SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks 2015; **8**:2257–2266
Published online 7 February 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/sec.1170

RESEARCH ARTICLE

Detect repackaged Android application based on HTTP traffic similarity

Xueping Wu, Dafang Zhang*, Xin Su and WenWei Li

College of Computer Science and Electronics Engineering, Hunan University, Changsha, China

ABSTRACT

In recent years, more and more malicious authors aim to Android platform because of the rapid growth number of Android (Google, Menlo Park, California, USA) applications (or apps). They embedded malicious code into Android apps to execute their special malicious behaviors, such as sending text messages to premium numbers, stealing privacy information, or even converting the infected phones into bots. We called the app, which has been embedded with malicious code, as embedded repackaged app. This phenomena leads a big security risk to the Android users and how to detect them becomes an urgent problem. Previous research efforts focus on extracting the app's characteristics for comparison from its static program code, which neither can handle the code obfuscation technologies, nor can analyze the app's dynamic behaviors feature. To address these limitations, we propose an approach based on extracting the app's characteristics from the HTTP traffic, which is generated by the app. Moreover, we have implemented a multi-thread comparison algorithm based on the balanced Vantage Point Tree (VPT), which can remarkably reduce the experiment time. In this experiment, we successfully detected 266 embedded repackaged apps from 7619 Android apps downloaded from six popular Android markets, and the distribution rate of each market ranges from 2.57% to 6.07%. Then based on the analyzing of the HTTP traffic generated by these embedded codes, we found that majority of them are advertisement traffic and malicious traffic. Copyright © 2015 John Wiley & Sons, Ltd.

KEYWORDS
Android app; embedded repackaged; HTTP traffic; similarity

***Correspondence**
Dafang Zhang, College of Computer Science and Electronics Engineering, Hunan University, Changsa, China.
E-mail: dfzhang@hnu.edu.cn

Received 17 June 2014; Revised 23 August 2014; Accepted 29 October 2014

1. INTRODUCTION

Mobile devices, such as smartphones, have gained great popularity in response to vast repositories of applications. Recent statistical data [1] show that Google's Android market has more than 400,000 applications downloaded by users in January 2012. At the same time, more and more malicious authors also aim to this platform. They embedded malicious code or third-party libraries into the apps to execute additional behaviors for achieving their special goals. We call such kind of apps as the embedded repackaged app. The repackaged app provides the same functions as the original one, but the embedded malicious code would execute malicious activities in the background, such as sending text messages to premium numbers [2], downloading additional apps [3], or even converting the infected phones into bots [4]. This phenomena leads to a big security risk to the Android users and how to detect the embedded repackaged apps becomes an urgent problem.

Recently, many research efforts focus on repackaged app detection. DroidMOSS [5] and PiggyApp [6] are two representative examples. DroidMOSS extracts the app's feature, instruction sequence, from its program code and generates its fingerprint based on a fuzzy hashing technique [7] to locate and detect the modifications applied over the original app. Because a repackaged app typically shares the same primary module as the original one, PiggyApp decouples the app's program code into primary module and non-primary module. Then it extracts certain semantic features(e.g., the requested permissions and the Android API calls etc) used in the primary module to compare. Both methods extract the app's characteristics for comparison from its static program code, which can easily operate and detect some repackaged apps. However, they cannot handle the code obfuscation technologies [8] or analyze the app's dynamic behaviors feature.

To address these limitations, we propose a detection approach based on extracting features from network traffic

generated by the apps. As more than 70% of traffic generated by the Android apps are the HTTP traffic [9], we focus on analyzing the app's HTTP traffic. To ensure that the HTTP traffic can precisely represent the app's feature, we divided it into 2 categories: (i) primary module traffic—the traffic generated to fulfill the app's main functionalities, and (ii) non-primary module traffic—the traffic generated by third-party libraries (e.g., advertisement library) and other traffic. Because the repackaged app typically shares the same primary module traffic with the original one, we use the HTTP Flow Distance algorithm and the Hungarian Method [10] algorithm to calculate two primary module traffic's similarity and use the similarity to identify the similar app pair, which are candidates for embedded repackaged apps. In order to quickly find out the similar app pairs from a large number of apps in popular Android markets(e.g., Google Play), we implemented a multi-thread comparison algorithm based on the balanced VPT tree [11]. Compared with the DroidMOSS's pair-wise comparison method with $O(n^2)$ time complexity, our comparison algorithm can achieve $O(nlogn)$ time complexity, which can remarkably reduce the search time and meet our scalability requirements. Finally, to exactly identify the embedding relationship, we will take into account non-primary modules of related apps.

The main contributions of this paper are as follows:

- We propose a similarity comparison approach to detect the embedded repackaged applications in the existing markets, including both official and unofficial ones, based on the HTTP traffic, which is generated by Android apps.
- We divide the generated HTTP traffic into primary module traffic and non-primary module traffic, use the HTTP Flow Distance algorithm and the Hungarian Method algorithm to calculate their primary module traffic's similarity, and use the similarity to identify the similar app pair.
- We implement a multi-thread comparison algorithm based on the balanced VPT tree to find out the similar app pairs from a large number of apps, which can remarkably reduce the search time compared to the existing search algorithms, and meet our scalability requirements.
- In our experiment, we successfully detected 266 embedded repackaged apps from 7619 Android apps downloaded from six popular Android markets, and the distribution rate of each market ranges from 2.57% to 6.07%. Moreover, based on the analyzing of the traffic generated by these apps' embedded code, we found that the majority of them are advertisement traffic and malicious traffic, which are generated to earn advertising revenue or for some specific purpose.

