Fast, Scalable Detection of "Piggybacked" Mobile Applications

Wu Zhou †, Yajin Zhou †, Michael Grace †, Xuxian Jiang †, and Shihong Zou ‡
†North Carolina State University
{wzhou2, yajin_zhou,mcgrace, xjiang4}@ncsu.edu
‡Beijing Univ. of Posts & Telecommunications
{zoush}@bupt.edu.cn

ABSTRACT

Mobile applications (or apps) are rapidly growing in number and variety. These apps provide useful features, but also bring certain privacy and security risks. For example, malicious authors may attach destructive payloads to legitimate apps to create so-called "piggybacked" apps and advertise them in various app markets to infect unsuspecting users. To detect them, existing approaches typically employ pair-wise comparison, which unfortunately has limited scalability. In this paper, we present a fast and scalable approach to detect these apps in existing Android markets. Based on the fact that the attached payload is not an integral part of a given app's primary functionality, we propose a module decoupling technique to partition an app's code into primary and non-primary modules. Also, noticing that piggybacked apps share the same primary modules as the original apps, we develop a feature fingerprint technique to extract various semantic features (from primary modules) and convert them into feature vectors. We then construct a metric space and propose a linearithmic search algorithm (with $O(n \log n)$ time complexity) to efficiently and scalably detect piggybacked apps. We have implemented a prototype and used it to study $84,767$ apps collected from various Android markets in 2011. Our results show that the processing of these apps takes less than nine hours on a single machine. In addition, among these markets, piggybacked apps range from 0.97% to 2.7% (the official Android Market has 1%). Further investigation shows that they are mainly used to steal ad revenue from the original developers and implant malicious payloads (e.g., for remote bot control). These results demonstrate the effectiveness and scalability of our approach.

Categories and Subject Descriptors C.4 [**Performance of Systems**]: Measurement techniques; K.6.5 [**Management of Computing and Information Systems**]: Security and protection – *Invasive software*

General Terms Security; Algorithms; Measurement

Keywords Mobile Application; Smartphone Security; App Repackaging; Piggybacked Application

CODASPY'13, February 18–20, 2013, San Antonio, Texas, USA.

1. INTRODUCTION

With the wide adoption of smartphones and mobile devices, mobile applications (or apps) are rapidly growing in number and variety. Recent statistics [6] show that since January 2012, Google's Android Market is home to more than $400,000$ apps for users to browse and download. What is more, the number is increasing at an astonishing rate: each month will see more than $20,000$ apps being published [6]. The convenience and functionality these apps offer greatly extend the capability and reach of mobile devices. Unfortunately, along with the above benefits, there are undesirable privacy risks and security issues associated with these mobile apps. For example, malware authors may piggyback destructive payloads on known good apps and then advertise the *piggybacked apps* in various app markets to infect unsuspecting users. For ease of presentation, we use the term *carrier* to refer to the original app being piggybacked and the term *rider* to denote the additional code injected into the original app. Notice that piggybacked apps are a special kind of repackaged apps [44, 26], which are created by modifying and re-signing legitimate apps for distribution. The distinction however is that piggybacked apps involve injecting (new) rider code into the original apps while repackaged apps may only make minor modifications, including tweaking resource files or replacing constant strings for new language support.

With the inclusion of new rider code, piggybacked apps pose greater security threats than other kinds of repackaged apps. In fact, a number of security alerts have been issued about the presence of piggybacked apps in various app markets. Specifically, these piggybacked apps embed malicious rider code into popular carrier apps, such as games and utility programs. Once installed, the rider code could perform a variety of malicious actions, such as sending text messages to premium numbers [32] and converting the infected phones into bots [31].

Recognizing these threats, researchers have explored different ways to detect them. The App Genome Project [26] and DroidMOSS [44] are two representative examples. They are designed to detect repackaged apps in general among third-party app markets – by assuming that apps in the official Android Market are original. This assumption is intuitive and reasonable in some aspect, but it prevents them from detecting repackaged apps in the official Android Market, where repackaged apps are also found in considerable amount [28]. In addition, while App Genome Project does not disclose its detection methodology, DroidMOSS uses a fuzzy hashing technique to generate app fingerprints based on their instruction sequences and then applies *pair-wise comparison* to detect repackaged apps. Pair-wise comparison does not scale to the large amount of apps available in modern marketplaces.

In this paper, we propose a fast and scalable approach called *PiggyApp* to effectively detect piggybacked apps in existing Android

markets, including both official and unofficial ones. PiggyApp meets the need for scalability and timeliness by accommodating the fast influx of a large number of apps in existing marketplaces, which dwarfs earlier approaches. Moreover, our system eliminates the previous assumption by considering apps from different marketplaces in the same manner, which uniquely enables the detection of piggybacked apps in the official Android Market (now part of Google Play).

Our approach is based on two main observations. First, in a piggybacked app, the rider code is relatively independent and does not tightly interweave, if any, with the primary functionality of the host app. Therefore, we propose a technique called *module decoupling* to effectively partition the app code into primary and non-primary modules. Each app has one unique primary module, which mainly implements the advertised functionality. Meanwhile, it may have a number of non-primary modules that are relatively standalone. Various support routines or libraries, advertisement packages, and mobile payment frameworks – as well as embedded rider code – fall in this category.

Second, a piggybacked app typically shares the same primary module as the original carrier app. Accordingly, we propose another technique called *feature fingerprinting* to extract certain semantic features (e.g., the requested permissions and used Android APIs) embodied in the primary module. To facilitate this comparison and meet our scalability requirements, we represent them as feature vectors, organize these feature vectors into a metric space, and then propose a linearithmic search algorithm (with $O(n \log n)$ time complexity – compared to the previous $O(n^2)$ complexity of pair-wise comparison) to detect piggybacked apps. From these piggybacked apps, we further derive the corresponding rider code and perform a systematic study about its functionality and purpose.

We have implemented a proof-of-concept prototype and used it to detect piggybacked apps in multiple Android markets worldwide, including the official market and six alternative ones: two from the US, two from Eastern Europe and two from China. Our study includes $84,767$ apps and 68,187 of them come from the official Android Market. These apps are collected by taking a snapshot of the available apps on these marketplaces in the first week of March 2011. By running our system on a standalone desktop machine (with 4 cores and 8G memory), it takes less than 9 hours to process all of these apps, which meets our scalability and timeliness requirements. The results show that 1.0% apps in the official Android Market are piggybacked. For the rest alternative ones, the piggybacked apps vary from 0.97% to 2.7%. By analyzing the rider code, we find that its main purposes are to embed ad libraries to steal the generated ad revenue from the original developers or to implant malicious payloads to compromise users' phones.

The rest of this paper is organized as follows: We describe the system design in Section 2, followed by its prototyping and evaluation results in Section 3. After that, we discuss the system's limitations and suggest possible improvements in Section 4. Lastly, we describe related work in Section 5 and conclude in Section 6.

