree is a tree data structure for indexing multi-dimensional information (rectangles can be treated as 2-dimension, width and height). R-tree groups nearby objects and represent them with their minimum bounding rectangle (MBR). An example is shown in Fig. 6. Fig. 6a is a screen capture of a demo layout. We mark the views in the layout with numbers. Fig. 6b is a visualization of R-tree constructed from this layout. The whole layout is covered by bound rectangle $MBR_0$, which is not shown in the figure. Fig. 6c shows the specific tree nodes of R-tree. From the figure we can see that the layout $MBR_0$ is divided into 3 parts, each part is the minimum bounding rectangle of 3 or 4 views. R-tree is a balanced search tree so it has good performance to search a given rectangle input.

Lastly, we calculate the similarity of two R-trees (supposing $tree_1$ and $tree_2$ extracted from $region_1, region_2$) as the similarity between $region_1, region_2$ by Algorithm 1. Since repackaged apps try to keep their looking similar with original apps, we consider that the positions and sizes of views in repackaged apps should be slightly changed, which means the corresponding rectangles can have high similarity scores. As a result, for each rectangle $rect_1$ in $tree_1$, we search the rectangle in $tree_2$ which has the most rectangle similarity score with $rect_1$. We use the average of these scores to measure the similarity of the two regions.

### Algorithm 1 Region Similarity

```plaintext
1: procedure REGIONSIMILARITY(region1, region2)
2: tree1, tree2 extracted from region1 and region2
3: sum ← 0, n ← 0
4: for all rect1 ← tree1 do
5:     rect2 ← arg max rect1 ∈ tree2 Sim_r(rect1, r)
6:     sum ← sum + Sum_r(rect1, rect2)
7:     n ← n + 1
8: end for
9: sim_n ← sum/n
10: return sim_n
11: end procedure
```

Additionally, supposing regions $r_i \in g_1.R, r_j \in g_2.R$, the similarity between groups $g_1$ and $g_2$ is shown in Equation 2.

$$Sim_{g}(g_1, g_2) = \max REGIONSIMILARITY(r_i, r_j)$$  (2)

### C. Graph Generation

We generate our RGG based on the graph generation method in Repdroid [20]. Repdroid constructs a graph called layout-group graph (LGG) as the birthmark of an app. It executes an app with a heuristic strategy by assigning weight properties to views and action types, and collect the runtime UI trace by traversing the app. When traversing an app, it prefers to interact with views which can lead to the layouts that they haven’t arrived at. However, there are opportunities for optimizations. Firstly, Repdroid tries to traverse the whole app, it may be trapped into a part of the app and could lead to different birthmarks if we run Repdroid several times for the same app. Intuitively speaking, when users try to tell whether two apps are a repackaged pair, they often check the first several layouts shown in each app, rather than comparing apps in every corner. We just need to reach the initial part of app to
generate a partial graph which is credible for app birthmark. Secondly, not all the triggered action types are necessary. In fact, only triggering click action is enough for generating good enough birthmark for repackaging detection.

As a result, we propose a strategy to generate RGG shown in Algorithm 2. The inputs of the algorithm are \(apk, d_m, r_m, c_m\). Here, \(apk\) is the target APK file, \(d_m\) means the maximum depth of region allowed to reach, \(r_m\) means the maximum rounds the main loop allowed to take, \(c_m\) means the maximum allowed rounds that the RGG does not change after a click action triggered. Firstly, we get the launcher activity, install \(apk\) and start the app. The launcher activity is also known as an entry point of an app. It should be declared clearly so that Android system can know where to start the app. So no matter how an app is obfuscated of encrypted, we can always get this configuration. Then we get current region \(r_c\) and group \(g_c\) by calling GetCurrentRegion. GetCurrentRegion returns current region of app and the group which the region belongs to. We mark \(r_c\) as the launcher region, and \(g_c\) as the launcher group. Then we run into the main loop (Line 7) to generate the graph. \(r\) represents current rounds the loop has taken. \(c\) counts the current rounds that the RGG does not change after click. If any of them is larger than the maximum threshold (i.e. \(d_m, r_m, c_m\)), the loop will break and the graph will be returned (Line 27).

