RepDroid: An Automated Tool for Android Application Repackaging Detection

Shengtao Yue*†, Weizan Feng*†, Jun Ma*†‡, Yanyan Jiang*†, Xianping Tao*†‡, Chang Xu*† and Jian Lu*†
*State Key Laboratory for Novel Software Technology, Nanjing University
†Department of Computer Science and Technology, Nanjing University
{mg1533079,141220029}@smail.nju.edu.cn, majun@nju.edu.cn, jiangyy@outlook.com, {txp,changxu,lj}@nju.edu.cn

Abstract*—In recent years, with the explosive growth of mobile smart phones, the number of Android applications (apps) increases rapidly. Attackers usually leverage the popularity of Android apps by inserting malwares, modifying the original apps, repackaging and releasing them for their own illegal purposes. To avoid repackaged apps from being detected, they usually use sorts of obfuscation and encryption tools. As a result, it's important to detect which apps are repackaged. People often intuitively judge whether two apps are a repackaged pair by executing them and observing their runtime user interface (UI) traces. Hence, we propose layout group graph (LGG) built from UI trances to model those UI behaviors and use LGG as the birthmark of Android apps for identification. Based on LGG, we also implement a dynamic repackaging detection tool, *RepDroid*. Since our method does not require the apps' source code, it is resilient to app obfuscation and encryption. We conducted an experiment with two data sets. The first set contains 98 pairs of repackaged apps. The original apps and repackaged ones are compared and we can detect all of these repackaged pairs. The second set contains 125 commercial apps. We compared them pair-wisely and the false positive rate was 0.08%.

***Keywords*—Android application, Repackaging detection, User interface, Obfuscation resilient**

I. Introduction

Recently, with the development of mobile devices, the number of mobile apps increases dramatically and there were more than 2.4 million apps available in the Google Play market by October 2016 [1]. However, Android apps can be modified and *repackaged* by reverse engineering tools (e.g., Apktool [2], dex2jar [3], and Soot [4]). Attackers leverage the popularity of Android apps and repackage them for illegal purposes including embedding advertisements and inserting malware. The prevalence of unofficial and third-party Android markets even further facilitates the growth of such illegal behaviors. Previous studies show that 5% to 13% of apps were plagiarisms in the official Android market [5] and 1083 of the analyzed 1260 malware samples (86.0%) were repackaged versions of legitimate apps with malicious payloads [6].

One common solution to detect Android repackaging is to generate and compare the *birthmarks* [7] of Android apps. A birthmark represents an app's unique characteristic which can be used to distinguish an app from the others, satisfying the following two properties [7].

Property 1: (*Resistance*) If program p' is obtained from p by any program semantic preserving transformation, the birthmarks of both p' and p should be the same. Obfuscation and encryption, shouldn't 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.

Birthmarks can be generated *statically* (via analyzing the binary source) or *dynamically* (via analyzing the execution traces). Static birthmarks [5], [8]–[13] are usually credible, however, may not resist code obfuscations [14] and could even hardly work when it comes to app encryption [15]. Since repackaged apps are generally functional equivalent, a promising direction is to profile actual app executions to obtain birthmarks. By using API invocation sequences [16] or runtime UI information [17], dynamic birthmarks are naturally resistant to code obfuscations. However, some semantic-preserving obfuscations, such as inserting useless activities into apps, or packaging the original layouts with multi nested views (both introduced in Section V), may heavily affect the effectiveness of such dynamic birthmarks.

In this paper, we propose RepDroid, a dynamic birthmark that is more resistance to semantic-preserving obfuscations. RepDroid is based on the observation that end-users are very good at finding repackaged apps by merely looking at the UI layouts and their interactions, without any source codes or professional knowledge. This is because attackers usually leverage the popularity of an original app and make their repackaged apps look and behavior similar with the original one, yielding similar UI traces.

We propose the *layout-group graph* (LGG) as dynamic birthmark of an app, which is constructed based on the runtime UI traces, and use the similarity between LGGs to determine whether apps are repackaged. Our approach is resilient to the semantics preserving obfuscation and encryption because (1) our approach only requires the runtime UI traces rather than the app's static codes. It means any obfuscation or encryption of the code, which does not change the UI much, will hardly affect the behavior of our detection; and (2) we can always dump the required UI and interact with the apps, no matter how the original app is encrypted or how the encrypted app works at runtime.

In summary, the main contributions of this paper are:

- We propose LGG, a dynamic app birthmark built from runtime UI traces.
- We implemented a fully automated tool RepDroid to

[‡] Corresponding authors

construct LGG and detect repackaged apps.

- We evaluated the credibility and resistance of RepDroid using real-world open-source and commercial apps and received promising results.

The rest of this paper is organized as follows: Section II introduces the background of our work. Section III presents the methodology of our approach. Section IV shows the architecture of RepDroid. Section V evaluates our tool, along with 2 previous works, by applying them to 2 different data sets. Section VI discusses the RepDroid and shows its limitation and future work. Section VII introduces the related work. Finally, we conclude the paper in Section VIII.

II. Background

A. Android Application Structure

Android apps are usually written in Java programming language. The Java program codes are compiled to a *.dex* file by Android SDK tools. The *.dex* file along with related data and resource files are finally packaged into an APK file with suffix *.apk*. An Android app consists of four types of app components: *Activities*, *Services*, *Content Providers* and *Broadcast Receivers* [18]. An *Activity* represents a single screen with a UI. A *Service* runs in the background to perform long-running operations or perform work for remote processes. A *Content Provider* manages a shared set of app data and a *Broadcast Receiver* responds to system-wide broadcast announcements.

Generally speaking, an Android app often consists of several activities. Each activity is given a *Window* in which to draw its UI. A window typically fills the screen, but it may be smaller than the screen and floats on top of other windows. A *View* is a particular rectangular space within the window and is able to respond to user interaction. The view can be divided into two categories, *Widgets* and *ViewGroups*. *Widgets* are views that provide visual and interactive elements for the screen, such as a *Button* for sending message, a *TextField* for message input, and so on. *ViewGroups* are views that can contain other views. All of these views are used to create and organize the layouts of the apps.

In the rest of our paper, in order to distinguish the meaning of the word 'layout', we use *Layout*, with capitalizing the first letter and never presented alone, to describe the views derived from *ViewGroup* which provide a unique presentation model for its child views, such as *LinearLayout*, *GridLayout*, and so on [19]. We use the single word *layout* to define the visual structure (Fig. 1 [20]) for a UI [21].

B. UIAutomator

UIAutomator is a testing framework through which Android testers can build their test cases to operate interactions with apps [22]. Besides, UIAutomator can dump the current layout into an XML file which is a kind of view hierarchy tree and Fig. 2 shows an example. Each node in the tree represents a view in the layout and the attribute *class* indicates the specific type of the view. The nodes always contain 17 attributes, such as *index, text, class, package, clickable, checkable, etc*. A layout can be declared in a predefined XML file or it can be created at runtime, but what we dump by UIAutomator is always the final presentation of the layout at runtime.