2. DESIGN

In Figure 1, we show the overall architecture of our system. While piggybacked apps leverage carrier apps to entice users into downloading and installing them, the main purpose is to execute the attached rider code unnoticed. Notice that the rider code is relatively independent and should not closely interweave, if any, with the primary functionality of the carrier app. Accordingly, we propose *module decoupling* to first isolate the primary modules from existing apps. Moreover, as piggybacked apps still share the same primary module code base as the originals, we then propose to mainly compare primary modules to infer the piggybacking relationship between two apps.

In our system design, there are three competing goals: *scalability*, *accuracy*, and *efficiency*. Scalability is needed to accommodate the large number of apps in existing markets; accuracy requires our system to effectively detect piggybacked apps with few false positives and negatives; and efficiency imposes the need for our system to handle existing apps in a timely and resource-efficient manner. Specifically, to meet the scalability requirement, our system must improve upon the $O(n^2)$ time complexity of pairwise comparison in existing systems [44]. To this end, we develop a *feature fingerprinting* technique that extracts semantic features from the primary module, including requested permissions and used Android APIs, and represents them as feature vectors. These feature vectors are used to construct a metric space from which we can efficiently identify similar apps using a linearithmic search algorithm ($O(n \log n)$ time complexity). By further examining the signing certificates and other non-primary modules of similar apps, we can effectively detect piggybacked apps as well as the related rider code.

In this work, we assume that piggybacking mainly occurs by adding Java code to a legitimate app, instead of native code. There are two main reasons: first, compared to native code, Java code is typically a more vulnerable target for piggybacking. A number of tools [40, 9] have been developed and can be readily misused for this purpose. Second, existing apps are still primarily written in Java, instead of C, which results in much less native code in existing apps. Considering the dataset used in this study, we find that only 5% of all apps contain native code. In addition, we assume that legitimate app developers do not disclose their private keys (for app signing) to others. Therefore, piggybacked apps will not share the same certificates as the original apps. Next, we detail each essential component in our system.

2.1 Module Decoupling

An Android app is typically composed of multiple relatively independent modules. The primary module implements the main functionality, which is advertised to attract user downloads. Other non-primary modules may serve the primary module with support routines and utility libraries, but could also be completely independent (such as ad libraries). Within each module, either primary or non-primary, the code is tightly coupled or organized; between modules, the code is loosely coupled or even not related. (Some standalone apps may only contain one module – the primary module.) Without the access to the app's source code, we resort to program comprehension techniques [5] to decouple internal modules within an app.

For a given app, our module decoupling process takes as input its `classes.dex` file and works in two main steps. First, based on the Dalvik bytecode, we build a program dependency graph (PDG). Within the graph, the node represents a Java class package that contains a number of Java class files declared within it. An edge connects two class packages if there exists an interaction or a dependency between these two packages. A weight is assigned to an edge to indicate how close these two class packages are connected. In our system, the edge essentially captures the following interaction or dependency relationship: class inheritance, package homogeny[1], method calls, and member field references. Each of this relationship in general represents certain degree of coupling and our system will collect it and assign a weight. As the class in-

[1]Two packages are homogeneous if they form a parent-child relation or share the same parent.

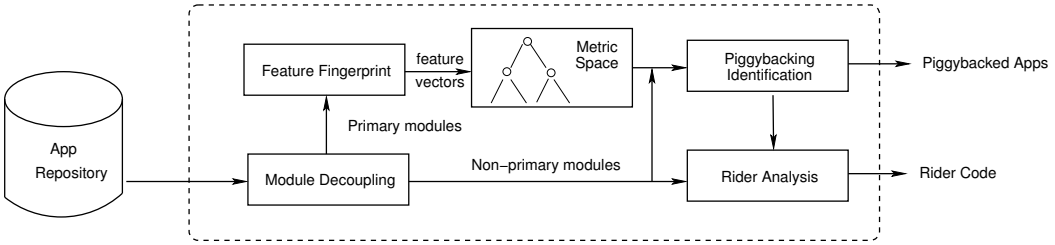

Figure 1: The overall system architecture

heritance relationship shows tighter coupling between two classes, we accordingly assign a higher weight to the edge than others that may simply indicate a single method call. Between two class packages, we use two cumulative weight values to unidirectionally sum all the edge weights from one to another.

Second, based on this program dependency graph, we use an agglomerative clustering algorithm (Algorithm 1) to group these class packages into different modules. To begin with, we initialize a number of singleton clusters one for each class package in the graph. After that, we repeat the process of checking whether any two clusters can be merged and, if they can, merging the pair of clusters that have the largest cumulative weight values. Otherwise, we report the resulting set of clusters as the modules contained in the app. Note the `merge_able` condition (`line 2`) in the algorithm examines the remaining largest cumulative weight values between any cluster pair. In our prototype, we empirically choose a cut-off value (Section 3).

Algorithm 1 Agglomerative clustering

Input: `Program dependency graph (PDG) of an app`
Output: `A list of primary and non-primary modules`

```
1: clusters = create_singleton_clusters(PDG)
2: while merge_able(clusters) do
3:    compute_coupling_between_each_pair(clusters)
4:    (c1, c2) ← select_the_most_coupled_pair(clusters)
5:    clusters ← merge(c1, c2, clusters)
6: end while
7: return clusters
```

In Figure 2, we show an example run of the clustering algorithm on a piggybacked app (`MD5: 09105460be466d0c024c37df8997b061`). Initially, it has six modules `com.rechild.advancedtaskkiller`, `com.google.ads`, `com.google.ads.util`, `org.json`, `com.android.root`, and `jackpal.androidterm`. The figure shows the cumulative weight values between each pair. After the run, our algorithm effectively merges `com.google.ads` and `com.google.ads.util` and reports five remaining clusters as standalone modules.

Among the reported modules, we then determine which one is the primary module. In particular, we leverage the information in the `AndroidManifest.xml` file that declares various components of an app, including its activities, services, receivers, and content providers. Specifically, it specifies concrete classes that will be invoked to handle certain events or actions. A special one is `ACTION.MAIN` that represents the main entry point of the app. Accordingly, we choose the module that contains this class as a candidate for the primary module. Meanwhile, notice that the primary module tends to provide the main interface for users to interact with. Therefore, we also select the module with classes that handle most activities as the primary module candidate. If there are multiple candidates, we choose the most similar one by

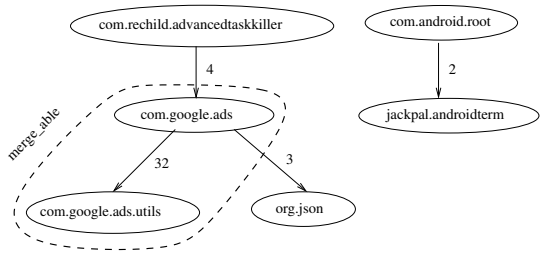

Figure 2: An example module decoupling run

calculating the similarity of the module name against the app name in the manifest file. In our experiments, we do not encounter any piggybacked apps that change the app name to a completely unrelated one.

