ViewDroid: Towards Obfuscation-Resilient Mobile Application Repackaging Detection

Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu
The Pennsylvania State University
University Park, PA, USA
{fuz104, hhuang, szhu}@cse.psu.edu, {dwu, pliu}@ist.psu.edu

ABSTRACT

In recent years, as mobile smart device sales grow quickly, the development of mobile applications (apps) keeps accelerating, so does mobile app repackaging. Attackers can easily repackage an app under their own names or embed advertisements to earn pecuniary profits. They can also modify a popular app by inserting malicious payloads into the original app and leverage its popularity to accelerate malware propagation. In this paper, we propose *ViewDroid*, a user interface based approach to mobile app repackaging detection. Android apps are user interaction intensive and event dominated, and the interactions between users and apps are performed through user interface, or views. This observation inspires the design of our new birthmark for Android apps, namely, *feature view graph*, which captures users' navigation behavior across app views. Our experimental results demonstrate that this birthmark can characterize Android apps from a higher level abstraction, making it resilient to code obfuscation. ViewDroid can detect repackaged apps at a large scale, both effectively and efficiently. Our experiments also show that the false positive and false negative rates of ViewDroid are both very low.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: [Security and Protection]; D.2.8 [**Software Engineering**]: Metrics

Keywords

Mobile application; Repackaging; Obfuscation resilient; User interface

1. INTRODUCTION

In recent years, as the wide use and rapid development of mobile devices such as smartphones and tablets, mobile application (app) markets are growing rapidly. There were over $1,100,000$ apps available on the Google Play Android app market [5] on March 2014. Since popularity has become the core value among mobile platforms, many popular Android apps have been "copied," or repackaged, as reported by Gibler et al. [17]. One of the major reasons behind the emerging of Android app repackaging is that it is easy to reverse-engineer an Android app. When a user purchases and downloads an Android app, the installation package (i.e., the .apk file) is downloaded and stored on the user's mobile device. Given the openness of the Android platform, it is very easy to obtain the installation package from the device. After that, reverse engineering can be performed based on readily available tools such as apktool [1] and Baksmali/Smali [8], which can dissemble the compiled Dalvik EXecutable (dex) from the .apk file into a human readable Dalvik bytecode format (e.g., .smali files). At this point, the content of the app can be easily manipulated, modified, repackaged, and signed into a re-publishable APK file. Signing is not required to be bound with any official real ID of the developer and there is no certificate authority to sign apps. Moreover, due to the popularity of the Android platform, many unofficial app markets exist. Most of them do not enforce sanity checks on the apps listed on their web pages. As a result, the severity of app repackaging in the Android platform has been observed higher than in any other mobile platforms.

Generally speaking, there are two types of Android app repackaging. The purpose of the first type is to use other developers' apps to earn pecuniary profits. An attacker can easily repackage an app under his own name or embed different advertisements to gain ad benefits, and then republish it to an app market. Zhou et al. [34] found 5% to 13% of apps in the third-party app markets repackaged the apps from the official Android market. The second type is related to malware, where attackers modify a popular app by inserting some malicious payloads, e.g., sending out users' private information and purchasing apps without users' awareness, and leverage the popularity of the original app to accelerate the propagation of the malicious one. According to a recent study [35], 1083 (or 86.0%) of 1260 malware samples were repackaged from legitimate apps, indicating repackaging is a favorable vehicle for mobile malware propagation. Clearly, to maintain the health of an app market as well as for the security of mobile users, app repackaging detection is a critical issue to be addressed.

However, the problem of app repackaging detection is very challenging. On the one hand, due to the huge number of apps on an app market such as Google Play, efficiency and scalability of a detection scheme are highly demanded. On

WiSec'14, July 23–25, 2014, Oxford, UK.

http://dx.doi.org/10.1145/2627393.2627395.

the other hand, the detection scheme must be resilient to code modification and existing automatic obfuscation techniques, because it is very easy to modify, obfuscate, and repackage Android apps without the source code of the original apps. Recently, several research works have been proposed for repackaging detection, based on Fuzzy Hashing [34], Program Dependence Graph (PDG) [15, 14], Feature Hashing [18], Module Decoupling [33], and Normal Compression Distance (NCD) [16]. These approaches can detect app repackaging efficiently based on certain "invariants" extracted from the app code. Such invariants are called *software birthmarks* as in the software engineering research [24, 25, 28, 30, 20]. A *software birthmark* is defined as a unique characteristic that a program or mobile app inherently possesses, and can be used to uniquely identify the program. All the above approaches use code-level birthmarks to characterize an app.

In this paper, we propose a novel repackaging detection system called *ViewDroid*, which leverages user interface based *birthmark* for detecting app repackaging on the Android platform. ViewDroid provides an alternative to the code-level detection approaches. It is motivated by two observations. First, smartphone apps are user behavior intensive and Android event-driven, and the interactions between users and apps are performed through user interfaces (i.e., app views). Some characteristics of views (e.g. the navigation between views) are unique for each independently developed app. Second, in both types of repackaging, because attackers want to leverage the popularity of a target app, they usually keep the repackaged apps' look-and-feel similar to the original one in the user interface level. Specifically, ViewDroid is built upon a robust birthmark called *view graph*. View graph is a graph constructed from all views through static analysis and catches the navigation relation among app views. In addition, we design features for both nodes and edges in view graph based on Android specific APIs. This can help pre-filter the non-relevant apps and improve the efficiency of the graph comparison algorithm.

ViewDroid is resilient to code obfuscation techniques for the following reasons. (1) View graph is a higher level representation of an app's behavior than the traditional code level birthmarks (e.g., opcode sequence, program dependence graph). In other words, ViewDroid does not need instruction-level details. Hence, it is resilient to code obfuscation such as noise instruction/data injection, instruction reordering, instruction splitting and aggregation and data dependence obfuscation, etc. (2) The generation of view graph relies on statically analyzing Android specific APIs (e.g., `startActivity` and `startActivityForResult`). These APIs are provided by the Android system and are hard to be replaced or modified. Therefore, view graph, as the birthmark, is more robust to obfuscation techniques such as API splitting, API renaming and API re-implementation. Our evaluation results demonstrate that ViewDroid is robust to many existing code obfuscation techniques.

Our paper makes the following contributions:

1. **View Graph**: We propose view graph, a user interface-based birthmark for Android apps. To the best of our knowledge, it is the first user interface level birthmark for software plagiarism or app repackaging detection.
2. **ViewDroid**: We propose ViewDroid, an Android app repackaging detection system based on view graph. ViewDroid is robust to many code obfuscation techniques, and both efficient and scalable. ViewDroid provides a complementary approach to current code-level repackaging detection methods.
3. **Obfuscation Resilience Evaluation**: We evaluated the obfuscation resilience of ViewDroid by 39 obfuscators from SandMarks [11] and KlassMaster [4], based on the evaluation framework proposed by Huang et al. [19]. The experimental results show that ViewDroid outperforms Androguard [16] in terms of obfuscation resilience.
4. **Large Scale Evaluation**: We tested ViewDroid on $10,311$ real-world apps ($573,872$ app pairs) from the Android market. It is detected that about 4.7% apps are repackaging cases. We also evaluated the false negative of ViewDroid on a known repackaging app set. The false negative rate is 1.3%. The evaluation results demonstrate the efficiency and effectiveness of ViewDroid.

