DroidEagle: Seamless Detection of Visually Similar Android Apps

Mingshen Sun, Mengmeng Li, John C.S. Lui
Department of Computer Science and Engineering
The Chinese University of Hong Kong
{mssun, mmli, cslui}@cse.cuhk.edu.hk

ABSTRACT

Repackaged malware and phishing malware consist 86 % [35] of all Android malware, and they significantly affect the Android ecosystem. Previous work use disassembled Dalvik bytecode and hashing approaches to detect repackaged malware, but these approaches are vulnerable to obfuscation attacks and they demand large computational resources on mobile devices. In this work, we propose a novel methodology which uses the layout resources within an app to detect apps which are "*visually similar*", a common characteristic in repackaged apps and phishing malware. To detect visually similar apps, we design and implement `DroidEagle` which consists of two sub-systems: `RepoEagle` and `HostEagle`. `RepoEagle` is to perform large scale detection on apps repositories (e.g., apps markets), and `HostEagle` is a lightweight mobile app which can help users to quickly detect visually similar Android app upon download. We demonstrate the high accuracy and efficiency of `DroidEagle`: Within 3 hours `RepoEagle` can detect 1298 visually similar apps from 99 626 apps in a repository. In less than one second, `HostEagle` can help an Android user to determine whether a downloaded mobile app is a repackaged apps or a phishing malware. This is the first work which provides both speed and scalability in discovering repackaged apps and phishing malware in Android system.

1. INTRODUCTION

Smartphones and tablets have become indispensable devices in our daily life. Due to their affordable prices and variety of models from different manufacturers, the market share of Android-based devices reached 78.4 % in 2013 [14]. Mobile applications (app for short) play a significant role in the Android ecosystem. Currently, there are around one million apps [9] in the Google official market (i.e., Google Play). However, unlike the Apple's iPhone/iPad system wherein developers can only distribute their apps on the App Store, Android system allows developers to distribute their apps in many third-party markets (e.g., websites, forums, or vendors' apps markets). Many of these third-party markets do not have a comprehensive apps review, and they are not as prudent as the Google Play in publishing or screening mobile apps. Hence, these third-party markets provide perfect breeding ground for cracked and malicious apps, such as phishing malware or trojans. According to a recent mobile threat report [18], Android accounted for 98 % of all mobile threats, and 99.9 % [12] came from many third-party markets. If left unaddressed, this chaotic Android ecosystem will harm both users and app developers.

WiSec '15, June 22 - 26, 2015, New York, NY, USA

ACM 978-1-4503-3623-9/15/06 ...$15.00
http://dx.doi.org/10.1145/2766498.2766508

Android malware contains trojan, backdoor, adware, fake apps, etc. According to a recent study [35], 86 % of these malware are using the *repackaging technique*, which is to disassemble a legitimate app (using some well-known tools [1, 3, 6]), and hackers can then add or modify logics of the original app, and then assemble it back and distribute the modified app in third-party markets. Using the repackaging technique, hackers can perform various malicious functions, e.g., they can crack paid apps to bypass payment function or they can replace developers' advertisement IDs. Moreover, malware writers can insert malicious logics into popular apps to deceive users and distribute malware widely. Besides repackaged malware, phishing malware attempts to acquire sensitive information (e.g., account password and credit card number) by masquerading as trustworthy bank apps, online shopping apps, or payment apps, etc. In summary, to secure the Android ecosystem, repackaged malware is the first form of Android malware that security researchers need to address.

To protect installation and update of apps, Android operating system provides a signing mechanism [2]. Developers should sign apps with certificates using their private keys before publishing their apps. However, many of these certificates are not issued by a certificate authority. This means that anyone can issue a certificate using his or her private key to publish an app. Hence, a signed certificate cannot absolutely ensure the authenticity of an app, and one cannot distinguish between fake app and real app by simply checking the existence of certificates. Therefore, this mechanism cannot effectively ensure the authenticity of an app.

Because repackaged apps consist 86 % of all Android malware [35], researchers have proposed two different approaches to detect repackage apps. Since a repackaged app is built from an original app with only *minor* modifications, so the majority of the app's functionalities and corresponding instructions are the same as the original one. Hence, one de-

tection approach is based on the "*instruction sequences*" of disassembled codes. Systems fall into this category include DroidMOSS [34] and DroidAnalytics [30], which use fuzzy hashing technique for similarity comparison so as to find repackaged apps. However, because this approach is sensitive to the instruction sequences of disassembled codes, it cannot defend against code obfuscation. For example, if hackers generate repackaged apps by adding some useless codes into the original app, the hash value of this modified app will be totally different from the original app. The second approach is based on the "*semantic information*" of disassembled code. Systems fall into this category include DNADroid [10], in which the system generates semantic program graphs (e.g., call reference graph) based on disassembled code, and the system uses subgraph isomorphism algorithms to determine similar graphs as that of the original app. Note that subgraph isomorphism algorithm is not scalable when one needs to deal with millions of apps. Because both approaches need to disassemble apps before analysis, this process takes a long time so they are not suitable for large volume apps analysis. Finally, hackers can use programming tricks [23] to easily bypass existing disassembling tools. In summary, existing approaches focus on the disassembled source code, it is not scalable and can be easily bypassed by code obfuscation.

The core idea of our repackage malware detection is as follows. We observe that many of these repackaged apps aim to modify logics of original apps. But to avoid being detected by users, repackaged apps need to have "*similar appearance*" as original apps. Furthermore, phishing malware relies on similar appearances as banks or shopping apps to deceive users. Therefore, all these repackaged malware need to have similar *visual characteristics* as the original apps. So by comparing their "*visual similarity*", one can quickly determine potential repackaged malware or phishing malware.

In this work, we propose `DroidEagle` and the system is based on visual characteristics to detect similar Android apps. `DroidEagle` has two sub-systems. One sub-system is for large scale apps repository analysis, while the other sub-system is for host-based detection. As we will demonstrate, these two sub-systems can accurately detect repackaged apps and phishing malware. Another major advantage as compared to some existing code-based malware detection systems is that `DroidEagle` is immune to code obfuscation and also efficient in large scale detections. We make the following contributions:

- We propose two methodologies to find visually similar apps in app repository and Android device respectively. Because both methodologies are based on visual characteristics instead of programming semantics, they are fast and accurate in detecting visually similar apps including repackaged apps and phishing malware. To the best of our knowledge, we are the first to use visual resources in Android systems to detect repackaged malware and phishing apps, and the methodologies provide scalability, speed and accuracy.
- We implement `RepoEagle` for Android apps repository analysis. `RepoEagle` can detect visually similar apps in a large scale app database. In particular, we were able to find 1298 visual similar apps from 99 626 apps from different third-party markets. The pre-processing time of `RepoEagle` is two times faster than previous code-based detection methods.
- We implement `HostEagle` to detect visually similar apps on Android devices. The detection time is around one second for each app, and it does not demand much computational resource, which makes it ideal to deploy on mobile devices.

The rest of the paper is organized as follows: In Section 3, we propose our detection methodologies and present our system implementation. Experimental results are presented in Section 4. In Section 5, we present the related work. Conclusion is given in Section 6.

2. BACKGROUND

In this section, we introduce some essential background about the Android app markets, Android app package file structure and user interface of Android apps.