2.2 Feature Fingerprint and Representation

After module decoupling, we then generate feature fingerprints for the primary module. More specifically, feature fingerprints are supposed to distinguish the functionality of one primary module from another. To this end, we extract various semantic features such as the requested permissions, the Android API calls used, involved intent types (which represent the way for inter-component or inter-process communication), the use of native code or external classes, as well as the authorship information (from the developer certificates in the `META-INFO` directory). The intuition is that it is rare for two different modules to be coincidentally the same in all the above items. Notice that we include the developer information to exclude different apps authored by the same developer as there is no such need.

With the collected features, we then represent them into a vector where 0 and 1 respectively represent the absence and presence of certain feature in the primary module. After that, we organize these feature vectors (each representing an app) into a metric space and transform the problem of detecting piggybacked apps into a nearest neighbor searching problem. A naive approach for nearest neighbor searching is to perform pair-wise comparison and choose the one with the smallest distance. Considering the number of apps in current app markets, this approach is not scalable with its $O(n^2)$ complexity. That is also the main reason why we choose to construct a metric space from the extracted feature vectors. By exploiting its triangle inequality property [43], we can effectively prune irrelevant portion during the search and achieve an $O(n \log n)$ time complexity, thus accommodating the scalability challenge.

In the metric space construction, we use the following Jaccard distance between the primary module features of two apps as the distance metric:

$$Jaccard(F_A, F_B) = \frac{|F_A \cup F_B| - |F_A \cap F_B|}{|F_A \cup F_B|} \quad (1)$$

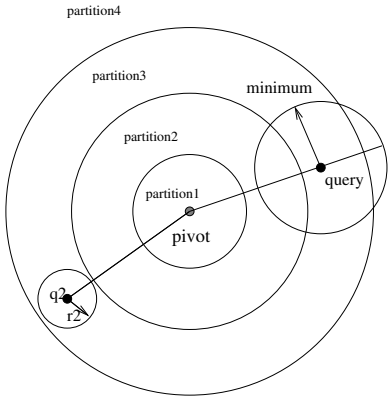

Figure 3: Triangle inequality-based VPT pruning

where F_A and F_B represent the feature vector of primary module A and B respectively. Recall that they are vectors of binary values (0 or 1) to indicate whether one specific API call, permission, intent type or any external code loading behaviors occur in the module code. Formula 1 essentially calculates the ratio of disjoint features over the union of features present in these two modules to characterize how different they are. As shown in [41], the Jaccard distance satisfies the property of the triangle inequality that is being exploited to prune the irrelevant part of the search space.

In our system, we use a Vantage Point Tree (VPT) [43] to construct the metric space. Specifically, we first select a primary module as the root pivot `P`, measure the Jaccard distances between `P` and all the rest of the modules, sort these modules in an ascending order of their distances to the pivot, and then divide them into a fixed number `N` of balanced partitions, represented as P_i, `i = 1,2,...N`. At the pivot, the distance range associated with each partition P_i is recorded, represented as $P_i.MIN$ and $P_i.MAX$. For each partition of the pivot, we will repeat this partitioning procedure to reduce its size to a manageable level.

To elaborate how the triangle inequality property enables efficient search pruning in the constructed VPT tree, we present in Figure 3 the partitions as concentric circles based on their distance ranges to the `pivot`. For an app `query`, suppose we discovered another app `nearest_neighbor` with the `minimum` distance to `query`. The search space is then reduced to locate another app, say `test`, whose distance to `query` is smaller than `minimum`. Due to the triangle inequality property in Jaccard distance, we have:

$$distance(query, test) > |distance(query, pivot) - distance(pivot, test)| \quad (2)$$

If `|distance(query, pivot) - distance(pivot, test)| > minimum` holds for any app `test` inside a partition, we can safely ignore this partition during the search. Thus we can use the following pruning conditions to skip any irrelevant partition P_i because it is not possible to find a shorter distance than `minimum`:

$$minimum < distance(pivot, query) - P_i.MAX \quad (3)$$

$$minimum < P_i.MIN - distance(pivot, query) \quad (4)$$

In Algorithm 2, we outline how nearest neighbor search works in VPT. It has two inputs: the query app `query` and the current root VPT node `currentNode`. During the search, we maintain two global variables, i.e., `nearest_neighbor` and the current `minimum` distance. If we reach a leaf node (`lines 1 to 10`), the algorithm

Algorithm 2 Nearest Neighbor Search in VPT: `nearestNeighborSearch(query, currentNode)`

```
1:  if currentNode is a leaf node then
2:    for each app in this leaf node do
3:      if app and query not from the same author then
4:        if minimum > distance(app, query) then
5:          minimum = distance(app, query)
6:          nearest_neighbor = app
7:        end if
8:      end if
9:    end for
10: end if

11: if currentNode is a pivot node then
12:   if pivot and query not from the same author then
13:     if minimum > distance(pivot, query) then
14:       minimum = distance(pivot, query)
15:       nearest_neighbor = pivot
16:     end if
17:   end if
18:   for each partition P_i of this pivot code do
19:     if minimum< distance(pivot, query)-P_i.MAX or
          minimum< P_i.MIN-distance(pivot, query) then
20:       continue
21:     end if
22:     nearestNeighborSearch(query, P_i)
23:   end for
24: end if
```

simply computes the distances between the `query` app and each `app` stored in this leaf node. If any distance is smaller than current `minimum` value and they are also from different developers, we locate a closer distance and accordingly update `minimum` and `nearest_neighbor` (`lines 4 to 7`). If we hit a pivot node (`lines 11 to 24`), the same procedure will be applied to the pivot app. Moreover, we will further examine each partition of this pivot node (`line 18`). If any of the pruning conditions in Formula 3 and Formula 4 are satisfied (`line 19`), this partition can be safely skipped; otherwise the nearest neighbor searching procedure will be recursively invoked on this partition (`line 22`). To speed up the search, we can also initialize the `minimum` to a small number that indicates the acceptable level for piggybacking detection. This is possible because our previous module decoupling technique only retains primary modules for comparison while removing other non-essential ones (i.e., non-primary modules) as noise. Also, instead of only returning one `nearest_neighbor`, we can adjust the algorithm to report a list of apps that fall in a range of distance with the `query` app. In either case, the algorithm has the time complexity of $O(n \log n)$.

2.3 Piggybacking Identification and Rider Analysis

By iterating through each app collected from an app market, our algorithm effectively reports a list of related apps which share similar primary modules and thus are candidates for piggybacked apps. To identify the exact piggybacking relationship, we take into account non-primary modules of related apps. Specifically, for each reported pair `A` and `B`, we match their non-primary modules. If the non-primary modules of an app `A` are a strict sub-set of `B`, any non-primary modules in `B`, but not in `A`, will be considered part of the rider code. Accordingly, we label the app with the rider

code as the piggybacked app, and the other as the corresponding carrier app. If both apps have non-matched modules standing out, we choose to report them as a piggybacked pair, as we are not able to determine which one is piggybacked.