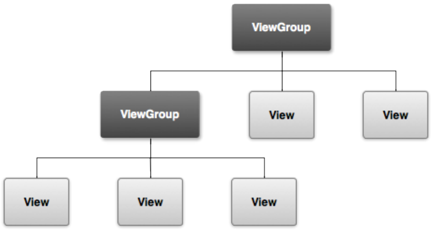
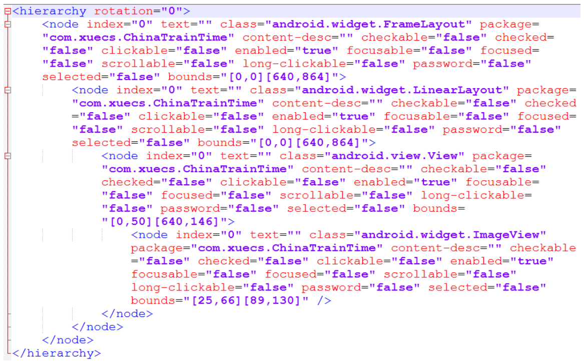


Fig. 1. Illustration of a view hierarchy, which defines a UI layout.



```
<hierarchy rotation="0">
    <node index="0" text="" class="android.widget.FrameLayout" package=
    "com.xuecs.ChinaTrainTime" content-desc="" checkable="false" checked=
    "false" clickable="false" enabled="true" focusable="false" focused=
    "false" scrollable="false" long-clickable="false" password="false"
    selected="false" bounds="[0,0][640,864]">
        <node index="0" text="" class="android.widget.LinearLayout" package=
        "com.xuecs.ChinaTrainTime" content-desc="" checkable="false" checked
        ="false" clickable="false" enabled="true" focusable="false" focused=
        "false" scrollable="false" long-clickable="false" password="false"
        selected="false" bounds="[0,0][640,864]">
            <node index="0" text="" class="android.view.View" package=
            "com.xuecs.ChinaTrainTime" content-desc="" checkable="false"
            checked="false" clickable="false" enabled="true" focusable=
            "false" focused="false" scrollable="false" long-clickable=
            "false" password="false" selected="false" bounds=
            "[0,50][640,146]">
                <node index="0" text="" class="android.widget.ImageView"
                package="com.xuecs.ChinaTrainTime" content-desc="" checkable
                ="false" checked="false" clickable="false" enabled="true"
                focusable="false" focused="false" scrollable="false"
                long-clickable="false" password="false" selected="false"
                bounds="[25,66][89,130]" />
            </node>
        </node>
    </node>
</hierarchy>
```

Fig. 2. The xml of a layout

III. Methodology

Intuitively, two functionally-equivalent repackaged clones should offer similar experiences (visual presentations, user interactions, etc.) to an end-user. Furthermore, considering that no static analysis could precisely analyze encrypted code, we take a dynamic approach that compares two apps' UI traces to detect repackaging in an app market (Fig. 3):

1) Each app is executed automatically and execution traces are collected.
2) Each app's birthmark is extracted from its execution traces. The birthmark is a graph constructed based on runtime layouts and their transitions, which is named layout-group graph (LGG).
3) A pair of repackaged apps are characterized by their similar birthmarks, i.e., the similarity score between their LGGs over a certain threshold. Even if the collected LGGs may be partial and repackaged apps may have slightly different LGGs, as long as the major functionality and visual signals remains the same, we are able to detect the repackaging.

In practice, the first two steps are conducted in graph generation strategy (introduced in Section III-C), which is used to traverse apps and generate LGGs as well. Details of our approach are expanded as follows.

A. The Layout-group Graph (LGG)

The LGG is defined in Definition 3. We use the runtime layouts to be the graph vertexes, representing how UI looks,

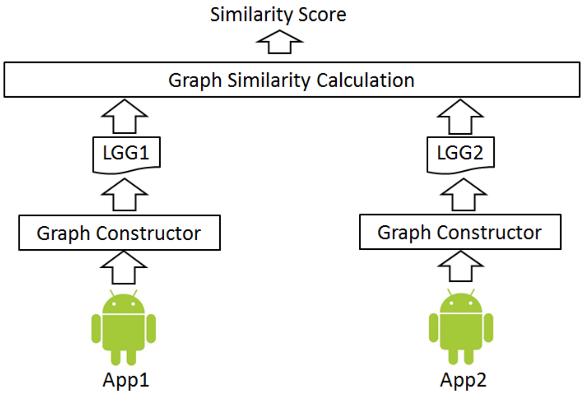
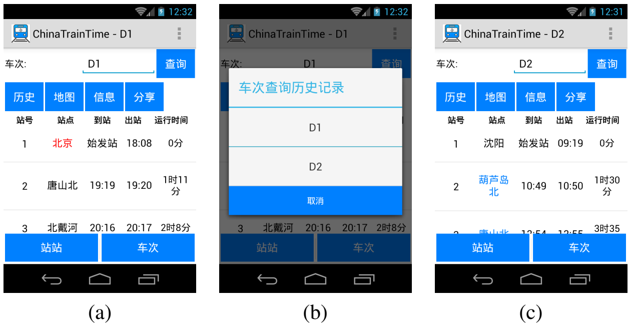


Fig. 3. Methodology architecture.



站号	站点	到站	出站	运行时间
1	北京	始发站	18:08	0分
2	唐山北	19:19	19:20	1时11分
3	北戴河	20:16	20:17	2时8分

站号	站点	到站	出站	运行时间
1	沈阳	始发站	09:19	0分
2	葫芦岛北	10:49	10:50	1时30分
				3时35

(a) (b) (c)

Fig. 4. Different layouts in the same activity. The app is used to query train timetables. Fig. 4a shows the layout of the query results with keyword ‘D1’. Fig. 4b shows the layout of search history list. Fig. 4c shows the layout of the query results with keyword ‘D2’

and use the layouts transitions to be the graph edges, representing the UI transitions. A transition is triggered by an action (Definition 1). Besides, people often consider that the layouts which differ a little are the ‘same’, so, accordingly, each graph vertex is not a single layout, but a layout sets in which similar layouts are grouped.

Definition 1 (Action): An action $a = (v, t)$ means an interaction with a view triggered by users, where v is the interacted view, and the action type $t \in AT = \{$*Click, LongClick, ScrollBackward, ScrollForward, SwipeToRight, SwipeToLeft, Menu, Home, Back*$\}$.

Definition 2 (Layout Group): A layout group g is defined as a layout set L in which similar layouts are grouped.

Definition 3 (LGG): A LGG is defined as a directed graph $LGG = (G, E, A, \alpha)$ where G is a set of layout 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 and A is a set of actions. Function $\alpha : E \rightarrow A$ represents an action when user triggers the particular view.Supposing an edge $e = g_1 \rightarrow g_2$, $a = \alpha(e)$ means the action that after user interacts with the view $a.v$ in layout l_1, the UI changes to layout l_2 where $l_1 \in g_1$ and $l_2 \in g_2$.

B. Layout Similarity Calculation

End-users often consider the layouts with few different views (e.g. Fig. 4a and Fig. 4c) can be treated as the layouts with same functions and behaviors. So we merge these layouts into same group.

An example of layout grouping is shown in Fig. 4. All three layouts are in the same activity *ChinaTrainTimeMain2*. From a human perspective, layouts in Fig. 4a and 4b are supposed to belong to different groups because they seem obviously different and are used for different app functions. But we can not distinguish these two layout by the name of the activity. On the other hand, layouts in Fig. 4a, 4c should be merged into the same group since they only differ little in their search lists.