2.1 Android App Markets

Android app market is an Internet site for developers to distribute their apps. Developers publish their apps on the market, and users can purchase apps of their liking. There are also free apps available. For this type of apps, developers usually gain profit by advertisement display or in-app purchases. Google Play [17] is the official app market for Android. However, there are many manufacturers who produce Android devices. In order to provide more features on their devices, these manufacturers provide vendor-customized firmwares (based on Android) and pre-install their own app markets. For instance, one can find app markets like Samsung Apps, HTC Hub or appXtra from Sony. These official markets will review any submitted apps, and scan these apps with their own proprietary detection systems before publishing to prevent malicious apps leakage. For instance, Google Play uses Bouncer [5] to scan newly submitted apps. Besides these official markets, there are many other third-party markets and forums (e.g., SlideMe, hiapk and AppChina) which provide Android apps download. According to the recent report [25], a large number of Android devices do not have the Google official market app. Furthermore, Google services are either blocked or have poor download performance in some countries. So users have to resort to third-party markets or forums for apps download. Moreover, some third-party markets intentionally publish cracked paid apps, no-ads apps and modified apps to attract users. Due to the variety of apps in third-party markets, it was reported that 72.6 % of apps are downloaded from these third-party markets [24]. It is also worth mentioning that these third-party markets have much less restrictions on publishing and provide no apps review. Hence, these markets become the perfect breeding ground for a large number of repackaged apps and phishing malware.

2.2 Android App Package File

Android app package file (i.e., `APK` file) is a single installation zip file for an Android app. It contains `classes.dex`, `AndroidManifest.xml` files and `lib`, `res`, `META-INF` directories. `classes.dex` is a Dalvik executable file (i.e., `DEX`

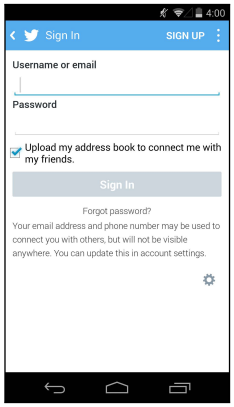
file) compiled from Java source code. This DEX file provides bytecodes for running on the Dalvik virtual machine. Several tools [6, 3, 1] can disassemble DEX bytecode to human-readable source code, so it is easy to reverse engineer an Android app. AndroidManifest.xml describes the name, version, permission usage and related app information. lib directory contains the native code for different platforms, res directory contains app resources such as layout structures (in /res/layout directory) and images (in /res/drawable directory). These files are in binary XML format and there are tools [3, 1, 4] which can translate these binary XML format to plain text XML format (again, making reverse engineering easy). The META-INF directory contains certificate of an app and signatures of each file in the APK file.

2.3 Android App User Interface

The basic elements of user interface in an Android app are View and ViewGroup. A View is an object on the screen which can interact with users and display objects. Android SDK provides several View objects such as Button, EditText, ImageView, etc. For example ImageView can load a image from an app's package. ViewGroup is an object container which holds other View or ViewGroup. ViewGroup (e.g., ScrollView, RelativeView, and LinerLayout in Android SDK) can define the layout arrangement of its elements. For example, LinearLayout is a ViewGroup specifying all the elements inside should either be horizontally or vertically displayed in a single direction. A layout structure is a hierarchy of View and ViewGroup which defines the visual user interface on the screen. An app can initiate *interface objects* in runtime. Alternatively, developers can also define a layout structure of the user interface in an XML file under the /res/layout directory so as to separate presentation and function logics.

Figure 1 illustrates a user interface and the corresponding layout file. There are two kinds of elements in the layout files. One is *empty-element*. This kind of element cannot include other elements. For example, "<CheckBox .../>" tag is an empty element. The other kind of element is *start-end element* which can include other elements. The *start-end element* contains a pair of tags, e.g., the LinearLayout element begins with the "<LinearLayout orientation= ...>" tag and ends with the "</LinearLayout>" tag. This start-end element is the *parent element* of any included elements, which are represented as *child elements*. The name of an element is the *first word* in the tag (e.g., "LinearLayout" in <LinearLayout ...> tag) and the following key/value pairs (e.g., orientation="vertical") are attributes of the element.

To illustrate, consider the layout file in Figure 1. The empty element corresponds to the View object (those under blue color), and the start-end element corresponds to the ViewGroup object (those under gray color). Note that developers can put layout files in other directories to support multiple screen types. The naming scheme of the directory will determine the screen type (e.g., size, density and orientation). For example, /res/layout-large directory contains layout files for large screen (i.e., screen size is at least 470dp×320dp). Layout XML file defines layout structure and element attributes including positions, width, height, etc. Because there may be multiple screens for an app, layout




```
<?xml version="1.0" ... ?>
<LinearLayout orinentation=...>
  <ToolBar ... />
  <!-- The tool bar in the top. -->
  <ScrollView background=...>
    <LinearLayout ...>
      <TextView text=... />
      <!-- Username or email. -->
      <EditText inputType=... />
      <LinearLayout ...>
        <TextView ... />
        <!-- Password field -->
        <EditText ... />
        <CheckBox ... />
        <!-- "Upload my ..." -->
      </LinearLayout>
      ...
    </RelativeLayout>
</LinearLayout>
```

Figure 1: The "Sign in" user interface of Twitter and corresponding layout file (login.xml).

directory may contain more than one layout XML file.

3. DETECTION METHODOLOGIES

In this section, we present our methodologies of how to detect visually similar apps, and how these methodologies can be effective to detect repackaged apps. We also present our system implementation and illustrate it via two usage scenarios: *app repository analysis* and *host-based detection*.

3.1 Overview

Our goal is to determine visually similar apps so as to detect repackaged apps and phishing malware. For our detection system, we set the following design goals:

- **Accuracy**. Accuracy is a basic requirement for a detection system. Our methodology should accurately determine visually similar apps so as to detect repackaged apps and phishing malware. Furthermore, code obfuscations should not affect our detection capability. Moreover, our methodology should be robust against modifications of app user interface.
- **Efficiency and Scalability**. The number of apps in third-party markets is over one million and is growing rapidly. Therefore, our methodology needs to be scalable to handle a large number of apps for repository analysis.
- **Flexibility**. The methodology will be used to analyze apps in third-party markets to determine visually similar apps. Furthermore, because users can download and install apps from third-party markets or some web/blog sites, our methodology should be able to perform real-time detection on an Android device for the app download so as to defend against malicious and repackaged apps' installation. Note that mobile devices are limited in storage and computation resources, therefore, our methodology needs to be flexible for both large scale analysis and real-time detection on resource constrained devices.

To addresses these challenges, we propose two methodologies: *repository analysis* and *host-based detection*. We implement two systems based on the methodologies for these

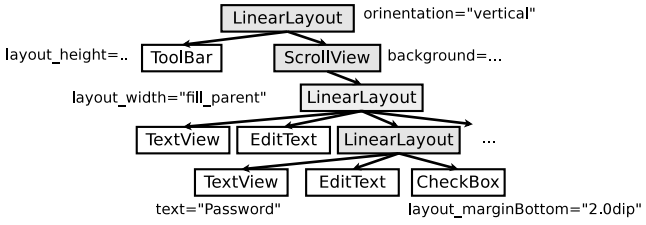


Figure 2: Layout tree of the layout file in Fig. 1

two usage scenarios. The methodologies are based on *visual resources* in an Android app. Visual resources are *layout files* and *drawable images* in an app package file. Layout files define user interfaces of an app, while drawable images will be used as a supplementary detection. Because visually similar apps have similar user interfaces, we can use this characteristic feature to detect repackaged apps. Layout file contains a hierarchy of `View` and `ViewGroup` objects. We formally represent a layout file as a data structure which we call *layout tree*. Let us first formally define the concept of a layout tree which we use in our methodologies.