Besides determining the piggybacking relationship, we are also interested in what functionality is implemented in the rider code. While manual analysis in general cannot be avoided, our investigation shows that the same rider code may be injected into multiple piggybacked apps. Accordingly, we elect to cluster the detected rider code and group them for correlation. By doing so, we are able to identify several clusters whose members are very similar to each other. In our prototype, we choose to reuse the previous algorithm and build another VPT tree (Section 2.2) *only* for these identified riders. Our experience shows that the number of rider-related non-primary modules is one magnitude smaller than that of apps, which allows us to select a smaller distance (as the range parameter). As to be shown in Section 3.5, such clustering quickly exposes several clusters with the same rider code piggybacking on a number of carrier apps.

3. PROTOTYPING AND EVALUATION

We have implemented a prototype of PiggyApp in Linux. In our prototype, the first component – module decoupling (Section 2.1) – is implemented by extending the open source Dalvik disassembler `baksmali` [3] (with an additional 1926 lines of Java code) to generate the program dependency graph (PDG) and then isolate primary modules from other non-primary modules. When generating the graph, we assign weights 10, 10, 2, 1 to edges representing class inheritance, package homogeny, method calls, and member field references, respectively. The cut-off value for the `merge_able` condition (Algorithm 1) is empirically set to 5, which works well in practice (Section 3.2).

The second component – feature fingerprint and representation (Section 2.2) – extracts the semantic features of $32,011$ APIs, 136 permissions, 122 intent types, 180 content provider features, and 2 additional code loading features, which essentially condense each app into a feature vector of length $32,451$. These feature vectors are then organized into a Vantage Point Tree (VPT) that is implemented in $2,731$ lines of C code. For search efficiency, we set the number of partitions at each pivot node to 3. To strike a good balance between accuracy and efficiency, we select Jaccard distance 0.15 for the similarity measurement of two primary modules. We will detail how we choose this Jaccard distance in Section 3.3.

The third component – piggybacking identification and rider analysis (Section 2.3) – is implemented in 611 lines of Python code. Basically, it scans the list of candidate app pairs reported from the second component, fetches the non-primary modules of related apps, determines the piggybacked apps, and exposes the rider code. In our implementation, we re-target the second component to organize the rider code with one exception: no author information is needed to constrain the nearest neighbor search.

To evaluate the scalability and efficacy of our system, we use it to detect piggybacked apps in a dataset with $84,767$ apps collected from seven different app marketplaces. In the following, we first present our evaluation setup and then assess the accuracy of our module decoupling technique. After that, we determine the Jaccard distance for similarity measurement and report the detection results, including the analysis of uncovered rider code. Finally, we report the performance overhead.

3.1 Evaluation Setup

In Table 1, we summarize the collected apps in our dataset. Basically, our crawler takes a snapshot of the available apps from seven different Android marketplaces in the first week of March, 2011. In total, the dataset contains $84,767$ distinct apps: $68,187$ of them appear in the official Android Market [25] and the rest come from other six popular third-party marketplaces: two in the US, two in China, and two in Eastern Europe. We highlight that we analyzed all these $84,767$ apps in this data set, which is made possible by our scalable analysis framework. Earlier systems such as DroidMOSS [44] employ pairwise comparison, which is not scalable and can only work on limited samples (e.g., 200 in [44]).

Table 1: The dataset for PiggyApp evaluation (†: the number in parenthesis shows the percentage of apps that are also hosted in the official Android Market.)

Marketplace	Total Number of Apps
slideme (US1)	3108 (29.8%†)
freewarelovers (US2)	3188 (13.2%†)
eoemarket (CN1)	8261 (30%†)
goapk (CN2)	4334 (13.5%†)
softportal (EE1)	2305 (19.6%†)
proandroid (EE2)	1710 (20.2%†)
Official Android Market	68187

For each app in our dataset, our system extracts $32,451$ semantic features and presents them in a vector. In total, our system produces 84,767 feature vectors. To understand the distribution of each app pair distance, we randomly select $2,000$ and $4,000$ samples and measure their distances with all other apps in the dataset. The results are shown in Figure 4 (with the y-axis in the log-scale). As expected, most apps are not similar to each other, which is reflected by the fact that a majority of distances (around 99.4%) are larger than 0.8 (the largest possible distance is 1). Also, there are a small fraction (0.06%) of apps whose distances fall below distance 0.2, suggesting most piggybacked apps are located in this range (Section 3.4). This distribution is helpful to create a balanced VPT tree and leads to efficient nearest neighbor search.

3.2 Module Decoupling Accuracy

Module decoupling is an essential component, which affects both the accuracy and efficiency of our system. To concretely evaluate its effectiveness, we randomly choose 200 samples from our dataset, apply the module decoupling algorithm (Algorithm 1), and then manually verify the decoupling results. As with the module decoupling process, verification involves two main aspects. The first one is to determine whether these apps are decoupled into the correct modules. Our results show that 193 apps (96.5%) are correctly decoupled. The second aspect is to determine whether primary modules are correctly labeled. For the correctly decoupled 193 apps, our system identifies 178 primary modules (92.2%). We further examine the 15 mis-labeled apps and find that most cases, especially game- and social network-related apps, use feature-rich engines or libraries (e.g., Scoreloop [30] and Openfeint [38]) for GUI rendering, user interaction, and virtual currency support. They are generally considered as the main functionality of an app, but are implemented as supporting frameworks and shared among related apps. As these mis-labeled cases are rare and usually come from a limited number of special-purpose SDKs, we choose to apply a quick patch to our prototype by using a short white-list.

3.3 Jaccard Distance Trade-Off

Next we present our experiments to determine the proper Jaccard distance in our study. As it measures the overlapped semantic

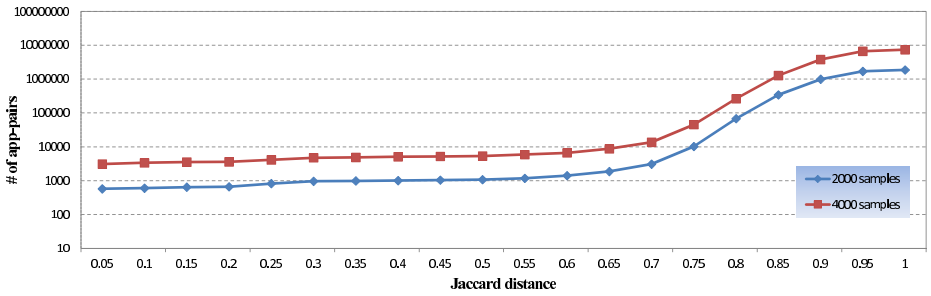

Figure 4: The cumulative distribution of pair-wise Jaccard distances

Table 2: Determining the right Jaccard distance

Jaccard Distance	0.05	0.1	0.15	0.2	0.25	0.3
True Positives	28	38	39	39	39	39
False Positives	0	0	2	2	4	4

features, our feature fingerprint is by design robust to existing code obfuscation techniques [2]. Moreover, it provides a "tuning knob" to adjust the trade-off between accuracy and efficiency. Specifically, a larger distance will likely tolerate more disjoint features between two apps, which has the benefit of reducing false negatives but at the cost of increasing false positives. A smaller distance leads to more false positives but less false negatives. As a general rule of thumb, if two apps have a Jaccard distance greater than 0.3, we consider the possibility of having a piggybacking relationship to be very low.