The remainder of the paper is structured as follows. Background of the Android platform and apps are given in Section 2. Section 3 generalizes the attack model and the design goals for repackaging detection. Section 4 describes the design of ViewDroid. Evaluation is presented in Section 5, followed by discussions in Section 6 and related work in Section 7. Finally we conclude the work with Section 8.

2. BACKGROUND

Android is a Linux-based platform for mobile devices. Users can download and install Android apps from various app markets. Android apps are published to the market in a compressed file format (i.e., .apk file). It contains a manifest file (i.e., AndroidManifest.xml), resource files (i.e., files in res directory), and compiled Dalvik Executable (i.e., classes.dex). The manifest file lists the package name, version number, critical components of the app, and the associate permissions to each component. The resource folder includes all the raw resource files, such as images and audio files, and the XML files which describe the layouts of user interfaces. The Dalvik executable contains all the classes that implement the functionality of all the primary components of an app. Some apps contain parts that are implemented by native languages. Since relatively few Android apps contain such components developed in the native languages C/C++ and they mostly serve as background services, our current ViewDroid design only takes into consideration the Dalvik executable, the relevant Android manifest file, and the layout files in the resource folder .

Components serve as the building blocks for Android apps. There are four types of components, namely, *Activity*, *Service*, *Broadcast Receivers*, and *Content Provider*. An *Activity* provides a screen for the user to interact with. An app requires one main activity to start but can have a number of other activities (roughly one per screen view). A stack is designed to organize activities. When a new activity starts, it goes to the top of the stack. A *Service* is a component that runs in the background, usually engaged in the performance of long-running tasks. In general, a service is used to perform any task that is asynchronous with respect to the main user interface. A *Broadcast Receiver* listens to special messages broadcasted by the system or individual apps and relays work to other services or activities. Finally, a *Content Provider* manages shared data and optionally ex-

poses query and update capabilities for other components to invoke. A message-like *intent* is used to help the communications among components.

The execution sequence of an Android app usually starts from the main activity, specified in the manifest file. When the launching icon of an app is pressed by user, the main activity will be launched. It serves as the main entry point to the user interface. The app switches between activities by invoking platform APIs, `Context.startActivity()` or `Activity.startActivityForResult()` with `Intent` objects as parameter. An `Intent` object contains the information of the target activity. A user interface is loaded when an activity is initialized by the `onCreate()` method, which creates a new user view through APIs like `setContentView()`. The view is then put on the top of the view stack and becomes the running activity. Therefore, by analyzing the Android specific APIs within each `Activity` class, the user interface navigation relation information can be constructed to build our view graph. Note that, in our work, we only consider apps that have interactions with users (e.g., by key pressing, button clicking). Some other apps, which only have background services and do not interact with users, are out of our consideration.

3. PROBLEM STATEMENT

The most fundamental challenge of app repackaging detection is to find unique *birthmarks* to characterize apps. The proposed birthmark should be *accurate* and *unique* enough to identify an Android app. Moreover, as reported by Zhou et al. [34], the plagiarists and malware writers tend to use obfuscation on the repackaged apps to evade detection. Hence, to significantly raise the bar for stealthy repackaging, the designed detection scheme must be resilient against most code obfuscation techniques. Finally, since the Android app repackage problem is prevalent among most Android markets, it is very important to build a detection tool that can perform detection in large-scale scenarios.

Scope of the paper. In this paper, our purpose is to detect repackaging Android app pairs, but not to identify which is the original one and which is the repackaged one. We only focus on non-trivial Android apps that interact with users through user interface and are implemented as Dalvik executables. Apps that contains components implemented by native-code languages are out of the scope of our paper. Those only providing background services without user interactions are not under our consideration either.

3.1 Attack model

The general attack model in the Android app repackaging problem is: an attacker has access to the plaintiff Android app package (.apk file); he repackages the app by copying the code, making a few modifications (e.g., replacing the advertisement, attaching malicious payloads), and applying automatic code obfuscation techniques in order to evade detection; the repackaged app is then signed with a private key and republished to the app markets.

Based on the level of modification on the original APK files and the effort an attacker is willing to pay on the repackaging process, we further classify the repackaging attacks into the following three categories:

Lazy attack: A lazy attacker can make some simple changes over an app without changing its code. For instance, repackaging an app with a different author name or with different advertisements is such rudimentary lazy attack. Non-developers can be easily trained to perform such tasks manually. More knowledgeable lazy attackers may apply current automatic code obfuscation tools to repackage an app without changing its functionality, following the procedure similar to what is shown in our evaluation section.

Amateur attack: An amateur attacker not only applies automatic code obfuscation but also changes/adds/deletes a small part of the functionalities. For example, an attacker can add some online social functionalities along with the online chat view to the original app. Attackers must pay more effort to understand and thus modify the code. For example, they have to read the Android manifest file to delete or append the components that they want to register for the app and to insert some interaction code into the original components to glue the newly added components.

Malware: A malware writer creates a malicious app that mimics a popular app by inserting some malicious payloads into the original program. In this way, the malicious app can leverage the popularity of the original app to increase its propagation speed. With this purpose, the attacker tries to make the functionalities and user interfaces of the repackaged app similar to the original one. Under this circumstance, an attacker actually has to perform most of tasks that an Amateur attacker has to do. In addition, the attacker needs to write the malicious payload either in Java or C/C++ and stealthily insert the payload into the app.

We further analyze how well ViewDroid can detect these attacks and other potential advanced attacks in Section 6.

3.2 Design Goals

In this paper, we design an app repackaging detection scheme named ViewDroid, with the following goals.

Accurate Birthmark: In order to measure the similarity between two Android apps, ViewDroid must select an accurate birthmark to characterize apps. This accurate birthmark should be able to reflect the primary semantics of Android apps and tell independently-developed apps apart. In other words, the designed birthmark should cause very few false positives.

Obfuscation Resilience: Code obfuscation is a technique to transform a sequence of code into a different sequence that preserves the semantics but is more difficult to understand or analyze. Obfuscation techniques can also be applied by attackers to evade repackaging detection. Hence, ViewDroid must be able to detect repackaging with the presence of various automated code obfuscation techniques. In other words, the designed birthmark should be robust against various obfuscators. Obfuscation resilient birthmarks will ensure low false negatives.

Scalable Detection: Because there are a huge number of apps on different Android app markets, ViewDroid must be efficient and scalable enough to detect repackaging in such a large-scale scenario.

4. DESIGN

4.1 Overview

It is critical while very challenging to identify an accurate and obfuscation resilient birthmark in the design of a repackaging detection tool. In the past, a good variety of birth-

marks have been proposed and evaluated for different types of program languages (C, Java) and platforms (Linux, Windows). Some of these traditional birthmarks have been proposed to detect Android app repackaging [14, 18, 16, 34]. They are all code-level birthmarks. In this paper, instead of applying traditional software birthmarks, we propose a novel user interface-based one, namely *feature view graph*. It fully leverages a unique characteristic of smartphone apps – they are mostly UI intensive and event dominated [26]. Feature view graph represents a higher level abstraction of an Android app's semantics. Therefore, it has the potential to be more robust to code obfuscation. In order to meet the scalability requirement of Android repackaging detection, feature view graph is generated by static analysis of the dissembled installation file of an Android app (i.e., the APK file).

We define *view* in Definition 4.1 and *view graph* in Definition 4.2. View graph describes the user interface navigation relations of an Android app.