The rest of this paper is organized as follows: we describe the related work in Section 2, After that, we describe the method design in Section 3, followed by its prototyping and evaluation results in Section 4. Lastly, we will conclude in Section 5.

2. RELATED WORK

Application similarity measurement and searching: The first category of related work includes prior efforts in effectively measuring apps's similarity [5,6,12,14]. Among all these works, DroidMOSS [5] is the most classic method. It extracts the app's feature, instruction sequence, from its program code and generates fingerprint based on a fuzzy hashing technique [7] to locate and detect the modifications repackagers apply over the original app. Our approach differs from it in three aspects: first, DroidMOSS detects repackaged apps based on feature extracts from the app's static program code, which cannot handle the code obfuscation technologies [8] or cannot analyze the app's dynamic behaviors feature when it is running. Second, our approach overcomes the scalability limitation from its pair-wise comparison, which could achieve $O(nlogn)$ complexity instead of $O(n^2)$. Third, DroidMOSS and the App Genome Project [12] are designed to detect repackaged apps in general third-party markets by assuming that the apps in the official market are original. This assumption is intuitive and reasonable in some aspect, but it prevents them from detecting repackaged apps in the official market, where repackaged apps are also found in considerable amount [13]. So our approach detects embedded repackaged apps in existing Android markets, including both official and unofficial ones. In addition, the App Genome Project does not disclose its detection methodology. DNADroid [14] uses program dependency graph (PDG) to characterize Android apps and compares PDGs between methods in app pairs, showing resistance to several evasion techniques. But it also applies pair-wise comparison as DroidMOSS, therefore lacking the scalability as presented in this paper. PiggyApp [6] is the most related one to our approach. It decouples the app's program code into primary module and non-primary module, because a repackaged app typically shares the same primary module as the original one. Then it extracts certain semantic features(e.g., the requested permissions and used Android APIs) embodied in the primary module to compare, which can reduce the error of apps similarity. But this approach also failed to deal with code obfuscation technologies [8].

Mobile app security and analysis: The second category covers a variety of projects [15–20] that have been undertaken to analyze or improve mobile security. Specifically, they can be loosely classified into two groups. The first group analyzes a single app from various perspectives to identify possible security and privacy problems. For example, both TaintDroid [15] and PiOS [16] focus on the privacy leak problem and respectively use dynamic taint analysis and static data flow analysis to infer potential privacy leaks. DroidRanger [17], instead, combines both static permission analysis and dynamic footprint monitoring to detect malicious applications in existing Android marketplaces. SCanDroid [18] examines an apps manifest file to automatically extract a data flow policy and then checks whether the data flows

DOI: 10.1002/sec

in the app are consistent with the extracted specification. The second group is more closely related, as it involves cluster apps. Bayer et al. [19] proposed a scalable malware-clustering algorithm based on the malware behavior expressed in terms of detailed system events. Perdisci et al. [20] further put forward a behavioral clustering of HTTP-based malware and signature generation using malicious network traces, which includes coarse-grained clustering, fine-grained clustering and cluster merging. Different from the aforementioned analysis, we focused on the Android app's HTTP traffic, which may not all malicious.

Network traffic monitor and classification: A network traffic model is expected to capture the prominent traffic properties including short and long range dependence, self-similarity in large time scale, and multi-fractality in short time scale. In [21], it has been reported and confirmed mathematically that Internet traffic behaves statistically self-similar, and aggregating streams of such traffic typically intensifies the self-similarity. Ollos et al. [22] proposed a signature extraction method to continuously extract the unique event sequences that are randomly occurring in the network. It adopts the unsupervised competitive Hebbian learning used in self-organizing Kohonen maps and the solution is fully adaptive, soft-state, iterative, and competitive. Our approach is inspired by the internet traffic self-similarity and the network signature-extraction method. After capturing the data, the traffic has to be classified. Wamser et al. [23] mentioned four kinds of classification methods: port-based classification, payload-based classification, host behavior classification and statistical classification. As more than 70% of traffic generated by the Android apps are the HTTP traffic [9] when using payload-based classification, we focus on analyzing the app's HTTP traffic in this paper. Moreover, Wei et al. [24] divided the HTTP traffic generated by the Android app into four categories: (i) the traffic generated by servers that belong to the app's provider, (ii) the traffic generated by ad providers and analytical services, (iii) the traffic generated by content distribution networks or cloud providers, (iv) all other traffic. To ensure the traffic can precisely represent the app's feature, we divided it into two categories: (i) primary module traffic—the traffic generated to achieve their functionalities, and (ii)non-primary module traffic—the traffic generated by third-party library(e.g., advertisement library) and other traffic. As the repackaged app typically shares the same primary module traffic with the original one, we can calculate the primary module traffic's similarity to find out the similar app pair.

3. APPROACH DESIGN

In this paper, our approach should achieve three goals: accuracy, scalability, and efficiency. Accuracy requires our approach detects embedded repackaged apps with few false positives and negatives; scalability represents that our approach should scale to a large number of apps in existing Android markets; and efficiency requires our approach is able to process a large number of apps fast and resource efficiency. Figure 1 shows the overall architecture of our approach, and we will describe each module in the following parts.