In our study, we aim to achieve a lower threshold to improve the search efficiency while still obtaining sufficient accuracy. To this end, we choose $4,000$ random samples and use a series of Jaccard distances to measure their accuracy. Specifically, for each distance, we calculate the true positives and false positives by examining each reported pair (as candidate piggybacked apps). The results are shown in Table 2. The experiments clearly indicate the Jaccard distance 0.15 as the threshold. In particular, the larger distances (> 0.15) detect two more pairs, but they are false positives. We also do not want to choose smaller distances because we still detect more true positives as we move closer to 0.15 threshold. This result is consistent with an earlier measurement reported in Figure 4.

We want to emphasize that the Jaccard distance threshold provides a desirable way to balance between efficiency and accuracy. Based on the resources available to scrutinize candidate piggybacked apps, we can adjust the distance accordingly. For a smaller dataset with less than 10,000 apps, we might choose to use a larger distance so as to catch as many piggybacked apps as possible. For a larger dataset with hundreds of thousands of apps, we might want to use a smaller distance to accurately return a high-density set that contains true piggybacked apps. In our above series of experiments, when we use the distance 0.15, our system reports 41 candidate pairs within 630 seconds and the distance of 0.05 returns 28 within 227 seconds.

3.4 Piggybacking Detection

From the previous section, we have empirically chosen 0.15 as the optimal Jaccard distance threshold. In this section, we apply it to our dataset and present our detection results. Overall, our system detects $1,094$ (1.3%) piggybacked apps in our dataset. For these repackaged apps, we further obtain the corresponding carrier apps and then classify them based on their sources. The results are shown in Table 3.

In the table, the second column shows the number of piggybacked apps in each market and the third column contains the ratio of piggybacked apps to the number of apps we collected from each market. Due to the large number of apps it hosts, the official Android Market contains the largest number of piggybacked apps, but its piggyback rate is one of the lowest. The fourth column reports the number of carrier apps that have been chosen for piggybacking, which in general reflects which market is of interest to piggybacking authors in order to find popular apps to piggyback on. Our results show that game, wallpaper, and electronic book apps are among the most popular targets. Notice that the numbers in the carrier apps column are smaller than those in the piggybacked apps column. The reason is that the same carrier apps may be piggybacked multiple times to include different rider code (one concrete example is shown in the next section).

Table 3: Piggybacking detection results

App Marketplace	# Piggybacked Apps	Piggyback Rate	# Carrier Apps
Slideme (US1)	49	1.6%	32
Freewarelovers (US2)	31	0.97%	52
Eoemarket (CN1)	224	2.7%	98
Goapk (CN2)	83	1.9%	108
Softportal (EE2)	39	1.7%	26
Proandroid (EE1)	32	1.9%	15
Official Android Market	683	1.0%	298

Interestingly, when examining these piggybacked apps inside the official Android Market, we find that 513 out of 683 (75%) are actually based on carrier apps also located in the official market. This clearly indicates the need for the official Android Market to adopt a rigorous policing to detect and potentially remove them. Also, notice that the remaining 170 piggyback on apps from third-party markets. This may sound counter-intuitive at first glance, but it is actually reasonable for two reasons: first, the official Android Market may not always be accessible or convenient to users outside the US, which partially explains the popularity of third-party markets in China; second, by choosing popular apps in third-party markets and uploading piggybacked apps into the official one, the app repackagers could reach more users for download and thus potentially maximize their impact.

To further measure the false negative rate, we study a list of 77 apps that were known to be piggybacked in our dataset (before our system was designed). PiggyApp correctly identifies 73 of them and misses four, indicating a false negative rate of 5.2%. Our manual analysis shows that these four failing cases are due to our module decoupling implementation, which incorrectly labels certain non-primary modules as primary and thus results in unnecessarily large Jaccard distances of related pairs.

3.5 Rider Analysis

After detecting these piggybacked apps, it is also interesting to find out answers to the following questions: what are the purposes behind these piggybacked apps? What rider code is injected into carrier apps? Are there (additional) permissions the rider code asks for? If there are, what are they? To answer these questions, we

Table 4: The statistics of piggybacked ad libraries

Ad Library	Module Name	# Piggybacked Apps
admob	com.admob.android.ads	724
wooboo	com.wooboo.adlib_android	321
youmi	net.youmi.android	197
adwhirl	com.adwhirl	173
google/ads	com.google.ads	170
zestadz	com.zestadz.android	101
millennialmedia	com.millennialmedia.android	97
urbanairship	com.urbanairship.push	85
mobclix	com.mobclix.android.sdk	45
wiyun	com.wiyun.ad	36
mobclick	com.mobclick.android	26
greystripe	com.greystripe.android.sdk	26
madhouse	com.madhouse.android.ads	5

perform a further analysis on these piggybacked apps. As it is not feasible to examine every single piggybacked app, we choose to use cluster analysis to group and correlate the rider code.

The clustering analysis is motivated from our detailed investigation of these piggybacked apps. Specifically, when analyzing specific samples, we observe two common characteristics: first, many piggybacked apps share similar or even the same rider code; second, the same carrier apps are found piggybacked with different rider code. Our clustering analysis helps identify both of them.

In particular, to identify these common carrier apps, we simply count the number of each carrier app that occurs in the set of identified app pairs. One such example is a popular game app named `com.appspot.swisscodemonkey.steam`, which has been piggybacked on at least six times: four of them are variants of the `Pjapps` malicious payload [31], one is the `ADRD` malicious payload [4], and the other is an ad library named `wooboo` [33].

To locate the related rider code, we again apply our feature fingerprint technique to fetch the feature vectors of the rider code (there are 2067 of them present in 1094 different piggybacked apps) and apply the same nearest neighbor search algorithm (Algorithm 2). In this case, instead of choosing the previous threshold 0.15, we select 0.2 to loosely group rider code. As a result, we identify 16 clusters (ranging in size from 5 to 724). For each cluster, we randomly choose some samples for manual investigation. By doing so, we significantly reduce the time and effort needed to analyze them. From the analysis, it becomes evident the inclusion of rider code mainly serves two purposes. The first one is to inject various ad libraries with the intention to collect ad revenue or steal it from the original developers. The second one is to enclose malicious payloads to directly control compromised phones or steal personal information on the phones. In the next two sections, we examine these two purposes in more detail.