For each round in the main loop, we first select a rectangle \(rect_s\) at current region according to rectangles’ weights. The larger the weight is, the more likely the rectangle will be selected. Then we trigger click action on \(rect_s\). Although the layout of app may or may not change after the action, we always get the current region \(r\) and its group \(g\). Afterwards, we get the depth \(d\) of group \(g\) by calculating distance between
Algorithm 2 Graph Generation Strategy

1: procedure GRAPHGENERATION(apk, d_m, r_m, c_m)
2:     \( RGG \leftarrow \emptyset \)
3:     Get launcher activity from apk
4:     Install apk and start launcher activity
5:     \([g_c, r_c] \leftarrow \text{GETCURRENTREGION}()\)
6:     \( g_s \leftarrow g_c, r \leftarrow 0, c \leftarrow 0 \)
7:     while \( r \leq r_m \land c \leq c_m \) do
8:         \( \text{rect}_s \leftarrow \text{WEIGHTEDSELECTRECT}() \)
9:         Do click action on \( \text{rect}_s \)
10:        \([g, r] \leftarrow \text{GETCURRENTREGION}()\)
11:        \( d \leftarrow \text{distance between } g_s \text{ and } g \)
12:        if \( d > d_m \) then
13:            Try to go back or restart app
14:            \( c \leftarrow c + 1 \)
15:        else
16:            Update RGG
17:            if RGG’s vertex or edge count changed then
18:                \( \text{rect}_s.w \leftarrow \text{rect}_s.w + 1 \)
19:                \( c \leftarrow 0 \)
20:            else
21:                \( \text{rect}_s.w \leftarrow \text{rect}_s.w - 1 \)
22:                \( c \leftarrow c + 1 \)
23:            end if
24:        end if
25:        \( r \leftarrow r + 1 \)
26:    end while
27:    return RGG
28: end procedure

D. Graph Similarity Calculation

After we generate the RGGs from apps, we need to calculate the similarity between them to detect repackaged apps. A common approach for graph comparison is the subgraph isomorphism algorithm. However, the subgraph isomorphism algorithm is not suitable for compare RGGs. Firstly, we do not try to traverse the whole apps and the generated RGGs are generally not complete but partial. Secondly, the subgraph isomorphism is time-consuming. Instead, we adopt the way proposed by Repdroid [20] with some improvement and try to find the matchings of elements in two RGGs. We first construct a bipartite graph from two RGGs and then use maximum weight matching to find the best matching in the bipartite graph. The matching score is used to measure the similarity of two RGGs.

More specifically, given two RGGs \( RGG_1 \) and \( RGG_2 \), we construct a bipartite graph (shown in Fig. 7). A bipartite graph has two parts of vertexes \( P_{left}, P_{right} \), which are corresponding the edge sets of \( RGG_1 \) and \( RGG_2 \). This means edges in RGGs are the vertexes in the bipartite graph. Edges rather than vertexes in RGGs are used here because edges contain information about groups and their transitions. For example, in Fig. 7, \( E_1, E_2, E_3 \) make up the left part of the bipartite graph, and \( E_{a}, E_{b}, E_{c}, E_{d} \) make up the right part. Then, we need to construct the edges in bipartite graph. Since we do not know which vertex in left part should match which one in the right, all of vertexes were supposed to be connected. Given an edge \( e = v_1 \rightarrow v_2 \), the weight is calculated by Equation. 3. In fact, we do not need to connect all the vertexes in bipartite graph after we get the weights of all vertex pairs. Unlike Repdroid, we add a filter to reduce the complexity of the bipartite graph for improvement: If the weight of two vertexes is smaller than 0.5, we believe they are too ‘different’ to be a matching candidate and we will drop this edge connecting these two vertexes.

After bipartite graph is constructed, we use Kuhn-Munkres algorithm [38] to calculate the maximum weight matching. Without loss of generality, we assume the amount of vertexes in \( P_{left} \) is no larger than that in \( P_{right} \) and find the vertexes in \( P_{right} \) which match vertexes in \( P_{left} \). Then we get the matching score \( M \). Finally, we normalize the matching score as the graph similarity by Equation. 4, where \( N \) is the size of vertexes in \( P_{left} \).

\[
\begin{align*}
    w(v_1, v_2) &= \frac{\text{sim}_d(v_1.g_1, v_2.g_1) + \text{sim}_d(v_1.g_2, v_2.g_2)}{2} \\
    \text{Sim}_G &= \text{norm}(M) = \frac{M}{N}
\end{align*}
\] (3)

If the similarity between two graphs is larger than a threshold \( \delta \), the corresponding pair of these two apps is considered as a repackaged pair.