We measure the similarity scores between layouts by converting the layout tree into a vector and calculate the distances between the vectors, which can be divided into the following steps. We first compress the layout tree by removing the tree nodes, which contain exactly one child, and making its parent and its child connected. The compression can not only reduces the size of tree for comparison, but also reduces the impacts of multi nested views. The class of view represents both of its visual presentation and functionality, which is the main attribute to represent the feature of the view. Therefore, we transform the compressed tree into a vector (in the breadth-first search order), and each element in the vector is the class name of the view. For example, the XML in Fig. 2 is dumped from the layout in Fig. 4b by UIAutomator, its breadth-first search order $V = \langle F, F, F, L, L, F, L, LV, V, F, L, T, LV, B, T \rangle$ where F means *FrameLayout*, L means *LinearLayout*, LV means *ListView*, V means *View*, T means *TextView*, and B means *Button*. After compression, the vector should be $V = \langle L, L, T, B, T, V \rangle$. We use the same method to convert the other layouts and result in vectors V_2 and V_3 which are corresponding to Fig. 4a and 4c. Then we calculate the edit distance between vectors in a class-name-level, which means this *edit distance* is the minimum amount of operations, which consist of removing, inserting or substituting a *class name* in the vector to transform vector A to B. For example, the edit distance between vectors $V_1 = \langle F, F, T, L \rangle$ and $V_2 \langle F, F, L \rangle$ is 1 (removing class name *TextView*). The views in layouts dumped by UIAutomator are all the standard views provided by Android system. The Android app developers often design their own custom views, but the final presentation dumped by UIAutomator at runtime is the combination of the standard views, i.e. the dumped hierarchy consists only standard views rather than custom ones. This make it meaningful to compare the class names of views in different apps.

Let V_1 and V_2 be the vectors of the compressed layouts l_1 and l_2, D be the edit distance between these two vectors, N_1 and N_2 be the amounts of the vectors' elements, the similarity between l_1 and l_2 is calculated by Sim_l shown in Equation 1. If the similarity between two layouts is larger than the threshold δ_l, we merge them into the same group.

$$Sim_l(l_1, l_2) = 1 - \frac{D}{max(N_1, N_2)} \tag{1}$$

Supposing that $l_i \in g_1.L, l_j \in g_2.L$, the similarity between groups g_1 and g_2 is shown in Equation 2.

$$Sim_g(g_1, g_2) = \max Sim_l(l_i, l_j) \tag{2}$$

C. Graph Generation

To generate LGG, we need to execute an app and collect the runtime UI trace by traversing the app. When end-users or artificial testers try to traverse an app, they often prefer to interact with the views which can lead to the layouts that they haven't arrived at. Therefore, we propose a heuristic strategy shown in Algorithm 1 to simulate such traversal approach. The basic idea of the strategy is that we add the weights w and w_t into each view of the layouts and the system trigger an action to interact with apps based on weights. Here, w means the weight of a view and w_t is an array where $w_t[t]$ means the weight of the action type t ($t \in AT$) on the given view. Each round the system selects an $action$ weighted-randomly, including an interactive view for $action.v$ and an action type for $action.t$. More specifically, 1) The larger the view's w is, the more likely the view will be selected. 2) For the selected view, the larger the $w_t[t]$ is, the more likely t will be selected. All of these weights are set to the initial values 10 and updated according to the behavior of LGG after each selection and execution. The variable *unChangedCount* counts the rounds since LGG has no more changes and resets to 0 as long as the graph changes. *unChangedCount* is set to 0 before the loop starts and the loop terminates when *unChangedCount* is larger than a threshold δ_c.

The main procedure (line 2 to line 35) returns the LGG of target app.

For each round, it first dumps the current layout of the app. Since the layout obtained from UIAutomator is a new instance of current dumped XML, the attributes w and w_t are set to initial values which may not be the correct values. For example, in the previous round, we clicked a button B in the layout tree $layout_1$, the weight of B in $layout_1$ was changed. Then in the current round we get a new layout tree $layout_2$. Although we can judge whether $layout_1$ and $layout_2$ are the similar layouts or not via Sim_l function, the weights of B in $layout_2$ are still the initial values because $layout_2$ is instantiated by current dumped XML which does not store the values of weights. Therefore, to create the groups and layouts with correct weight values, we use GETCURRENTGL(LGG) (line 3, line 37) to get current compressed layout $layout_c$ and its corresponding group $group_c$. In this function, when we get a new layout $layout_c$ from UIAutomator, we can always find a layout $layout_m$ in group $group_m$ (if not empty), which has the maximum similarity with $layout_c$ (line 45). If the similarity between $layout_m$ and $layout_c$ equals to 1, we consider that they are exactly the same. So we return $group_m$ and $layout_m$ directly. If the similarity is larger than threshold δ_l, we consider that they are similar and $layout_c$ should be grouped into $group_m$. So we copy the weight attributes of each node in $layout_m$ to $layout_c$ by MERGEWEIGHT($layout_c, layout_m$) (line 50). The *xpath* of a view (line 68) means the XML path, which is used to locate a node in an XML. Otherwise, we consider that $layout_c$ has no similar layouts so it need to be added into a new empty group.

Then we need to find all views in l_c which can be interacted according to their attributes (line 10), e.g. attributes *clickale, long-clickable, scrollable, or foucsable* are set *true*. Action types *Back, Menu, Home* are always considered as the special views and added into the return result in FINDALLINTERACTIVEVIEW($layout$) (line 10). After that, we choose one of these nodes and the action type weighted-randomly (line 11, line 13). After execution of the action (line 22), the LGG may or may not change. To determine it, we check whether the graph changes (i.e. the amount of the vertex and edge set in LGG changes) (line 24). If so, the action view's and type's weights increase and *unChangedCount* is reset to 0. Otherwise, the weights decrease and *unChangedCount* increases by one. TRYSTAYINCURRENTAPP(i)s used to ensure that the target app is always the current focused app.

D. Graph Similarity Calculation

After we generate the LGG of each app, we need to calculate the similarity between of these graphs (Algorithm 2). A common approach is the subgraph isomorphism algorithm, but such algorithm is sometimes time-consuming. Besides, since we can not always traverse the whole app and the LGG may be partial, the subgraph isomorphism algorithm is too strict to compare different LGGs. Therefore, we can just try to find the matchings of elements in LGGs rather than the subgraphs. Since the repackaged apps always share the similar UI features in common, we can try to find which features from 2 apps can be matched as a pair that results in highest similarity score. That is, we need to find a matching between graphs LGG_1 and LGG_2, so that the sum of similarities between each matched pair can be the maximum. We use the edges of LGG as the matching elements because an edge is the smallest substructure of a graph which contains both the information about its connected groups and their transition. The graph similarity algorithm can be split into the following three steps.

1) Filter Pairs: Firstly, we calculate the difference between the sizes of two LGGs' groups. If the size of the group of one LGG is 2 times larger than the other's, we believe that these two LGGs are different so we set their similarity score equal to 0. This can reduce the graph pairs to be calculated. The remaining pairs will proceed to the following steps.