3.5.1 Collecting Ad Revenue

In the first purpose, the piggybacked apps are used to insert additional ad libraries, which help the repackagers, instead of the original developers, to collect ad revenues (generated from users' views or clicks). As most of existing apps are free, developers want to monetize by including ad libraries and there are a variety of them [24, 29, 33], which are provided as standalone packages for simple reuse. Many of them require little or no change on the original code. Examples include `admob` [24] and `mobclix` [29]. Such convenience also makes it easy for repackagers to integrate them into popular apps as their carriers. Among the detected 1,094 piggybacked apps, 1,068 (97.6%) fall in this category. In Table 4, we show 13 top ad libraries in the rider code.

Table 5: The statistics of piggybacked malicious payloads

Malware Family	Module Name	# Piggybacked Apps
Geinimi	com.geinimi	6
ADRD	com.xxx.yyy	1
Pjapps	com.android.main	8
DDream	com.android.root	10
BgServ	com.mms.bg	1

In the table, the first column shows the library name, the second column contains its detailed module name, while the third column counts the number of piggybacked apps that have it embedded. Among these 13 ad libraries, `admob` tops the list by being present in 724 carrier apps in our dataset. These ad libraries naturally request their own permissions for the provided functionality, some of which may not be requested by the carrier apps. It turns out that all these ad libraries ask for the `INTERNET` permission, 9 of them request the `LOCATION` permission, 5 need `READ_PHONE_STATE`, 1 demands `CALL_PHONE`, 1 uses `ACCESS_WIFI_STATE`, and 1 makes use of `ACCESS_NETWORK_STATE`. On average, these modules ask for 2.3 permissions.

3.5.2 Injecting Malicious Payloads

In the second purpose, repackagers implant malicious payloads into chosen carrier apps. In our dataset, we discover 5 different malicious rider payloads embedded in 26 different carrier apps that are present in various app markets. These malware are all listed in the yearly report of Android malware [39]. In Table 5, we show the list of detected malicious rider payloads. In the following, we choose representative samples and present our analysis.

`Geinimi` [27] is one of the earliest Android malware discovered in the wild that piggybacks on legitimate apps to perform malicious activities on the background. Our system identifies 6 piggybacked apps that have similar `Geinimi` code embedded (two different variants with 96% of their code in common). Both variants add their own activity, which once triggered invokes the embedded malicious code, including the bootstrap of a new service. These variants also add a new receiver to register for callbacks when certain events such as `SMS_RECEIVED` and `BOOT_COMPLETED` happen. By doing so, the malware can immediately run once the system boots or when a short message is received. To accomplish all these tasks, `Geinimi` needs 17 different permissions.

Compared to `Geinimi`, `ADRD` [4] is less complicated. Based on our analysis, the rider code is composed of four new receivers, which listen on the system boot completion event `BOOT_COMPLETED`, phone state change `PHONE_STATE`, network connection state change `CONNECTIVITY_CHANGE` and its own alarm timer `com.lz.myservicestart`. It also defines a new service that will send device-specific information to a remote server and receive instructions from it. In the piggybacked app, the main entry point remains the same as in the carrier apps. In our dataset, we only find one `ADRD`-piggybacked app with a new module named `com.xxx.yyy`. Overall, `ADRD` demands 7 different permissions.

`Pjapps` [31] is another malicious rider embedded in a number of carrier apps. In our dataset, there are 8 of them. All the related rider code share the same class package named `com.android.main`. In essence, it adds two new receivers and one more service. The internal mechanism of these new components works similar to that of `ADRD`. In total, `Pjapps` requests 9 permissions.

`DroidDream` [28] was reported in the official Android Market and our dataset contains 10 infected apps. Similarly, all of its rider code share a common module named `com.android.root`. As

with Geinimi, it adds its own activity and starts a new service to set up an alarm timer, which in return triggers another service to launch one root exploit to elevate its privilege. With the root exploit, DroidDream requests fewer permissions, but can essentially do whatever it wants on the compromised devices.

The last piece of malicious rider code is from the BgServ malware, which injects one module named com.mms.bg into carrier apps to transport device-specific information (via short messages) to a remote party. One interesting thing about this payload is that it leverages an open source project hosted at Google code projects [1]. For its wrongdoings, BgServ asks for 9 permissions.

Overall, these malicious payloads all request more permissions than original carrier apps, which imply that the request for a bulk of permissions may be an indicator for potentially suspicious apps.

3.6 Performance

In this section, we report the performance measurement of our system. Our test runs on a Ubuntu Server 10.04 Linux machine with an four-core Intel Xeon CPU (2.67HZ) and 8G memory. Our current prototype runs on a single thread, which leaves room for future improvement to take advantage of multiple threads for speed-up.

In our test, we run the module decoupling and feature extraction separately, which is easily parallelized. Each app, depending on its complexity, takes from 0.167 to 5.492 seconds to process. On average, it takes 0.952 seconds to process one app. Our module decoupling task of all these 84,767 apps takes 5 hours and 36 minutes in total. The construction of the VPT tree requires 126 seconds. Based on the VPT tree, given a single app, our algorithm takes between 0.00001 seconds and 0.576 seconds to find its nearest neighbor with 0.133 seconds on average. To iterate the apps in our dataset to locate possible piggybacked apps, it takes 3 hours and 15 minutes in total. The memory footprint seems small as only 127M main memory is used.

When streamlining the processing of these components, it takes less than nine hours to analyze our dataset with 84,767 apps from seven different Android markets. As mentioned earlier, improvements exist to better parallelize various components so that we can further reduce the processing time.

4. DISCUSSION

Our prototype demonstrates promising results by allowing for fast and scalable detection of piggybacked apps. In this section, we examine possible limitations in the current prototype and discuss future improvements.

First, to infer piggybacking relationship, we extract semantic features based on primary modules of existing apps. These semantic features are mainly based on the Android APIs, requested permissions, and various intents, etc. Though these features satisfy our current needs, they are still limited in a number of ways. To improve the system, we could extend these semantic features to include syntactic instruction sequences [44] or control-flow graphs. The addition will be helpful to better characterize and identify a particular app. Meanwhile, our core prototype remains intact as we can easily expand feature vectors to accommodate them and reuse the same VPT tree for construction and lookup.

Second, our current prototype largely depends on the existence of authentic carrier apps in order to detect piggybacked apps. Unfortunately, due to the variety, scale, and dynamic nature of existing app markets, it is not possible to build a centralized repository having every app in existence; our current collection is far from complete. For example, there are cases where we can infer a potential piggybacking relationship but can not determine which one is actually piggybacked (Section 2). Also, our collection is comprised of only free apps and does not include paid apps, which are sold for their features and will likely be attractive targets for piggybacking. This also indicates the need to continuously expand our current data set with more comprehensive samples.

Finally, our current prototype basically serializes the execution of different components. As mentioned earlier, for improved performance, there is a need to re-engineer our prototype for a parallel version. Fortunately, the overall system design of PiggyApp is parallelizable in nature and does not require complete revamp.