DEFINITION 4.1. ***(View)*** *A view is a user interface that is displayed to users for interaction with the mobile app. Each view has a corresponding activity class that defines the view's functionality. A view contains one or more visible components (e.g. buttons, trackball) on the screen. When touched, the components might trigger other activities or services.*

DEFINITION 4.2. ***(View Graph)*** *A view graph of a mobile app is a directed graph $G(V, E)$, where V is a set of nodes, each of which represents a user interface view. E is a set of edges $< a, b >$ such that $a \in V$, $b \in V$ and the smartphone display can switch from view a to view b by user interaction or other triggers.*

By adding features to each view and each edge, a view graph can represent an app more accurately and also improve the efficiency of app similarity measurement at the later stage. The features of a view could be the number, types or layout of the visible components (e.g., buttons, menus), or a set of Android platform specific APIs invoked in this view's activity. However, the former one (e.g., layout) is much easier for an attacker to manipulate because it does not represent the fundamental semantics of an app. The latter one (i.e., Android specific APIs) is much more stable and can reflect app's semantics; as a result, we only consider that as view features.

In a feature view graph, the feature of an edge is the event listener function (e.g., `onClick()`, `onLongClick()`, `onTouch()`, etc.) that is directly triggered by user generated events. Generally, there are two types of events in Android platform, user-generated events and system-generated events. We only focus on user-generated events in our birthmark creation. This is because these events are highly associated with the functionality of an individual app and the corresponding user interaction with the app. For instance, the `onClick()` method of a registered listener is triggered when the corresponding button is pressed by a user, so we consider it as an edge feature. An example of system-generated events is when the system sends a short-message-received event, which triggers the `onReceive()` method registered for the SMS_RECEIVED intent in the manifest file. Clearly, this `onReceive()` method is not triggered by direct user interaction, so it is not considered as an edge feature. The *feature view graph* is defined in Definition 4.3.

DEFINITION 4.3. ***(Feature View Graph)*** *A feature view graph of a smartphone app is based on its view graph, $G(V, E)$, where certain features are selected and attached to V and E.*

After creating the feature view graph of a plaintiff app and a suspicious app, ViewDroid measures the similarity between the two graphs by an applying subgraph isomorphism algorithm. Since it is an NP-complete problem, we need to improve the performance of graph matching. To this end, we apply a pre-filter to eliminate those more obvious non-matching pairs in advance.

4.2 System Architecture

Figure 1 shows the system architecture of ViewDroid, which has three primary components. Given two Android apps in *.apk* format, the *Code Extractor* will extract and parse the smali code (the dissembled version of Dalvik bytecode), human-readable Android Manifest file and layout XML files from each app's installation package. After that the *View Graph Constructor* performs some static analysis to generate a feature view graph for each app. A pre-filter is applied to remove app pairs that are not likely to be similar. Then the *Graph Similarity Checker* compares two feature view graphs and calculates a similarity score. Note that if two apps are signed by the same developer, we do not consider them as a repackaging case.

Code Extractor. The user interface layouts of an Android app are usually defined in XML files in the res/layout/ directory. The activity of a view is implemented in the *classes.dex*, which is compiled as Dalvik bytecode. Instead of focusing on the view layout that can be easily modified, we conduct the analysis on the activity class, which defines the functionality of a view and indicates the navigation between views. We choose to perform the static analysis directly on the smali code, which is an intermediate representation of Dalvik bytecode. This is because smali code is the direct dissembled version of Dalvik binary with rich annotation information. Our static analysis also uses some information from the Android manifest files and the layout component files. We leverage an existing tool apktool [1] to extract smali code and human-readable XML files from android app packages.

View Graph Constructor. As discussed in Section 2, the layout of an app view is usually defined in XML resource files and loaded by activity code during the execution to be presented to users. Most activities load a view by invoking the `setContentView()` function with an XML file name as the parameter, in its `onCreate()` function. A few special views are loaded by other functions, e.g., the Settings view is loaded by the `addPreferencesFromResource()` function. View navigation is implemented by activity switching. When an activity calls another activity, an instance of the callee activity is created, a new view associated with the callee will be loaded and put on the top of the system's view stack to be presented to users. An activity switches to another activity by invoking function `startActivity()` or `startActivityForResult()` with an `Intent` object as the parameter. As a result, we can construct the view graph by statically analyzing these function invocations.

The detailed steps of view graph construction are as follows:

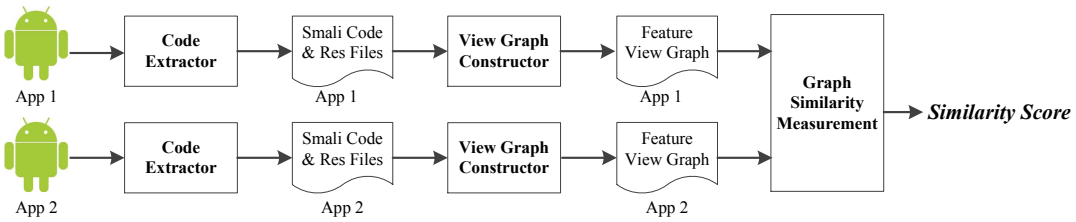


Figure 1: The ViewDroid system architecture.

1. **Generate view nodes**: We need to collect all the activities that are associated with potential UI views, each of which is usually a separate smali file loading a view layout in its `onCreate()` function. In each activity, we parse and grep the view loading function, such as `setContentView()` and `addPreferencesFromResource()` in the `onCreate()`function. The parameters of these view loading functions are the names of the XML resource files. After parsing all this relevant information, every view node and its relation to the corresponding activity class is generated.

2. **Extract view node features**: For the features of the view nodes, we only focus on the Android framework specific APIs. Since the Android platform use Java APIs that are built on a subset of the Apache Harmony Java implementation, we consider this set of APIs are more vulnerable to renaming attacks. Attackers can easily find semantic similar or equivalent APIs from other sources. However, the set of Android specific APIs, e.g., methods from the *android.security.KeyChain* or *android.nfc.NfcManager* classes, are very hard to be replaced. In order to interact with the Android platform, an app has to register certain permissions in the manifest files and use the relevant APIs to perform tasks. Based on this observation, we build the feature for each view node accordingly. We first analyze each activity class file associated with a view node to extract a set of invocations of the Android specific APIs. Then we can build an *invocation vector* for each view node. In such invocation vectors, instead of making a counter for each API, we only set a flag for the APIs that are invoked in the activities. This can protect ViewDroid from dummy code insertion attack and can also improve the efficiency on the invocations vector pattern matching.

3. **Generate edges**: Edges in the feature view graph represents the activity switch relationship among the set of views. The source view is associated with the caller activity of the `startActivity()` or `startActivityForResult()` functions. The target view is associated with the activity declared by the `Intent` object. There are six kinds of `Intent` constructors [3]:

   (1) `Intent()`
   (2) `Intent(Intent o)`
   (3) `Intent(String action)`
   (4) `Intent(String action, Uri uri)`
   (5) `Intent(Context packageContext, Class<?> cls)`
   (6) `Intent(String action, Uri uri, Context packageContext, Class<?> cls)`

   As described in [32], constructors (5) and (6) specify the target activity in an explicit way with a particular class name. We can perform analysis to trace back to this hard-coded class name. Constructors (3) and (4) initialize an implicit Intent object by an action name, with or without a URI. The associated target activity, which could be within the same app or in another app, is selected by matching *intent filters* in the Android manifest files. The external target is undecidable without knowing other apps installed in a smartphone. In ViewDroid, we create a general destination node `external_activity` to represent all external targets and add an edge from the source activity to this node. Constructor (1) initializes an empty intent, which is surrounded by `setClass()`, `setComponent()` or `setAction()`. Hence, the identification of the target activity is the same as constructors (3)-(6). Constructor (2) copies another Intent object *o*. In this case, our analysis needs to trace back to the activity, which is specified by the constructor of the object *o*.

