Towards a Scalable Resource-driven Approach for Detecting Repackaged Android Applications

Yuru Shao[†], Xiapu Luo[†‡*], Chenxiong Qian[†], Pengfei Zhu[†], and Lei Zhang[†]
Department of Computing, The Hong Kong Polytechnic University[†]
The Hong Kong Polytechnic University Shenzhen Research Institute[‡]
shaoyuru@gmail.com, {csxluo, cscqian, cspzhu,cslzhang}@comp.polyu.edu.hk

ABSTRACT

Repackaged Android applications (or simply apps) are one of the major sources of mobile malware and also an important cause of severe revenue loss to app developers. Although a number of solutions have been proposed to detect repackaged apps, the majority of them heavily rely on code analysis, thus suffering from two limitations: (1) poor scalability due to the billion opcode problem; (2) unreliability to code obfuscation/app hardening techniques. In this paper, we explore an alternative approach that exploits core resources, which have close relationships with codes, to detect repackaged apps. More precisely, we define new features for characterizing apps, investigate two kinds of algorithms for searching similar apps, and propose a two-stage methodology to speed up the detection. We realize our approach in a system named ResDroid and conduct large scale evaluation on it. The results show that ResDroid can identify repackaged apps efficiently and effectively even if they are protected by obfuscation or hardening systems.

1. INTRODUCTION

Repackaged apps have been one of the major sources of mobile malware on Android for many years [16,18]. A recent study showed that 86% malware samples were repackaged version of legitimate apps [48]. BitDefender even found that 1.2% of apps on Google Play have been repackaged to deliver ads and collect information [2]. Apps repackaging has also become a major threat to app economy [9]. By modifying the embedded ad's client ID or replacing it with new ad libraries, an attacker can make profits through apps developed by others [24]. As another example, repackaging paid apps and uploading the modified versions to third-party markets will result in revenue losses to developers. Moreover, repackaged financial apps not only cause financial loss to customers [31], but also compromise companies' reputation and users' experiences [9].

*The corresponding author

ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA

http://dx.doi.org/10.1145/2664243.2664275

Although a number of systems have been proposed to identify repackaged apps, how to effectively and efficiently detect them remains a challenging problem. One possible reason is that the majority of existing systems heavily rely on codes' features to quantify the similarity between apps, such as diverse hash values [1, 27, 47], abstract syntax trees (AST) [37], control flow graph (CFG) and its variants [13, 22, 43], program dependency graph (PDG) [19, 20], etc., thus suffering from two limitations:

Poor scalability due to the billion opcode problem. Hanna et al. estimated that the total amount of opcodes in all apps is around 1.45 billion [27] not to mention the rapid increase in the number of new apps. Therefore, processing millions of apps in many Android markets demands scalable and efficient solutions.

Unreliability to code obfuscation/app hardening techniques. Since most of existing solutions borrowed methods from the area of code clone detection that has been studied for 20 years [11, 39, 41], evasion solutions are available to dedicated attackers, let alone new obfuscation and hardening techniques [17, 29, 43, 44].

We explore an alternative scheme, a novel resource-driven approach, to detect repackaged apps. Our approach is motivated by two observations. First, Android apps usually contain various resources, such as layout and style resources for graphical user interface (GUI). Although these resources are separated from the executable `.dex` file, there are close relationships between resources and codes. Second, attackers seldom modify original resources in order to let the repackaged apps work properly and have the same look-and-feel. Moreover, existing obfuscation/hardening systems rarely handle resources. Although not all resources are critical to an app, some core resources cannot be easily modified by attackers without affecting the app's functionality or quality-of-experience (QoE). Therefore, we exploit core resources and the related codes to quantify apps' similarity.

To scale up this approach, we propose a two-stage methodology for grouping similar apps. More precisely, statistical features are used in the first stage to quickly divide apps into groups and then at the second stage structural features are employed to further cluster apps in each group. Note that the second stage can be performed in parallel. Since the methodology is general, we investigate the performance of two kinds of algorithms to search for similar apps: nearest neighbor search (NNS) [8] and clustering [7]. The former may quickly locate apps that are very similar to a target app but may miss other similar apps. The latter can partition apps into different clusters from a global view but has higher

computational complexity.

We have realized our approach in a system named ResDroid and conducted extensive evaluation on it. The experimental results not only validate its effectiveness and efficiency but also reveal interesting observations. In summary, this paper makes the following contributions:

1. We propose a novel resource-driven approach to detect repackaged apps. To our best knowledge, it is the first systematic examination on leveraging resources for repackaged apps detection.

2. We propose a two-stage methodology to scale up the approach and investigate the performance of two kinds of algorithms for searching similar apps: nearest neighbor searching and clustering.

3. We realize our approach in a system named ResDroid with 2770 lines of Python code, 1157 lines of Java code, and 309 lines of C code. Although a simultaneous research, ViewDroid [45], also adopts GUI-related features, there are significant differences between it and ResDroid in terms of feature selection, comparison algorithms, and scalability, as detailed in Section 7.

4. We conduct the first study on the effect of commercial app hardening systems on detecting repackaged apps. We also develop DexDumper for dynamically dumping hardened apps from memory for detection.

5. We conduct extensive evaluation on ResDroid with 169,352 apps crawled from 10 markets and 200 real repackaged apps. The results show that ResDroid can detect repackaged apps effectively and efficiently.

The rest of this paper is organized as follows. Section 2 introduces the problem and background knowledge. Section 3 and Section 4 detail our methodology and the implementation of ResDroid, respectively. The experimental results are reported in Section 5. Section 6 discusses ResDroid's limitations and future work. After introducing the related work in Section 7, we conclude the paper in Section 8.

```
1  MainActivity.java:
2  public class MainActivity extends Activity {
3      protected void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          setContentView(R.layout.main_act_layout);
6          ...
7      }
8  }
9
10 TextFragment.java:
11 public class TextFragment extends Fragment {
12     public View onCreateView(LayoutInflater inflater,
           ViewGroup container, Bundle ...) {
13
14         View view = inflater.inflate(
15             R.layout.text_fragment, container, false);
16
17         return view;
18     }
19 }
20
21 MyEditText.java:
22 public class MyEditText extends EditText {
23     ...
24 }
25
26 main_act_layout.xml:
27 <RelativeLayout ... >
28     <fragment
29         android:id="@+id/text_fragment"
30         android:name="com.example.TextFragment" />
31 </RelativeLayout>
32
33 text_fragment.xml:
34 <RelativeLayout ... >
35     <include layout="@layout/titlebar" />
36     <TextView
37         android:id="@+id/text_view1"
38         android:text="@string/tv_text"
39         ... />
40     <com.example.MyEditText
41         android:id="@+id/edit_text"
42         ... />
43 </RelativeLayout>
44
45 my_editor.xml:
46 <view
47     class="com.example.MyEditText"
48     ... />
```

Figure 1: Example of the interaction between app code and resources.