2) Match Graphs: Secondly, we try to find a maximum weighted matching of edges between graphs LGG_1 and LGG_2 based on bipartite graph matching. We construct a bipartite graph $G_B = (V, E, \omega)$ where the vertices $v \in V$ in G_B can be divided into two disjoint sets V_1, V_2 and we define $V_1 = LGG_1.E$, $V_2 = LGG_2.E$ (Each edge in LGG is represented as a vertex in the bipartite graph). The edge set $E = V_1 \times V_2$, i.e. the bipartite graph G_B is complete, since each edge in LGG_1 is possible to be matched to any edge in LGG_2. Function $\omega : E \rightarrow W \subseteq [0, 1]$ represents the weight of each edge. Supposing an edge $e = v_1 \rightarrow v_2$, the weight $w = \omega(e) =$

Algorithm 1 Graph Generation Strategy

```
$LGG \leftarrow \emptyset$                                          ▷ create empty LGG
procedure MAIN
    $[group_c, layout_c] \leftarrow$ GETCURRENTGL($LGG$)
    while $r \leq r_m$ do                          ▷ $r_m$, the maximum rounds allowed
        $action \leftarrow NoMoreAction$
        $unChangedCount \leftarrow 0$
        if $unChangedCount >= \delta_c$ then
            $action \leftarrow NoMoreAction$
        else
            $viewList \leftarrow$ FINDALLINTERACTIVEVIEW($layout_c$)
            $view_s \leftarrow$ WEIGHTEDRANDSELECT($viewList$)
            if $view_s \neq null$ then
                $type_s \leftarrow$ WEIGHTEDRANDSELECT($view_s.w_t$)
                Update $view_s.w$ and $view_s.w_t$
                $action.v \leftarrow view_s$
                $action.t \leftarrow type_s$
            end if
        end if
        if $action = NoMoreAction$ then
            $break$
        else
            DOACTION($action$)
            Update LGG to according to the transition after action
            if LGG's vertex or edge count changed then
                increase weight $action.v.w$ and $action.n.w$
                $unChangedCount \leftarrow 0$
            else
                $unChangedCount \leftarrow unChangedCount + 1$
            end if
            TRYSTAYINCURRENTAPP()
        end if
        $r \leftarrow r + 1$
    end while
    return $LGG$
end procedure

procedure GETCURRENTGL($LGG$)
    $layout \leftarrow$ DUMPLAYOUT()
    $layout_c \leftarrow$ COMPRESSLAYOUT($layout$)
    if LGG is empty then
        $group \leftarrow$ new group
        $group.addLayout(layout_c)$
        return $[group, layout_c]$
    end if
    $[group_m, layout_m] \leftarrow$ GETSIMILARLAYOUT($LGG, layout_c$)
    $sim = Sim_l(layout_c, layout_m)$
    if $sim = 1$ then
        return $[group_m, layout_m]$
    else if $sim \geq \delta_l$ then
        MERGEWEIGHT($layout_c, layout_m$)
        $group_m.addLayout(layout_m)$
    else
        $group_m \leftarrow$ new group
        $group_m.addLayout(layout_c)$
    end if
    return $[group_m, layout_c]$
end procedure

procedure GETSIMILARLAYOUT($LGG, layout_c$)
    Find $group_m \in LGG.G, layout_m \in group_i.L$
    s.t. $Sim_l(layout_m, layout_c)$ can be the maximum.
    return $[group_m, layout_m]$
end procedure

procedure MERGEWEIGHT($layout_m, layout_c$)
    for all $view \leftarrow layout_m.V$ do
        $view_c \leftarrow layout_c.find(view.xpath)$
        if $view_c \neq null$ then
            $view_c.w \leftarrow view.w$
            $view_c.w_t \leftarrow view.w_t$
        end if
    end for
end procedure
```

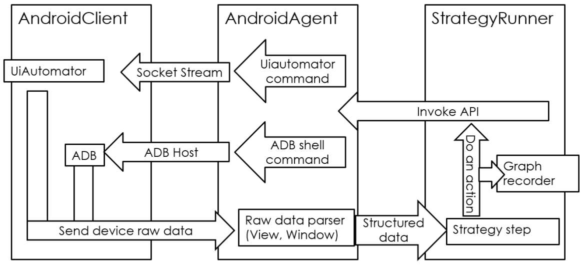


Fig. 5. Structure of graph constructor.

WEIGHT$(v1, v2)$ (line 16). Then we use the Kuhn-Munkres algorithm [23] to calculate the maximum weight matching of the bipartite graph G_B. Without loss of generality, we assume the amount of V_1 is no larger than that of V_2 and find the vertices in V_2 which match vertices in V_1. Then we get the matching array M where $M[i]$ means the weight of edges between $v_i \in V_1$ and v_i's matched vertex. We call $M[i]$ the matching score of the vertex v_i.

3) Normalization: Finally we normalize the matched score as the similarity. The graph similarity is normalized by Equation 3, where N is the size of V_1:

$$Sim_G = norm(M_f) = \frac{\sum_i M_f[i]}{N} \tag{3}$$

Algorithm 2 Graph similarity calculation

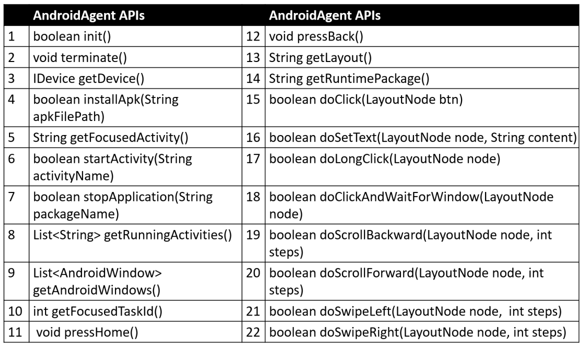
```
 1: procedure SIMILARITY($LGG_1$, $LGG_2$)
 2:     if amounts of two graphs' groups differ more than 2 times then
 3:         return 0
 4:     end if
 5:     $G_B.V_1 \leftarrow LGG_1.E$
 6:     $G_B.V_2 \leftarrow LGG_2.E$
 7:     for all $v_i \in V_1, v_j \in V_2$ do
 8:         $G_B.addEdge(v_i, v_j,$ WEIGHT$(v_i, v_j))$
 9:     end for
10:     $M \leftarrow$ KUHN-MUNKRES($G_B$)
11:     $M_n \leftarrow norm(M_f)$
12:     return $M_n$
13: end procedure
14: procedure WEIGHT($v_1$, $v_2$)
15:     $w \leftarrow Sim_g(v_1.g_1, v_2.g_1) + Sim_g(v_1.g_2, v_2.g_2)$
16:     return $w/2$
17: end procedure
```

IV. System Architecture

A. Overview

We implement the tool RepDroid based on the methodology to generate LGG and finally calculate the similarity score. The main part of RepDroid is the LGG graph constructor and Fig. 5 shows the structure of it. Graph constructor can be split into three sub-parts: *AndroidClient*, *AndroidAgent* and *StrategyRunner*. AndroidClient runs in the Android Virtual Device (AVD) and the other two parts run in a Java host. AndroidClient is used to execute actions and gather the runtime data of apps. StrategyRunner generates the actions and record the graph based on the UI. AndroidAgent plays a key role that it bridges the AndroidClient with the StrategyRunner.