   In order to figure out all the possible switching relationships among views, static analysis is performed. By analyzing all `startActivity()` and `startActivityForResult()` functions, we can stitch the caller activity and callee activity and therefore create an edge from the view of the caller activity to the view of the callee activity. Our view switching based invocation graph is more robust to code obfuscation than the traditional call graph, because it does not rely on the exact call sequence starting from one view node and ending at another view node. Whenever there is a view switching relationship, an edge is built to link the two views. It captures the user's real experience of view switching. Even though there might be several method invocations between an actual view switching, we ignore all the intermediary method calls, but just stitch the source and end view nodes for the corresponding activity classes. As long as attackers want to keep most functionality of the original app, the view switching relationship cannot be changed.

4. **Extract edge features**: In order to minimize false matches and improve the efficiency of similarity measurement in a latter stage, we add a feature to each edge. It is the user-generated event that triggers the view switch. During the static analysis, we can locate the `startActivity()` or `startActivityForResult()` functions and analyze which function call actually triggers the view switching. The trigger could be library provided event listener, such as `onClick()`, `onTouch()`, `OnItemSelected()` etc, or app developer self-defined functions. We consider these triggers to be the fea-

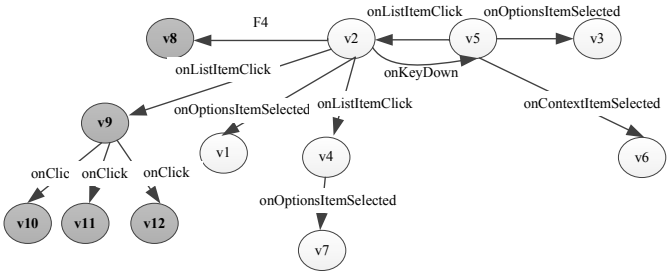


Figure 3: The feature view graph of a repackaging app.

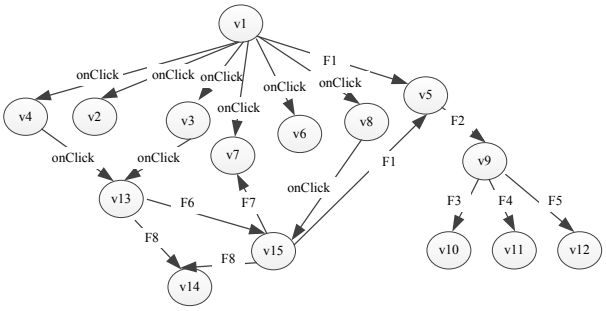


Figure 4: The feature view graph of an independent app.

tures of edges. Note that because the names of self-defined functions can be easily modified by an attacker, so we label all the developer self-defined trigger functions with the same name `self_defined_trigger` and consider them potentially matched with each other.

Figure 2 illustrates the steps of view graph construction for a simple Sudoku app. Figure 3 shows the feature view graph of an app that repackages the original app in Figure 2. The repackaged app copies the original app, adds an AdActivity (node $v8$) and additional social network functions (nodes $v9$, $v10$, $v11$, $v12$). Figure 4 shows the feature view graph of an independent app. Note that to make the graph clear in these figures, we omit the node features.

Graph Similarity Checker. We apply the VF2 [13] subgraph isomorphism algorithm to measure the similarity between two feature view graphs.

A pre-filter is leveraged to reduce the graph pairs that need to be compared. If one of the following three criteria meets, we will consider that they are not repackaging cases:
(1) If the size of two view graphs differs a lot (specifically, the size of the bigger graph is at least 3 times of the smaller graph);
(2) If the node features (i.e., those Android specific APIs considered in feature view graphs) in two view graphs have limited overlap (i.e., the number of overlapped features is below 1/3 of the size of the smaller graph).
(3) If the sets of edge features in two view graphs have limited overlap (i.e., the number of overlapped edge features is below 1/3 of the edge number in the smaller graph).

When two graphs are compared by the subgraph isomorphism algorithm, only nodes and edges with similar features can be matched. We consider two view nodes are similar when their API invocation vectors have the Jaccard distance below 0.5. The Jaccard distance between two sets A and B is calculated with Formula 1. Edges with the same event listener are considered as a matched pair. Not only can this feature pre-comparison reduce false matches of nodes and edges, thus decreasing the false positives caused by simple view graphs, but it can also improve the efficiency of subgraph isomorphism computation.

$$J_d(A,B) = 1 - \frac{A \cap B}{A \cup B} \tag{1}$$

If apps A and B have m matched nodes, with n_A and n_B nodes in their feature view graphs, respectively, their similarity score is calculated as:

$$\text{similarity score} = \frac{m}{\min(n_A, n_B)} \tag{2}$$

5. EVALUATION

ViewDroid is implemented in Python and Shell-script. The whole system consists of 2400 lines of Python code and 400 lines of Shell-scripts. Our experiments were conducted on a commodity machine with 1.6 GHz Intel Core i5 processor and 4 GB memory.

We have two sets of experiments. First, we conduct evaluation on a large set of real-world apps to measure the effectiveness and efficiency of ViewDroid. We also test the percentage of the repackaged malware cases. Second, we evaluate the obfuscation resilience of ViewDroid by applying different obfuscation techniques on existing apps and using ViewDroid to detect their similarities.

5.1 Real-world Large-scale Experiment

5.1.1 *False Positive and Efficiency*

We crawl 10,311 top Android apps from Google Play. These apps belong to 20 categories. We randomly choose 100 samples from each category and compare them with apps in the same category in a pairwise way. Totally 573,872 app pairs are compared.

We set the similarity score threshold at 0.7. After applying ViewDroid to detect the repackaged apps, we manually check the detected pairs to measure false positives. The manual checking has two criteria: (1) We execute the app on a smartphone to check the similarity of their functionality; (2) We check the code, including smali files, layout files and the permissions. Only when both criteria are similar, we consider them as the real repackaging cases. We find 129 false matched pairs in total in 11 categories. Most of the false matches (112 out of 129) are caused by the invocations of ad libraries. When two apps share the same ad libraries and one app's graph size is relatively small, the matched nodes related to the common ad libraries will result in a high similarity score. These false matches can be eliminated by whitelisting known ad libraries. That is, we can simply ignore views that are generated by whitelisted libraries. The other 17 false matches are due to that one of the apps in each pair is very simple. For their view features, no special API is invoked and therefore nodes are not distinguishable. Moreover, their view graphs are small and easy to find matchable (sub)graphs. Our detection results, after adding a whitelist to rule out the known ad libraries, are shown in Table 1. The percentage column is the proportion

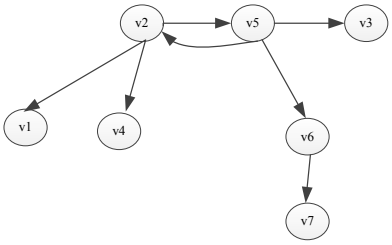
Step 1: Generate Nodes

Node	Smali Code	XML File
v1	FileListActivity	file_list
v2	FolderListActivity(Main Activity)	folder_list
v3	SudokuEditActivity	sudoku_edit
v4	SudokuExportActivity	sudoku_export
v5	SudokuListActivity	sudoku_list
v6	SudokuPlayActivity	sudoku_play
v7	GameSettingsActivity	game_settings

Step 2: Extract Node Features

Node	Features
v1	(ListActivity ->onPrepareDialog, ...)
v2	(Cursor->getColumnIndex,...)
v3	(Activity->onWindowFocusChanged,...)
v4	(ProgressDialog->setTitle,...)
v5	(MenuItem->setshortcut,...)
v6	(Window->setFlags,...)
v7	(MenuItem->setIcon,...)