Definition 1. *A layout tree is a tree data structure over a layout file where:*

- *A node in the layout tree represents an element in the layout file. The node name is the corresponding element name and the node values contain attributes of the corresponding element.*
- *The parent/child relationship of nodes in a layout tree is the same as that in the layout file. For example, if a `ViewGroup` object contains other objects like `View` or `ViewGroup`, then the `ViewGroup` is the parent node of these `View` or `ViewGroup` objects.*

Let us take the layout file in Figure 1 as an example. We can represent this layout file as a layout tree and it is illustrated in Figure 2. In this layout tree, each node has node values and it corresponds to an element with a element name and attributes in the layout file. For example, the node "`ScrollView`" in the layout tree corresponds to the "`ScrollView`" start-end element in the layout file. The "`CheckBox`" node corresponds to the "`CheckBox`" empty element in the layout file with `layout_marginBottom="2.0dip"` attribute as the node value.

We can use this layout tree to detect visually similar apps. Firstly, the structure of a layout tree defines the visual structure of an app's user interface. Secondly, repackaged apps and phishing malware have similar layout trees because they need to rely on the appearance of the original app, in particular, same user interface, so as to deceive users. Thirdly, nodes in a layout tree with detailed attributes (e.g., positions, width and height) accurately represent visual appearances for each element in the screen. Fourth, we can obtain layout files from an app easily by extracting from them from the unzipped package file. This is much easier than previous proposals which need to go through the disassembling process of an entire app, therefore, our methodology requires much less computational resource as compared to previously proposed methods of using disassembled code. Finally, slight modifications of a layout structure can easily mess up the user interface, so hackers cannot easily obfuscate layout files. In the following sections, we present the layout-tree-based detection methodology, as well as our system implementations for detecting visually similar apps both in a repository and in Android smartphones.

3.2 RepoEagle: Repository Analysis

Repository analysis is to analyze all apps in an app repository (say, on some third-party markets) so as to discover all visually similar apps. We propose a methodology to calculate layout similarity for detection. Based on this methodology, we design a system, `RepoEagle`, which can accurately determine the visual similarity of two apps, thereby further detecting repackaged apps and phishing malware.

3.2.1 Layout Edit Distance

We define *layout edit distance* (LED for short) as a metric to measure the similarity between two layout trees. Given two apps, we first represent their layout files as two layout trees, and the LED is the *minimum number of operations* required to transform from one layout tree to another tree. The operations in layout tree transformation are *node deletion*, *node insertion* and *node substitution*. Figure 3 illustrates these three transformation operations. For example, a deletion operation is issued to delete the node "`CheckBox`" from layout A and it results in layout B. From the screenshots, the check box is deleted from the user interface. A node insertion operation of the "`TextView`" node is performed on layout A which results in layout C (which is highlighted in red in the figure). Finally, a node substitution operation of the node "`LinearLayout`" is performed on layout A which results in layout D. This substitution operation causes the nodes to have a messy display.

When the LED between two layout trees is equal to n, it means that there are *at least* n operations performed to a layout tree so as to transform it to another layout tree. Therefore, LED describes the dissimilarity of two layout trees, and a small LED value implies that the two layout trees are very similar. In Figure 3, the LED values between layout tree A and B (or C or D) is equal to 1.

3.2.2 Repository Analysis System

Based on the LED metric, we implement `RepoEagle`, a repository analysis system to detect visually similar apps in apps repository. Figure 4 illustrates the design of `RepoEagle`. There are two inputs to our system. The first input is a set of apps which are from an app repository (these apps are retrieved by our crawler from an official market or third-party markets). The second input is an official app from a trusted authority (e.g., Google Play). `RepoEagle` will determine apps within the repository which are visually similar to the official and trusted app.

`RepoEagle` consists of four main components: (1) *layout tree extractor*, (2) *certificate extractor*, (3) *similarity comparator* and (4) *certificate verifier*. Layout tree extractor extracts layout files from an `apk` file and translate the binary XML

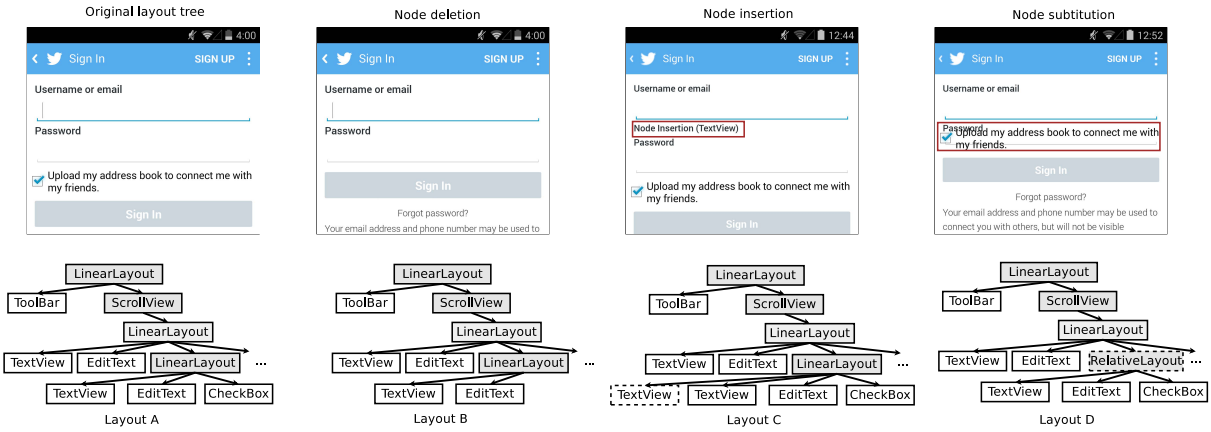


Figure 3: Three operations of layout tree transformation on the "Sign in" user interace of Twitter

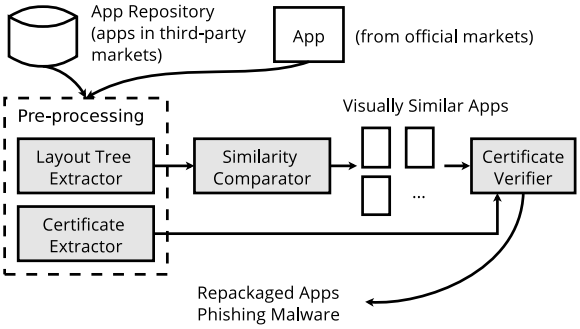


Figure 4: Architecture of `RepoEagle`.

files to plain text. Certificate extractor extracts certificate in the `APK` file. Layout tree extractor and certificate extractor pre-process the input apps for further detection. Similarity comparator will use the layout edit distance algorithm to determine the similarity distance between two layout trees. By comparing the layout in the repository, similarity comparator will find out a set of similar apps. Certificate verifier will determine the fake apps by verifying the certificate of these similar apps.

3.2.3 *Implementation*

In the pre-processing stage, because layout files in `APK` files are in binary `XML` format, layout extractor uses `apktool` to translate these files to plain `XML` files. Since some layout files use "`<include />`" or "`<merge />`" statement to include other layout files. Hence, the layout tree extractor needs to first combines all referenced layout files into a single file. Furthermore, when constructing a layout tree, `RepoEagle` will not consider any invisible `View` objects. For example, nodes with "`visibility="invisible"`" or zero width and height attribute. This is because this kind of objects will not display on the user interface, and hackers often like to use this trick to add a number of invisible nodes with the hope to bypass detection. Finally, the layout tree extractor stores all plain layout trees of the apps in the repository into database. The layout tree extractor will also store the number of elements of an app into the database as a metadata. This metadata will help similarity comparator to optimize the comparison process (which we will elaborate next). For the certificate extractor, it unzips all `APK` files and extracts certificate files (e.g., `CERT.RSA` and `CERT.DSA`) in the `/META-INF/` directory. Then the system utilizes the `keytool` to extract certificate information such as issuer and certificate fingerprint, and stores the information in the certificate database. Note that the certificate information will be used in our detection to identify the authority of the apps.