2. BACKGROUND

2.1 Problem Statement

The majority of existing approaches heavily depend on code-level features, thus suffering from two limitations: (1) poor scalability to process billions of opcodes; (2) unreliability to code obfuscation/app hardening techniques. The goal of this paper is to explore an alternative scheme, a resource-driven approach, to detect repackaged apps. Motivated by the observation that existing attacks and code obfuscation/app hardening techniques seldom handle resources, we investigate how to employ resources to detect repackaged apps from four aspects, including, feature selection, feature extraction, scalable approaches for searching similar apps, and limitations. Note that the new approach complements the existing code-level systems instead of replacing them.

2.2 Application Resources

Android developers are recommended to externalize resources from the codes so that they can be maintained independently [6]. Figure 1 shows an example of app codes and resources. `MainActivity`.java defines an activity that provides users a GUI for interaction. When an activity is launched, the Android runtime creates an activity object and invokes the `onCreate` method defined at lines 3-7.

An activity may contain other GUI components, which can be added to its `View` dynamically or defined by a layout file (in XML format). Lines 27-31 define the layout used by `MainActivity`. Line 5 indicates that `MainActivity` loads its view from main_act_layout.xml.

`Fragment` is a special component that represents a behavior or a portion of user interface within an activity. It can be considered as a modular section of an activity, which has its own layout. main_act_layout.xml defines a fragment associated with the class com.example.TextFramgment, whose content is shown in lines 10-19. When `MainActivity` is started, its layout file main_act_layout.xml will be loaded and the activity will present users the fragment, whose layout is defined in text_fragment.xml and loaded in line 14 via the `inflate` method. In this example, the final GUI presented to users is actually the view of the fragment defined in text_fragment.xml.

The GUI component text_fragment (lines 40-42) is a customized component com.example.MyEditText. It is defined in my_editor.xml (lines 46-48), and the corresponding codes are in com.example.MyEditText.java, as specified in line 47.

When interacting with an app, users can navigate between (i.e., transition between) different activities [10]. We define the transition among activities as *activity transition graph* (ATG), where each vertex represents an activity and an edge indicates the existence of transitions through Android methods startActivity() or startActivityForResult().

App resources are referenced through IDs. For example, after main_act_layout.xml is parsed, an ID will be signed to it. The auto-generated file `R.java` records all resources and their IDs. App codes access the resources through their IDs. For example, `MainActivity` sets its view defined in main_act_layout.xml by referencing `R.layout.main_act_layout`. Moreover, developers can use the scheme `@type/name` to reference other resources. For instance, line 38 references a string resource named "tv_text" using `@string/tv_text`.

2.3 Event Handlers

There are two kinds of event handlers in Android apps.

GUI event handlers. GUI objects can be associated with event handlers. For example, given a button, the execution will reach its `onClick` function, defined in the interface android.view.View.OnClickListener, after a user clicks it.

Lifecycle event handers. An activity instance may transition among different states in its lifecycle [5]. Developers can define how an activity behaves when it transitions from one state to another in callback methods. For example, when an activity is started, its `onCreate` and `onStart` callback methods will be invoked successively.

For apps without GUI/activities, we consider callbacks in their Services and Broadcast Receivers as event handlers.

3. METHODOLOGY

3.1 Overview

Figure. 2 depicts our resource-driven solution for detecting repackaged apps, which is realized in ResDroid and the implementation is detailed in Section 4.

The feature extraction module first identifies an app's major packages according to their importance measured by the PageRank algorithm [34] (Section 3.3). Then, core resources along with their statistical features and structural features will be determined according to major packages and the app's manifest file. The statistical features (Section 3.4) are lightweight in terms of computation and comparison but may not provide precise information about an app. In contrast, structural features (Section 3.5) can better characterize an app at the cost of the complexity of computation and comparison.

To scale up the detection for millions of apps, we propose a two-stage methodology that employs the divide-and-conquer strategy to identify similar apps within small groups of apps. More precisely, ResDroid first uses statistical features to divide apps into small classes in the coarse-grained processing module and then employs structural features to identify similar apps within each group in the fine-grained processing module. The output is a set of *potential repackaging groups* (PR-Groups) containing suspicious repackaged apps. Our approach is rational because repackaged apps are usually similar to original apps and the process of clustering apps in different groups can be parallelled.

Finally, ResDroid verifies whether those similar apps are repackaged apps according to their signatures. Since the percentage of suspicious apps is usually small, they could be further inspected by in-depth code analysis systems [19], dynamic analysis systems [23, 38], malware detection systems [25], or even manual verification if necessary.

Based on the selected features, our solution can adopt different algorithms to find similar apps. Here, we examine the performance of two kinds of popular algorithms: clustering algorithms and nearest neighbor search (NNS) algorithms in Section 3.6 and Section 3.7, respectively.

3.2 Hardening Detection

To secure Android apps, hardening techniques and services have emerged [3] (e.g., Bangcle[1], iJiaMi[2]). Typically, they encrypt an app's `classes.dex` and load it into memory through java native interfaces (JNI). Hardening not only raises the bar for attackers to repackage apps, but also thwarts ResDroid to extract apps' features. To tackle this problem, we design and implement DexDumper (Section 4.1) to dynamically restore the `classes.dex`.

Before extracting features, ResDroid will check whether the app is hardened or not by looking for patterns of hardening services. For example, since Bangcle inserts a shared library `libsecexe.so` into hardened apps, an app containing such file is considered as a hardened app. Moreover, ResDroid will invoke DexDumper to dump the original `classes.dex` from a hardened app for feature extraction.

3.3 Major Packages and Core Resources

Not all resources are critical to an app and/or have close relationship with major codes. To raise the bar for an attacker to evade the detection by modifying features, we define core resources, which are used by major packages, with the following requirements: (1) it should be difficult for an attacker to manipulate these resources. In particular, random manipulations by an attacker will impair an app's functionality and/or QoE, or such manipulations can be easily filtered out; (2) they are representative.

Major packages refer to important codes in an app excluding imported libraries. We created a blacklist to filter out frequently-used ad and third-party libraries. Motivated by the module decoupling technique in [46], ResDroid first constructs a package dependency graph, an undirected and weighted graph, where each vertex represents a package, and an edge between two vertexes indicates the existence of method invocations between them. The weight of an edge is increased by one if it spots a method invocation between the two packages. Then, ResDroid ranks the packages using the PageRank algorithm [34] and selects the top 5 packages as major packages.