IV. IMPLEMENTATION AND EVALUATIONS

To evaluate the proposed approach for generating and comparing the dynamic birthmark RGG, we implemented a tool name RegionDroid to generate RGGs and calculate the
similarities among RGGs. In RegionDroid, we use adb [39]
tool to control the Android system and app runtime execution,
such as installing APKs, triggering click actions, etc. We also
use UIAutomator [40] tool to dump layouts to analyze the
runtime UI.

We conducted the experiments on Genymotion [41] virtual
device with Android 6.0 (API 23). The virtual device is
configured with 4GB memory, and 1080*1920 resolution. We
set thresholds \(d_m = 2, r_m = 1000, c_m = 200\) and the graph
similarity threshold \(\delta_s\) is 0.78. The following two properties
are used to evaluate the birthmarks [7].

**Property 1:** (Resistance) If program \(p'\) is obtained from \(p\)
by any program semantic preserving transformation, the birth-
marks of both \(p'\) and \(p\) should be the same. Obfuscation
and encryption, should not affect the birthmarks of the transformed
programs.

**Property 2:** (Credibility) For program \(p\) and \(q\) implementing
the same specifications, if \(p\) and \(q\) are written independently,
the birthmarks of \(p\) and \(q\) should be different.

**A. Datasets**

We design our experiment with two parts. In the first part,
we use datasets \(S_1\) and \(S_2\) from Repdroid [20] to evaluate
the credibility and resistance when apps are obfuscated or
encrypted. In the second part, we apply both RegionDroid
and RepDroid to a dataset \(S_3\) consisting of 157 hybrid apps
to compare their performances.

1) \(S_1\) and \(S_2\): In dataset \(S_1\), there are 98 pairs of An-
droid apps. Each pair is an original app from F-Droid and
the corresponding obfuscated or encrypted app. 20 apps are
downloaded from F-droid, and they are obfuscated by 2
different obfuscation tools (FakeActivity, and NestedLayout,
both implemented by Repdroid’s work) and encrypted by
AndroCrypt [18]. These original and corresponding apps make
up totally 60 pairs. The rest 38 apps are download from
Wandoujia and they are encrypted by Ijiami [35]. In dataset
\(S_2\), there are 125 real world apps from Wandoujia and all of
them are the top ranked apps from 8 categories.

FakeActivity repackages apps with useless activities by Soot
[4]. These activities will never be started during apps’ normal
execution NestedLayout modifies layout files by inserting
nested views such as LinearLayout, FrameLayout. With nested
views attributes set properly, the repackaged apps still look
the same as the original ones, but their layout XMLs are quiet
different.

AndroCrypt encrypts the original APK as an asset file and
pack it into a new shelled Android app. When new app is
running, it dynamically decrypt and load original APK asset as
if original app was running. Ijiami is a commercial encryption
tool, which has similar theory with AndroCrypt. However,
Ijiami loads assets by native operation while AndroCrypt uses
Android APIs (e.g. dalvik.system.DexClassLoader).

2) \(S_3\): In dataset \(S_3\), we downloaded 117 hybrid apps from
androzoo [42], AndroZoo is a collection of Android apps
collected from several sources, including the official Google
Play app market. We select hybrid apps from androidzoo
datasets by checking whether the layouts of apps contain
element ‘webview’ or HTML, css files. Besides, since many
developers build their apps with development frameworks, we
also downloaded 40 hybrid apps from the showcases of 7
hybrid frameworks (shown in Table I). To sum up, we have
157 hybrid apps for dataset \(S_3\) in total.

**B. Resistance and Credibility**

1) Resistance: To evaluate the resistance of RegionDroid,
we extract RGGs from 98 pairs of apps in \(S_1\). An original app
and the corresponding obfuscated or encrypted app are defined
as a RGG pair. For each pair, we calculate the similarity to
detect whether they are repackaged. With the threshold \(\delta_s\)
0.78, we can detect all repackaging pairs. Fig. 8 shows the
distribution graph of the similarity. The distribution of RGG
size is shown in Fig. 10a. We use the vertex count of graphs
as the size of RGG. There are 156 RGGs generated in total, which
are 20 graphs from F-droid apps, 20 graphs each from 3 kinds
of obfuscation and encryption, 38 from apps in Wandoujia and
38 from their encrypted apps by Ijiami.