Before graph constructor runs, we need to set the properties of the AVD, such as target SDK version, so that each Android app can be executed in the Android environment correctly. After configuration, the constructor launches AndroidAgent to create a new AVD and backups the *userdata-qemu.img* file. To ensure that each app can be started from same initial environment, the original *userdata-qumu.img* file of the AVD is



	AndroidAgent APIs		AndroidAgent APIs
1	boolean init()	12	void pressBack()
2	void terminate()	13	String getLayout()
3	IDevice getDevice()	14	String getRuntimePackage()
4	boolean installApk(String apkFilePath)	15	boolean doClick(LayoutNode btn)
5	String getFocusedActivity()	16	boolean doSetText(LayoutNode node, String content)
6	boolean startActivity(String activityName)	17	boolean doLongClick(LayoutNode node)
7	boolean stopApplication(String packageName)	18	boolean doClickAndWaitForWindow(LayoutNode node)
8	List<String> getRunningActivities()	19	boolean doScrollBackward(LayoutNode node, int steps)
9	List<AndroidWindow> getAndroidWindows()	20	boolean doScrollForward(LayoutNode node, int steps)
10	int getFocusedTaskId()	21	boolean doSwipeLeft(LayoutNode node, int steps)
11	void pressHome()	22	boolean doSwipeRight(LayoutNode node, int steps)

Fig. 6. The operation interfaces that *AndroidAgent* provides

backed up and will be restored before each execution. In order to speed up the experiment, AndroidAgent can creates multi AVDs at same time so that programs can be run in parallel. Each AVD has an AndroidClient installed by AndroidAgent and is connected with AndroidAgent all the time.

B. *AndroidClient*

AndroidClient is a class extending UIAutomator and works in the AVD. Our approach is designed for dynamic detection based on UI, so it's important for us to obtain the current layout information shown on the screen and interact with the apps. With the help of UIAutomator, we can dump the view hierarchy of UI and interact with the views. We create a socket thread both in the UIAutomator and AndroidAgent, and implement a certain protocol through socket communication. The protocol consists of a number of commands so that the AndroidAgent can tell UIAutomator which action will be executed.

The AndroidClient protocol consists of most basic actions API in terms of interactions including *PressHome*, *PreesBack*, *Click*, *LongClick*, *Scroll*, *GetLayout*, etc. These operations can help us deal with most operations in Android apps by sending and receiving commands.

C. *AndroidAgent*

AndroidAgent is designed to build a bridge between AndroidClient and StrategyRunner. Its detailed functions are as follows:

1) Provide Android-related API: The StrategyRunner needs a high-level API to accomplish its strategy. AndroidAgent connect with AndroidClient for action-related APIs. Besides, we still need some other functions to achieve certain purposes. For example, we need to create and control AVDs, install or uninstall the apps, start the apps. Fig. 6 shows the interface that AndroidAgent provides.

2) Parse raw data: The data obtained from AVD can not be dealt with directly, so we need a parser to structure the raw data. We create a tree structure for the layout XML, called *layout tree*, which can be convenient for StrategyRunner to use. The nodes in a layout tree represent the responding views.

TABLE I
OBFUSCATED OR ENCRYPTED APPS

Repackaging Type	Original App Count	Repackaged App Count
IJiami	38 (Wandoujia)	38
AndroCrypt	20 (F-droid)	20
FakeActivity		20
NestedLayout		20

D. *StrategyRunner*

StrategyRunner is used to generate and record the actions and finally construct a LGG for the app by implementing the *Graph Generation Strategy*. As soon as an app is launched, the whole system is in such a loop that for each round, the StrategyRunner generates one action according to layout data and records the action, the AndroidAgent controls AndroidClient to execute the action, the AndroidClient returns layout data back to AndroidAgent, and accordingly StrategyRunner generates a new action for next round. The loop ends when there satisfies the termination condition. Finally, the LGG is constructed based on the runtime UI traces.

V. EVALUATION

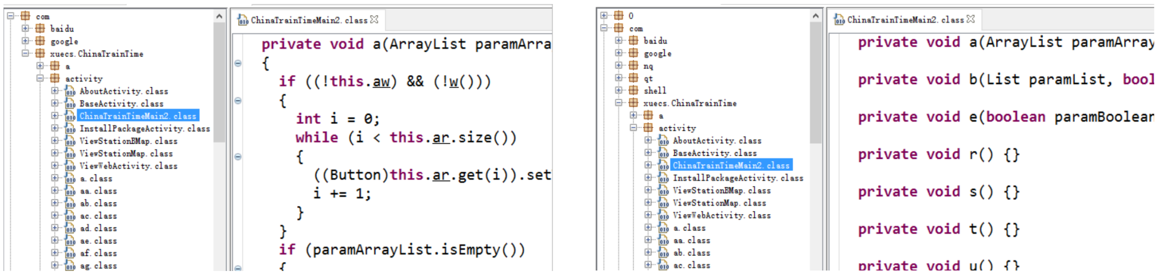
We conducted the experiments[1] on a Dell Workstation with a quad-core Xeon processor and 64GB memory running Microsoft Windows 10. We created four Virtual machine to obtain birthmark in parallel, each runs Android 4.2.2 and has 1GB memory. We set the thresholds $\delta_c = 200$ and $\delta_l = 0.75$ in the experiments. The impact of changing δ_c and δ_l are discussed in Section V-E.

A. *Dataset*

In order to evaluate the two properties of our approach, we create two data sets. The first data set S_1 contains 58 Android apps which were downloaded from *Wandoujia* [24], one of the most popular Android app market in China, and *F-droid* [25], an installable catalogue of free and open source Android applications. These apps were obfuscated or encrypted by *Ijiami* [26], *AndroCrypt* [16], *FakeActivity*, and *NestedLayout*. The apps, along with obfuscated of encrypted ones totally consists of 98 pair of apps (shown in Table. I), which is designed for evaluating the *Resistance* of our approach. The data set from *F-droid* is provided by the dynamic detection work [16]. The second data set S_2 contains 125 Android apps, which are the top ranked apps from 8 categories (shown in Table. II) in *Wandoujia* and each of them is downloaded at least 100,000 times. S_2 is used for evaluating the *Credibility* of our approach.

Ijiami, a kind of Android packer [27], encrypts the original app without change its execution semantics [28]. Ijiami provides a complete pack service with its own shelled classes, *NativeApplication* and *SuperApplication*, and the original file is encrypted as an asset file packed into a new APK and is dynamic loaded during runtime [27]. 38 apps have been

[1] RepDroid and datasets are available at http://moon.nju.edu.cn/people/junma/RepDroid



```
com
  baidu
  google
  xuecs.ChinaTrainTime
    a
    activity
      AboutActivity.class
      BaseActivity.class
      ChinaTrainTimeMain2.class
      InstallPackageActivity.class
      ViewStationBMap.class
      ViewStationMap.class
      ViewWebActivity.class
      a.class
      aa.class
      ab.class
      ac.class
      ad.class
      ae.class
      af.class
      ag.class
```

ChinaTrainTimeMain2.class

```
private void a(ArrayList paramArra
{
  if ((!this.aw) && (!w()))
  {
    int i = 0;
    while (i < this.ar.size())
    {
      ((Button)this.ar.get(i)).set
      i += 1;
    }
  }
  if (paramArrayList.isEmpty())
  {
```

```
0
com
  baidu
  google
  nq
  qt
  shell
  xuecs.ChinaTrainTime
    a
    activity
      AboutActivity.class
      BaseActivity.class
      ChinaTrainTimeMain2.class
      InstallPackageActivity.class
      ViewStationBMap.class
      ViewStationMap.class
      ViewWebActivity.class
      a.class
      aa.class
      ab.class
      ac.class
```

ChinaTrainTimeMain2.class

```
private void a(ArrayList paramArray

private void b(List paramList, bool

private void e(boolean paramBoolear

private void r() {}

private void s() {}

private void t() {}

private void u() {}
```

Fig. 7. Difference between dex files from the original app (left) and the encrypted app by *Ijiami* (right).