3.4 Statistical Features

3.4.1 Definition

We define 15 statistical features, which can be easily retrieved, and use them in the coarse-gained processing module. Let $\mathcal{A} = \{a^{(i)}\}_{i=1...N}$ be a set of apps. The statistical features of each sample $a^{(i)}$ is represented as a vector

[1] `http://www.bangcle.com`
[2] `http://www.ijiami.cn`

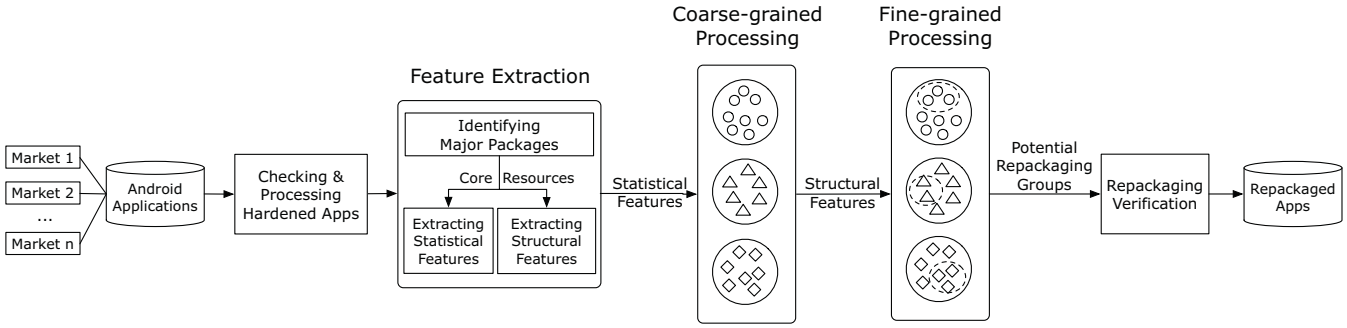


Figure 2: The procedure of our resource-driven approach.

$v^{(i)} = (f_1^{(i)}, ..., f_{15}^{(i)})$. The first five dimensions $(f_1^{(i)}, ..., f_5^{(i)})$ include (1) number of activities; (2) number of permissions, both system-defined and custom ones are included; (3) number of intent filters; (4) average number of `.png` files per `drawable*` directory; and (5) average number of `.xml` files per directory in `res`. The next 10 dimensions $(f_6^{(i)}, ..., f_{15}^{(i)})$ include the average number of references to the 10 most-referenced resources. Table 1 lists the top resources in 24,810 randomly selected apps. More features, such as number of services, can be added to better profile an app without dramatically increasing the complexity.

Note that we use the average number of `.png` and `.xml` files instead of the total amount, because they are more representative. For example, apps that support multiple display resolutions will have several set of drawables. For example, `drawable-hdpi` contains bitmap drawables for high screen densities while `drawable-ldpi` has objects for low-density displays. As another example, files in `values-zh` and `values-fr` provide Chinese and French supports, respectively. If an app is repackaged with extra language support, the total number of certain kinds of resources will increase but the average values will not be affected.

#	Type	Total	Average
1	id	6,631,598	267
2	drawable	3,616,963	146
3	string	1,964,378	79
4	color	1,024,896	41
5	style	823,848	33
6	dimen	623,906	25
7	layout	248,097	10
8	xml	136,730	6
9	integer	88,130	4
10	array	76,670	3

Table 1: 10 most referenced resource types in 24,810 apps randomly selected from our dataset.

3.4.2 Comparison

Since the features' ranges are quite different, we normalize $v^{(i)}$ and get a new vector $v_n^{(i)} = (F_1^{(i)}, ..., F_{15}^{(i)})$, where each dimension ranges in $[0, 1]$. We calculate $F_j^{(i)}$ using the following function:

$$F_j^{(i)} = norm(f_j^{(i)}) = \sqrt{\frac{f_j^{(i)} - min(f_j^{(1...N)})}{max(f_j^{(1...N)}) - min(f_j^{(1...N)})}} \quad (1)$$

Note that $min(f_j^{(1...N)})$ and $max(f_j^{(1...N)})$ are the minimal and maximal values of the jth feature for all apps in $\mathcal{A}$. The similarity between two apps $a^{(k)}$ and $a^{(h)}$ according to the statistical features is defined as:

$$s^{(k,h)} = e^{-D_c(v_n^{(k)}, v_n^{(h)})}, \quad (2)$$

where $D_c(v_n^{(k)}, v_n^{(h)})$ is the *Euclidean distance* of the two normalized feature vectors.

3.5 Structural Features

The structural features cover two types of information: (1) activity layout; (2) event handler. These features are reliable and representative because: (1) repackaged apps usually have the same GUI as the original apps to avoid affecting their functionality and QoE; (2) although an dedicated attacker can re-implement an existing layout, it takes time to achieve the completely same appearance and it is difficult to apply this process to all apps automatically; (3) since GUI objects and the associated *event handlers* determine the functionality of an app, attackers usually keep existing event handlers to avoid impairing the original app's functionality. Therefore, ResDroid compares two apps' event handlers instead of all codes.

As an app may have multiple activity layouts and different kinds of event handlers, features extracted from a single layout or event handlers should be well organized to form the final structural features. To achieve this goal, for each app $a^{(i)} \in \mathcal{A}$, we first perform static analysis to construct its ATG and then traverse the ATG using depth-first search algorithm starting from the main activity of $a^{(i)}$. If an app has no main activity, we choose the first activity defined in its `AndroidManifest.xml`. We remove the activities that do not belong to core resources from the generated activity sequence $AS(a^{(i)})$. The following layout features (Section 3.5.1) and event handler features (Section 3.5.2) are arranged with the order defined in $AS(a^{(i)})$.

3.5.1 Activity Layout Feature

For an activity *act*, we denote its layout as $L(act)$. We traverse $L(act)$ using the pre-order traversing algorithm and obtain an element sequence $\langle e_1, ..., e_m \rangle$, where e_i $(i = 1, ..., m)$

stands for a GUI object, such as `Button`, `EditText`. We define a function α for mapping e_i to an English letter ('a' to 'z', 'A' to 'Z') according to its type. For example, `Button` is labelled as 'b', `EditText` as 'e', and `View` as 'v', etc. Since customized components are usually derived from existing components and their names may be obfuscated, we use their ancestors' type to label them. For instance, a `PhotoView` extending `View` is also labeled as 'v'.