Step 3: Generate edges



Step 4: Extract edge features:

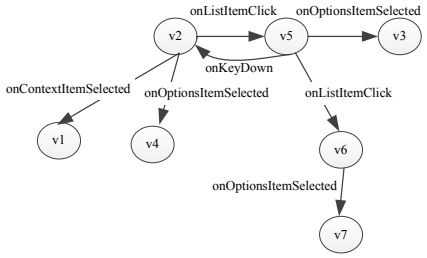


Figure 2: An example of view graph construction

of apps, which either repackage other apps or are repackaged by others, in all apps of each category. On average 4.7% among tested apps are found to be the real repackaging cases. The book and comic categories have more repackaging cases than other categories, because in both categories, there are existing products that can convert an ebook into an Android app. The apps generated by the same converting product are detected as repackaging pairs by ViewDroid. They are true positives since they share the same code base and the same views.

Among all 542 repackaging pairs, 262 of them belong to lazy attacks. The malware cases are analyzed in Section 5.1.3. The other pairs belong to the amateur attacks. Note that ViewDroid only measures the similarity between two apps. It does not identify which one is the original app and which one is the repackaged one.

The average execution time of ViewDroid for each testing is listed in Table 2. It is about 11s per pair. In rare cases, the graph construction time and graph comparison time may take minutes. Only 0.6% apps take more than 1 minute to construct view graph and 0.2% pairs need more than 1 minute to conduct graph comparison. In addition, when applying ViewDroid to check a large number of apps, code extraction and view graph construction for each app is only performed once.

5.1.2 False Negative

In this section, we use a set of repackaged apps provided by a research group to measure the false negative rate of ViewDroid. These apps were collected from multiple Android markets. The app dataset includes totally 901 pairs of apps, whose view graphs have more than 3 view nodes. By setting the similarity score threshold at 0.7, as in Section 5.1.1, ViewDroid detects 868 pairs as repackaging cases. Among 659 of them, each pair of apps have the similarity score 1.0.

We then manually check the 33 pairs that are not detected by ViewDroid. They can be divided into three different categories. (1) For 11 pairs, two apps of each pair do not share or share very little common code. They do not have common functionalities or views either. As a result, not reporting them is the correct detection result for these 11 pairs. They were falsely included in the app dataset. (2) Another 10 pairs are not real repackaging cases either, although they do share some code between each other. The shared code is not related to the functionalities or the views of these apps, but is used as malicious payload to create ad shortcuts or to send out messages without users' awareness. That is, attackers use different apps to propagate the same malicious payload. Therefore, ViewDroid is correct again not reporting them. (3) The other 12 pairs are false negatives of ViewDroid at detection threshold 0.7. Here, each pair of apps have repackaged code related to their major functionalities, but have different code that implements

Table 1: The repackaged apps detected by ViewDroid

Category	Pair#	App#	Repackaged Pair	Repackaged App	%
Books	34,550	495	81	55	11.1
Business	23,882	455	10	13	2.9
Comics	40,850	558	110	75	13.4
Communication	20,582	487	0	0	0.0
Education	40,950	559	7	11	2.0
Entertainment	25,758	512	10	16	3.1
Finance	37,650	526	9	13	2.5
Game arcade	30,496	543	64	37	6.8
Game cards	27,329	545	11	13	2.4
Game casual	20,662	509	12	18	3.5
Health	36,550	515	13	20	3.9
Lifestyle	20,538	509	10	13	2.6
Media	39,150	541	56	35	6.5
Medical	38,650	536	14	21	3.9
Music	19,655	496	21	20	4.0
News	10,466	495	21	24	4.8
Personality	37,050	520	31	25	4.8
Photography	23,914	518	17	22	4.2
Shopping	28,185	495	23	23	4.6
Social	17,005	497	22	26	5.2
Total	573,872	10,311	542	480	4.7

Table 2: The execution time of ViewDroid (in seconds)

	Code Extraction	Graph Construction	Graph Comparison
Max	15	146	590
Avg	4	6	1

Table 3: The malware attacks detected by ViewDroid

Type	Number
Trojan.FakeApp/FakeFlash	25
Adware.Airpush	14
Adware.Plankton	17
Adware.LeadBolt	26
Other Adware	10
Virus	1

"add-on" functions. These add-on components are relative large and complex compared to their carrier code. For example, two apps both implement a Ninja game. The matched view nodes detected by ViewDroid are the game itself, while the unmatched view nodes represent different social network functions. It is very likely that these two apps both repackaged another benign app by inserting their own customized social network library, which targets a specific market. The similarity scores of false negative cases are all between 0.5 and 0.7. It indicates that ViewDroid is able to find their common views. The false negative rate of ViewDroid at detection threshold 0.7 is 1.3%.

5.1.3 Malware

We use VirusTotal (`https://www.virustotal.com/en/`), an online malware detection service, to scan all the repackaged pairs detected in Section 5.1.1. Among the 480 apps identified as involved in repackaging cases (either the original ones or the repackaged ones) in our previous experiment, we detect 93 malware, which is 19.3% of repackaged apps. The malware types are listed in Table 3. They mainly belong to two different categories: Adware and Trojan horse. Adwares aggressively show advertisements to smartphone users. Trojan horses usually pretend to be legitimate apps, but steal sensitive information covertly. There is one virus detected. It is labeled as `Virus:BAT/Rbtg.gen`.

5.1.4 Category-based Evaluation

Next we illustrate a different kind of evaluation on real-world apps. We first search by some keywords in Google Play, and then download the returned apps and conduct pairwise measurement of their similarities. While our previous large-scale experiment randomly chooses pairs in the app market to evaluate the effectiveness and scalability of ViewDroid, this experiment is more interesting to individual app developers and app users to understand how repackaging may affect them.

We list two examples here. The first keyword is "sudoku" and we download 20 sudoku game apps. Based on the similarity scores, we cluster these apps as shown in Figure 5. An edge indicates two apps have a similarity score higher than 0.7. The largest cluster has 9 apps. The app with dashed circle is similar to all the other 8 apps in the cluster. The other two clusters both have 3 apps. Our manual checking verifies that the result has no false positives and false negatives. Further analysis indicates that there are 3 pairs belonging to lazy attack, where plagiarists only repackage the original apps without changing their functionality. The similarity scores of these pairs are 1.0. ViewDroid also discoveries one malware case, the red big node in Figure 5. VirusTotal identifies it as the Airpush Adware, which aggressively shows ads in the Android notification bar. This app inserts Airpush ads module into the original app and

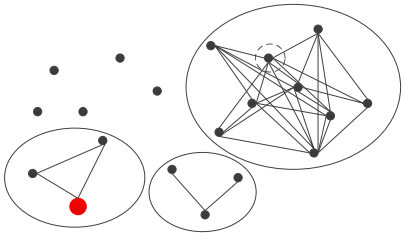


Figure 5: The cluster of sudoku apps based on the similarity scores.