3.1. HTTP flow classification

After collected generated network traffic during the apps running time, we begin to parse and classify these traffic traces. We parse the HTTP traffic into HTTP flows by using six-tuple (host, method, page, parameter, values, type), then we classify the HTTP flows into primary module traffic and non-primary module traffic. First, we build the third-party library to label the non-primary module traffic. The third-party library contains three libs: public lib, which stored public platform flow, the ad lib which stored ad libraries, and the malicious lib stored malicious flow. Public platform is a platform that provides the public API interfaces. For example, Moji Weather app provides a various public login interface for users to log in, such as Tencent micro-blog account and Sina micro-blog account. The flows generated by these interfaces are public platform flows. We need to filter these flows out of the primary

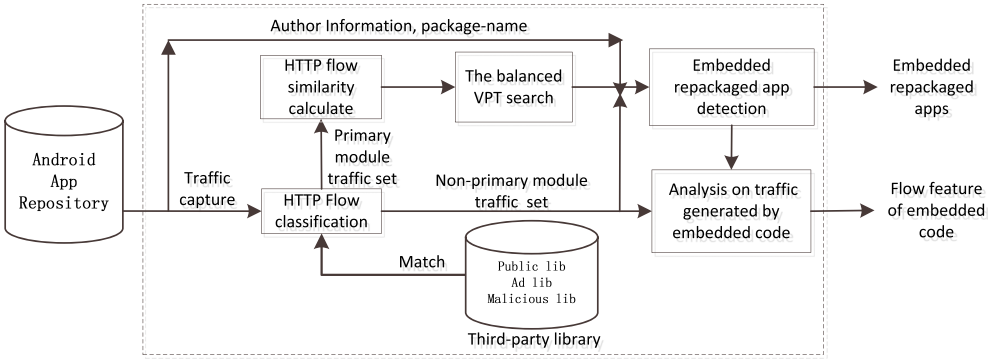

Figure 1. The overall architecture.

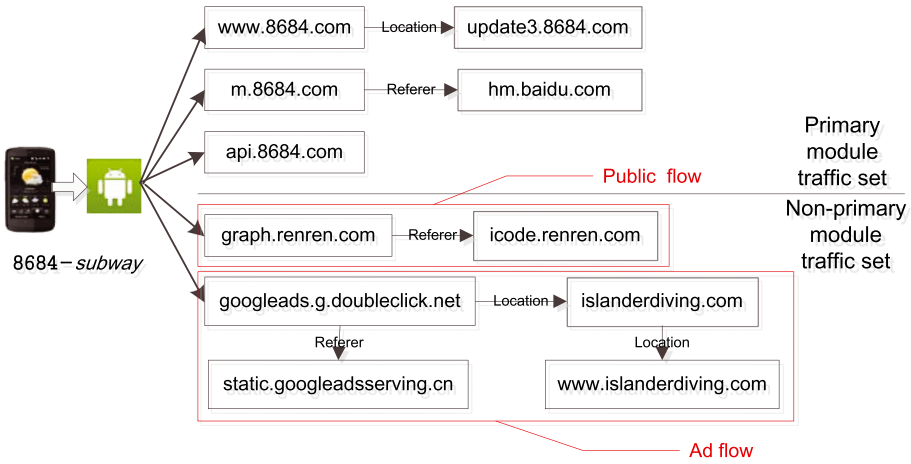

Figure 2. The HTTP flow classification result of the *8684 subway* app.

module traffic, otherwise it will increase the similarity between the apps using the same interfaces and then lead to false positive. Therefore, we collected 24 public available APIs for Android application development from the perspective of the Android application developers, such as Baidu map API, captured their traffic, parsed the traffic into flows with the three-tuple (host, method, page) pattern, and stored them into the public lib. Based on our observations, in our Android app data set (7619 apps), there are 4758 (62.45%) apps that contain ad library and may generate ad traffic. We collected 100 ad libraries from the results of paper [25] and stored them into the ad lib. Moreover, traffic generated by the malicious code also should belong to the non-primary module traffic. In order to correctly identify the malicious traffic, we captured the traffic traces generated by the 49 malware families, 1206 malware [26], and used the clustering algorithm in paper [20] to cluster each family malicious traffic, obtain their signatures, and store them into the malicious lib. For each app's flow, when it is matched with a record in one of the libs in the third-party library, we label it as a non-primary module traffic flow. In additional, when the flow is page-linked or page-redirected by a labeled non-primary module traffic flow, we also label it as a non-primary module traffic flow. To better illustrate how to classify primary module traffic and non-primary module traffic, we use the 8684 subway app as an example. Figure 2 shows that the HTTP flow dividing results of the 8684 subway app, in which the *Referer* denotes the page link and the *Location* denotes page redirection.

Furthermore, we use package name or host name to label the rest of the HTTP flows, which belong to primary module traffic or not. Because the package name is the unique identification of an app, each app has a special meaningful package name, which is similar with its company domain name. Because the primary module traffic are the flows generated by the apps access to their own servers or CDN+cloud servers to fulfill their functionalities. In Figure 2, the primary module traffic of the 8684 subway app are those that access to the **.8684.com* domain name and come from those flows by *Referer* or *Location*.

3.2. HTTP flow similarity calculate

We obtain two traffic sets, primary module traffic set and non-primary module traffic set, after HTTP traffic classification. Because embedded repackaged app embedded code into the original app, which primary module traffic set would be similar to the original one. In this paper, we use the HTTP Flow Distance algorithm and the Hungarian Method [10] algorithm to calculate distance (similarity=1-distance) between two primary module traffic. In order to calculate the distance, we first define two HTTP requests r_k and r_h generated by two different apps. Figure 3 shows the structure of an HTTP request.