In the similarity comparator module, we implement the LED metric based on Zhang and Shasha [28] tree edit distance algorithm to calculate the visual similarity score among apps. The algorithm can achieve $O(m^2n^2)$ time complexity with $O(mn)$ space complexity, where m and n is the number of nodes in two trees. The similarity score of two apps is the LED between the layout trees of these two apps. Note that an app may have multiple layout trees, therefore, one should compare all pairs of the two apps. Suppose there are m layout trees in app A and n layout trees in app B. The similarity score between A and B is the minimum number of all layout edit distances among mn layout tree pairs. Because there can be many comparisons, we utilize two methods to *optimize* our comparison process. Firstly, some layout trees may only contain few nodes, the result of LED cannot accurately represent the visual similarity of two layout trees. Hence, we will not use those layout trees whose number of nodes is less than a threshold, say $\mathcal{N}$, for LED comparison. Secondly, we can use the number of elements to reduce the comparison pairs. Suppose we want to find apps in the repository whose similarity distance between the input app is lower than a threshold $\mathcal{T}$. Before comparing the two layout files, the layout comparator will first compare the element number in these two files. If the difference between these two numbers is larger than $\mathcal{T}$, the system will not compare these pairs. This is because according to our layout edit distance, insertion is one of the transformation operations. The difference (say d) on the number of elements in two layout files represents that one needs at least d insertion operations to transform into another. Because we store the number of elements as metadata in the database during the pre-processing stage, the layout comparator will not compute this number every time. Using these two optimization methods, we significantly reduce the number of layout tree comparison. We will present the effectiveness of

these optimization techniques in Section 4.

For the certificate verifier, we use certificate information including issuer and certificate fingerprint from certificate database. Certificate fingerprint is the cryptographic hash value of the certificate. Apps published by the same developers will have the same certification fingerprint. The certificate verifier compares the fingerprints of visually similar apps to the fingerprint of the input official app. In Section 1, we discussed that if a hacker repackages an app or creates a phishing app, he needs to sign this app using his own private key. Hence, if an app has a similar appearance with the input app (i.e., the official app) but has a different certificate fingerprint, this means that the app is published by other unknown third party (e.g., hackers).

3.3 HostEagle: Host-based Detection

Because users may install apps from unknown sources (e.g., shared file in cloud storage, forums or third-party markets), one cannot ensure the safety and authenticity of these apps. We develop `HostEagle`, a host-based detection system for Android devices. `HostEagle` can detect repackaged apps and phishing malware based on visually similar detection on mobile devices. Due to the computation and storage limitation of smartphones, we cannot directly use the previously proposed disassembling approaches in smartphones. Hence we design a system based on layout hashing (LH) method. This method not only has an accurate detection rate and fast response time, but also has low computation demand on mobile devices.

3.3.1 Layout Hashing

Layout hashing (LH) is a method to generate layout hash (i.e., a hash value) for a layout tree. The layout hash values of visually similar layout trees are likely to be same. Figure 5 illustrates the process of generating a layout hash of a layout tree. It consists of two steps: (1) leaf pruning and (2) tree hashing. Firstly, we prune all leaves in the layout tree. Leaf nodes in a layout tree consist of different kinds of `View` objects. After removing all `View` objects in the leaves, the remaining subtree contains all `ViewGroup` objects representing the *layout skeleton* of the user interface. For example, in Figure 5, we prune the leaves (i.e., "`ToolBar`", "`CheckBox`", "`TextView`" and "`EditText`" objects) in the original layout tree and get a subtree with all `ViewGroup` objects. The second step is to generate a hash value for the remaining layout tree. For each node, we generate a node hash value by hashing the concatenated string of node name and specific node attributes. Because there are many attributes of one node, some of them do not have visual effects. Therefore, to make LH value more accurate, we only choose node attributes which have visual effects. In our implementation, we use "`width`", "`height`", "`text`" attributes for the calculation of a node hash value. To keep the attributes in order, we first sort these attributes and then concatenate them as a string. For each layer, we concatenate all node hash values in this particular layer and generate a layer hash value. Lastly, we obtain the final layout hash value by hashing the concatenated layer hash values of each layer. Algorithm 1 summarizes the process of generating layout hashing. The algorithm generates one single hash value representing this whole layout file. For implementation, one can use a cryptographic hash function (e.g., MD5 and SHA1) to compute

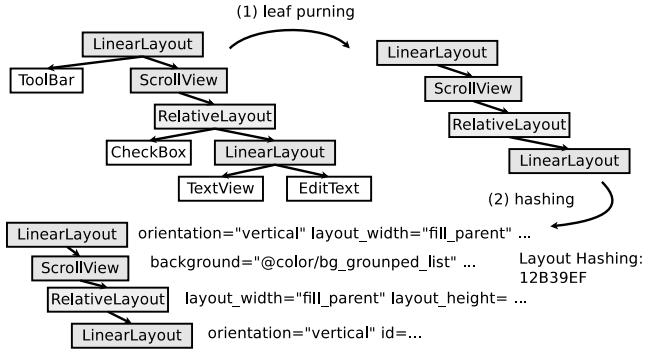


Figure 5: Layout Hashing

Algorithm 1 Layout Hashing

```
1: procedure HASH LAYOUT(layoutTree)
2:     initialize layerHashList
3:     for all layer in layoutTree do
4:         initialize nodeHashList
5:         for all node in layer do
6:             nodeValues ← getNodeAttributes(node)
7:             sortedNodeValues ← sort(nodeValues)
8:             valueStr ← concatenate(sortednodevalues)
9:             nodeHash ← hash(valueStr)
10:            append nodeHash to nodeHashList
11:        end for
12:        append nodeHashList to layerHashList
13:    end for
14:    return hash(sortedLayerHashList)
15: end procedure
```

hash values of string. The time complexity of generating a layout hashing is $O(V+E)$, where V is the number of nodes and E is the number of edges in layout tree.

There are several reasons why we use the layout hashing algorithm to detect visually similar apps on Android devices. Firstly, layout hashing generates a unique hash value for a given layout structure, and this hash value is easy and fast to query and compare. Secondly, the algorithm prunes all leaves in the layout tree before hashing. We need the pruning step because visually similar layout trees are likely to have the same layout structure, but some `View` objects (which are the leaf nodes) may have different names and attributes. So by removing these leaf nodes, we generate a hash value only for the layout skeleton. Hence, any modifications of the leaves (i.e., `View` objects) will not affect the final hash value. In addition, because any modifications on the remaining tree (i.e., pruned layout tree) will dramatically change the appearance of a user interface, which defeats the requirement of creating a repackaged app since it aims to appear as similar as the original and legitimate app. Therefore, this hash value can accurately represent the user interface of an app. Besides, the algorithm requires low computation resources and the transmission of a hash value over the network consumes only a small amount of network bandwidth. Another important feature of our algorithm is that it can be easily implemented on smartphones. We will evaluate the effectiveness of this algorithm in Section 4.