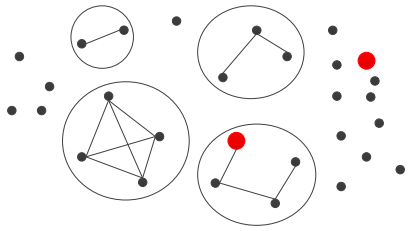


Figure 6: The cluster of flashlight apps based on the similarity scores.

slightly modifies the functionality by removing a strategy help view. The other repackaging pairs are all amateur attacks, where functionalities are added or removed, such as social network modules, help view, strategy hint views and advertisements.

In the second example, we search by the keyword "flashlight" and download 29 apps. We find 15 pairs with similarity scores higher than 0.7. Our manual checking indicates that 3 pairs are false positive cases. They are all caused by one app that has 4 views, only one Of which relates to its functionality whereas the other three are generated by an ad library. When compared to apps that share the same library with it, the three ad views are matched and the similarity scores are 0.75. Again, such false positives can be eliminated by whitelisting the ad libraries. The similarity cluster is shown in Figure 6. Four clusters have more than one app. Among all the 12 repackaged pairs, 2 belong to the lazy attacks, and 9 belong to the amateur attack where views are added or removed (e.g., the "about" view, "setting" view). One malware attack, shown as a big red node in Figure 6, is found. It is reported as a trojan horse by VirusTotal (there is indeed another malware in the 29 downloaded apps, but it is not the app repackaging case. It is identified as Plankton [7]).

5.2 Obfuscation Resilience

To test the obfuscation resilience of ViewDroid, we try to obfuscate the existing apps and malware with different obfuscators, and then check with ViewDroid the similarity score between each original app and its corresponding obfuscated one. Most existing popular obfuscation tools (e.g. ProGuard [6] and DexGuard [2]) work on Java source code level and their obfuscators are limited to method renaming, string encryption and class name encryption, etc. Therefore, we choose to use an obfuscation resilience evaluation tool developed by Huang et al. [19]. This evaluation framework can obfuscate and repackage apps by using one or multiple ob-

fuscators from different Java bytecode obfuscation platforms (e.g., Sandmarks [11]). It directly targets the Dalvik bytecode. This actually mimics the real world scenarios where a plagiarist or repackager who only has access to the compiled Dalvik bytecode but not the Java source code and is eager to use various obfuscation techniques to evade detection. In our current obfuscation resilience test, we equip the framework to perform 39 obfuscators from both SandMarks [11] and KlassMaster [4]. To our knowledge, this is the broadest obfuscation resilience evaluation on Android repackaging detection.

First of all, we generate pairs of APK files from the obfuscation resilience evaluation tool. Then, we use ViewDroid to measure the similarity pairwisely between the obfuscated APK file and the original APK file. The higher similarity scores our ViewDroid returns for each specific obfuscator, the better resilience against that particular obfuscation.

We choose 50 apps from the Android app market based on different categories and 50 malwares from the malware Gnome project based on different families [35]. With this 100 Android app set, we perform *broadness* analysis and *depth* analysis to evaluate the obfuscation resilience aspect of ViewDroid provided by the evaluation framework. The broadness analysis result shows the general weakness and strength of ViewDroid against a broad range of obfuscation techniques. In this analysis, each obfuscator is applied individually. On the other hand, the depth analysis result evaluates the overall obfuscation resilience of ViewDroid against deep code manipulation by serializing a set of obfuscators. In this analysis, ViewDroid is evaluated against repackaged apps that have been obfuscated by multiple obfuscators. For example, an app may be obfuscated by variable renaming, followed by noise injection and/or control-flow flattening. With depth analysis, we can test the robustness of our detection scheme against more sophisticated obfuscation attacks.

5.2.1 Applying Single Obfuscation Algorithm

In our current evaluation setup, the *broadness* analysis is based on 39 obfuscation algorithms from *SandMarks* and *KlassMaster*. Table 4 shows the resilience comparison between ViewDroid and AndroGuard. The *Obfuscation Algorithm* columns indicate the names of the obfuscation algorithms applied in our framework. The *ViewDroid* columns list an average similar score for each obfuscation case. Specifically, in each obfuscation case, ViewDroid computes a similarity score for each original app (among totally 100 apps) and its obfuscated version and finally reports the average over 100 apps. The *AndroGuard* columns are the results reported by Huang et al. in [19], and we also compute three average similarity scores for AndroGuard based on three obfuscators from KlassMaster, which were not provided in the previous case study. All these three obfuscators have a *K*-tag at the beginning of the obfuscators' names in Table 4.

Based on the classification by Collberg et al. [12], all the single obfuscators can be categorized as *layout obfuscation*, *control based obfuscation* and *data based obfuscation*, which are tagged *L*, *C* and *D* after each obfuscator. The detailed explanation of the difference between these categories can be found in Huang et al. [19]. Overall, ViewDroid has better obfuscation resilience than AndroGuard. This is because in ViewDroid, repackaging detection is performed based on the similarity of the high level semantics of the app using the created feature view graph, while ignoring the detailed control/data dependency or data structure. From the result, we can see that only 4 out of 39 obfuscators have an effect on ViewDroid, and the average similarity scores of all the other 35 obfuscators tested against ViewDroid are all 1.00.

The *Class Encrypter* obfuscator reduces the similarity score to 0, which is the only obfuscator that ViewDroid returns a lower score than AndroGuard. However, the score for AndroGuard is .03, which is very close to zero. This indicates that static analysis based detection schemes are not well-suited for encryption based obfuscation. By encrypting class files and decrypting them at runtime, Class Encrypter can completely hide the static structure of the program. However, certain heuristic can be built to preprocess these extreme encryption cases. For instance, whenever decryption or decoding is used in the program very intensively or is identified for a very large portion of the code, it can be flagged as suspicious. Usually, dynamic analysis based detection is needed in this situation, which is, however, lack of scalability. Overall, handling the heavy encryption and encoding based obfuscation is an interesting topic to explore in the future.

The other obfuscation algorithms that have some influence on ViewDroid are *Node Spliter*, *Method Madness* and *Class Splitter*. After further analysis of the feature view graph pairs computed from the 100 apps' obfuscated versions, we find that some graph nodes are split by obfuscators Node Spliter and Class Splitter, and the names of the methods that trigger view switching are replaced by some random names by Method Madness, which can potentially modify the feature of our view graph. However, from the overall similarity scores of these four obfuscators, we can see that these types of obfuscation cannot be performed frequently, as certain conditions have to be satisfied before these obfuscators make the actual manipulations. For instance, some class inheritance relationship has to be met in order to perform Node Splitter or Class Splitter, and also relevant specification in the Android manifest file should be changed accordingly. Furthermore, the obfuscation of method randomization in Method Madness cannot be performed directly on the Android framework APIs, tedious method rewriting work has to be performed before replacing the invocation of the Android APIs. For instance, simply changing the invocation *Landroid/app/Activity.dispatchTouchEvent (Landroid/view/MotionEvent;)Z* into *Landroid/app/Activity. M103456d(Landroid/view/MotionEvent;)Z* does not work. As a result, we find that most APK files become non-executable after the Method Madness obfuscation.