- m represents the request method(e.g., GET, POST, and HEADER). We define a distance function $d_m(r_k, r_h)$ that is equal to zero if the requests r_k and r_h both use the same method (e.g., both are GET requests), otherwise it is equal to one.
- p stands for page, namely the first part of the URL that includes the path and page name, but does not include the parameters. We define $d_p(r_k, r_h)$ to be equal to the normalized Levenshtein distance between the strings related to the path and the pages that appear in the two requests r_k and r_h.
- n represents the set of parameter names (i.e., n = {session_id, appver, dbver} in the example in Figure 2). We defined $d_n(r_k, r_h)$ as the Jaccard distance between the sets of parameters names in the two requests.
- v is the set of parameter values. We define $d_v(r_k, r_h)$ to be equal to the normalized Levenshtein distance between strings obtained by concatenating the parameter values (e.g., 01013.11.3820121227001126).
- t represents the generation type of HTTP flow, it only has three values: 0, 1, and 2, respectively, indicate Direct access, Referer and Location. We define

Figure 3. Structure of an HTTP request.

$d_t(r_k, r_h)$ to be the distance, which will be zero if both requests have the same generation type, otherwise be one.

We define the overall distance between two HTTP requests as Formula 1:

$$d_r(r_k, r_h) = w_m \cdot d_m(r_k, r_h) + w_p \cdot d_p(r_k, r_h) + w_n \cdot d_n(r_k, r_h) + w_v \cdot d_v(r_k, r_h) + w_t \cdot d_t(r_k, r_h) \quad (1)$$

where the factors $w_x, x \in \{m, p, n, v, t\}$ are predefined weights. Assigning different weights will directly affect the two requests distance (the actual value assigned to the weights w_x will discuss in Section 4.1). Because each primary module traffic set contains one or more flows, we defined two primary flow sets, P1 and P2, and their similarity is the maximum value among the similarity of all HTTP flows. We calculate the maximum similarity between P1 and P2 based on a weighted X-Y bipartite graph. Each HTTP flow corresponds to each node, the HTTP flows from P1 belong to the partition X of the bipartite graph, and the HTTP flows from P2 belong to Y. Matching from an HTTP flow from P1 with an HTTP flow from P2 corresponds to inserting an edge from a node in X to a node in Y. Similarity between two nodes corresponds to weights of edges. We use the Hungarian algorithm [10] to calculate the maximum similarity, which is able to solve the problem in polynomial time, and time complexity of this algorithm is $O(n^3)$. The algorithm performs minimization by default, we use distance value instead of similarity value. The primary flow set distance obtained by the Hungarian algorithm is the minimum distance between two primary flow sets. To the contrary, the primary flow set similarity is the maximal similarity. The Hungarian algorithm formula is shown in Formula 2. For easy understanding, we call the two primary module traffic set distance as the Hungarian distance in this paper.

$$d_p(P1, P2) = \frac{\sum_{(r_k, r_h) \in match(P1, P2)} d(r_k, r_h)}{|P1| + |P2|} \quad (2)$$

3.3. The balanced VPT search

One of our goals is to search the similar app pairs from a large number of apps. The most simple search method is the pair-wise comparison method, but its time complexity is $O(n^2)$. For the purpose of speeding up the search process, we use the balanced VP-tree [11] to build the search space, which is a data structure where neighbors in the tree are likely to be neighbors in the space. Then use the triangle inequality property to create search pruning condition, which is able to reduce the time complexity of the algorithm to $O(nlogn)$.

First, we construct the balance VPT tree, each primary module traffic set denotes a node in the VPT tree. We randomly select a primary module traffic set as the root pivot P, measure the Hungarian distance between P and all the rest of primary module traffic sets, and sort these primary module traffic sets in an ascending order of their distances to the pivot. Then divide them into a fixed number N of balanced partitions, define them as $P_i, i = 1, 2, \ldots, N$. Each partition has their own distance range from $P_i.MIN$ to $P_i.MAX$. For each partition of root node, our approach repeats the aforementioned process to limit the search size in balanced range. Second, we create search pruning condition to search the similar primary module traffic set pairs quickly. For an app *query*, we assume that we already found an app its distance is less than the *min*. Therefore, the search goal is the node called *test*, which distance with *query* is less than the *min*. Because the triangle inequality property, we calculate the distance between *query* and *test* as 3:

$$distance(query, test) > |distance(query, pivot) - distance(pivot, test)| \quad (3)$$

If $|distance(query, root) - distance(root, test)| > min$ holds for any app *test* inside a partition, we can ignore this partition during the search. Thus, we can use the following pruning conditions to skip any irrelevant partition P_i, which is shown in 4 and 5.

$$min < distance(query, pivot) - P_i.MAX \quad (4)$$

$$min < P_i.MIN - distance(query, pivot) \quad (5)$$

In Algorithm 1, we outline how to search the nearest node in the VPT, which has two inputs, the query app *query* and the current node *currentNode*. During the search process, we maintain two global variables, *nearestNode* and the current minimum distance *min*. Start from the pivot node *root*, if distance between the *query* and *root* is smaller than *min*, we update the *min* to the distance between *root* and *query*, the *nearestNode* to root(line 2 4). If any of the pruning conditions in Formula 4 and Formula 5 are satisfied(line 7), this partition can be skipped. Otherwise, the nearest node-searching procedure will be recursively invoked on this partition(line 10). If we reach a partition node(lines 13–20), the algorithm simply calculates the distances between *query* and each app stored in this Partition node. If any distance is smaller than *min*, we locate a closer distance and accordingly update *min* and *nearestNode*(line

15–17). To speed up the search process, we can also initialize the *min* to a small number that indicates the acceptable level for repackaging detection. Moreover, we can adjust the algorithm to report a list of apps that fall in a range of distance with the *query* app to instead of only returning one *nearestNode*. In any case, the algorithm has the time complexity of $O(nlogn)$.