2) Credibility: To evaluate the credibility, we extract RGGs
from \(S_2\), and compare these RGGs pairwisely. Since Android
version is different from that in Repdroid, we only successfully
run 114 apps in our experiment. We applied RegionDroid to
these apps and there about to be 6441 similarity scores. Like
Repdroid did, we also filter the pairs that the differences be-
tween the amount of their groups are more than two times. The
similarities of filtered pairs are set 0 directly. The distribution
graph of similarity is shown in Fig. 9. There are 4 pairs found
‘repackaged’ with threshold \(\delta_s\) 0.78. We check these pairs
manually, and all of these 4 pairs are real repackaged pairs.
Fig. 10b shows the distribution of RGG size in \(S_2\).

**C. Hybrid Applications**

To evaluate RegionDroid for hybrid applications, we com-
pare apps in \(S_3\) pairwisely like in \(S_2\), and with threshold \(\delta_s\)
0.78, three pairs are found ‘repackaged’. We check these three
pairs manually. One of them is the real repackaged pair. The
other 2 pairs are false positives. The first false-positive pair is
because one of the app is a game which failed to run in our
virtual device and the dumped layout is almost empty. The
other app also has only one simple layout coincidentally, which
make their similarity score high. The second false-positive pair
is because they use same UI framework without much changes.
They only changes some texts and the corresponding links to

TABLE I

<table>
<thead>
<tr>
<th>Hybrid Framework</th>
<th>Apk Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appcelerator [30]</td>
<td>9</td>
</tr>
<tr>
<td>Framework7 [29]</td>
<td>7</td>
</tr>
<tr>
<td>ionic [27]</td>
<td>4</td>
</tr>
<tr>
<td>Mobile Angular UI [43]</td>
<td>5</td>
</tr>
<tr>
<td>Ossen UI [44]</td>
<td>8</td>
</tr>
<tr>
<td>PhoneGap [28]</td>
<td>4</td>
</tr>
<tr>
<td>Xamarin [45]</td>
<td>3</td>
</tr>
<tr>
<td>Total</td>
<td>40</td>
</tr>
</tbody>
</table>
by chance. Since we often generate birthmarks for each app and the number of traversed layouts can be greatly different in one repackaging generation, it is trapped into different parts of app layouts. We check this pair. Repdroid does not limit the depth of graph generation, it is trapped into different parts of app layouts and the number of traversed layouts can be greatly different by chance. Since we often generate birthmarks for each app just once in practice, this uncertainty will heavily affect the accuracy of repackaging detection.

Table II shows that the birthmark generation in RegionDroid (3.12 minutes on average) takes much shorter time than Repdroid does (14.09 minutes on average). The limited depth of graphs also helps speed up the similarity calculation because fewer groups in RGGs are compared during calculation.

E. Action Type and Graph Depth

We mentioned that our approach only use action click and do not traverse the whole app. To prove that it is reasonable, we conduct the experiment on Repdroid with the data set $S_2$. Two parts of Repdroid have been modified: 1) We only trigger click action during the graph generation strategy. 2) We only generate LGG with certain depth. We run Repdroid with different depth thresholds as those in original Repdroid. The result of experiment is shown in Table III. From the table we find that different depth of graph can significantly affect the time consumption of graph generation, while rarely affect the accuracy of repackaging detection.

F. Threats to Validity

We have tried to design our experiments for the representative evaluations, but there are still some threats which may affect the validity. Firstly, we have tried to find hybrid apps built by famous hybrid frameworks as many as possible. However, the showcases of these frameworks contains few apps and some of them failed to run in our virtual devices. Apps built from same framework may have features in common, which can be a testpoint for evaluating credibility. So the lack of such kinds of apps may affect the representativeness of our work. Secondly, we selected hybrid apps by statically checking whether the layouts of apps contain element ‘webview’ or HTML, css files. Such kinds of layouts may be in a deep position, i.e. the depth of them is larger than the threshold. We can not guarantee that RegionDroid will certainly visit the layouts which contain these web elements during executions of all apps.

V. DISCUSSION

A. Attacks and Defenses

Generally speaking, the Android app repackaging attacks can be classified into the three categories [13], lazy attack, amateur attack, and malware.

1) Lazy attack: Lazy attackers often modify apps with automatic code obfuscation tools and does not change the functionality of original apps. They may also change the information of apps, such as author name, company name, etc. However, these changes will not affect the UI features and the execution traces. Our birthmark will not be affected and the repackaging detection is still effective.