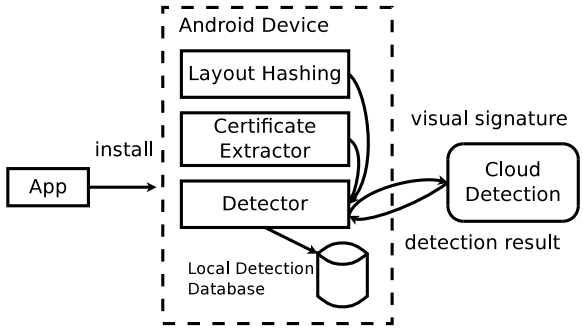


Figure 6: Architecture of `HostEagle`.

3.3.2 *Host-based Detection System*

Based on the layout hashing algorithm, we design `HostEagle`, a host-based detection system to actively defend against repackaged apps and phishing malware. Figure 6 shows the architecture of `HostEagle`. The system monitors app installation events and then generates a layout hash for the installed app. Using the layout hash value and its certificates, `HostEagle` can verify the authority of the app. `HostEagle` can conduct detection locally on the smartphone if there is no network connection. Due to the flexibility of our hashing algorithm, `HostEagle` can also upload the hash to a remote server for cloud-based detection. The detection consumes low network bandwidth and provides accurate result.

`HostEagle` consists of five building blocks: (1) layout hashing, (2) certificate extractor, (3) detector, (4) local detection database and (5) cloud detection server. Layout hashing generates layout hashes for every layout file of an installed app. Certificate extractor obtains certificate fingerprint from the app package. We represent layout hashes and certificate fingerprint as visual signature. With this visual signature, detector first detects in its local detection database. The local detection database contains visual signatures of apps which are frequently repackaged (e.g., popular paid games) and sensitive apps (e.g., banking apps and shopping apps). In our current implementation, the size of the local database is about 100KB. The detector will compare the visual signature of installed app with that in local detection database. If the layout hashes are same but certificate fingerprints are not, the detector will notify users that the installed app are most likely a repackaged app or phishing malware. If the local detection database does not contain the layout hashes of the installed app. The detector will upload visual signature to cloud server for further detection.

3.3.3 *Implementation*

In our implementation, we first extract visual signatures for apps from trusted authority (e.g., Google Play) and store them in the cloud detection server. For simplicity, we put visual signatures of the top five hundred apps from the Google Play in the local detection database on an Android device. For the system in Android devices, it listens to the `android.intent.action.PACKAGE_INSTALL` and `android.intent.action.PACKAGE_ADDED` broadcasts so as to monitor the installation event. When receiving these broadcasts, the system generates layout hashes for every layout file and certificate fingerprint in the newly installed app as visual signature. We utilize SSL/TLS to encrypt the messages between detector and cloud detection server to secure the communication channel. Note that we do not need to modify the Android firmware or request a higher privilege (e.g., root privilege). `HostEagle` only requires package and network related permissions. Therefore, `HostEagle` can be easily deployed on users' devices to prevent repackaged and phishing malware.

Table 1: Apps Statistics

Category	Name	URL	# of Apps	Size
Official	Google Play	play.google.com	500	7.0 GB
Third-party	appchina	appchina.com	34 989	238 GB
	appfun	appfun.cn	12 427	154 GB
	hiapk	apk.hiapk.com	5287	87 GB
	android.d.cn	android.d.cn	4064	163 GB
	jimi168	jimi168.com	23 723	76 GB
	anzhi	anzhi.com	18 736	118 GB
Cloud Storage	Baidu	pan.baidu.com	200	3.5 GB
	Huawei	dbank.com	200	3.1 GB
Total			100 126	849.6 GB

3.4 Supplementary Detection

Note that some apps may not have any layout files or there are only few elements in their layout files. For example, some mobile game apps only use 3D engines to render the user interface or game graphics on the screen so these apps have few elements in their layout files. Hence, we cannot utilize the layout files for detection using the above two methodologies. To supplement our method, we utilize drawable images for detection. In our implementation, if there is no layout file or if the number of elements in layout files is less than a threshold, we use perceptual hash values (pHash) [26] of images in the `/res/drawable` directory as visual signatures for detection. pHash algorithm can tolerate less than 25 % modifications of the image. In Section 4, we will discuss the usage of this supplementary detection.

4. EXPERIMENTS & EVALUATION

In this section, we first present the analysis of `RepoEagle`, our apps repository detection system. Then we evaluate the effectiveness of `HostEagle` and demonstrate its effectiveness in detecting repackaged apps or phishing malware on Android devices.

4.1 Repository Statistics

We crawled and collected 100 096 apps from the Google official market and various third-party markets. Table 1 shows the statistics of the apps in our app repository. There are five hundred apps from a trusted authority (via Google Play). Other apps are obtained from various third-party markets and shared links from cloud storage.

Let us first discuss some characteristics of the layout files in our app repository. We analyze the disassembled layout files from 500 apps which is randomly selected from our repository. Figure 7 shows the distribution of the total number of elements in a layout file. Layout files are stored in two

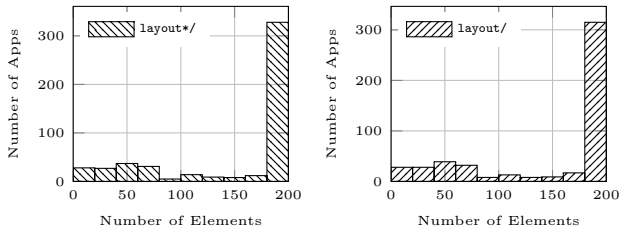


Figure 7: Distribution of the number of elements in layout files for an app from `layout*/` directory ( ) and `layout/` directory ( ).

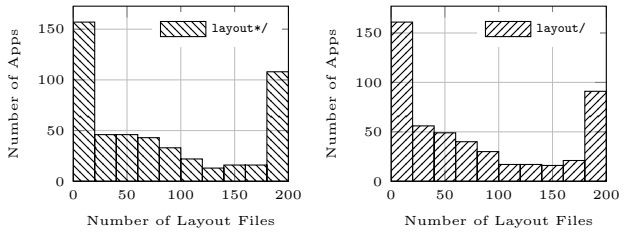


Figure 8: Distribution of the number of layout files for an app from `layout*/` directory ( ) and `layout/` directory ( ).

directories: (a) `layout*/` and (b) and `layout/`. The directory "`layout*/`" stores all layout files of that app packages for different screen sizes (e.g., `layout-xlarge/` and `layout-hdpi/` directory), while the directory "`layout/`" stores those files for general screen size. From Figure 7, one can observe that most of the apps contains more than two hundred elements within a layout file. Figure 8 shows that the number of layout files per application. The number of layout files are mostly distributed in two ranges. For general apps (e.g., tools or social networking apps), the number of layout files is between 1 to 50. On the other hand, game apps usually have above 200 layout files. Figure 9 shows the distribution of the average number of elements within a layout file in an app. Note that the average number of elements for most of the apps in our repository is between five to 15. From this analysis, we know that the visual resources (i.e., the number of layout files) and visual characteristics (i.e., the number of elements in layout files) in the majority of apps can be utilized for our LED and LH detection methodologies.

4.2 Experiments

We conduct an analysis on our repository with 99 596 apps which were obtained from third-party markets and cloud storage. We downloaded five hundred official apps from the Google Play, and analyze whether there is any visually similar apps within our app repository. Table 2 depicts our experimental results. There are 1159 visually similar apps found in third-party markets. Via static analysis, we discover that most of these apps are cracked games with unlocked in-app paid functions. We also use an anti-virus engine (i.e., Kaspersky) to scan these repackaged apps and discover there exist some malware within these repackaged apps. These results show that hackers like to put their repackaged apps and malware on the cloud storage, and they publish these URL links of the cloud storage to many mobile related forums and social media. This way, hackers not only can hide their identity, but at the same time, take advantage that cloud storage providers usually do not scan any uploaded files, so hackers can easily distribute their repackaged apps or malware.