Algorithm 1 Nearest Node Search(*query*, *currentNode*)

```
1: if currentNode is a root node then
2:   if min > distance(root, query) then
3:     min = distance(root, query;
4:     nearestNode = root;
5:   end if
6:   for each partition P_i of this root node do
7:     if min <distance(root, query)-P_i.MAX or min <
       P_i.MIN-distance(root, query) then
8:       continue;
9:     end if
10:    NearestNodeSearch(query, P_i);
11:  end for
12: end if
13: if currentNode is a Partition node then
14:   for each node in this Partition node do
15:     if min > distance(node, query) then
16:       min = distance(node, query);
17:       nearestNode = node;
18:     end if
19:   end for
20: end if
```

4. EXPERIMENT

4.1. Experiment setup and data set

We have implemented our approach on a Windows 7 machine with a four-core Intel Xeon CPU (1.80HZ) and an 8G memory. First, we classify the generated HTTP traffic as non-primacy module traffic based on third-party library. This library collected traffic generated by 24 public platforms, 100 ad libraries [25], and 1026 malwares [26], and stored their flow features into a corresponding lib. Second, we assigned the weight in the HTTP Flow Distance algorithm formula as: $w_m = 10, w_p = 8, w_n = 3, w_v = 2, w_t = 1$ after many times evaluations. Because there is progressive relationship between the five tuples, the weight is decreasing relationship. Third, we build the balanced VPT tree to increase search efficiency and set the number of partitions at each pivot node to four. To obtain better trade-off between accuracy and efficiency, we select the Hungarian distance 0.15 for the similarity measurement of two primary module traffic sets. We will discuss how to choose this Hungarian distance in Section 4.3.

Table I. The number of collected apps from official and third-party Android markets

Android markets	The number of apps
Anzhi	1286(13.84%)
Hiapk	828(10.87%)
Gfan	715(6.57%)
AppChina	601(22.46%)
ZOL	511(22.31%)
Google Play	3678
Total	7619

Finally, we detect embedded repackaged app from 7619 Android apps. These apps are all popular apps downloaded from six different Android markets in the second week of January 2013. Table I summarized the distribution of the apps collected from each market. Of them, 3678 appear in the official Android Market and the rest comes from other five popular third-party markets. The number in parenthesis shows the percentage of apps that are also hosted in the official Android Market.

4.2. HTTP traffic classification results

The HTTP flow classification is an essential component in our approach, which affects both accuracy and efficiency to detect embedded repackaged app. To evaluate efficiently, we randomly choose 100 samples, which generated a total of 2034 HTTP flows. After the HTTP flow classification, we manually verified the results. The result shows that 1973 HTTP flows (97%) are correctly classified and 61 HTTP flows are not. We further examine the 61 mislabeled flows and find that most of them are mislabeled because our collected ad libraries are not complete enough. Therefore, we will complete the ad lib in future. Even so, the correct rate of 97% indicates that our HTTP flow classification method is able to obtain a good accurate.

4.3. Hungarian distance trade-off

In this section, we discuss how to choose a reasonable value for Hungarian distance, which provides a better trade-off between detection efficiency and accuracy. To analyze the distribution of app pair Hungarian distance, we randomly select 1500 and 3000 sample app pairs and calculate their Hungarian distances, respectively. From Figure 4, about 77.51% of apps Hungarian distance are larger than 0.7, and 19.15% are less than 0.3. Because the repackaged app shared a similar primary module traffic set with the original one, we should determine a reasonable Hungarian distance as two similar primary module traffic set distance threshold. A larger distance threshold may detect more repackaged apps but also generates more false positive. A smaller distance can reduce false positives but more false negatives. As a general rule of thumb, if two apps have a Hungarian distance greater than 0.3, we consider the possibility of having a repackaging relationship

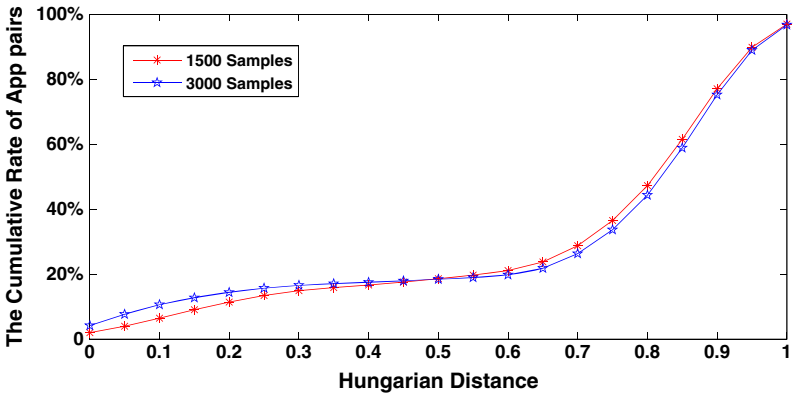

Figure 4. The cumulative distribution of Hungarian distance.