TABLE II
THE CATEGORIES IN S_2

Categories	Amounts	Categories	Amounts
Office	15	News	15
Communication	16	Reading	16
Finance	15	Health	16
Education	16	Tool	16

uploaded to *Ijiami* and encrypted. Fig. 7 shows the difference between the dex files from the original apps and the encrypted apps by Ijiami. It shows that the original package structure doesn't change a lot, but all of the original methods, e.g. *com.xuecs.ChinaTrainTime.activity.ChinaTrainTimeMain2* in the figure, are set to empty functions. As a result, although the encrypted apps can be decompiled (with errors thrown by decompiler dex2jar, however), there remains little useful information for code analysis.

AndroCrypt is another Android app encrypted tool proposed by [16]. AndroCrypt also encrypt the original APK as an asset file and loads it during runtime. One of the major difference between AndroCrypt and Ijiami is that AndroCrypt loads original classes by Android system APIs (e.g. *dalvik.system.DexClassLoader*), while *Ijiami* uses the native binary.

We implemented an obfuscation tool *FakeActivity* which repackage the original apps with several useless activities with the help of Soot [4]. These activities will never be started during apps' normal execution so they won't change the behaviors of repackaged apps. Besides, we override the method *onCreate* in each fake activity so that it can create a layout if these activities are started by the testing tools.

We also implemented another obfuscation tool *NestedLayout* which packs the original layouts with multi 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.

B. Resistance

We define that an original app and the corresponding encrypted app are a graph pair. As a result, there will be 98 graph pairs. For each pair, we calculate the similarity. From the Fig. 8, where the horizontal axis represents the similarity range, and the vertical axis represents the number of pairs with the corresponding similarity, we can conclude that with the threshold $\delta = 0.75$, all of the pairs are found to be the same.

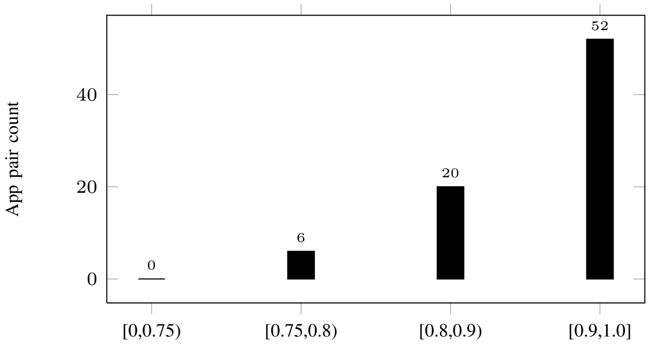


Fig. 8. The distribution graph of the similarity (Resistance)

C. Credibility

We successfully applied RepDroid to all of the 125 apps in S_2 and obtained the LGGs of each app accordingly. With the pair-wise detection for each app, there should be 7750 similarity scores. After we filtered the pairs that the difference between the amount of their groups are more than two times (4930 pairs), there are 2820 similarity scores remaining, and the score distributions are shown in Fig. 9, where the horizontal axis represents the similarity range, and the vertical axis represents the number of pairs with the corresponding similarity. We also split the range '0.7-0.8' into two parts, '0.7-0.75' and '0.75-0.8'. From the figure we can see that most pairs can be distinguished since their similarity scores are lower than threshold $\delta = 0.75$. However there still exists 10 pairs considered as 'repackaged' so we checked these pairs manually. 4 of these 10 pairs are in the range '0.9-1' and are actually the repackaged apps. They are the different versions for same apps. For example, both two apps are doctor apps developed by same company, but one is male version and the other is the female version, which just differs in the colors of app styles. Another 6 pairs are found to be false positives. The reason for it is that these are too few groups generated from these 6 pairs which would affect the accuracy of the detection.

D. Comparisons with Existing Approaches

We have applied both of these two data sets to the *ViewDroid* [13], Soh et al.'s work [17] (called *UI Method*), and

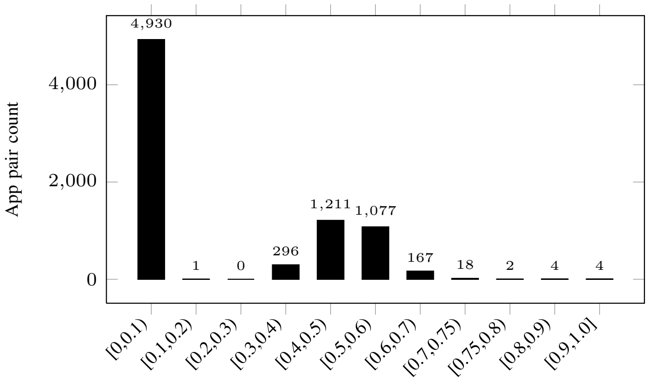


Fig. 9. The distribution graph of the similarity (Credibility)

Kim et al.'s work [16] (called *API Method*). *ViewDroid* generates birthmarks statically, the others dynamically. *Viewdroid* generates view feature graphs based on specific Android APIs as the birthmarks of apps. *API Method* executes apps and collects API call sequence as the birthmarks. *UI Method* orients the UI information with the help of UIAutomator as well. It counts the frequency of selected attributes in each view created by corresponding activities and constructs vectors upon them as the birthmarks. The thresholds and properties for each approach are from corresponding papers. Since we failed to get the source code of *UI Method*, we implement this tool ourselves according to its paper. The comparison results are shown in Table. III.

1) Comparisons in S_1: In data set S_1, the false negative (FN) rate of the result shows in Table. III, column S_1. The FN rate of *ViewDroid* is 59.2%. To the encrypted apps, *Viewdroid* can only analyze the shelled activities, which is totally different from those of the original apps. So all of the apps encrypted by Ijiami and AndroCrypt are found different from the original ones. *API method* can't work when it comes to *Ijiami* set. The encrypted apps by Ijiami partially run with the JNI native binary, instead of Android API, so *API method* can't gather the correct API traces of the app. The similarity scores of *API Method* for each pairs are all zeros and the FN rate is 38.8%. *UI Method* does well in the apps encrypted by *Ijiami* and *AndroCrypt*. It can find all of these repackaged pairs. However, it doesn't work for the apps obfuscated by *FakeActivity*. Since *FakeActivity* insert several fake activities into the apps and the similarity score of two apps is inversely proportional to the maximum of activity amount, *UI Method* will result in a small similarity for each pair, which means *UI Method* can not find out these repackaged apps. *UI Method* is also heavily affected by *NestedLayout* because each layout contains several nested views compared with original ones, which directly affect the accuracy of their birthmarks.