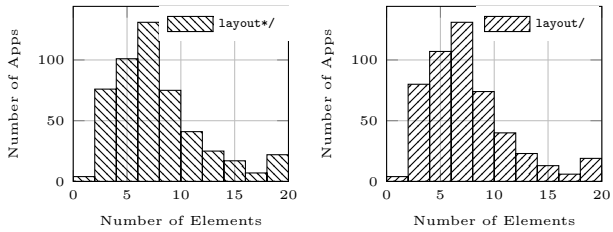


Figure 9: Distribution of the average number of elements in layout files for an app from `layout*/` directory ( ) and `layout/` directory ( ).

Table 2: Results of Repository Analysis

Market	# of Visually Similar Apps (Percentage)	# of Malware
Third-party Market	1159 (1.6 %)	10
Cloud Storage (Baidu)	50 (10.0 %)	0
Cloud Storage (Huawei)	89 (17.8 %)	15
Total	1298 (1.3 %)	25

Repackaged Apps. Let us now illustrate the effectiveness of LED (i.e., layout edit distance) and LH (i.e., layout hashing) methodologies. In our repository analysis, we find eight apps which are visually similar with the official app "`Angry Bird`" from the Google Play. Table 3 illustrates the information of these apps, including LED and LH as compared with the original app, certificate issuer and certificate fingerprint. Three apps have zero LED with the original app. However, they have different certificate issuers and fingerprints. We can confirm that these are repackaged apps. From the issuers, we discover that one app (ID: `666cc8f79a32cc8`) is repackaged by a tool called "`Virtuous Ten Studio`" [8] and another app (ID: `0f6f93bbc7c6f00`) is signed by a debug key from the Android SDK. Via static analysis, we find that `666cc8f79a32cc8` cracked the in-app paid function in the original "`Angry Bird`" so that users do not need to purchase any virtual commodities, and both `666cc8f79a32cc8` and `0f6f93bbc7c6f00` are reported as adware. Furthermore, `3b524dd4a7bbd2d` is reported as `DroidKungFu` trojan. This trojan will contact a remote server (`http://data.flurry.com/aap.do`) to download an updated file (`logo`). Once users install and open this update app, this app will gain the root privilege and copy itself to the `/system/` directory so that users cannot uninstall the app. This malware also has a number of backdoor functions (e.g., deleting files, installing `APK` and launch app), and can respond to a command & control server. One suspicious app (ID: `ed0e5232ded2314`) has an LED value equal to two as compared with the original app. We check the certificate fingerprint of this suspicious app and find out it is in fact a legitimate app signed by a trusted company. This app is in fact an updated legitimate app. It has an LED value of 2 is because the updated app contains similar "`login`" user interface, and has two additional `TextView` objects.

[1]For display purpose, we cut off the last 16 characters of the hash value. This will not affect the results shown in the table.

Table 3: Case Study of Repository Analysis for Angry Bird

App ID[1]	LED	LH[1]	Certificate Issuer	Cert Fingerprint [1]	Repackaged	Market	Detection Result
22d397b900f86e3	0	1cd5ec09fd1bfad7	Rovio Mobile Ltd.	5557CBE41C109409		Google Play	
233ce0378da3eec	0	1cd5ec09fd1bfad7	Rovio Mobile Ltd.	5557CBE41C109409		appfun.com	
5803487d1a44ccf	0	1cd5ec09fd1bfad7	Rovio Mobile Ltd.	5557CBE41C109409		appfun.com	
666cc8f79a32cc8	0	1cd5ec09fd1bfad7	Virtuous Ten Studio	A925D13B36571880	✓	appfun.com	Android.Adware. Jumptap.a
7a433c193466744	0	1cd5ec09fd1bfad7	Rovio Mobile Ltd.	5557CBE41C109409		appfun.com	
9ee436e7b365485	0	1cd5ec09fd1bfad7	databin	FC007C2E2C0A43CD	✓	appfun.com	
22d397b900f86e3	0	1cd5ec09fd1bfad7	Rovio Mobile Ltd.	5557CBE41C109409		android.d.cn	
0f6f93bbc7c6f00	0	1cd5ec09fd1bfad7	android-debug	264BF7D71E0EDC4F	✓	android.d.cn	Android.Adware. Dowgin
3b524dd4a7bbd2d	0	1cd5ec09fd1bfad7	keystore3	990B1C84F6558298	✓	dbank.com	Android.Trojan. DroidKungFu
ed0e5232ded2314	2	1cd5ec09fd1bfad7	Rovio Mobile Ltd.	5557CBE41C109409		jimi168.com	

Table 4: Experimental Result of Layout Hashing

Name	# of Samples	Layout File	LH[1]	Time*
FakeAV	8	activity_scanning.xml	5a7ee504cc3a58e61	0.466
FakeMart	3	main.xml	d41d8cd98f00b204e	0.355
Agent	5	activity_main.xml	93447baff373b9b90	0.501

* Generation time of LH in second on Nexus 5.

Phishing Malware. In this case study, we use `HostEagle` to detect a phishing malware to evaluate the effectiveness of visual detection on Android devices. We consider a phishing malware, *Android.FakeDefender*, and install it on an Android device, Nexus 5. Upon launching, this malware displays a fake security alert and deceives users to purchase some fake anti-virus products. Usually, the appearance of the fake alert is similar to some popular anti-virus apps in the official markets. We conduct an experiment on a fake malware sample which masquerades as the "`Avast`" anti-virus software. We also download the original "`Avast`" from its official website. By generating layout hash values, we find that both of the malware and the original apps have a layout file (`scan.xml`) with the same layout hash values (`27501e19883cf1ab`). From this experiment, it shows that layout hashing can *accurately detect* visually similar layout in phishing malware. Table 4 depicts three phishing malware, their layout files, and the corresponding layout hash values and generation time. All samples in a malware family contain the same layout files. The generation time of layout hash value is less than one second. This shows the effectiveness of `HostEagle` in detecting phishing malware.

4.3 Methodology Analysis & Evaluation

To analyze our system, we use true positive (TP), true negative (TN), false positive (FP), false negative (FN) and accuracy to evaluate the effectiveness of our detection methodologies. If the system successfully detects a repackaged app, we call it as a TP event. If a legitimate app is reported as repackaged app, then it is a FP event. If the system successfully determines a legitimate app as legitimate, then we call it as a TN event. If an illegal app is reported as legitimate, we call it a FN event. Suppose that the total number of apps in our evaluation is N, the accuracy of the system is $(\sum TP + \sum TN)/N$.