Table II. Choosing the right Hungarian distance

Hungarian distance	0.05	0.1	0.15	0.2	0.25	0.3
True positives	7	13	18	20	23	24
False positives	0	0	2	5	6	10

Table III. Embedded repackaged apps detection results

Android markets	# of repackaged apps	Repackaged rate	# of original apps
Anzhi	22	2.66%	10
Hiapk	65	5.05%	40
Gfan	24	3.36%	17
AppChina	29	4.83%	38
ZOL	31	6.07%	26
Google Play	95	2.57%	79
Total	266	3.49%	210

to be very low. Hence, we consider the distance threshold range from 0.05 to 0.30.

In order to determine the Hungarian distance threshold more precisely, we randomly select 500 apps and used six groups of distance threshold for detection, and the final results are shown in Table II. From Table II, the results indicate that the Hungarian distance is 0.15 as the distance threshold, which can detect the most repackaged apps and cause minimal false positives.

4.4. Embedded repackaged apps detection

From the previous section, we have empirically chosen less than 0.15 as the Hungarian distance threshold. In this section, we apply it to our data set and present our detection results.

In this experiment, we successfully detected 266 (3.49%) embedded repackaged apps from our Android app data set. The results are shown in Table III, the second column shows the number of embedded repackaged apps in each market and the third column shows the repackaged app ratio. The official Android Market contains the largest number of embedded repackaged apps, but its repackaged rate is the lowest. The fourth column represents the number of original apps that have been repackaged, which in general indicates which market contains the most repackaged app. Our results also show that game, wallpaper, and electronic book apps are the most popular repackaged.

To further measure the false negative, we study 52 apps that were known to be embedded repackaged apps in our data set(before our method was designed). We can correctly detect 50 of them and miss two, namely false negative is 3.85%. Our manual analysis shows that because of the results of HTTP flow classification, two apps been missed.

After detecting these embedded repackaged apps, it is also interesting to find out answers to the following questions: Whether all embedded ad libraries are benign? How much additional traffic would generate by embedded code or library? How many apps have been embedded malicious code? To answer these questions, we analyzed the non-primary module traffic generated by the detected embedded repackaged apps and find these flows generated by embedded code with the following main features:

Embedded Advertisement Library: Because most of the apps are free, many Android developers prefer to embed advertisement library to earn money. There have been many open sources of ad libraries, developers can easily reuse them through minor code change or even no change, such as Admob. Such a simple, fast, and efficient reuse pattern makes the embedded ad libraries become very common in repackaged app. In our data set, we find 4758 (62.45%) apps are embedded ad libraries. We also find that five ad libraries are dangerous, namely Casee, Sosceo, Malaup, Adsage, and Adwo. These ad libraries locate the mobile phone, read the call records and phone messages, and send SMS messages and even download additional apps, and so on. Obviously, besides embedding some benign ad libraries to earn advertising revenue, embedded repackaged apps even embedded malicious ad libraries to achieve a specific purpose.

Additional Traffic Generation: Embedded repackaged apps used to embed extra payload into the original app,

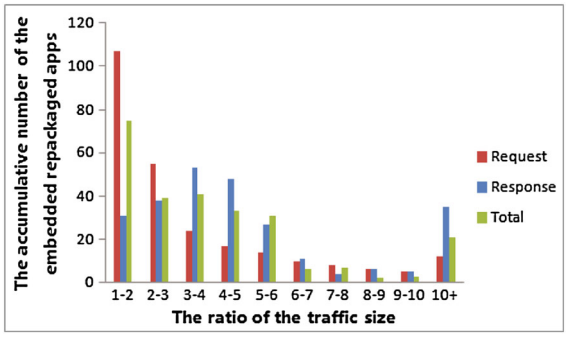

Figure 5. The request, response and total traffic ration between repackaged and original apps.

Table IV. The statistics of embedded malicious payloads

Malware family	# of repackaged apps
Anserverbot	2
Droidkungfu	3
Geinimi	2
Plankton	1
YZHCSMS	1

and they would generate more traffic than the original one. Figure 4 shows the ratio of HTTP request traffic, HTTP response traffic, and total traffic among the 266 embedded repackaged apps and their corresponding original one. From Figure 4, we find that the request traffic size between the embedded repackaged apps and their original is not significantly different. Its ration mainly distributes in the range of 1–2 times. But the response traffic is remarkably different, the size of response traffic of the embedded repackaged apps is much larger than their original one. Its ration distributes in the range of 3–6 times. Because a very brief request flow would lead to a large number of the response sometimes, such as, the "Push Ad" will continue to send advertising information to mobile phone.

In addition, the last set of columns representing the embedded repackaged apps traffic is more than 10 times to their corresponding original apps. After Analyze these apps, we find some repackaged apps have more ad libraries than the corresponding original one. For example, Japanese.lookup.Dictionary, the original app only has two ad libraries, namely Adsmogo and Umeng. However, the repackaged one has two extra ad libraries, Adwo and Domob, which generated more traffic. The most important is that it downloaded a new additional app, *brother-play*. Therefore, the generated request traffic, response traffic, and total traffic ration between them are 39.5, 79.4 and 59.9 times more than the original one, respectively.