Besides, we have conducted another experiment to show how the encryption affects these approaches. We applied these approaches to the 58 encrypted apps (mentioned above). The apps are detected pair-wisely and finally have 1653 pairs. The result is shown in Table. III, column *Encrypted Apps*. *RepDroid* and *UI Method* can successfully distinguish all of encrypted apps and both have 0% false positive (FP) rate. However, *ViewDroid* can only access the shelled activities which is almost the same for all the encrypted apps with same encryption, so the generated birthmarks result in 42.0% FP rate. As mentioned above, *API method* can't gather the correct API traces of the app so the generated of birthmarks are not reliable and has 40.2% FP rate.

2) Comparisons in S_2: In data set S_2, we calculate the similarities between the unique apps pair-wisely with each approach and calculate the FP rate. The results of each approach are checked manually to ensure that there is no true positive in them. The results are shown in Table. III, column S_2. Since *API Method* only supports Android 2.3.3, most apps in S_2 were not able to be installed. Only 30 of 125 apps successfully executed and generated the API traces. So the total amount of app pairs for *API Method* is 435. From the table we can see that, among the 4 approaches, RepDroid has the lowest FP rate, 0.08%.

In conclusion, compared with other approaches, *RepDroid* not only has an obvious advantage in credibility, but also performs the best in resistance if the apps is obfuscated or encrypted in different ways.

3) Comparisons in Time Consumption: The average time consumption of S_2 is shown in Table. III, column *Time Consumptions*. The average time of *RepDroid* graph construction for each app takes 12.62 minutes (equals to the sum of each apps' time consumption divided by total app count). After the graphs are constructed, the similarity calculation needs much less time which only takes 0.032 seconds for each graph on average. Compared with other approaches, our approach takes the most time for each app. However, since each app only need to run once to generate LGG and RepDroid can create multi AVDs at the same time to speed up the experiment, the time consumption of graph construction can be reduced. In our experiment, we created 4 AVDs in parallel and the time consumption were reduced to 4.23 minutes on average (equals to the total time consumption for all apps divided by total app count), which can be acceptable for repackaging detection. The time consumption can be further reduced if we use more threads.

E. Threshold Values

We determine the threshold values δ_c and δ_l by conducting two experiments on another dataset with 80 apps different from those in S_1 and S_2. δ_c is used for the loop termination condition in *Graph Generate Strategy*. The larger δ_c is, the longer the strategy will take. To find δ_c, we first set $\delta_c = 500$, which is large enough based on prior experience. We then run RepDroid for 80 apps and record the maximum rounds when LGG changes. The maximum rounds distribution is shown in table IV. We can find that most apps need less than 200 rounds. Considering the balance between the time consuming and accuracy, we set $\delta_c = 200$. Although there exists 4 in 80 apps that need more than 200 rounds to change the LGG and thus their graphs may be incomplete, we believe that apps

TABLE III
COMPARISON RESULTS

	S_1		Encrypted Apps		S_2		Time Consumptions	
	FN Count	FNR	FP Count	FPR	FP Count	FPR	Birthmark Generation	Similarity Calculation
RepDroid	0	0%	0	0%	6	0.08%	12.62 min	0.032 sec
ViewDroid	58	59.2%	694	42.0%	108	1.39%	0.82 min	1.97 sec
UI Method	40	40.8%	0	0%	191	2.46%	0.27 min	13.70 sec
API Method	38	38.8%	665	40.2%	6[1]	1.38%[1]	0.92 min	0.09 sec

[1] Only 30 of 125 apps (totally 435 pairs) in S_2 can be installed in Android 2.2.3 used by *API Method*.

TABLE IV
THE MAXIMUM UNCHANGED ROUNDS DISTRIBUTION

Rounds	App Count
0-50	28
50-100	14
100-150	16
150-200	18
more than 200	4
total	80

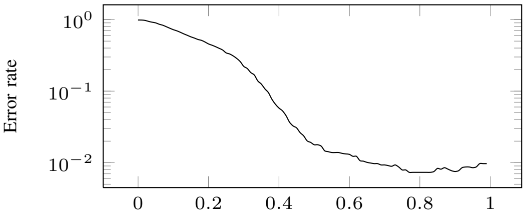


Fig. 10. Error rates with different δ_l

and their repackaged ones must share the graph features in common.

δ_l is used to judge whether two layouts can be grouped into same group in LGG. To find δ_l, we manually execute several apps and dump 100 layouts. We check these layouts one by one and merage similar layouts into same group manually. These groups are used for the ground truth. Then we use different δ_l to group these layouts by Sim_l between each layout and calculate the error rate compared with the ground truth. The Fig. 10 shows the error rate with different δ_l. The horizontal axis means δ_l and the vertical axis means the corresponding error rate. From the figure we can see that δ_l between 0.75 and 0.85 has the lowest error rate, so we set $\delta_l = 0.8$ for our experiment.

F. Threats to Validity

Although we have tried to make our experiment representative, there are still some threats to the validity of the result. Firstly, since we couldn't get the source code of *UI method* and we implemented it ourselves, the result of *UI method* may not be exactly accurate. Secondly, most of apps in S_2 for *API method* failed due to the Android version compatibility, which may affect the result of *API Method*. Finally, the data sets are only from two markets and they are all free apps, which can not necessarily generalise.

VI. DISCUSSIONS

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*.

A lazy attacker often makes some simple changes without changing its code, such as repackaging an app with a different author name. Thus, it can not change the runtime UIs. Therefore, our approach can effectively detect such attacks.

An amateur attacker not only applies automatic code obfuscation but also changes, adds, or deletes a small part of the functionalities. The LGG could be changed if the attack affect the app runtime UIs. However, since we always try to find the maximum weighted matching of the graph and small changes only affect a few matching weights, the attack will slightly reduce the similarity score of whole LGG and can not affect the detection result.

A malware attacker inserts some malicious payload into the original program but still makes repackaged apps look and behavior similar with the original one to leverage the popularity of the original apps. This means original and repackaged apps would share much in common with their UIs, and the LGG would be similar as well. So our approach can also detect such repackaging.

Besides, there are other potential attacks against our approach. Firstly, they can insert a number of invisible views into the layouts (e.g. views that visibility is set to 'invisible', view color is set to 'transparent', or view size is set to zero). However, if the views are designed to be invisible, it will not be presented in runtime and those views can not affect the dumped layouts. Such kind of attack can not affect our approach. Secondly, they can obfuscate the layout XML far different from the original one but still ensure that the runtime presentation does not change. For example, they may add some views and set their colors to be the same as background color. However, as far as we know, there has been no layout preserved transformation tool yet for obfuscation. As a result, if attackers want to conduct such attack, they must modify layout XML files which can be a great burden for repackaging apps.

B. Limitations and Future Works

Firstly, RepDroid can not reach the layouts which need the specific user input, such as password authentication. However, as our work is used for repackaging detection, if scripts for fulfilling these inputs were provided, we can generate the LGG reaching those layouts. Besides, repackaged apps have similar behavior with the original ones. If layouts in original apps are unaccessible, the corresponding layouts in repackaged ones are supposed to be unaccessible as well. So RepDroid can reach similar layouts between the repackaged and original apps. Secondly, since our work is based on UI, if the apps contain few layouts, the similarity scores could be heavily affected and we may not detect the repackaging correctly. Lastly, the accuracy of our approach may be affected when it comes to the hybrid apps, where some views (such as *WebView*) are constructed by web technology such as HTML, JavaScript and CSS. We can not obtain the contents of such views so that we can not analysis them. Our future work will include the research on detecting repackaged hybrid apps.