Following this rule, $L(act)$ is converted into a sequence $LS(act) = \langle\alpha(e_1), ..., \alpha(e_m)\rangle$. For an app $a^{(i)} \in \mathcal{A}$ and $AS(a^{(i)}) = \langle act_1, ..., act_n\rangle$, its layout feature will be:

$$LF(a^{(i)}) = \langle LS(act_1), ..., LS(act_n)\rangle \quad (3)$$

3.5.2 Event Handler Feature

Event handler feature refers to the corresponding callback methods. We perform static analysis on each event handler and extract their *method signatures* [21]. Let $\sigma(m)$ denote the signature of an event handler m. Since an activity act may have several event handlers, we represent its event handler feature as a sequence that consists of method signatures in *lexicographical order*: $EH(act) = \langle\sigma(m_1), ..., \sigma(m_k)\rangle$. Given an app $a^{(i)} \in \mathcal{A}$ and $AS(a^{(i)}) = \langle act_1, ..., act_n\rangle$, its event handler feature $EF(a^{(i)})$ is:

$$EF(a^{(i)}) = \langle EH(act_1), ..., EH(act_n)\rangle \quad (4)$$

3.5.3 Comparison

When quantifying the similarity between two apps $a^{(k)}$ and $a^{(h)}$ using the structure features, we define the distance between $a^{(k)}$ and $a^{(h)}$ as:

$$\begin{aligned} D_f(a^{(k)}, a^{(h)}) = w_l * d_s\Big(LF(a^{(k)}), LF(a^{(h)})\Big) \\ + w_e * d_s\Big(EF(a^{(k)}), EF(a^{(h)})\Big), \end{aligned} \quad (5)$$

where w_l and w_e are pre-defined weights (the actual values assigned to w_l and w_e are discussed in Section 5). $d_s(s_1, s_2)$ measures the distance between two sequence s_1 and s_2, which is defined as:

$$d_s(s_1, s_2) = 1 - \frac{length(LCS(s_1, s_2))}{min(length(s_1),\ length(s_2))}, \quad (6)$$

where $LCS(s_1, s_2)$ is the *longest common sequence* of s_1 and s_2, and $length(s)$ is the length of sequence s. The advantage of LCS is that even if noise is inserted into one sequence the final output of d_s may remain unchanged. Therefore, even though an attacker may insert junk resources to change an app's structural features, ResDroid will not be affected.

3.6 Clustering-based Processing

We adopt the *spectral clustering* [7] algorithm to cluster apps according to the normalized statistical features. Spectral clustering techniques leverage the spectrum (eigenvalues) of the similarity matrix of the input data to perform dimensionality reduction before performing the clustering. It allows us to add more features for further improving the performance in future work.

We apply the complete-linkage *hierarchical clustering* algorithm and the DB cluster validity index [26] to split each coarse-grained cluster into fine-grained clusters (i.e., PR-Groups). In hierarchical clustering, the linkage criterion determines the distance between sets of observations. We chose the complete-linkage criterion because it usually results in compact clusters with small diameters. Although it may be sensitive to outliers, we can filter out outliers before conducting the hierarchical clustering.

3.7 NNS-based Processing

Clustering algorithms may compare every pair of apps, thus leading to high computation complexity. NNS algorithms allow us to only compare apps that are likely to be similar and therefore dramatically reduce the comparisons. When applying NNS algorithms, we adopt the k-d tree technique [35]. In the coarse-grained processing, we select candidate app pairs with the help of k-d tree. In the fine-grained processing we compare apps in pair to obtain PR-Groups.

3.7.1 Selecting Candidate Pairs

We first build a k-d tree and insert statistical feature vectors $v^{(i)}$ $(i = 1, ..., N)$ to this tree ($k = 15$ because of the 15 dimensions). Since different apps may have identical statistical features, we cannot insert both of them into the k-d tree for the sake of avoiding duplicated nodes. For example, given two apps $a^{(i)}$ and $a^{(j)}$ with the same feature vectors (i.e., $v^{(i)}$ and $v^{(j)}$), if $a^{(i)}$ is in the k-d tree, we cannot insert $a^{(j)}$. To solve this problem, we sort the statistical feature vectors of all apps according to the *lexicographical order* and then group apps having identical feature vectors. As shown in Figure 3, $v^{(5635)}$ and $v^{(9440)}$ are identical, and therefore we group $a^{(5635)}$ and $a^{(9440)}$ into the same cluster c_n. Then, we insert clusters containing one or more apps into the k-d tree, where each node represents a cluster. When we query n nearest neighbors for each cluster C_i, we regard C_i and its neighbors as candidate cluster pairs. For two clusters C_i and C_j, the candidate app pairs include: (1) app pairs in C_i; (2) app pairs in C_j; and (3) app pairs between C_i and C_j.

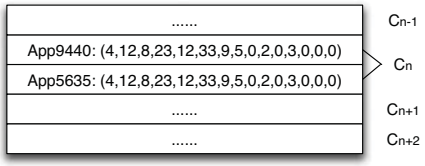


Figure 3: Statistical feature vectors are sorted in Lexicographical order. Apps that have identical feature vectors are grouped in the same cluster.

3.7.2 Pair-wise Comparison

The distance between two apps $a^{(k)}$ and $a^{(h)}$ is measured by D_f in Eqn.(5). Given a threshold θ, if $D_f(a^{(k)}, a^{(h)})$ is smaller than θ, they are in the same PR-Group.

3.8 Repackaging Verification

Apps in PR-Groups have similar appearance and functionalities. However, since developers may create a bunch of similar apps that cannot be considered as repackaged apps, we perform a verification on apps in each RP-Groups. More precisely, we extract developer certificates from apps and distinguish each certificate with their MD5 checksum. If all apps in a PR-Group share the same certificate, they are not repackaged. Otherwise, apps repackaging is detected.

4. IMPLEMENTATION

We have implemented Resdroid with 2770 lines of Python code, 1157 lines of Java code, and 309 lines of C code.

4.1 DexDumper

To handle hardened apps whose original `classes.dex` is encrypted or hidden in shared libraries, we design and implement *DexDumper* that dumps the original `classes.dex` of a hardened app from memory during runtime. It is motivated by the observation that the Dalvik virtual machine (DVM) cannot run encrypted `classes.dex` and therefore the original `classes.dex` will be restored before being executed.