5. RELATED WORK

Software similarity measurement and searching The first category of related work includes prior efforts in effectively measuring software similarity and detecting plagiarized code [44, 12, 34, 23]. Among all these works, DroidMOSS [44] is a closely related one to measure the similarity of mobile apps. PiggyApp differs from it in three aspects: First, DroidMOSS detects repackaged apps while PiggyApp focuses on piggybacked apps. As mentioned earlier, piggybacked apps are a subset of repackaged ones but pose greater security threats. Second, our system overcomes the scalability limitation from the pair-wise comparison in DroidMOSS. Specifically, by proposing a new distance metric design and the associated nearest neighbor search algorithm, our system achieves better detection efficiency and scalability (with $O(n \log n)$ complexity, instead of $O(n^2)$), and enables a large scale evaluation instead of sampling based study. Third, for app comparison, DroidMOSS depends on syntactic instruction sequences of entire apps while PiggyApp extracts semantic features *only* from their primary modules, which leads to better accuracy and efficiency. DNADroid [12] uses program dependency graph (PDG) to characterize Android app and compares PDGs between methods in app pair, showing resistance to several evasion techniques. But it also applies pair-wise comparison as DroidMOSS, therefore lacking the scalability as presented in PiggyApp.

Smit [23] leverages a similar Vantage Point Tree but for large scale malware indexing and queries. In particular, by focusing on detecting malware variants, it does not have the need to further decouple internal modules, which is essential for our system. Similarly, BitShred [34] focuses on large-scale malware triage analysis by using feature hashing techniques to dramatically reduce the dimensions in the constructed malware feature space. After reduction, pair-wise comparison is still necessary to infer similar malware families. In comparison, PiggyApp focuses on a different problem, i.e., piggybacking detection among existing mobile apps, which necessitates module decoupling-like techniques to partition apps into primary and non-primary modules. Also, our linearithmic nearest neighbor search algorithm avoids the need of performing pair-wise comparison.

Program comprehension As our system centers on module decoupling to identify piggybacking relationships, we also consider the second category of related work from existing program comprehension techniques. Specifically, module decoupling has been an important tool to comprehend and manage large-scale legacy software that may not be well understood or easily maintained. To reduce a given large system into smaller and more manageable units, different techniques have been proposed in two general subcategories. The first one concerns different clustering techniques, which use a variety of software metrics to group smaller units of code into larger modules [5, 36]. Our module decoupling technique is based on one such clustering method – agglomerative clustering. In particular, our system initially considers Java class packages

as the smallest unit and then infers different kinds of program dependencies as metrics to merge them. The second one is concept analysis [35, 42], which in general uses functions as the smallest unit and employs lattice theory to group functions that have some specific common attributes. We consider function-level granularity may not be appropriate for our purpose and thus have not explored it further.

Mobile app security and analysis The third category covers a variety of projects [15, 37, 16, 47, 19, 17, 11, 21, 46, 22, 45, 20] that have been undertaken to analyze or improve mobile security. Specifically, they can be loosely classified into two groups. The first group of projects analyze a single app from various perspectives to identify possible security and privacy problems. For example, both TaintDroid [15] and PiOS [14] focus on the privacy leak problem, and respectively use dynamic taint analysis and static data flow analysis to infer potential privacy leaks. DroidRanger [47] instead combines both static permission analysis and dynamic footprint monitoring to detect malicious applications in existing Android marketplaces. SCanDroid [19] examines an app's manifest file to automatically extract a data flow policy, and then checks whether the data flows in the app are consistent with the extracted specification. Stowaway [17] studies a set of 940 apps and finds that about one-third are not following the principle of least privilege. Enck *et al.* [16] crafted a byte code decompiler to study $1,100$ popular Android apps for characterization. All these tools use static analysis, or dynamic analysis, or both techniques to infer some specific security or privacy properties for individual mobile app. In contrast, PiggyApp uses feature vectors, rather than more expensive and complicated static or dynamic analysis techniques, to enable the rapid comparison of *pairs* of apps.

The second group is more closely related to PiggyApp, as it involves the interactions between apps. For example, one line of research [37, 11, 18, 21, 13, 10] studies the security risks causes by inter-application interaction. Among them, both ComDroid [11] and Saint [37] examine the interfaces third-party apps export in order to uncover possible unintended consequences. Woodpecker [21] focuses on a similar "capability leak" problem in Android firmware apps preloaded on the device. Numerous solutions to this problem have been proposed. For example, Saint [37] further extends the Android framework to enforce a user-configurable inter-application policy. Felt *et al.* [18] proposes a mechanism called IPC Inspection that allows the framework to inspect the complete call chain that leads to a request for a dangerous feature. QUIRE [13] addresses the same permission delegation problem by proposing IPC call chain tracking to identify the provenance of these IPC requests and then enforce security checks. Bugiel et al. [10] use a runtime monitor to regulate communications between apps, to protect Android against confused deputy and colluding apps attacks. While PiggyApp is concerned with the relationship between apps and the interfaces they export, it differs substantially from these systems in that it does not attempt to model the flow of information or control through an app; PiggyApp is concerned with the similarity between two apps, not what they do or whether they may be tricked into doing something inappropriate.

Another line of research [7,8] focuses more broadly on entire app markets. For example, Stratus [7] explores the security problem of the whole app ecosystem composed of multiple markets, each of which has its own security policy, and proposes a new app installation method to retain the original single-market security semantics (e.g., kill switches or developer name consistency). While its focus is markedly different from ours, both directly concern the issues that face app marketplaces today. Barrera *et al.* [8] uses a self-organization map to analyze 1,100 popular Android apps and identifies common usage patterns of permissions shared by different apps. Our approach also employs clustering techniques and extracts certain semantic features from the set of permissions an app requests. However, Barrera *et al.* uses them to visualize the relationships (in terms of requested permissions) among popular apps. PiggyApp instead makes use of them to build a VPT tree for efficient piggybacking detection.

6. CONCLUSION

In this paper, we present PiggyApp, a system for fast and scalable detection of piggybacked apps in existing Android markets. Based on the observation that in a piggybacked app, the added rider code is loosely coupled with the primary functionality (or module) of the original app, we develop a module decoupling technique to effectively locate the primary module for comparison. To avoid pair-wise comparison, we further propose a scalable approach to extract semantic features from the decoupled primary modules and organize them in a metric space, which allows for fast and efficient search (with $O(n \log n)$ complexity). We have implemented a prototype and used it to detect piggybacked apps in a dataset collected from seven different markets. Our results show that 0.97% to 2.7% of apps hosted in these markets, including the official Android Market, are piggybacked. Based on these results, we further analyze rider code and find that it mainly serves two purposes: stealing ad revenue and implanting malicious payloads. These results call for a rigorous vetting process for their detection.

Acknowledgements The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. We also want to thank Chiachih Wu, Minh Q. Tran, Lei Wu and Kunal Patel for the helpful discussion. This work was supported in part by the US National Science Foundation (NSF) under Grants 0855297, 0855036, 0910767, and 0952640. 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 NSF.