VII. Related work

A. Static Birthmarks

DroidMOSS [5] applies a fuzzy hashing to each method of the app as fingerprints. Based on the edit distance of two fingerprints, they calculate the similarity of corresponding apps. DNADroid [8] constructs the program dependency graphs (PDGs) for each method in every class of apps and compare the graphs by VF2 subgraph isomorphisms. AnDarwin [9] is another tool which also uses the PDGs. Compared with DNADroid, AnDarwin calculates the local sensitive hashing (LSG) for the semantic vectors of the PDG, which increases the scalability of the detection. Juxtapp [10] uses k-grams of opcodes and feature hashing to determine the similarity between two apps and detect the reuse of Android apps. PiggyApp [11] is a scalable tool to detect piggybacked apps. It applies a module decoupling technique to find the primary part of the comparison. It also improves the detection efficiency by the semantic features from decoupled primary modules. Centroid [12] construct the 3D control flow graphs (3d-CFGs) of each method and calculate the centroid of these graphs. Then, it defines the centroid difference degree to measure the similarity among apps. Viewdroid [13] generates view feature graph, a UI-based birthmark for Android apps, depending on specific Android APIs. To measure the similarity between the graphs, it uses the VF2 subgraph isomorphism algorithm.

B. Dynamic Birthmarks

The work [16] analyzes the Android APIs as well. Compared with Viewdroid, it generates the app execution trace, an API call sequence, instead of the view graph. Besides, the trace is collected during the run-time of the app by testing tool Monkey. The work [17] orients the UI information with the help of UIAutomator. It counts the frequency of certain selected attributes in each view created by the corresponding activities and constructs vectors upon them as the birthmark of the app. To compute the similarity between the apps, it uses E2LSH to find the near neighbors for all activities in each app, and applies the Hungarian algorithm to find the pairs of activities within the nearest neighbors which result in the highest similarity score.

C. Similar App Detections

There are some other related works about detecting similar apps. Different from app repackaging, two apps are similar to each other if they share many features in common even though they are developed dependently [29]. McMillan et al. [29] calculate the similarity index by using the notion of semantic layers to detect closely related applications (CLAN). They introduce a new abstraction that is relevant to semantic spaces that are modeled as existing inheritance hierarchies of API classes and packages. Thung et al. [30] detect similar apps based on CLAN. Instead of API calls, they use collaborative tagging to model the system. Linares et al. [31] propose an approach to detect similar apps in Android (CLANdroid) by analyzing the five semantic archors: identifiers, Android APIs, intents, permissions, and sensors.

VIII. Conclusion

In this paper, we propose the LGG, built from an Android app's runtime UI traces, to model its runtime behavior and use it as the birthmark to detect Android app repackaging. Our approach orients the runtime UI and doesn't require to analyze the static codes of apps. We also implement RepDroid, a dynamic detection tool by constructing the LGG, to detect the app repackaging. The experiment shows that our approach performs well both in its credibility and resistance when it comes to the obfuscated and encrypted apps.

Acknowledgment

This work was supported in part by the National Key Research and Development Plan of China under Grant No. 2016YFB1000802, the National High-Tech Research and Development Program of China under Grant No. 2015AA01A203, the National Natural Science Foundation of China under Grant Nos. 61373011, 61690204, and the Collaborative Innovation Center of Novel Software Technology and Industrialization.

References

[1] AppBrain. (2016, Sep) Number of available applications. [Online]. Available: http://www.appbrain.com/stats/number-of-android-apps

[2] Apktool. (2016, Sep) A tool for reverse engineering android apk files. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[3] dex2jar. (2016, Sep) Tools to work with android .dex and java .class files. [Online]. Available: https://sourceforge.net/projects/dex2jar/

[4] Soot. (2016, Sep) A framework for analyzing and transforming java and android applications. [Online]. Available: https://sable.github.io/soot/

[5] 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*. ACM, 2012, pp. 317–326.

[6] M. Ballano, "Android threats getting steamy," *[Online] February*, vol. 28, 2011.

[7] H. Tamada, M. Nakamura, A. Monden, and K.-i. Matsumoto, "Design and evaluation of birthmarks for detecting theft of java programs." in *IASTED Conf. on Software Engineering*, 2004, pp. 569–574.

[8] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *European Symposium on Research in Computer Security*. Springer, 2012, pp. 37–54.

[9] J. Crussell, C. Gibler, and H. Chen, "Scalable semantics-based detection of similar android applications," in *Proc. of Esorics*, vol. 13. Citeseer, 2013.

[10] 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 *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2012, pp. 62–81.

[11] 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*. ACM, 2013, pp. 185–196.

[12] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 175–186.

[13] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "Viewdroid: towards obfuscation-resilient mobile application repackaging detection," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. ACM, 2014, pp. 25–36.

[14] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 95–109.

[15] M. Dalla Preda and F. Maggi, "Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology," *Journal of Computer Virology and Hacking Techniques*, pp. 1–24, 2016.

[16] D. Kim, A. Gokhale, V. Ganapathy, and A. Srivastava, "Detecting plagiarized mobile apps using api birthmarks," *Automated Software Engineering*, pp. 1–28, 2015.

[17] C. Soh, H. B. K. Tan, Y. L. Arnatovich, and L. Wang, "Detecting clones in android applications through analyzing user interfaces," in *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 2015, pp. 163–173.

[18] A. Delelopers. (2016, Sep) Android applications fundamentals. [Online]. Available: https://developer.android.com/guide/components/fundamentals.html

[19] A. Delelopers. (2016, Sep) Android activities. [Online]. Available: https://developer.android.com/guide/components/activities.html

[20] A. Delelopers. (2016, Sep) Ui overview. [Online]. Available: https://developer.android.com/guide/topics/ui/overview.html

[21] A. Delelopers. (2016, Sep) Layouts. [Online]. Available: https://developer.android.com/guide/topics/ui/declaring-layout.html

[22] UiAutomator. (2016, Sep) Testing support library. [Online]. Available: https://developer.android.com/guide/components/activities.html

[23] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval research logistics quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[24] Wnadoujia. (2016, Dec) Wandoujia. [Online]. Available: http://www.wandoujia.com/apps

[25] F-Droid. (2016, Dec) F-droid. [Online]. Available: https://f-droid.org/repository/browse/

[26] Ijiami. (2016, Dec) Ijiami. [Online]. Available: http://www.ijiami.cn/

[27] R. Yu, "Android packers: facing the challenges, building solutions," in *Proceedings of the Virus Bulletin Conference (VB'14)*, 2014, pp. 266–275.

[28] F. Guo, P. Ferrie, and T.-C. Chiueh, "A study of the packer problem and its solutions," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2008, pp. 98–115.

[29] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 364–374.

[30] F. Thung, D. Lo, and L. Jiang, "Detecting similar applications with collaborative tagging," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 600–603.

[31] M. Linares-Vásquez, A. Holtzhauer, and D. Poshyvanyk, "On automatically detecting similar android apps," in *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 2016, pp. 1–10.