DexDumper first invokes the `ptrace` system call to attach to the process of a running hardened app, and then reads the app's memory and searchs for the dex files. As the memory space of a process is very large, DexDumper manages to narrow the searching scope to efficiently locate the target dex code. More precisely, it reads the process mapping file of the hardened app, namely `/proc/PID/maps` (`PID` stands for the app's process ID), where we can get the start address, the end address, and the attributes of each memory piece. Note that the memory where dex files are mapped into has attributes "r-xp", meaning that the area is readable (r), executable (x) and private (p). From memory pieces with such attributes, we can dump a set of `dex` format files since some pre-loaded runtime libraries (e.g., `ext.jar@classes.dex`) are also located.

For each dumped dex files, DexDumper looks for the app's package name (defined in the app's manifest) in their string constants pool. If found, it is considered as the original `classes.dex`. Since hardening will not alter the app's resources, ResDroid takes in the dumped dex file and the resources from the hardened apps for further process.

4.2 Feature Extraction

We employ *apktool* to decompile apps. Apktool can restore an app's resources and translate its `dex` code into `smali` format simultaneously. However, we found that apktool may crash when processing some apps. For such apps, we use *aapt* and *baksmali* to obtain their resources and smali code separately. To construct ATG, we use A3E [10] to perform static analysis on apps. It took around 22 hours to generate ATGs for all apps in our data set. Then we run PageRank to pick 5 major packages, which took another 3 hours. In total, the extraction of statistical features consumed 25 hours. Such a long period is not unexpected because both A3E and the PageRank algorithm are time-consuming. It is acceptable as we only need to do this computation once.

When extracting layout features of apps, we observed that 93.7% apps define their layouts in XML under the `res/layout` directory. Therefore, we can obtain their layout structures by traversing the XML files. Since some apps generate their layouts dynamically, we use GATOR (v1.0) [4, 40] to process them, which can construct an app's layouts from the codes by conducting static reference analysis for GUI objects. We further enhanced GATOR from two aspects. First, it does not handle certain GUI components (i.e., fragments and dialogs). As fragment is widely used to realize components reuse in different layouts, we added functions to handle it. Second, as GATOR only takes in source codes, we empowered it to accept dex files.

Similar to activities, layouts of fragments can be defined in static XML files or dynamically created at runtime. Therefore, we first identify fragment objects in activities, and then examine the fragment and check whether its layout is defined in resource file. If so, we directly parse the XML file to obtain its structure. Otherwise, we construct its layout structure by re-using the code logic designed for processing dynamically-generated activity layouts. The extraction of structural features took 49.7 hours.

4.3 Clustering and NNS

In coarse-grained clustering, we use a parallel implementation of spectral clustering [14], which can effectively handle large-scale data. Since the last step of spectral clustering is actually running *k-means*, the number of clusters should be specified. We use *x-means* [36] to estimate a proper number, instead of choosing it arbitrarily.

5. EXPERIMENTS

Our data set contains 169,352 apps crawled from 10 Android markets, including the offical market Google Play and 9 other third-party markets. Our experiments were conducted on a PC running Ubuntu Linux 12.04LTS with an 8-core Intel i7 3.50GHz CPU and 32GB memory.

5.1 Ground Truth

We use 200 pairs (400 apps) of repackaged and original apps as ground truth to evaluate ResDroid. All the repackaged apps are real malware or adware (121 from SandDroid[3] and other 79 from ContagioMobile[4]). The original apps were downloaded from Google Play. There are chances that different versions of the same app present dissimilar GUIs. Therefore, to make the ground truth more reliable, we only select original apps that have the nearest (or the same) version codes with the repackaged ones.

5.2 Clustering-based Approach

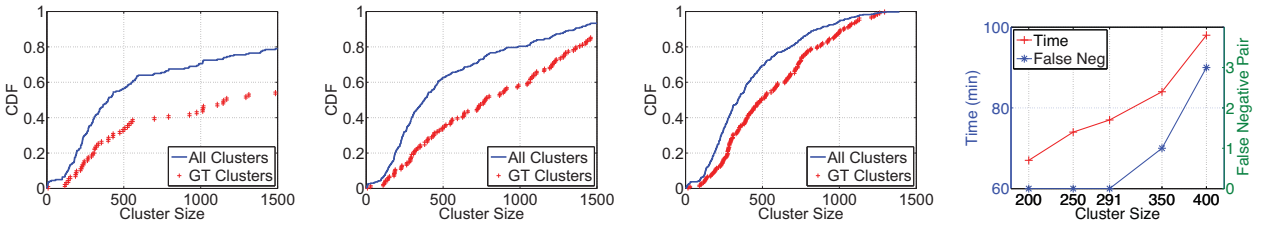
During the coarse-grained processing, the x-means algorithm [36] was used to estimate the number of clusters (denoted as C). The recommended value was 291. We also tried other values close to 291 (i.e., 200, 250, 350, and 400) when performing the spectral clustering. Figure 4 shows the CDFs of the sizes of clusters. With the increment of C, the average cluster size decreases. Figure 4(b) shows that when $C = 291$ nearly 10% clusters (around 20% groud truth clusters) contain more than 1,500 apps. As shown in Figure 4(c), when $C = 400$, all clusters contain less than 1,500 apps.

It is obvious that the smaller a cluster is, the more quickly (and hopefully more accurate) the hierarchy clustering will be finished. Furthermore, although we need to perform more clustering, it is easy to parallelize the tasks as they are independent. So a large C may be expected. However, as shown in Figure 4(d), more false negatives appear along with the increment of C. The false negative is 0 when $C = 291$, but it increases to 3 when $C = 400$. Moreover, a larger C requires longer time to perform the spectral clustering. Considering both cluster sizes and false negative rate, we finally chose 291 as the number of clusters in coarse-grained clustering.

We apply hierarchical clustering to each cluster produced by coarse-grained clustering to generate PR-Groups. In hierarchical clustering, we set the cutoff value to 0.1. That

[3] `http://sanddroid.xjtu.edu.cn`

[4] `http://contagiominidump.blogspot.com`

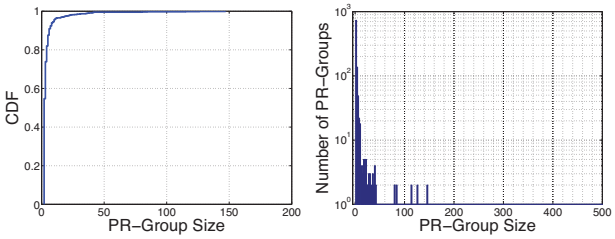


(a) $C = 200$ (b) $C = 291$ (c) $C = 400$ (d) Consumed time and false negatives with different C

Figure 4: CDFs of sizes of clusters produced by spectral (coarse-grained) clustering with different numbers of cluters

means, if two clusters have a distance less than or equal to 0.1, they will be merged into the same PR-Group.