7. REFERENCES

[1] MMSBG: An Open-Source Project. `https://code.google.com/p/mmsbg`. Online; accessed at Dec 1, 2011.

[2] ProGuard | Android Developers. `http://developer.android.com/guide/developing/tools/proguard.html`. Online; accessed at Dec 1, 2011.

[3] Smali - An Assembler/Disassembler for Android's dex Format. `http://code.google.com/p/smali/`. Online; accessed at Dec 1, 2011.

[4] AndroidCommunity. [ALERT] New Trojan Called Hong Tou Tou Lurking. `http://androidcommunity.com/android-trojan-alert-hong-tou-tou-20110216/`. Online; accessed at Dec 1, 2011.

[5] Nicolas Anquetil, Cédric Fourrier, and Timothy C. Lethbridge. Experiments with Clustering as a Software Remodularization Method. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, WCRE '99, pages 235–, Washington, DC, USA, 1999. IEEE Computer Society.

[6] AppBrain. Number of Available Android Applications. `http://www.appbrain.com/stats/number-of-android-apps`. Online; accessed at Dec 1, 2011.

[7] David Barrera, William Enck, and Paul Oorschot. Seeding a Security-Enhancing Infrastructure for Multi-market Application Ecosystems. Technical report, School of

Computer Science, Carleton University, http://www.scs.carleton.ca/shared/research/tech_reports/2010/TR-11-06%20Barrera.pdf. Online; accessed at Dec 1, 2011.

[8] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil Somayaji. A Methodology for Empirical Analysis of Permission-Based Security Models and Its Application to Android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, 2010.

[9] Joany Boutet. Malicious Android Applications: Risks and Exploitation - A Spyware Story about Android Application and Reverse Engineering. http://www.sans.org/reading_room/whitepapers/malicious/malicious-android-applications_risks-exploitation_33578. Online; accessed at Dec 1, 2011.

[10] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *19th Annual Network & Distributed System Security Symposium (NDSS)*, Feb 2012.

[11] Erika Chin, Adrienne Felt, Kate Greenwood, and David Wagner. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys 2011, 2011.

[12] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Computer Security âĂŞ ESORICS 2012*, volume 7459 of *Lecture Notes in Computer Science*, pages 37–54. Springer Berlin Heidelberg, 2012.

[13] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the 20th USENIX Security Symposium*, USENIX Security '11, San Francisco, CA, 2011.

[14] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, NDSS '11, February 2011.

[15] William Enck, Peter Gilbert, Byung-gon Chun, Landon Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. 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 '11, 2011.

[16] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium*, USENIX Security '11, San Francisco, CA, 2011.

[17] Adrienne Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS' 11, 2011.

[18] Adrienne Felt, Helen Wang, Alexander Moschhuk, Steve Hanna, and Erika Chin. Permission Re-Delegation: Attacks and Defense. In *Proceedings of the 20th USENIX Security Symposium*, USENIX Security '11, San Francisco, CA, 2011.

[19] Adam Fuchs, Avik Chaudhuri, and Jeffrey Foster. SCanDroid: Automated Security Certification of Android Applications. http://www.cs.umd.edu/~avik/projects/scandroidascaa/paper.pdf. Online; accessed at Dec 1, 2011.

[20] Michael Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[21] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, NDSS '12, February 2012.

[22] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *10th International Conference on Mobile Systems, Applications and Services*, June 2012.

[23] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. Large-Scale Malware Indexing using Function-Call Graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 611–620, New York, NY, USA, 2009. ACM.

[24] Google Inc. Admob for Android Developers. http://developer.admob.com/wiki/Android.

[25] Google Inc. Android Market. https://market.android.com/. Online; accessed at Dec 1, 2011.

[26] Lookout Inc. App Genome Report: February 2011. https://www.mylookout.com/appgenome/. Online; accessed at Dec 1, 2011.

[27] Lookout Inc. Security Alert: Geinimi, Sophisticated New Android Trojan Found in Wild. http://blog.mylookout.com/2010/12/geinimi_trojan/. Online; accessed at Dec 1, 2011.

[28] 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 at Dec 1, 2011.

[29] MobClix Inc. Mobclix SDK Integration Guide. http://support.mobclix.com/attachments/token/lvbgrqsfpjgvgxb/?name=Detailed_Start_Guide_for_Android.pdf. Online; accessed at Dec 1, 2011.

[30] Scoreloop Inc. Scoreloop : Cross Platform Mobile Gaming SDK for Virtual Currency, Social Games and Distribution. http://www.scoreloop.com/developers/.

[31] Symantec Inc. Android Threats Getting Steamy. http://www.symantec.com/connect/blogs/android-threats-getting-steamy. Online; accessed at Dec 1, 2011.

[32] Symantec Inc. Android.Basebridge: Technical Details. http://www.symantec.com/security_response/writeup.jsp?docid=2011-060915-4938-99&tabid=2. Online; accessed at Dec 1, 2011.

[33] Wooboo Inc. How to Add Wooboo Advertisement SDK into Android. http://admin.wooboo.com.cn:9001/cbFiles/down/1272545843644.swf.

[34] Jiyong Jang, David Brumley, and Shobha Venkataraman. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *Proceedings of the 18th ACM*

conference on Computer and communications security, CCS '11, pages 309–320, New York, NY, USA, 2011. ACM.

[35] Christian Lindig and Gregor Snelting. Assessing Modular Structure of Legacy Code based on Mathematical Concept Analysis. In *Proceedings of the 19th international conference on Software engineering*, ICSE '97, pages 349–359, New York, NY, USA, 1997. ACM.

[36] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In *Proceedings of the 6th International Workshop on Program Comprehension*, IWPC '98, pages 45–, Washington, DC, USA, 1998. IEEE Computer Society.

[37] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, 2009.

[38] OpenFeint. OpenFeint Developers - Mobile Open Source Social SDK & Tools for iOS & Android. `http://openfeint.com/developers`. Online; accessed at Dec 1, 2011.

[39] Paolo Passeri. One Year of Android Malware (Full List)). `http://paulsparrows.wordpress.com/2011/08/11/one-year-of-android-malware-full-list/`. Online; accessed at Dec 1, 2011.

[40] Google Code Project. Android-apktool - Tool for Reengineering Android apk Files. `http://code.google.com/p/android-apktool/`. Online; accessed at Dec 1, 2011.

[41] Helmuth Spaeth. *Cluster Analysis Algorithms for Data Reduction and Classification of Objects*. J. Wiley and Sons, 1980.

[42] Paolo Tonella. Concept Analysis for Module Restructuring. *IEEE Trans. Softw. Eng.*, 27:351–363, April 2001.

[43] 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, pages 311–321, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[44] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 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, February 2012.

[45] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[46] Yajin Zhou and Xuxian Jiang. Detecting Passive Content Leaks and Pollution in Android Applications. In *Proceedings of the 20th Annual Symposium on Network and Distributed System Security*, 2013.

[47] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 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, February 2012.