Malicious Payload Embedded: Besides embedded ad libraries, malicious authors even embedded malicious payloads into normal apps. In order to detect the malicious flow, we captured the 49 malware families, 1206 malwares traffic, and used the clustering algorithm in paper [20] to cluster each family's malicious traffic, obtained their signatures, and stored them into malicious lib. Lastly, we use the signatures to find malicious flow in the detected 266 embedded repackaged apps. At last, we find nine embedded repackaged apps belonging to five malware families, which are all coming from the third-party markets. The results are shown in Table IV.

From Table IV, the DroidKungFu [27] is the mostly used malware family for embedded repackaged apps. DroidKungFu is capable of rooting the vulnerable Android phones and may successfully evade the detection from current mobile anti-virus software. It will add into the infected app a new service and a new receiver. The receiver will be notified when the system finishes booting so that it can automatically launch the service without user interaction. Once the service gets started, it will collect a variety of information on the infected mobile phone, including the IMEI number, phone model, as well as the Android OS version. With the collected information, the malware phones home by making a HTTP POST to a hard-coded remote server. For example, one built-in payload of DroidKungFu is to install a hidden app named legacy after getting the root privilege. The app is embedded as part of the infected host app and pretends to be the legitimate Google Search app bearing with the same icon. It turns out that the fake app is a backdoor, which connects back to the remote server for instructions and sends the collected information through http://search.googfu-android.com:8511/search/sayhi.php, as shown in Figure 6.

4.5. Performance evaluation

In this experiment, for each app, the HTTP flow dividing process spends 0.383–7.692 s. The 7619 apps of this process spend 5538.293 s in total, and the average processing time of each app is 0.727 s. Then the construction of the balanced VPT tree spent 6.752 s.

In this paper, we use the Java thread pool to implement multi-thread. In this evaluation, we set the thread

```
POST /search/sayhi.php HTTP/1.1
Content-Length: 174
Content-Type: application/x-www-form-urlencoded
Host: search.googfu-android.com:8511
Connection: Keep-Alive
User-Agent: Apache-HttpClient/UNAVAILABLE(java 1.4)
Expect: 100-Continue

POST Data
imei=000000000000000&ostype=AOSP&osapi=8&mobile=13549646178
&mobilemodel=generic+sdk&netoperater=internet&nettype=mobile
&managerid=Coolpad&sdmemory=0.00B&aliamemory=27M&root=1

HTTP/1.1 200 OK
Date: Thu, 12 Mar 2013 13:15:05 GMT
Server: Apache/2.2.3(CentOS)
X-Powered-By: PHP/5.1.6
Content-Length: 2
Connection: Close
Content-Type: text/html; charset-UTF-8
```

Figure 6. Payload content of DroidKungFu.

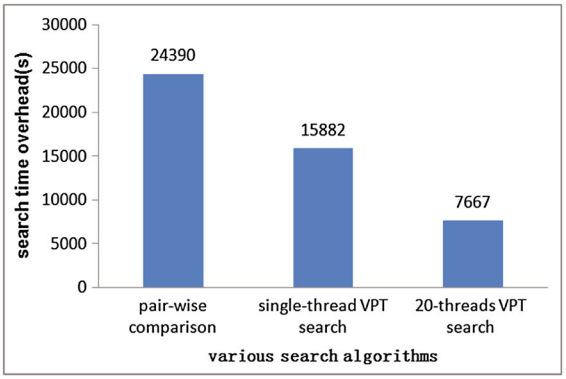

Figure 7. The time overhead of various search algorithms.

number as 20, its search time is 7667 s, which is the minimum search time. Figure 7 shows the time overhead of the pair-wise comparison algorithm, and single-thread and 20-threads search algorithm based on the balanced VPT tree, respectively. We can observe that the search algorithm based on the balanced VPT tree spent less search time than the pair-wise comparison algorithm. Moreover, the 20-threads search algorithm based on the balanced VPT tree reduces 51.73% of search time compare to the single-thread search algorithm and 68.56% of search time compare to the pair-wise comparison algorithm. We can see that the multi-thread technology can greatly use the resources and improve the efficiency.

5. CONCLUSION

In this paper, we propose a fast, scalable, accurate approach to detect the embedded repackaged applications in the existing markets based on the similarity of the Android apps' HTTP traffic. In order to make the HTTP flow represent the app's feature more accurately, we classified the generated traffic into 2 categories: primary module traffic and non-primary module traffic. Then we use the HTTP Flow Distance algorithm and the Hungarian Method algorithm to calculate the two primary module traffic's similarity. Moreover, we implement a multi-thread comparison algorithm based on the balanced VPT tree, which can remarkably reduce the search time compared to the existing comparison algorithms, and quickly find out the similar app pairs. Finally, we evaluate our approach through 7619 apps collected from six popular Android markets. We successfully detected 266 embedded repackaged apps, and the distribution rate of each market is from 2.57% to 6.07%. Furthermore, we analyze the non-primary module traffic, we find that the majority of embedded code are ad or malicious code, and traffic generated by these kinds of code are to earn advertising revenue or for some malicious purposes. These results demonstrate that the Android application markets, whether the official or third-party markets, are in urgent need of a stringent detecting process to better regulate the mobile applications market order.

Acknowledgements

This work is based upon work supported in part by the National Basic Research Program of China (NBRP or 973 Program) under grant No. 2012CB315805, the National Science Foundation of China (NSF) under grants No. 61173167, 61473123 and 61173168. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the 973 Program or the NSF.