1,605 PR-Groups are obtained and they include 6,906 apps in total. Figure 5 shows the distribution of the sizes of PR-Groups. Over 98% of PR-Groups have sizes smaller than 50. The average size of PR-Groups is 4.03, which is small enough for manually checking if need. Among all of our ground truth pairs, the repackaged app and the original app fell into the same PR-Groups, meaning that the false negative rate is 0.



(a) CDF (b) Histogram

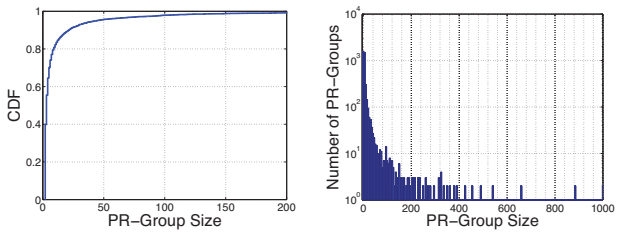
Figure 5: Distribution of the sizes of PR-Groups that are produced by hierarchical clustering with cutoff value 0.1.

5.3 NNS-based Approach

We first build a k-d tree according to statistical features and query nearest neighbors of each tree node for selecting candidate app pairs. More precisely, we set the number of neighbors (n) to 10. Then, we compare their structural features and calculate their distance following Eqn.(5). Note that w_l and w_e can be adjusted to support different criteria. For example, if $w_l > w_e$, more emphasis is paid to the layout features. Otherwise, the event handler features may be regarded as more important. In our experiments, $w_l = w_e = 0.5$. The distance threshold θ was set to 0.15, meaning that if the distance between two apps is smaller than 0.15 they will be classified into the same PR-Group.

The total number of PR-Groups produced by NNS-based approach is 2,070, including 20,867 apps. The average size of PR-Groups is 9.9. Figure 6 illustrates the distribution of the sizes of PR-Groups. Similar to the clustering-based approach, most of PR-Groups (over 90%) are smaller than 50. However, there are several PR-Groups whose sizes are larger than 500 and the largest size is 997. Only two pairs of repackaged and original apps are not grouped into the same PR-Groups, and therefore the false negative rate of NNS-based approach is 1%.

5.4 Accuracy



(a) CDF (b) Histogram

Figure 6: Distribution of the sizes of PR-Groups when NNS-based approach is employed, with distance threshold 0.15.

Section 5.2 and Section 5.3 show that both clustering-based approach and NNS-based approach can effectively detect repackaged apps with very low false negative. To evaluate the false postive rate of our approaches, we randomly selected 100 groups from the PR-Groups produced by clustering-based approach, and manually checked their resources and `smali` codes. We also executed these apps in Android emulator to check whether they have similar GUI and functionality. We did the same thing to the PR-Groups produced by NNS-based approach.

In the results of clustering-based approach, we found three PR-Groups that contain false positives. So the false positive rate this approach is 3%. But not surprisingly, the sizes of these three PR-Groups (i.e., 80, 84 and 127) are larger than the average value. Our analysis reveals two major reasons for these false positives. First, those apps have only a few activities and their simple functionalities lead to similar statistical features and structural features. Second, apps created by some online tools (e.g., App Makr[5]) may use the same template, and therefore these apps have similar appearance and event handlers. We discuss how to handle them in Section 6.

In the results of NNS-based approach, we found 5 PR-Groups containing false positives, and hence the false positive rate is 5%. Similar to the results of clustering-based approach, these false positives were all found from groups with large size (i.e., 520, 548, 641, 885 and 991). It shows that NNS-based approach is not as accurate as clustering-based approach. The reason may be that when querying nearest neighbors, k-d tree (or other NNS algorithms) only considers most similar ones "locally" whereas in clustering-based approach each sample will be compared with others (i.e., construct a global view). It demonstrates a tradeoff be-

[5] `http://www.appmakr.com`

tween efficiency and effectiveness. That is, clustering-based approach suffers from high computational complexity but is more accurate. In contrast, NNS-based approach is much more efficient but less accurate.

5.5 Complexity

Since the time complexity of both spectral clustering and hierarchical clustering are $O(N^3)$, the overall complexity of clutering-based approach is $O(N^3)$. Although the computational cost of clustering-based approach is high, we may leverage parallel computing to speed up the process. For NNS-based approach, we sort feature vectors in lexicographical order before building k-d tree. This step can be accomplished within $O(NlogN)$. Moreover, the complexity of building k-d tree and that of querying nearest neighbors are also $O(NlogN)$. Therefore, the overall time complexity of NNS-based solution is $O(NlogN)$.

5.6 Code Obfuscation/App Hardening

We implement DexDumper to extract the original `classes.dex` from apps protected by typical hardening systems. Therefore, attackers could not employ such hardening techniques to evade ResDroid.

We also tested ResDroid's ability to confront obfuscation techniques. We first employed Proguard to obfuscate apps from source codes. The results showed that both statistical features and structural features were not affected. Then we used SandMark[6] to generate obfuscated code from bytecode. However, since SandMark does not support Android's dex format, we converted dex files into jar files through dex2jar and then fed jar files to SandMark. Unfortunately, although SandMark provides 39 kinds of obfuscation methods, only three of them (static method bodies, method merger and class encrypter) could successfully process these jar files. After converting the obfuscated jar files back to dex through `dx`, we found that *static method boides* and *method merger* had no influence on ResDroid. However, ResDroid failed to extract structural features when the jar was encrypted by the class encrypter, and we propose possible solutions in Section 6.

5.7 Observations

Apart from the ground truth, ResDroid identified 64 repackaged apps. We examined some of them manually and report the observations.

5.7.1 App Plagiarism

We found that some plagiarizers repackaged apps and republished these apps as their own. For example, an app (package name: com.bluedog1893.android.translate) has identical structural features with another app (package name: com.dollars.translate). However, they have completely different icons. After manual examination, we found that the former one contains most of the resources and the whole codes from the latter one, but the string resources have been converted from English to Chinese. Moreover, while the original app does not have ads, the plagiarizer added ad libraries to make profits.

5.7.2 Massively-produced Apps

We found a set of apps from the same developer having the same structural features. Figure 7 shows three of them.

[6] `http://sandmark.cs.arizona.edu`

(a) App 1 (b) App 2 (c) App 3

Figure 7: Three apps have exactly the same layout

To create these apps, the developer just wrote the codes once and then applied the same codes to different resources for quickly producing "new" apps.