5.2.2 Serializing Multiple Obfuscation Algorithms

Practically, especially when detection algorithms become more powerful, it is very possible that an attacker will try a combination of various obfuscation algorithms. Hence, besides the *broadness* analysis performed on ViewDroid, for *depth* analysis we also apply multiple obfuscators by serializing the top-three obfuscators reported from our broadness analysis, excluding the *Class Encrypter*. Due to the conflicts among various obfuscators, not all the obfuscated APK files are complete. We test various permutation cases with these three obfuscators and find two of all the permutations can be performed more successfully for the testing apps. One can output 99 out of 100 obfuscated APK files and the other outputs 96 out of 100 for all the tested ones. These two interesting permutations are shown as follows:

Table 4: Average similarity score by ViewDroid compared with AndroGuard for each obfuscator used in broadness analysis

Obfuscation Algorithm	ViewDroid	AndroGuard	Obfuscation Algorithm	ViewDroid	AndroGuard
Const Pool Reorder (L)	1.00	0.92	Node Spliter (D)	0.94	0.94
Static Method Bodies (C)	1.00	0.88	Class Encrypter (D)	0.00	0.03
Method Merger (C)	1.00	0.65	Reorder Parameters (D)	1.00	0.92
Interleave Methods (C)	1.00	0.56	Promote Prim Register (D)	1.00	0.92
Opaque Pred Insert (C)	1.00	0.92	Promote Prim Types (D)	1.00	0.93
Branch Inverter (C)	1.00	0.77	Bludgeon Signatures (D)	1.00	0.96
Rand Dead Code (C)	1.00	0.92	Objectify (D)	1.00	0.83
Class Splitter (C)	0.97	0.87	Publicize Fields (D)	1.00	0.91
Method Madness (C)	0.92	0.43	Field Assignment (D)	1.00	0.86
Simple Opaque Pred (C)	1.00	0.92	Variable Reassign (D)	1.00	0.85
Reorder Instructions (C)	1.00	0.89	Parameter Alias (D)	1.00	0.92
Buggy Code (C)	1.00	0.67	Boolean Splitter (D)	1.00	0.85
Inliner (C)	1.00	0.89	String Encoder (D)	1.00	0.87
Branch Insert (C)	1.00	0.87	Overload Names (D)	1.00	0.91
Dynamic Inliner (C)	1.00	0.84	Duplicate Registers (D)	1.00	0.89
Irreducibility (C)	1.00	0.86	Rename Registers (D)	1.00	0.96
Opaque Branch Insert (C)	1.00	0.85	False Refactor (D)	1.00	0.95
Exception Branch (C)	1.00	0.81	Merge Local Int (D)	1.00	0.94
K-Flow Obfuscation (C)	1.00	0.77	K-Name Obfuscation (D)	1.00	0.89
			K-String literals Encrypter (D)	1.00	0.91

1. *[Node Spliter ⇒ Method Madness ⇒ Class Splitter]* Average Similarity Score of 99 apps : 0.915;
2. *[Class Splitter ⇒ Method Madness ⇒ Node Spliter]* Average Similarity Score of 96 apps : 0.906;

Both cases have the same three obfuscators but at a different serializing order. Although they can slightly reduce the average similarity scores by ViewDroid compared to the solely applying the obfuscator *Method Madness* case (with score 0.92), these scores are both above .90, sufficiently large for the obfuscated apps to be detected. *Case* 1 reduces the average score from 0.92 to 0.915, which shows that applying serialized multiple obfuscators has only slightly higher influence on ViewDroid than applying a single obfuscator. For *Case* 2, the average similarity score is a little bit lower than *Case* 1. However, there are four apps that cannot finish the whole serialized obfuscations. This indicates that although serialized obfuscation is slightly more powerful, the attacker has to take the risk of ending up with incomplete obfuscation. We encountered more failures when performing other orders of serialization. Overall, our evaluation demonstrates that multiple obfuscations are hard to be serialized, and even if successfully performed, they have little impact on ViewDroid's detection capability. Huang et al. [19] also reported that, in some scenarios, applying multiple obfuscations can lower the similarity scores reported by tools such as *AndroGuard*. Our experiment shows that ViewDroid's high-level abstracted birthmark is not affected much by the low-level (multiple) code obfuscation.

6. DISCUSSION

6.1 Attack Analysis on ViewDroid

As discussed in Section 3, based on different repackaging purposes, ViewDroid might face various types of attacks.

- **Lazy attack:** In this attack, the attacker does not change the functionality of original apps but applies automatic code obfuscation tools to repackage an app. As a result, a lazy attack does not change the view navigation relations of an app. In addition, code obfuscation has little impact on the feature view graph generation, as demonstrated by evaluation in Section 5.2. Therefore, ViewDroid can effectively detect such attacks.

- **Amateur attack:** An attacker not only applies automatic code obfuscation but also makes small modifications on the functionalities. The feature view graph could be changed slightly. However, because we use the subgraph isomorphism algorithm to compare graphs, small changes of the view graph may reduce the similarity score a little but will not affect the overall detection result much. As a result, ViewDroid can tolerate small changes on app functionalities and views.

- **Malware:** An attacker inserts some malicious payload into the original program while trying to make the repackaged app look the same or similar to the original one in order to leverage the popularity of the original program for wide propagation. Clearly, their feature view graphs would also be very similar. Therefore, ViewDroid can effectively detect such repackaging.

Other Potential Professional attacks: An attacker, who knows ViewDroid, may attempt to change feature view graphs to evade detection. Attackers may (1) insert a dummy view into the path of two directly connected views; (2) split one view node into two view nodes; (3) self implement or obfuscate the invocation of `startActivity()` and `startActivityForResult()` functions. Since we use the subgraph isomorphism algorithm with a certain matching threshold (e.g., 0.7), in order to affect the detection result, attackers need to modify many views of the original apps. On one hand, it will significantly increase the workload of repackaging an app. On the other hand, the dummy nodes, edges and self-implemented functions will increase the code size and decrease the performance of apps. We have not seen such attacks in the real world yet.

ViewDroid helps defenders stay ahead of the current arms race with attackers.

6.2 Limitations and Future Work

ViewDroid can detect the repackaging of non-trivial apps effectively, but for the detection of apps with few views, more false positives may be reported. Even so, the API vector node features can significantly reduce such false positives, because only nodes with very similar API vectors can be matched.

ViewDroid can effectively detect the following three types of mobile app repackaging attacks: lazy attacks, amateur attacks and malware. However, some professional attacks can potentially change view graphs, regardless of the workload of attackers and the performance overhead of the repackaged apps. Dummy view insertion may be defeated by examining the trigger function of the view switches. If a switch is not triggered by user behavior, we can merge the target view with its predecessor/successor in the feature view graph. A similar strategy has been used by Chen et al. [10] to check for malicious behavior. In our future work, we are going to enhance ViewDroid to deal with such attacks.

As shown in our evaluation, ViewDroid has false negatives when the encrypter obfuscation is used. This is because encryption changes the code completely and hides all the static characteristics of an app. This is also a common problem of all static analysis based detection. To defeat against such attacks, dynamic analysis may be applied. However, dynamic analysis is not efficient enough to be used as a large-scale detection approach. This is the fundamental tradeoff between accuracy and performance. How to build a hybrid approach to leverage both dynamic and static analysis for encrypter obfuscation is also a very interesting and important topic.