REFERENCES

1. AppBrain. *Number of available Android applications. [EB/OL].* http://www.appbrain.com/stats/number-of-android-apps. [Accessed on 1 December 2011].
2. Kaspersky Lab. *First SMS trojan detected for smartphones running Android. [EB/OL].* http://www.kaspersky.com/about/news/virus/2010/First_SMS_Trojan_detected_for_smartphones_running_Android. [Accessed on 9 August 2010].
3. Symantec Inc. *Android threats getting steamy. [EB/OL].* http://www.symantec.com/connect/blogs/android-threats-getting-steamy. [Accessed on 28 February 2011].
4. *Lookout Inc. Security Alert: Geinimi, Sophisticated New Android Trojan Found in Wild*. [EB/OL], http://blog.mylookout.com/2010/12/geinimi_trojan/. [Accessed on 29 Decenber2010].
5. Wu Z, Yajin Z, Xuxian J, Peng N. DroidMOSS: detecting repackaged smartphone applications in third-party Android marketplaces, In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, (CODASPY'12), San Antonio, Texas, USA, February 7–9, 2012.
6. Wu Z, Yajin Z, Michael G, Xuxian J, Shihong Z. Fast, scalable detection of piggybacked mobile applications, In *Proceedings of the 3nd ACM Conference on Data and Application Security and Privacy*, (CODASPY'13), San Antonio, Texas, USA, February 18–20, 2013.
7. Dustin Hurlbut. Fuzzy hashing for digital forensic investigators. *Technical report Inc.* http://access-data.com/downloads/media/FuzzyHashingforInvestigators.pdf.Online; [Accessed on 17 May 2011].
8. Developers. *ProGuard*. [EB/OL], http://developer.android.com/tools/help/proguard.html. [Accessed on 1 December 2011].
9. Shuaifu D, Alok T, Xiaoyin W, Antonio N, Dawn S. NetworkProfiler: towards automatic fingerprinting of Android app, In *Proceedings of the 32nd IEEE*

International Conference on Computer Communications (INFOCOM'13), Turin, Italy, 2013.
10. Kuhn H. The Hungarian method for the assignment problem. *Naval Research Logistics* 2005; **52**(1): 7C21.
11. Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces, *In Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms, SODA 93*, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics; 311C321.
12. *Lookout Inc. App genome report*, February 2011. https://www.mylookout.com/appgenome/.Online; [Accessed on 1 December 2011].
13. *Lookout Inc. Update security alert: DroidDream malware found in official Android Market.* http://blog.mylookout.com/2011/03/security alert malware found in official android market droiddream/ Online; [Accessed on 1 December 2011].
14. Jonathan C, Clint G, Hao C. Attack of the clones: detecting cloned applications on Android markets, In *Proceeding of the 17th European Symposium on Research in Computer Security (ESORICS)*, Pisa, Italy, September 10–12, 2012.
15. William E, Peter G, Byung-gon C, Landon C, Jaeyeon J, Patrick MD, Anmol S. TaintDroid: an information flow tracking system for realtime privacy monitoring on smartphones, In Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, USENIX OSDI'10, Vancouver, BC, Canada, October 4–6, 2010.
16. Egele M, Kruegel C, Kirda E, Giovanni V. PiOS: detecting privacy leaks in iOS applications, In *Proceedings of the 18th Annual Network and Distributed System Security Symposium, NDSS 11*, San Diego, California, February 6–9, 2011.
17. Zhou Y, Wang Z, Zhou W, Xuxian J. Hey you, get off of my market: detecting malicious apps in official and alternative Android markets, In *Proceedings of the 19th Annual Network and Distributed System Security Symposium, NDSS 12*, San Diego, February 5–8, 2012.
18. Adam Fuchs, Avik Chaudhuri, Jeffrey Foster. *SCanDroid: automated security certification of Android applications*. http://www.cs.umd.edu-/avik/projects/scandroidascaa/paper.pdf.Online;, [Accessed on 1 December 2011].
19. Bayer U, Milani P, Comparetti C, Hlauschek CK, Kirda E. Scalable, behavior-based malware clustering, In *Proceeding of the 16th Annual Network and Distributed System Security Symposium, NDSS 2009*, San Diego, February 8–11, 2009.
20. Perdisci R, Lee W. Behavioral Clustering of Http-based Malware and Signature Generation using Malicious Network Traces, In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, San Jose, USA, April 28–30, 2010.
21. Tavernier W, Papadimitriou D, Colle D, Pickavet M, Demeester P. Packet loss reduction during rerouting using network traffic analysis. *Telecommunication Systems* 2013; **52**(2): 861–879.
22. Ollos G, Vida R. Event signature extraction and traffic modeling in WSNs. *Telecommunication Systems* 2014; **55**(4): 513–523.
23. Wamser F, Pries R, Staehle D, Heck K. Phuoc Tran-Gia: traffic characterization of a residential wireless Internet access. *Telecommunication Systems* (October 2011); **48**: 1-2.
24. Wei X, Gomez L, Neamtiu I, Faloutsos M. Profiledroid: multilayer profiling of Android applications, In *Proceeding of the 18th Annual International Conference on Mobile Computing and Networking, MobiCom*, lstanbul Turkey, August 22–26, 2012.
25. Grace M, Zhou W, Jiang X, Sadeghi AR. Unsafe exposure analysis of mobile in-app advertisements, In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, Tucson, Arizona, USA, April 16-18, 2012.
26. Zhou Y, Xuxian J. Dissecting Android malware: characterization and evolution, In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, San Francisco, Canada, May 20–23, 2012.
27. *DroidKungFu. [EB/OL]*. http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html. [Accessed on 31 May 2011].