Table 5: Adversary Models and Descriptions

Adversay Model	Description
Adversary I	inserting one malicious class into the package
Adversary II	injecting some garbage instructions into one class
Adversary III	injecting garbage instructions into every class in the disassembled code
Adversary IV	splitting one method into several methods in different classes
Adversary V	adding a button `View` object in one layout file
Adversary VI	modifying one `ViewGroup` object in a layout file from `LinearLayout` to `RelativeLayout`

Table 6: Detection Results for Adversary Models

Methodology	Adversary Model					
	I	II	III	IV	V	VI
LED	✓	✓	✓	✓	✓	✓
LH	✓	✓	✓	✓	✓	✗
FH [34]	✓	✓	✗	✗	✓	✓
PDG [10]	✓	✓	✓	✗	✓	✓

To evaluate the effectiveness of our layout-based detection methodologies, we use several adversary methods to compare our systems with existing systems. Firstly, we download an official app, "`Twitter`", from Google, and then we use six adversary models to repackage this app. Table 5 shows the descriptions of these six adversary models including normal repackaging, disassembled code obfuscation and obfuscation on layout files. We evaluate the effectiveness of four detection methodologies under different adversary models. The four detection methodologies are layout edit distance (LED), layout hashing (LH), fuzzy hashing (FH) [34] and program dependency graph (PDG) [10]. Because the authors of FH and PDG did not provide source codes of their systems, so we implement their detection methodologies. Table 6 illustrates the results of four methodologies under different adversary models. We see that by obfuscating the disassembled codes, FH and PDG *cannot* detect the repackaged apps. LH fails to detect the repackaged apps of adversary VI. However, this adversary model will also cause a messy arrangement of layout elements, so hackers will abandon to use this adversary model to deceive users.

4.3.1 *RepoEagle*

In Section 4.1, we discussed the statistics of layout files. Because the layout edit distance is sensitive to the number of nodes in a layout tree. We evaluate the impact of nodes number in our repository analysis system. We first define an *unqualified app* as an app whose number of nodes is less than a given threshold (say $\mathcal{N}$), while others are called *qualified apps*. Because unqualified apps have few nodes in their layout tree, we cannot directly adopt LED to these apps, so we resort to our previously discussed supplementary detection method. We analyze the impact of $\mathcal{N}$ to both the number of qualified apps and on the accuracy of detection. Firstly, we randomly select five hundred apps from our repository and extract their layout files. The red line in Figure 10 illustrates the impact of $\mathcal{N}$ on the number of qualified apps. In other words, with the increase of $\mathcal{N}$, the number of qualified apps decreases. So we cannot adopt LED to the apps using a large $\mathcal{N}$. Secondly, to demonstrate the effectiveness of LED, we use a set of repackaged apps to evaluate the detection accuracy. We choose two sets of repackaged apps from third-party markets as our ground truth. Each of these sets contains fifteen repackaged apps. We manually check the disassembled code and certificate information via static analysis, and confirm that these apps are repackaged from a game "`Fruit Ninja`" and a tool "`PPS`". Then we choose five hundred legitimate apps from official markets. We use `RepoEagle` to detect repackaged apps of "`Fruit Ninja`" and "`PPS`" from these 515 apps respectively. Because we want to know the detection rate for various numbers of nodes in layout tree. Hence, in the detection, we only use layout trees whose number of nodes is in the range of $[\mathcal{N}-5, \mathcal{N}+5]$ for LED comparison. We record the number of TP and TN, and compute the accuracy of each detection. Two blue lines in Figure 10 illustrate the accuracy of detection for different $\mathcal{N}$. With the increase of $\mathcal{N}$, the accuracy of detection goes up. If we choose layout tree with nodes number greater than a larger $\mathcal{N}$, the accuracy of detection is about 1. However, if we use layout trees with small nodes number, because the number of TN is low, the system performs with a lower accuracy. However, when we combine LED and our supplementary detection, we can achieve high accuracy in detection, and this is depicted as an orange line in the Figure 10.

In Section 3.2.3, we discussed the optimization of layout edit distance methodology. The threshold $\mathcal{T}$ will influence the comparison time. We analyze the impact of $\mathcal{T}$ on our repository analysis system `RepoEagle`. Figure 11 illustrates the impact of $\mathcal{T}$ on the execution time of `RepoEagle` with different number of apps in our repository. `RepoEagle` is deployed on a server with dual cores 2.80 GHz CPU and 4 GB memory. From our experimental result in the previous section, most of the repackaged apps and phishing malware have a low LED score. Hence, we choose a small $\mathcal{T}$ in `RepoEagle` to reduce the detection time.

We compare the performance of `RepoEagle` and disassembled-code-based system. In the pre-processing stage, we only need to extract layout files in apps, while disassembled-code-based system needs to disassemble compiled Dalvik bytecodes. Figure 12 shows the performance comparison between `RepoEagle` and other disassemble-code-based systems. We plot the pre-processing time vs. numbers of apps,

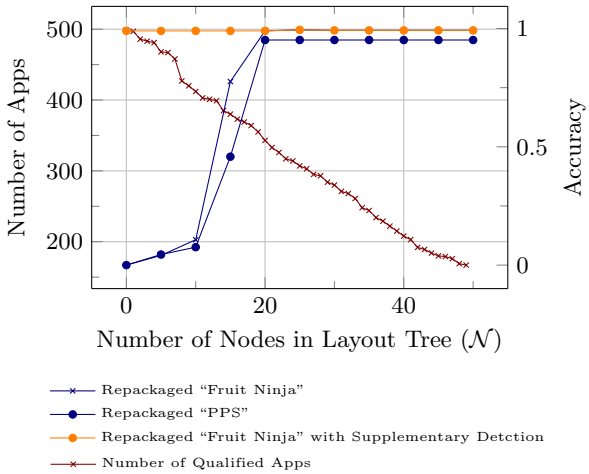


Figure 10: Relationship between threshold and the number of qualified apps.

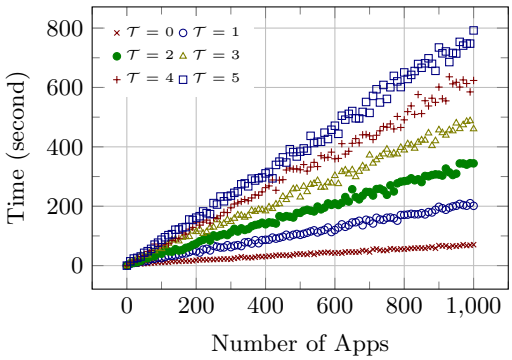


Figure 11: Relationship between threshold and the analysis time.

which are randomly selected from our repository. `RepoEagle` is about two times faster.

4.3.2 *HostEagle*

`HostEagle` aims to detect visually similar apps on Android devices which have limited resources. So we evaluate its performance on mobile devices. For comparison, we also implement fuzzy hashing algorithm on Android. We compare the time of hash value generation for detection on an Android device. In the experiment, we use LG Nexus 5 which has 2.26 GHz CPU and 2 GB memory with Android 4.4 installed. We randomly choose eight apps with different sizes of `DEX` files in our repository. Figure 13 depicts the generation time under different `DEX` file sizes. Since fuzzy hashing algorithm is based on disassembled instructions, the generation time depends on the size of `DEX` file. On the other hand, `HostEagle` is based on layout files, and the size of `DEX` file will not affect the time of hash value generation. In our experiment, the average hash generation time for fuzzy hashing is 61.8 seconds per app, while `HostEagle` only takes 0.8 second. The experiment shows that `HostEagle` is appropriate for malware detection on mobile devices since it takes much less them to generate a signature hash for detection.

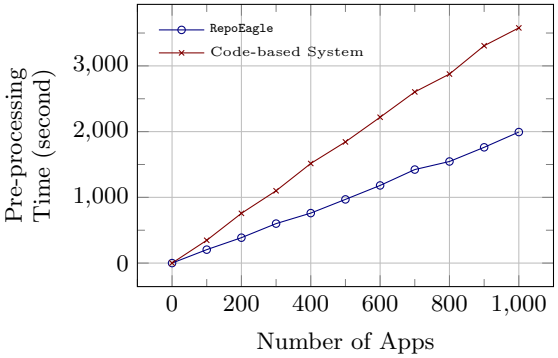


Figure 12: Pre-processing time for different numbers of apps.