Another example comes from a PR-Group where the package names of all 83 apps follow the pattern: com.lvping.mobile.cityguide.*. These apps provide guidance for travellers, and the last piece of their package names represents the city's name. For example, the app with package name com.lvping.mobile.cityguide.sydney236 offers a travelling guide for Sydeny. However, these 83 apps were signed with two different certificates. Specifically, 4 of them are signed by the certificate whose MD5 fingerprint is '246DA3F3F52830A9E3FD04111BA4C1D4', while other 79 apps are signed by the certificate whose MD5 is '76439FA93B09D3FA51874769C74486AB'. We examined the owners and issuers of the two certificates. The first one is "Android Debug" used for signing debugging version of apps. The second one is the company's domain (i.e., `lvping.com`). We are not sure whether the ones signed by the debugging certificate are repackaged apps created by attackers or it is just because the developer forgot to export release version and sign them using official certificates.

5.7.3 Misusing Certificates

We found a pair of repackaged apps that have the same package name (i.e., `a5game.leidian2`) but different certificate fingerprints. One was signed by 'EE1C7585428F65BAC2D156B0792D2358', and the other app was signed by '86544D775DCBA00275CD304C5C37BCC7'. We investigated the two certificates and found the former is owned by "5agame.com", but the latter's owner is in Chinese characters, which is the name of the website `http://www.5agame.com`. After careful examination of their codes, we did not find anything abnormal. It is likely that both the two apps were actually published by the same developer "5agame.com", but they chose different certificates for the same app for some unknown reasons. This may result in failures of updates, because the Android system does not allow the newer app and the older one have different certificates. Therefore, it is an example of misusing app certificates.

6. DISCUSSION

6.1 Attack Analysis

Zhang et al. described the behaviors of three kinds of repackaging attacks [45], including (1) lazy attacks that use automatic code obfuscation tools to repackage apps without modifying the functionality of original apps; (2) malware that embeds malicious payload into original apps without

changing them; (3) amateur attacks that may make some changes to original apps besides employing the automatic code obfuscation tools. Since none of them will modify the resources and the related codes, ResDroid can detect them.

Advanced attackers who know ResDroid may change features to evade detection. For example, they can insert many junk resources into the repackaged app for affecting the statistical features. However, we consider the relations between resources and codes (e.g., references from XML files, loading from code) and identify core resources, from which statistical features are extracted. Therefore, junk resources that are not carefully crafted will be filtered out. Dedicated attackers may change statistical features by inserting resources and altering the dex file simultaneously at a cost of increased app size and degraded performance. We may apply dead-code detection techniques [12] to identify such junk functions and remove them along with the corresponding resources when computing the features.

As explained in Section 3.5, it is difficult for attackers to re-implement a new layout while keeping the same looks and feels. Moreover, to retain the normal functionality and QoE of the repackaged app, attackers will not remove event handlers. Although attackers can add new event handlers, they cannot modified the result of normalized LCS as explained in Section 3.5.3. Therefore, the structural features are robust and can raise the bar for attackers to evade detection.

6.2 Limitations and Future Work

We notice that apps created by automatic building tools will cause false positives, because these tools provide developers a set of templates to create apps. Apps using the same template will shared similar appearance and the event handlers, therefore, their statistical features and structural feature are alike. ResDroid cannot differentiate them. In future work, we will use components' attributes to differentiate them. Another possible approach is to use code-level detection systems such as DNADroid [19] to handle them.

If code obfuscation/app hardening systems employ various dynamic loading techniques and encryption methods to prevent static analysis, ResDroid may not be able to handle such apps through static analysis. However, since these techniques will eventually load codes and resources into memory, we will design a kernel-based dynamic approach like [42], which keeps monitoring the behavior of a hardened app and dumps selected codes and resources after they are loaded. Moreover, we will examine how to fingerprint apps protected by different code obfuscation/app hardening systems.

7. RELATED WORK

Code clone detection. Considerable research has been conducted on code clone detection [11, 39, 41]. Existing approaches can be roughly classified into four groups [41]: (1) textual analysis that extracts fingerprints from code directly; (2) lexical analysis that converts codes into lexical tokens and then detects duplicated token sequences (e.g., CP-Miner [32], etc.); (3) syntactic analysis that first turns codes into abstract syntax trees (AST) and then applies tree matching or structural metrics to detect clones (e.g., Deckard [30], etc.); (4) semantic analysis that employs static program analysis to extract more precise information about the code, such as program dependency graph, for detection (e.g., GPLAG [33], etc.). Some of these methods have been used to detect repackaged apps by analyzing dex files, or the converted Java class files, or the disassembled smali codes.

Repackaged apps detection. Assuming apps from the official market are original, DroidMOSS applies fuzzy hashing to each app's opcodes and then compares it to original apps' fingerprint for detecting repackaged apps [47]. Similarly, Androguard [1] uses several standard similarity metrics to hash methods and basic blocks for comparison. Juxtapp characterizes apps through k-grams of opcodes and feature hashing and then clusters the corresponding bitvectors to identify app clones [27]. PiggyApp was designed to detect piggybacked apps, a special kind of repackaged apps, which contain injected code [46]. It first decouples modules according to their dependency relationship and then construct fingerprint for the primary module by collecting various features, such as requested permissions, Android API calls used, etc. [46]. These methods are vulnerable to simple obfuscation techniques because they consider few semantic information about codes [29, 43, 44].

Dresnos used normalized compression distance (NCD) [15] to compare the similarity of apps according to their method signatures, including external API used, exceptions, and control flow graph (CFG) [22]. Potharaju et al. proposed an approach to detect plagiarized apps according to symbol tables and method-level AST fingerprints. This approach can handle two kinds of obfuscation techniques that mangles symbol table or inserts random methods with no functionality [37]. DroidSim utilizes component-based control flow graph (CB-CFG) to quantify the similarity between apps [43]. DNADroid constructs a program dependency graph (PDG) for each method and then performs subgraph isomorphism comparison on PDGs after filtering out unnecessary methods [19]. To speed up DNADroid, AnDarwin splits PDGs into connected components (i.e., semantic blocks), each of which will be represented by a semantic vector containing the number of specific types. After that, it employs locality sensitive hashing (LSH) to identify code clones that have similar semantic vectors [20]. Chen et al. proposed a novel approach that uses the centroid of control dependency graph to measure the similarity between methods for detecting cross-market app clones [13]. Although these methods are better than the previous ones, they could still be easily evaded by obfuscation techniques (e.g., inserting dummy codes or adding data related variables) [43, 45].