2) Amateur attack: Amateur attackers will make modifications on the functionalities or UI layouts. Parts of code or resources will be modified, added, or deleted for attackers’ certain purposes. For example, they may try to add a new layout to lead users to browsing attackers’ advertisements.
TABLE II

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Average Time Consumption (S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Birthmark Generation</td>
</tr>
<tr>
<td>S2</td>
<td>3.12 min</td>
</tr>
<tr>
<td>S3</td>
<td>14.09 min</td>
</tr>
</tbody>
</table>

Table II: Comparison Results

(a) Dataset $S_1$.
(b) Dataset $S_2$.
(c) Dataset $S_3$.

Fig. 10. The distribution graphs of the RGG size.

3) Malware: Malware attackers will insert malicious code into apps. For example, they may upload user privacy information, damage device system, or hijack and tamper web links. Malwares often try to hide themselves to avoid being discovered, and it will not affect app UI features. So RegionDroid still works for this kind of attack.

4) Other Potential Attacks: There are other potential attacks against our approach. Firstly, attackers can obfuscate UI layouts far different from the original ones but still ensure that the runtime presentation does not change. For example, they can add a completely transparent layout overriding the original one. The looking of current layout is still the same as original one, but the contents dumped is replaced by the transparent layout. However, to make apps behave similarly, they need to handle the event listeners so that the transparent layout can response to user interface correctly. As a result, such attacks may be a great burden for repackaging. Secondly, attackers may change the positions and size of views in layouts. If the changes are small or the numbers of changed layouts are small, the overall similarity can still be high and our approach can tolerate such changes. If attackers tries to re-design the apps’ layouts but make layouts still look similar (e.g. symmetrical views to the center), it could affect our work and can be a potential threat. However, as far as we know, there are no such automatic tool for this kind of attack. Attackers need to modify XML or HTML files carefully so that new layout can be similar enough to confuse users.

TABLE III

<table>
<thead>
<tr>
<th>Depth</th>
<th>Time consumption (minutes)</th>
<th>False positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5.11</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5.38</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>6.08</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5.25</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>7.31</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>7.05</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>6.23</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>6.85</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8.21</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>8.34</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8.81</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8.78</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>15.77</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>15.05</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>12.52</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>14.22</td>
<td>4</td>
</tr>
</tbody>
</table>

Fig. 11. A false positive in $S_3$.

Fig. 10. The distribution graphs of the RGG size.

However, to leverage the popularity of the original app, the remaining parts of repackaged app should be similar as the original one. As a result, the similarity score can still be high and the repackaging can be detected as well.
B. Limitations and Future Works

Firstly, since our approach is based on UI regions, if the app have few layouts, e.g. service apps which almost run in the background, we can not detect repackaging over these kinds of apps. Secondly, like most existing works, RegionDroid can not reach the layouts which need specific user input, such as password authentication. However, since repackaged apps try to maintain the same behaviors with the original ones, if layouts in original apps are unaccessible, the corresponding layouts in repackaged apps are also unaccessible. As a result, the RGG extracted from runtime UI features can still share similar characteristics. Thirdly, our approach can not deal with the apps where graphic UI are designed by OpenGL directly, e.g. most of games. The runtime dumped contents do not reflect the UI view hierarchies of current layout, not to mention analyzing their features. Lastly, third-party webview bring some troubles to RegionDroid. Since hybrid apps are becoming more and more popular, some organizations or companies have developed third-party webviews for hybrid app developers, such as CrossWalk [46] and TBS [47]. They contribute to their webviews for better app performance. Although Android native webview (since Android 4.4) has been replaced by Chromium, which is a powerful open-source webview core for hybrid apps (CrossWalk is no longer updated due to this replacement [48]), there are still apps embedded with third-party webviews. We can not dump contents from third-party webviews easily with existing tools for analysis. Although it does not affect the model of our approach itself, it is still a challenge for repackaging detection which can be our future works.

VI. Conclusion

In this paper, we proposed a dynamic birthmark RGG based on UI region features to detect Android app repackaging. Our approach execute apps automatically and analyze the runtime UI of apps. We partially traverse apps within a limited depth and focus on the region of layouts to deal with weakly typed views in hybrid apps. We also implement the tool RegionDroid based on RGG and conducted the experiment to evaluate our approach. The experiment shows that our approach has good credibility and resistance when it comes to the obfuscated and encrypted apps. Additionally, our approach also has good performance in detecting repackaging among hybrid apps.

ACKNOWLEDGMENT

This work was supported by the National Key R&D Program of China (Grant No. 2017YFB1001801), National Natural Science Foundation (Grant Nos. 61690204, 61472174) of China, the Fundamental Research Funds for the Central Universities, and the Collaborative Innovation Center of Novel Software Technology and Industrialization.

REFERENCES