7. RELATED WORK

Smartphone App Similarity Measurement. The smartphone app repackaging problem has drawn great attention from the research community. There are several relevant works on measuring the similarity between Android apps on code level. DroidMOSS [34] leverages fuzzy hash to detect app repackaging. A hash value is computed for each local unit of opcode sequence of the classes.dex, instead of computing a hash over the entire program opcode set. It can efficiently and effectively identify the opcode segments that were left untouched by the lazy repackager and works well when the bytecode is only manipulated at a few interesting points (e.g., the string names or hard-coded URLs). However, some obfuscation, such as noise injection, can evade the detection. DNADroid [14] proposed a program dependence graph (PDG) based detection approach, which considers the data dependency as the main characteristic of the apps for similarity comparison. The efficiency of the comparison is further improved in AnDarwin [15] by building semantic vectors from PDG for each method. In general, PDG is resilient against several control flow obfuscation techniques and noisy code insertion attacks that do not modify the data dependency. However, some specific data dependence obfuscations can be designed to evade this approach. For example, PDG can be changed by inserting intermediate variable assignment instructions into the code. Juxtapp [18] proposed a code-reuse evaluation framework which leverages k-grams of opcode sequences to build feature for the feature hashing approach. A sliding window will move within each basic block to map the features into bit vectors, which are further combined into a feature metric to help birthmark each app. This detection scheme is able to effectively detect different code reuse situations, including piracy and code repackaging, malware existence, vulnerable code. Special designed code manipulation can potentially destruct the normal opcode pattern of Dalvik bytecode in a very dense fashion. Chen et al. [9] proposed a novel app birthmark, which is the geometry-characteristic-based encoding of control flow graph. This approach can effectively and efficiently detect cloned code which is syntactically similar with the original code. However, it cannot deal with app repackaging using code obfuscation techniques.

Traditional software plagiarism detection is another category of literatures that are relevant to smartphone app repackaging detection. MOSS [27] applies local fingerprinting to detect source code plagiarism. Liu et al. [22] proposed a program dependence graph (PDG) based approach. Lim et al. [21] used stack pattern based birthmark. These static analysis methods require the source code and are vulnerable to some code obfuscations. Myles et al. [25] statically analyzed executables and used K-gram techniques to measure the similarity. This approach is vulnerable to instruction reordering and junk instruction insertion. The dynamic software birthmarks include core values based birthmark [20, 31], dynamic opcode n-gram birthmark [23], whole program path (WPP) birthmark [24], dynamic API birthmark [29] and system call based birthmark [30]. The dynamic methods are not efficient enough to be performed on a large scale plagiarism detection scenario, like Android app markets.

Smartphone App Security. There are several publications in this category related to ViewDroid. They leverage the user interface feature of the Android platform, but use it for different purposes. SmartDroid [32] leverages user interfaces to find user interactions that will trigger sensitive APIs. It combines the static analysis and dynamic analysis. Chen et al. [10] developed a Permission Event Graph (PEG) to detect, or prove the absence of malicious behavior that is not authorized by users. Zhou et al. [33] proposed a module decoupling method to partition an app's code into primary and non-primary modules and thus to identify the malicious payloads reside in the benign apps. They also develop an approach to extracting feature vectors from those piggy backed apps to help improve the efficiency of the piggyback relationship detection. Our approach pays more attention to the repackaging detection from the primary functionalities of the Android apps and takes into account obfuscation resilience. We also identify certain features for the nodes and edges during the view graph construction to improve the efficiency of our detection.

8. CONCLUSION

In this paper, we proposed a user interface based Android app repackaging detection method, ViewDroid. The evaluation results show that ViewDroid can effectively detect Android app repackaging with the presence of various obfuscation techniques. ViewDroid is also efficient enough for performing large-scale experiments.

9. ACKNOWLEDGMENTS

This research was supported in part by the NSF Grant CCF-1320605, NSF Grant CNS-1223710, ARO W911NF-09-1-0525 (MURI), and ARO W911NF-13-1-0421 (MURI).

10. REFERENCES

[1] Android-Apktool: A tool for reverse engineering Android apk files. http://code.google.com/p/android-apktool/.

[2] Dexguard. http://www.saikoa.com/dexguard.

[3] Intent android developers. developer.android.com/reference/android/content/Intent.html.

[4] KlassMaster. http://www.zelix.com/klassmaster/docs/index.html.

[5] Number of avaliable Android applications. http://www.appbrain.com/stats/number-of-android-apps.

[6] Proguard. http://developer.android.com/tools/help/proguard.html/.

[7] Security alert: New stealthy android spyware - plankton - found in official android market. http://www.csc.ncsu.edu/faculty/jiang/Plankton/.

[8] Smali: An assembler/disassembler for Android's dex format. http://code.google.com/p/smali/.

[9] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *36th International Conference on Software Engineering (ICSE)*, 2014.

[10] K. Z. Chen, N. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. X. Wu, M. Rinard, and D. Song. Contextual policy enforcement in Android applications with permission event graphs. In *NDSS'13*, 2013.

[11] C. Collberg, G. Myles, and A. Huntwork. Sandmarks - a tool for software protection research. In *IEEE Security and Privacy, vol. 1, no. 4*, 2003.

[12] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, 1997.

[13] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(10), 2004.

[14] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *ESORICS*, pages 37–54, 2012.

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

[16] A. Desnos and G. Gueguen. Android: From reversing to decompilation. In *Black hat 2011, Abu Dhabi*.

[17] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. Adrob: Examining the landscape and impact of Android application plagiarism. In *Proceedings of 11th International Conference on Mobile Systems, Applications and Services*, 2013.

[18] 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 *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2012.

[19] H. Huang, S. Zhu, P. Liu, and D. Wu. A framework for evaluating mobile app repackaging detection algorithms. In *Proceedings of the 6th International Conference on Trust & Trustworthy Computing*, 2013.

[20] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu. Value-based program characterization and its application to software plagiarism detection. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 756–765. ACM, 2011.

[21] H. Lim, H. Park, S. Choi, and T. Han. Detecting theft of Java applications via a static birthmark based on weighted stack patterns. *IEICE - Trans. Inf. Syst.*, E91-D(9), 2008.

[22] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: detection of software plagiarism by program dependence graph analysis. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006.

[23] B. Lu, F. Liu, X. Ge, B. Liu, and X. Luo. A software birthmark based on dynamic opcode n-gram. *International Conference on Semantic Computing*, 2007.

[24] G. Myles and C. Collberg. Detecting software theft via whole program path birthmarks. *Information Security*, 3225/2004, 2004.

[25] G. Myles and C. Collberg. K-gram based software birthmarks. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, 2005.

[26] J. Ostrander. *Android UI Fundamentals: Develop and Design*. Peachpit Press, 2012.

[27] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, 2003.

[28] D. Schuler, V. Dallmeier, and C. Lindig. A dynamic birthmark for Java. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007.

[29] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K. ichi Matsumoto. Dynamic software birthmarks to detect the theft of windows applications. In *Int. Symp. on Future Software Technology*, 2004.

[30] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Detecting software theft via system call based birthmarks. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 149–158. IEEE, 2009.

[31] F. Zhang, Y. Jhi, D. Wu, P. Liu, and S. Zhu. A first step towards algorithm plagiarism detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012.

[32] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. SmartDroid: an automatic system for revealing UI-based trigger conditions in Android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104. ACM, 2012.

[33] 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*, pages 185–196. ACM, 2013.

[34] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, 2012.

[35] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. *Security and Privacy, IEEE Symposium on*, 2012.