Recently, Hao et al. showed that it is possible to detect app clones using UI state transition graphs [28]. In a simultaneous research, Zhang et al. proposed ViewDroid that first constructs feature view graph and then applies subgraph isomorphism algorithm to measure the similarity between two apps [45]. Although both ViewDroid and ResDroid exploit UI for detecting repackaged apps, there are three major difference between them. First, ViewDroid only uses the relationship among activities while ResDroid employs both the layout of activities and the relationship among activities. Moreover, we take into account Android's fragment component that provides similar functionality as an activity. However, ViewDroid does not handle such component. Second, by only examining core resources, ResDroid would be more efficient and more robust to third-party libraries than ViewDroid that considers all views. Third, whereas ViewDroid targets on comparison between a pair of apps, ResDroid is built on top of a two-stage methodology and equipped with two kinds of algorithms.

8. CONCLUSION

We propose a novel approach that leverages new features extracted from core resources and codes to detect repackaged apps. These features do not require processing all opcodes and are resilient to code obfuscation/app hardening techniques. To speed up the detection, we adopt the divide-and-conquer strategy to reduce the comparison and support parallel processing. Our solution have been realized in ResDroid and the extensive evaluation using real repackaged apps has demonstrated its effectiveness and efficiency.

9. ACKNOWLEDGMENT

We thank reviewers for their comments, and thank Wenjun Hu and Kai Chen for providing us samples. This work is supported in part by the Hong Kong GRF (No. PolyU 5389/13E), the National Natural Science Foundation of China (No. 61202396), the PolyU Research Grant(G-UA3X), and the Open Fund of Key Lab of Digital Signal and Image Processing of Guangdong Province(2013GDDSIPL-04).

10. REFERENCES

[1] Androguard. `https://code.google.com/p/androguard/`, 2011.

[2] 1.2% of apps on google play are repackaged to deliver ads, collect info. http://www.net-security.org/secworld.php?id=15976, 11 2013.

[3] App hardening emerges as a key component of mobile security strategies. http://betanews.com/2013/12/03/app-hardening-emerges-as-a-key-component-of-mobile-security-strategies, 2014.

[4] Gator: Program analysis toolkit for android. `http://dacongy.github.io/gator/`, 2014.

[5] Managing the activity lifecycle. `http://developer.android.com/training/basics/activity-lifecycle/index.html`, 2014.

[6] Resources overview. `http://developer.android.com/guide/topics/resources/overview.html`, 2014.

[7] C. Aggarwal and C. Reddy. *Data Clustering: Algorithms and Applications*. Chapman and Hall/CRC, 2013.

[8] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1), 2008.

[9] Arxan Technologies Inc. State of security in the app economy. http://www.arxan.com/resources/state-of-security-in-the-app-economy/.

[10] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proc. ACM SIGPLAN*, 2013.

[11] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Software Eng.*, 33(9), 2007.

[12] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Trans. Software Eng.*, 31(2), 2005.

[13] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proc. ACM ICSE*, 2014.

[14] W.-Y. Chen, Y. Song, H. Bai, C.-J. Lin, and E. Y. Chang. Parallel spectral clustering in distributed systems. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(3), 2011.

[15] R. Cilibrasi and P. Vitanyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4), 2005.

[16] Cisco Systems Inc. Annual security report, 2014.

[17] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, 2009.

[18] S. Corporation. Internet security threat report, 2014.

[19] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *Proc. ESORICS*, 2012.

[20] J. Crussell, C. Gibler, and H. Chen. Scalable semantics-based detection of similar android applications. In *Proc. ESORICS*, 2013.

[21] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software bertillonage. *Empirical Software Engineering*, 18(6), 2013.

[22] A. Dresnos. Android: Static analysis using similarity distance. In *Proc. HICSS*, 2012.

[23] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. OSDI*, 2010.

[24] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. Adrob: examining the landscape and impact of android application plagiarism. In *Proc. ACM MobiSys*, 2013.

[25] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proc. ACM MobiSys*, 2012.

[26] M. Halkidi, Y. Batistakis, and M. Vazirgiannis. On clustering validation techniques. *Journal of Intelligent Information Systems*, 17(2-3), 2001.

[27] 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 *Proc. DIMVA*, 2012.

[28] S. Hao, B. Liu, S. Nath, W. Halfond, and R. Govindan. PUMA: Programmable ui-automation for large scale dynamic analysis of mobile apps. In *Proc. ACM MobiSys*, 2014.

[29] H. Huang, S. Zhu, P. Liu, and D. Wu. A framework for evaluating mobile app repackaging detection algorithms. In *Proc. TRUST*, 2013.

[30] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proc. IEEE ICSE*, 2007.

[31] J.-H. Jung, J. Y. Kim, H.-C. Lee, and J. H. Yi. Repackaging attack on android banking applications and its countermeasures. *Wireless Personal Comm.*, 73(4), 2013.

[32] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3), 2006.

[33] C. Liu, C. Chen, J. Han, and P. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *Proc. ACM KDD*, 2006.

[34] M. Newman. *Networks: An Introduction*. Oxford University Press, 2010.

[35] A. Papadopoulos. *Nearest Neighbor Search: A Database Perspective*. Springer, 2004.

[36] D. Pelleg, A. W. Moore, et al. X-means: Extending k-means with efficient estimation of the number of clusters. In *Proc. ICML*, 2000.

[37] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang. Plagiarizing smartphone applications: Attack strategies and defense techniques. In *Proc. ESSoS*, 2012.

[38] C. Qian, X. Luo, Y. Shao, and A. Chan. On tracking information flows through JNI in android apps. In *Proc. IEEE/IFIP DSN*, 2014.

[39] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7), 2013.

[40] A. Rountev and D. Yan. Static reference analysis for gui objects in android software. In *Proc. IEEE/ACM CGO*, 2014.

[41] C. Roy, J. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7), 2009.

[42] Y. Shao, X. Luo, and C. Qian. Rootguard: Protecting rooted android phones. *IEEE Computer*, June 2014.

[43] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, and L. Xie. Detecting code reuse in android applications using component-based control flow graph. In *Proc. IFIP SEC*, 2014.

[44] M. Vasquez, A. Holtzhauer, C. Bernal-Cardenas, and D. Poshyvanyk. Revisiting android reuse studies in the context of code obfuscation and library usages. In *Proc. IEEE MSR*, 2014.

[45] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proc. ACM WiSec*, 2014.

[46] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of piggybacked mobile applications. In *Proc. ACM CODASPY*, 2013.

[47] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proc. ACM CODASPY*, 2012.

[48] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. IEEE Symp. Security and Privacy*, 2012.