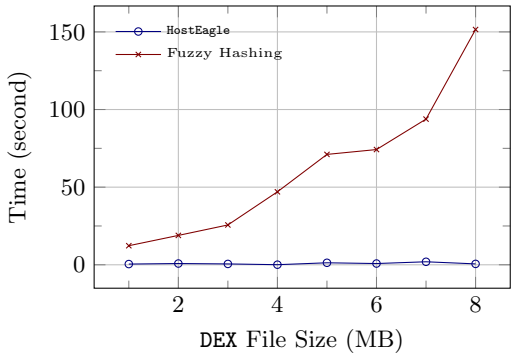


Figure 13: Time of hash value generation on Android device for different sizes of `DEX` files.

5. RELATED WORK

In conventional web security, several works [13, 20, 21] focus on detecting phishing webpages based on visual similarity assessment. Because elements in webpages are different from mobile applications, their techniques are not applicable to Android system. For smartphone security, there are several studies focusing on detecting repackaged apps. All of them are based on the approach of disassembled Dalvik bytecode of the apps. DroidMOSS [34] adopts fuzzy hashing [7, 19] to disassembled code for Android repackaged apps detection. However, the approach is based on pieces of instruction sequence, and malware writers can easily obfuscate instructions to bypass detection. To make the result more accurate, DNADroid [10] proposes program dependency graphs and uses subgraph isomorphism algorithm for similarity comparison to detect cloned apps. But constructing graphs and subgraph comparison are not scalable to analyze a large number of apps in a repository. AnDarwin [11] designs a detection method and it categorizes statements of the disassembled code into several semantic types and constructs semantic vectors. AnDarwin leverages on a locality sensitive hashing algorithm to speed up finding similar apps. ViewDroid [27] proposes feature view graph for detecting repackaged malware. The feature view graph is based on the calling relations between user interfaces (UI). However, phishing malware tend to utilize part of privacy sensitive UIs (e.g., login UIs) to cheat users. Therefore, the feature view graph cannot address the privacy leakage problem. In addition, Zhou *et al.* [33] propose using the decoupled program dependency graph to detect "piggybacked" apps. Juxtapp [16] uses machine learning algorithm to cluster similar apps based on disassembled codes. However, these methods are highly based on feature selections and learning-based methods cannot be used for host-based detection.

All the above systems rely on the *disassembled codes*. There are several reasons that using disassembled codes for detection is not sufficient and scalable. Firstly, the file structure will be different when hackers add some garbage instructions in the repackaged apps, so it is not difficult to bypass detection. Secondly, there are approaches [29, 22, 23] to obfuscate disassembled codes or make disassembling tools crashed. Thirdly, constructing program dependency graph is time consuming and it is not scalable for large apps repository analysis, in particular, when we have to deal with apps repository which has over millions of apps. Therefore, code-based detection system cannot quickly and accurately detect repackaged apps. Last but not least, because of the power and computation limitations of smartphones, the above mentioned approaches cannot be directly applied to end users' devices.

Gibler *et al.* [15] study the impact of Android app plagiarism. To prevent application repackaging, AppInk [32] proposes an algorithm to generate watermark for an app to prevent repackaging. DIVILAR [31] is proposed to encrypt original instructions so that hackers cannot modify or repackage the application. On devices, DIVILAR interprets or decrypts these obfuscated instructions and execute at runtime. However, to make it deployable, DIVILAR needs to customize the Android firmware so as to add a special interpreter in the Dalvik virtual machine. In summary, app repackaging is still a challenging and open problem.

6. CONCLUSION

We present a framework to detect visually similar Android apps. This is important because close to 90 % of Android malware are repackaged apps and phishing malware which are visually similar to the original and legitimate apps. We present our detection methodologies based on visual resources. Based on these methodologies, we implement two systems, `RepoEagle` and `HostEagle`, for repository analysis and host-based detection respectively. `RepoEagle` can quickly and accurately detect visually similar apps within a large app repository. `HostEagle` is designed for resource constrained Android devices, and it can efficiently detect repackaged apps or phishing malware upon apps download. We conduct extensive experiments to demonstrate the effectiveness and scalability of our systems. From our experiments, we show that `RepoEagle` only needs around 3 hours to detect 1298 visually similar apps from 99 626 apps in our repository, and `HostEagle` only takes less than one second to determine whether a downloaded app is a repackaged app or a phishing malware.

7. REFERENCES

[1] Androguard. https://code.google.com/p/androguard/.
[2] Android signing mechanism. http://developer.android.com/tools/publishing/app-signing.html.
[3] Apktool. https://code.google.com/p/android-apktool/.
[4] Axml. https://code.google.com/p/axml/.
[5] Google play bouncer.

http://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/.

[6] Smali/baksmali. https://code.google.com/p/smali/.

[7] ssdeep. http://ssdeep.sourceforge.net/.

[8] Virtuous ten studio. http://virtuous-ten-studio.com/.

[9] AppBrain. Number of android applications. http://www.appbrain.com/stats/number-of-android-apps.

[10] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *Computer Security–ESORICS 2012*. Springer, 2012.

[11] J. Crussell, C. Gibler, and H. Chen. Andarwin: Scalable detection of semantically similar android applications. In *Computer Security–ESORICS 2013*. Springer, 2013.

[12] F-Secure. Threat report h2 2013. http://www.f-secure.com/static/doc/labs_global/Research/Threat_Report_H2_2013.pdf.

[13] A. Y. Fu, L. Wenyin, and X. Deng. Detecting phishing web pages with visual similarity assessment based on earth mover's distance (emd). *Dependable and Secure Computing, IEEE Transactions on*, 2006.

[14] Gartner. Gartner says annual smartphone sales surpassed sales of feature phones for the first time in 2013. http://www.gartner.com/newsroom/id/2665715.

[15] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. ACM, 2013.

[16] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 62–81. Springer, 2013.

[17] G. Inc. Google play. https://play.google.com.

[18] Kaspersky. Kaspersky security bulletin 2013. overall statistics for 2013. http://media.kaspersky.com/pdf/KSB_2013_EN.pdf.

[19] J. Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation*, 3:91–97, 2006.

[20] W. Liu, X. Deng, G. Huang, and A. Y. Fu. An antiphishing strategy based on visual similarity assessment. *Internet Computing, IEEE*, 2006.

[21] Y. Pan and X. Ding. Anomaly based web phishing page detection. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 2006.

[22] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 2013.

[23] T. Strazzere. Dex education: Practicing safe dex. In *Blackhat USA 2012*, 2012.

[24] Umeng. 2013 umeng insight report. http://www.slideshare.net/umengnews/2013-umeng-insight-report.

[25] Yourstory. Google play is not the place to be in china, "app in china" connects you to top 20 chinese android app stores. http://yourstory.com/2014/02/google-play-place-china-app-china-connects-top-20-chinese-android-app-stores/.

[26] C. Zauner. *Implementation and benchmarking of perceptual image hash functions*. na, 2010.

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

[28] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 1989.

[29] M. Zheng, P. P. Lee, and J. C. Lui. Adam: An automatic and extensible platform to stress test android anti-virus systems. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2013.

[30] M. Zheng, M. Sun, and J. Lui. Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, pages 163–171. IEEE, 2013.

[31] W. Zhou, Z. Wang, Y. Zhou, and X. Jiang. Divilar: diversifying intermediate language for anti-repackaging on android platform. In *Proceedings of the 4th ACM conference on Data and application security and privacy*. ACM, 2014.

[32] W. Zhou, X. Zhang, and X. Jiang. Appink: watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 2013.

[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*. ACM, 2012.

[35] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE.