Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces

Wu Zhou, Yajin Zhou, Xuxian Jiang, Peng Ning

North Carolina State University
890 Oval Drive, Raleigh, NC 27695
{wzhou2, yajin_zhou xuxian_jiang, pning}@ncsu.edu

ABSTRACT

Recent years have witnessed incredible popularity and adoption of smartphones and mobile devices, which is accompanied by large amount and wide variety of feature-rich smartphone applications. These smartphone applications (or apps), typically organized in different application marketplaces, can be conveniently browsed by mobile users and then simply clicked to install on a variety of mobile devices. In practice, besides the official marketplaces from platform vendors (e.g., Google and Apple), a number of third-party alternative marketplaces have also been created to host thousands of apps (e.g., to meet regional or localization needs). To maintain and foster a hygienic smartphone app ecosystem, there is a need for each third-party marketplace to offer quality apps to mobile users.

In this paper, we perform a systematic study on six popular Android-based third-party marketplaces. Among them, we find a common "in-the-wild" practice of repackaging legitimate apps (from the official Android Market) and distributing repackaged ones via third-party marketplaces. To better understand the extent of such practice, we implement an app similarity measurement system called DroidMOSS that applies a fuzzy hashing technique to effectively localize and detect the changes from app-repackaging behavior. The experiments with DroidMOSS show a worrisome fact that 5% to 13% of apps hosted on these studied marketplaces are repackaged. Further manual investigation indicates that these repackaged apps are mainly used to replace existing in-app advertisements or embed new ones to "steal" or re-route ad revenues. We also identify a few cases with planted backdoors or malicious payloads among repackaged apps. The results call for the need of a rigorous vetting process for better regulation of third-party smartphone application marketplaces.

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 Algorithms, Measurement, Security

Keywords Smartphones, Privacy and Security, Repackaging

CODASPY'12, February 7–9, 2012, San Antonio, Texas, USA.

1. INTRODUCTION

Smartphones have recently gained much popularity and demand. A recent report shows that in July of 2011, the number of Android smartphone activations reached $550,000$ each day [34]. Besides the portability and mobility, the popularity is also possibly due to the large amount and wide variety of feature-rich smartphone applications (or apps) mobile users can install and experience. As an example, there already have more than $200,000$ apps in the official Android Market [3] (starting May 2011). These feature-rich apps extend the capability of smartphones by empowering users to browse, socialize, entertain, and communicate on the go with unprecedented convenience and experiences, instead of limiting users for basic phone calls or simple text messages.

To allow for mobile users to conveniently browse and install these smartphone apps, platform vendors created centralized marketplaces, including Apple's App Store [23] and Google's Android Market [26]. Through the centralized marketplaces, developers can submit their apps to these marketplaces and make them available to thousands of users. Platform owners can also better control the quality of apps and block malicious ones to protect users. Meanwhile, a number of third-party marketplaces were also created for various purposes (e.g., to meet regional or localization needs). Cydia [10] and Amazon AppStore [22] are two such examples that host thousands of apps for iPhone and Android users, respectively.

To maintain and foster a hygienic smartphone app ecosystem, there is a need for third-party marketplaces to offer quality apps to mobile users. In this paper, we focus on six popular Android-based third-party marketplaces and perform a systematic study on the $22,906$ apps collected from them. Particularly, we observe that the apps hosted in these third-party marketplaces can be classified into three categories: The first category includes apps that are also available in the official Android Market. It is possible as app developers may choose to submit their apps to both official and alternative marketplaces to reach more users. The second category contains apps that are only available from the third-party marketplaces. The reason is that developers may create apps targeting specific customers (e.g., in their own regions, countries, or languages). The third category, which is the main focus of our study, consists of apps that are repackaged from the official Android Market and re-distributed to third-party marketplaces. Specifically, repackaged apps are based on legitimate ones, but for whatever reasons, include some "value-added" functionality or modification. Unfortunately, repackaged apps could lead to a number of problems. For example, if a legitimate app is repackaged with additional malicious payloads or exploits, users may find their phones compromised, phone bills increased or sensitive information (stored on the phones) stolen. App developers are also impacted as they find their intellectual properties violated, the

in-app revenues stolen, and their reputation impaired. From the marketplace perspective, they are also seriously affected because mobile users and developers are their customers, who might turn away to other marketplaces for better-quality apps. As a result, the entire smartphone app ecosystem could seriously suffer from repackaged apps.

In this paper, we are motivated to systematically detect repackaged apps on third-party Android marketplaces. In particular, we aim to shed some light on questions such as: How serious is the overall app repackaging situation in our current marketplaces? What purposes are these repackaged apps used for? Can we systematically identify them? Note that the answers can be greatly helpful in assuring users that a downloaded app is legitimate and does not contain any malicious payload. Moreover, developers are also protected since their intellectual properties are not violated. For marketplace owners, they can also ensure that their marketplaces are not populated with repackaged or trojanized apps.

As our first step, we present a system called DroidMOSS[1] to measure the similarity between two different apps and use it as the basis to detect repackaged apps. Specifically, given each app from a third-party Android marketplace, we measure its similarity with those apps from the official Android Market. In order to handle a large number of apps in the official and alternative marketplaces, we choose to extract some distinguishing features from apps, and generate app-specific fingerprints. Our fingerprint generation is based on a fuzzy hashing technique to localize and detect the modifications repackagers apply over the original apps. After that, we calculate the edit distance to gauge how similar each app pair is. When the similarity exceeds certain threshold, we consider one app in the pair is repackaged.

We have implemented a DroidMOSS prototype and used it to study six third-party Android marketplaces worldwide, including two from United States (with $6,296$ apps), two from China (with $12,595$ apps), and two from East Europe (with $4,015$ apps). These apps were collected in the first week of March, 2011 and are measured against the $68,187$ apps collected from the official Android Market in the same time frame. To perform a concrete analysis, we randomly picked up 200 apps from each of these six marketplaces, and measured their similarity with the total $68,187$ apps in the official Android Market. From the resulting $81,824,400$ pair-wise similarity scores, DroidMOSS systematically reports the repackaged apps. For each reported one, we perform a manual analysis and then calculate the false positive rate. Our results show that 5% to 13% of apps hosted in these six marketplaces are re-packaged (with false positive rates ranging from 7.1% to 13.3%). Also, we found that 13.5% to 30% of apps in these alternative marketplaces are simply redistributed from the official Android Market. A further manual investigation indicates that these repackaged apps are mainly used to replace existing in-app advertisements or embed new ones to "steal" or re-route ad revenues. We also identified a few serious cases with planted backdoors or malicious payloads in repackaged apps. These worrisome facts call for the imperative need of a rigorous vetting process in third-party marketplaces.

The rest of this paper is organized as follows: We describe the DroidMOSS system design for app similarity measurement in Section 2, followed by its prototyping and evaluation results in Section 3. After that, we discuss the limitations of our system and suggest possible improvement in Section 4. Finally, we describe related work in Section 5 and conclude this paper in Section 6.

[1]The name comes from an earlier system called MOSS [39] that is designed to measure software similarity and has been primarily used in detecting plagiarism in programming classes (based on source code submissions).

2. DESIGN

To systematically detect repackaged apps in third-party marketplaces, we have three key design goals: *accuracy*, *scalability*, and *efficiency*. Accuracy is a natural requirement to effectively identify app-repackaging behavior in current marketplaces. However, challenges arise from the fact that the repackaging process might dramatically change the function naming or code layout in the repackaged app, which renders whole-app hashing schemes ineffective. Also, due to the large number of apps in various marketplaces, our approach needs to be scalable and efficient. As a matter of fact, our current data set for app similarity measurement has $81,824,400$ app pairs, which makes expensive semantics-aware full app analysis not feasible. Accordingly, in our design, we choose to collect syntactic instruction sequences from each app and then distill them for fingerprint generation. The generated fingerprints need to be robust in order to accommodate possible changes from app-repackaging behavior.

Assumption In this paper, we aim to uncover repackaged apps in current marketplaces and understand the overall situation. We focus on Java code inside Android apps without considering native code. One reason is that native code is harder for repackager to modify. Also our dataset shows that only a small number of (5%) apps contains native code. Moreover, the apps from the official Android Market are assumed to be trusted and not re-packaged. There may exist exceptions to this assumption, but DroidMOSS is still helpful in distinguishing app pairs with repackaging relationship (Section 4). Finally, we assume that the signing keys from app developers are not leaked. Therefore, it is not possible that a repackaged app will be signed by the same author as the original one.

2.1 Overview

Repackaged apps share two common characteristics: First, due to the repackaging nature, the code base is similar between the original app and the repackaged app. Second, since the developers' signing keys are not leaked, the original app and the repackaged app must be signed with different developer keys. DroidMOSS capitalizes on these two insights by extracting related features from apps and then discerning whether one app is repackaged from the other.

Figure 1 shows an overview of our approach. In essence, DroidMOSS has three key steps. The first step is to extract from each app two main features, i.e., instructions contained in the app and its author information. These two features are used to uniquely identify each app. After that, the second step is to generate a fingerprint for each app. The reason is that each app may contain hundreds of thousands of instructions. There is a need to significantly condense it into a much shorter sequence as its fingerprint (for similarity measurement). Finally, based on app fingerprints, the third step discerns the source of apps, i.e., either from the official Android Market or from the third-party marketplaces, and measures their pair-wise similarity scores (so that we can detect repackaged apps). In the following, we examine each step in more detail.

2.2 Feature Extraction

Each Android app is essentially a compressed archive file, which contains the *classes.dex* file and a *META-INF* subdirectory. The *classes.dex* file contains the actual Dalvik bytecode for execution while the *META-INF* subdirectory contains the author information.

To extract Dalvik bytecode from *classes.dex*, we leverage existing Dalvik disassemblers (i.e., *baksmali* [1]). Initially, we use the Dalvik bytecode (with opcodes and operands) as the code

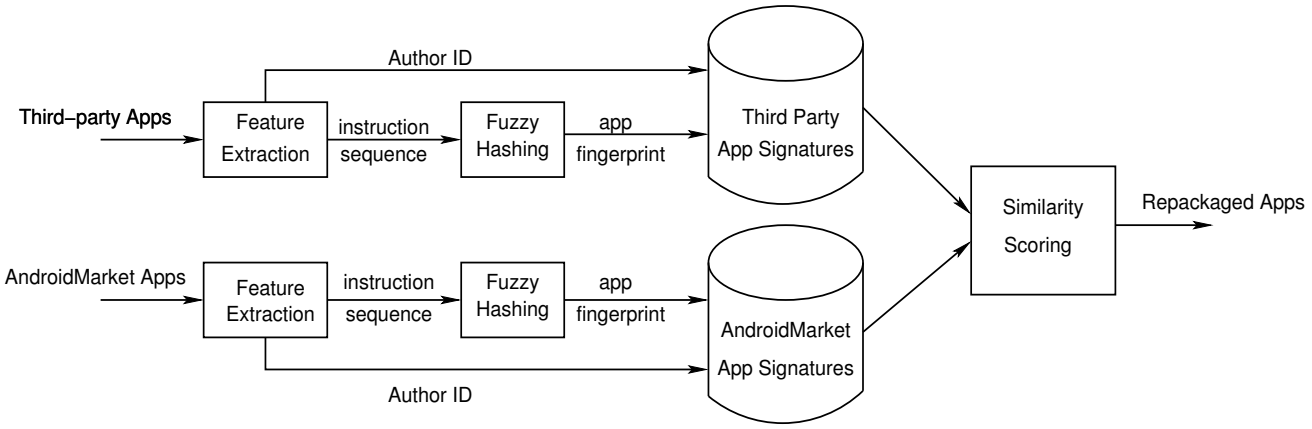

Figure 1: An Overview of DroidMOSS

feature directly. It turns out that it is not robust even for simple obfuscation that could just change some string operands (such as string names or hard-coded URLs). Because of that, we opt to make further abstraction by removing the operands and retaining only the opcode. The intuition is that it might be easy for repackagers to modify or rename the (non-critical) operands, but much harder to change the actual instructions. In the meantime, we also observe that apps intend to include various ad SDK libraries to fetch and display ads. After being disassembled, these shared ad libraries unnecessarily introduce noise to our feature extraction. Fortunately, there are a limited number of them and our current prototype builds a white-list to remove them from the extracted code.

For the author information, the *META-INF* subdirectory contains the full developer certificate, from which we can obtain the developer name, contact and organization information, as well as the public key fingerprints. For simplicity, we map each developer certificate into one unique 32-bit identifier (or authorID). This unique identifier is then integrated into the signature for comparison.

2.3 Fingerprint Generation

For each app, our second step generates a fingerprint from the extracted code. A common way of achieving that is through hashing. Although hashing the entire code sequence of an app can uniquely determine whether two apps are the same, they are not helpful to determine whether two files are similar. The reason is simply because one minor modification will dramatically change the hashing value. From another perspective, calculating the edit distance between two given sequences is a well-known technique to measure their similarity. Unfortunately, it cannot be directly applied either. Considering each instruction sequence (of an app) could have hundreds of thousands of instructions, it will be very expensive to calculate one single edit distance between two apps, not to mention the large number of apps each needs to be paired and compared with others.

In DroidMOSS, we adopt a specialized hashing technique called *fuzzy hashing* [21]. Instead of directly processing or comparing the entire (long) instruction sequences, it first condenses each sequence into one much shorter fingerprint. The similarity between two apps is then calculated based on the shorter fingerprints, not the original sequences. Therefore, a natural requirement for fuzzy hashing is that the reduction into shorter fingerprints should minimize the change, if any, to the similarity of two sequences.

To achieve that, we first divide the instruction sequence into smaller pieces. Each piece is considered as an independent unit to contribute to the final fingerprint. Therefore, if the repackaging process changes one piece, its impact on the final fingerprint is effectively localized and contained within this piece. For the rest pieces that are not changed, their contributions to the final fingerprint are still valid and persistent through the repackaging process, thus reflecting the similarity between the original app and the repackaged one. However, the challenge lies on the determination of the boundary of each piece. In DroidMOSS, we use a sliding window that starts from the very beginning of the instruction sequence and moves forward until its rolling hashing value equals a pre-selected *reset point*, which determines the boundary of the current piece. Specifically, if a reset point is reached, a new piece should be started. The concrete process is presented in Algorithm 1 and visually summarized in Figure 2.

Algorithm 1 Generate the app fingerprint

Input: Instruction sequence *iseq* of the app
Output: Fingerprint *fp*
Description: wsize - sliding window size, rp - reset point value, sw - content in sliding window, ph - the piece hash

```
1: set_wsize(wsize)
2: set_resetpoint(rp)
3: init_sliding_window(sw)
4: init_piece_hash(ph)
5: for all byte d from iseq do
6:     update_sliding_window(sw, d)
7:     rh ← rolling_hash(sw)
8:     update_piece_hash(ph, d)
9:     if rh = rp then
10:        fp ← concatenate(fp, ph)
11:        init_piece_hash(ph)
12:    end if
13: end for
14: return fp
```

For further elaboration, suppose a repackaged app has added a new instruction to invoke an external function. For simplicity, we assume the new instruction is inserted in the first piece of the instruction sequence (i.e., *piece 1* in Figure 2). Since our fuzzy hashing scheme uses a sliding window to calculate the rolling hash to determine the piece boundary, there are two possibilities about the placement of the new instruction in the first piece, either falling outside or inside the last sliding window. The former affects only

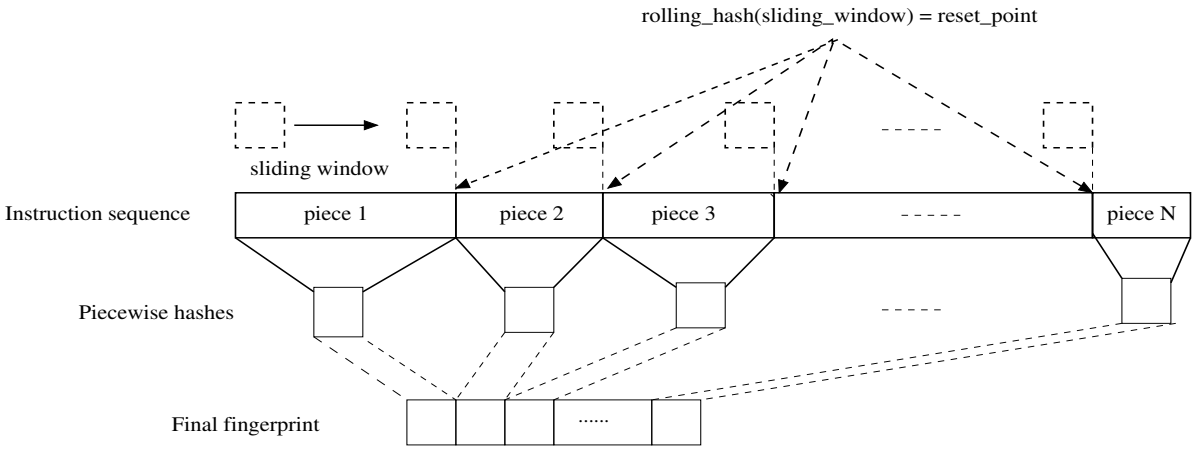

Figure 2: Fuzzy Hashing for Fingerprint Generation

the calculated hash value in the first piece while the rest pieces are intact. The latter changes the rolling hash value of the last sliding window (of the original *piece 1*). As a result, instead of stopping at the original boundary, we keep moving forward the sliding window until it hits the last sliding window in the second piece. In other words, it merges the first two pieces into one. Notice that it does not affect the determination of boundaries of the subsequent pieces. Therefore, for the final fingerprint generation, it only changes the hash values of the first two consecutive pieces. In either way, our scheme effectively localizes the changes.[2]

In our design, to derive the fingerprint, we need to apply traditional hash function twice. The first is to calculate the hash value of each piece (after its boundary is determined) and the calculated hash values of all pieces are combined into the final fingerprint. The second is to calculate a hash value on the content of the sliding window, which is matched against the reset point. In our prototype, we use a prime number as the reset point to enhance the randomization or robustness of our scheme against possible repackaging attacks.

2.4 Similarity Scoring

Our first two steps are applied for each app regardless of its source. In the third step, we divide the apps into two groups, one from the official Android Market and one from alternative marketplaces, and then calculate pair-wise similarity scores between the two. The similarity is based on the derived fingerprints, not the detailed (long) instruction sequence. Note that our fuzzy hashing scheme is deterministic in that if two apps from two groups are identical, the same fingerprints will be generated. In addition, it can also effectively localize the changes possibly made in repackaged apps.

Based on the above analysis, the similarity between the (shorter) fingerprints represents how similar their corresponding apps are. With that, our similarity scoring algorithm is to compute the edit distance between these two fingerprints, which is the number of minimum edit operations, including insertion, deletion and substitution of a single byte, needed to convert one fingerprint into another. The algorithm DroidMOSS adopts is a dynamic programming algorithm as presented in Algorithm 2. In particular, for two fingerprints *fp1* and *fp2* (with lengths of *len1* and *len2*, respectively), we reserve a two-dimensional matrix (each value in the matrix is initialized to 0) to hold the edit distance between all prefixes of the first fingerprint and all prefixes of the second, and then compute the values in the matrix by flood filling the matrix. The distance between the two full strings will be the final value of the edit distance between the two fingerprints. The edit distance of any prefix subsequences of *fp1* and *fp2* can be derived from the minimum of three values: (1) $matrix(i-1,j)+1$, which means to add one insertion operation in *fp1*; (2) $matrix(i,j-1)+1$, which means to add one deletion operation in *fp2*; and (3) $matrix(i-1,j-1)+cost$, which means to add one substitution operation between *fp1* and *fp2*.

Algorithm 2 Calculate the edit distance between two apps

Input: Two fingerprints *fp1* and *fp2*
Output: Edit distance between *fp1* and *fp2*

```
1: len1 ← strlen(fp1)
2: len2 ← strlen(fp2)
3: initialize_two_dimensional_matrix(matrix, len1, len2)
4: for i = 0 → len1 do
5:    for j = 0 → len2 do
6:       if fp1[i] = fp[j] then
7:          cost = 0
8:       else
9:          cost = 1
10:      end if
11:      matrix[i,j] = min(matrix[i−1,j]+1, matrix[i,j−
         1] + 1, matrix[i − 1, j − 1] + cost)
12:   end for
13: end for
14: return matrix(len1, len2)
```

Based on the calculated edit distance, we can derive a similarity score between two fingerprints. The formula we are using is as follows:

$$similarityScore = [1 - \frac{distance}{max(len1, len2)}] * 100 \qquad (1)$$

If the calculated similarity score between two apps exceeds certain threshold and these two apps are signed with two different

[2]Note that more advanced techniques could possibly mitigate our scheme. However, our prototyping experience and evaluation shows that they are not being used. From another perspective, the off-line nature of analyzing existing apps also makes our scheme easy to adapt and evolve.

Table 1: The Numbers of Collected Apps from Official or Alternative Android Marketplaces (†: the number in parenthesis shows the percentage of apps that are also hosted in the official Android Market.)

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

developer keys, our system reports the one not from the official Android Market as repackaged. The threshold selection affects both false positives and false negatives of our system. Specifically, a high threshold likely leads to low false positives but also high false negatives while a low threshold introduces high false positives but with low false negatives. During our experiments, we empirically found the threshold 70 is a good balance between the two metrics (Section 3).

3. PROTOTYPING AND EVALUATION

We have implemented a prototype of DroidMOSS in Linux. In our prototype, the first step – feature extraction – is based on two open-source tools. Specifically, we use *baksmali* [1], a popular Dalvik disassembler to reverse *classes.dex* into an intermediate representation and then map it into Dalvik bytecode. A publicly available tool named *keytool* [25], which is already a part of Android SDK, is used to extract the author information. To glue them together, we created a number of *perl* scripts. For the next two steps, we implement our own C programs for fingerprint generation and similarity scoring. For efficiency reason, our rolling hash function is based on the spamsum algorithm proposed by Andrew [2] (originally for spam detection) and the sliding window size in our prototype is 7. The input to fingerprint generation is essentially those instruction sequences generated from the first step, while the output is used for similarity scoring. As mentioned earlier, the similarity scoring will also take into account the app author information: If two apps has the same authorID, we exclude them from repackaged app detection. If not, our prototype calculates the edit distance and derive the similarity score. The larger the score, the more similar the app pair.

To detect repackaged apps, we chose six popular third-party Android marketplaces worldwide: two in US, two in China, and two in Eastern Europe[3]. For each marketplace, we use a crawler to collect hosted apps. Our study is based on those apps collected in the first week of March, 2011. Meanwhile, we also collect more than sixty thousand apps from the official Android Market in the same time frame. The exact numbers of collected apps from official and alternative marketplaces are shown in Table 1. For each alternative marketplace, we also report the percentage of apps that are hosted in it but also have an identical copy in the official Android Market (i.e., the first category in Section 1). Table 1 shows our results.

[3]One domain is registered in Ukraine, but the resolved IP is actually located in US.

Table 2: Repackaged App Detection from Six Studied Third-party Android Marketplaces (200 samples)

Third-party Marketplace	# Repackaged Apps Apps from DroidMOSS	# Repackaged Apps from Manual Analysis	Percentage
US1	24	22	11%
US2	13	12	6%
EE1	11	10	5%
EE2	15	13	6.5%
CN1	27	25	12.5%
CN2	28	26	13%

3.1 Repackaged Apps in Alternative Marketplaces

To perform a concrete study on the repackaged apps and measure the effectiveness of our approach, we randomly choose 200 samples from each third-party marketplace and detect whether they are repackaged from some official Android Market apps. Specifically, for each chosen app, we measure its similarity score with each of these $68,817$ ones inside the official Android Market. Among the calculated $68,817$ similarity scores, we choose the highest one for manual investigation. Among the total $1,200$ app pairs, we apply the threshold 70 to infer whether an app is repackaged or not.

Our results are shown in Table 2. The first column lists the name of these third-party marketplaces; the second column indicates the number of repackaged apps detected by DroidMOSS out of the 200 samples; the third column shows the manual analysis results; and the fourth one reports the corresponding repackaging rate. For each marketplace shown in the table, DroidMOSS reports that 5% to 13% of apps hosted on it are repackaged. Among the reported ones, we manually verify them and for each marketplace we only find one or two false positives, demonstrating the effectiveness of our approach. By further looking into the false positive cases, we notice that one main contributing factor is that our white-list of ad SDKs or shared libraries is incomplete. Note that by iterating the process to complete the white-list, there is a room for our system to be further improved.

Overall, our experiments show that the repackaging rates range from 5% to 13% among these third-party marketplaces. This is alarming as the repackaged apps seriously affect the entire smartphone app ecosystem. In the following paragraphs, we further look into individual repackaged apps and classify them into different categories for better understanding.

Injecting New In-App Advertisements In the first category, we observe new ad SDKs are added into the original app. Note that ad SDKs typically require adding a certain publisher identifier in the *AndroidManifest.xml* file, inserting layout description into the resource file, importing their own ad class files into the class directory, or even modifying the app bytecode. Recall that DroidMOSS considers ads as noise and thus filters them out for fingerprint generation and similarity scoring. With that, our system can easily spot them – as they share similar (or even identical) code sequences but are signed by different authors.

One example repackages a legitimate app *com.mmc.life49* by including the admob [24] SDK in the class hierarchy and adding a publisher identifier ADMOB_PUBLISHER_ID in *AndroidManifest.xml*. All the original bytecode remains intact. But some ad SDKs (e.g., wooboo [31] and youmi [42]) do require modifying existent class files in the original app to invoke ad-displaying code. Merely looking into the modified manifest file (or resource files)

Table 3: The Comparison of App Manifest Files from the Original App and the Repackaged App

Original *Angry Birds* (in the official Android Market)	Repackaged *Angry Birds* (in a US alternative marketplace)
<manifest android:versionCode="142" android:versionName="1.4.2" android:installLocation="preferExternal" package="com.rovio.angrybirds" xmlns:android="..."> <application android:label=...> ... <meta-data android:name="ADMOB_PUBLISHER_ID" android:value="**a14c9c5b4602e23**" /> <meta-data android:name="ADMOB_INTERSTITIAL _PUBLISHER_ID" android:value="**a14ca2471ee0891**" /> ... </application> </manifest>	<manifest android:versionCode="142" android:versionName="1.4.2" android:installLocation="preferExternal" package="com.rovio.angrybirds" xmlns:android="..."> <application android:label=...> ... <meta-data android:name="ADMOB_PUBLISHER_ID" android:value="**a14ce0cb83321d2**" /> <meta-data android:name="ADMOB_INTERSTITIAL _PUBLISHER_ID" android:value="**a14ce0cbd3cc9a1**" /> ... </application> </manifest>

and newly added class files is not enough to identify this kind of repackaging. In general, the modification is applied on small part of the original code. DroidMOSS can readily localize such kind of modification and detect the repackaging.

Usurping Existing In-App Advertisements In the second category, we also observe repackaged apps where existing ad SDKs still remain, but the corresponding publisher identifiers have been replaced likely with the repackagers' identifiers. Note that each developer can sign up various ad networks to get his own app publisher identifier. The publisher identifier is assigned and used by an ad network to correctly distinguish user clicks or ad traffics and then return the resulting ad revenues. For example, Admob, one of the most popular ad networks in Android, uses two identifiers *ADMOB_PUBLISHER_ID* and *ADMOB_INTERSTITIAL_PUBLISHER_ID*, whose values are assigned by Admob to the app developers during their enrollment. By repackaging apps with their own publisher identifiers, repackagers can collect ad revenues from ad networks, resulting in a financial loss for the original app developers.

In our experiments, we found that one popular repackaged target is the *Angry Birds* app (*com.rovio.angrybirds*) [40]. The vendor of this app (i.e., Rovio) does not charge for the download and installation. Instead it embeds certain ad SDKs (i.e., Admob) into this app to collect ad revenues. One repackaged *Angry Birds* DroidMOSS identified in a US marketplace did not modify any code in the original app. Instead, the only modification is on the Admob-specific identifiers. Table 3 shows the comparison of two corresponding manifest files.

During our evaluation, we initially thought that applying a common Unix utility program, i.e., *diff* [20], on these two manifest files and their corresponding class files can easily identify such repackaging behavior. However, our experience indicates that repackagers explored various unusual ways to substitute publisher identifiers (Table 3). For instance, besides modifying the app manifest file, they may modify the string resource file instead without changing the bytecode at all. Fortunately, with its capability of effectively localizing the changes from repackaging behaviors, DroidMOSS can help detect them.

Trojanizing Legitimate Apps with Malicious Payloads In the third category, we also observe trojanized apps with malicious payloads. Our findings are consistent with recent reports about discovered Android malware [7]. Specifically, the added payloads can be used to conduct a variety of malicious activities, such as sending text messages to premium-rated numbers [35], downloading additional apps from the Internet [30], rooting the phone [32], and even registering the compromised phones as bots [28].

One example found by DroidMOSS is a repackaged app from *com.tencent.qq*, a popular instant messaging app. During our analysis, we found that the trojanized version requests more permissions as embodied in the first four lines of Figure 3. These permissions are requested to facilitate its wrongdoings. But having these added permissions is not sufficient to determine that one app is the repackaged version of another one. (Newer versions of an app may ask for more permissions than previous ones, and vice versa.) As a result, we need to further look into the code to collect additional evidences. In this particular case, the manifest file shows that a new receiver and a new service are added to the original app, and the receiver will be triggered when the system finished booting. Looking into the disassembled code, we know its purpose is to bootstrap a background service named *com.android.MainService*, whose code fetches and executes instructions from a remote server, effectively turning the compromised phones into bots. A further in-depth investigation of the code shows that the trojan app supports a number of commands, such as sending SMS messages to premium numbers, modifying the bookmarks of the built-in browsers, and downloading and installing additional apps onto the phones. All these actions are dispatched through a member function named *execTask* in *com.android.MainService*. The function is invoked to check a command and control (C&C) server using a hard-coded URL (*http://xml.XXX.com:8118/push/androidxml/?[parameters]*) to fetch and execute commands. In Figure 4, we show a code snippet from this function that demonstrates how different payloads are called according to the command it receives.

Another example found by DroidMOSS is a repackaged app based on *com.intsig.camscanner*. A similar background service is needed in this case, but it is triggered in a different way. Instead of using a new receiver to trigger the service, the repackager directly modifies the main activity of the original app to achieve the same purpose. Our analysis also indicates that some obfuscation techniques are being adopted by repackagers to evade analysis and detection. It seems that these malicious payloads are getting more powerful and harder to be analyzed.[4]

[4]Our study also shows that there is a fourth category of apps. In this category, repackagers essentially decompose original apps and re-package them by signing with their own developer keys. One possible reason is that repackagers want to build their own

```
. . . . .
<uses-permission android:name="android.permission.READ_SMS" />
<uses-permission android:name="android.permission.SEND_SMS" />
<uses-permission android:name="com.android.browser.permission.READ_HISTORY_BOOKMARKS" />
<uses-permission android:name="com.android.browser.permission.WRITE_HISTORY_BOOKMARKS" />
. . . . .
<receiver android:name="com.android.AndroidActionReceiver">
   <intent-filter>
      <action android:name="android.intent.action.SIG_STR" />
      <action android:name="android.intent.action.BOOT_COMPLETED" />
   </intent-filter>
</receiver>
. . . . .
<service android:name="com.android.MainService" android:process=":remote" />
```

Figure 3: The Manifest File of A Repackaged App *com.tencent.qq* (the listed receiver and service do not exist in the original app)

```
.method private execTask()V
   .registers 14
   ......
   const-string v10, "push"    ...
   invoke-direct p0, v0, v2, Lcom/android/MainService;->execPush(Ljava/lang/String;[Ljava/lang/String;)V
   ......
   const-string v10, "soft"    ...
   invoke-direct p0, v0, v2, Lcom/android/MainService;->execSoft(Ljava/lang/String;[Ljava/lang/String;)V
   ......
   const-string v10, "xbox"    ...
   invoke-direct p0, v0, v2, Lcom/android/MainService;->execXbox(Ljava/lang/String;[Ljava/lang/String;)V
   ......
   const-string v10, "mark"    ...
   invoke-direct p0, v0, v2, Lcom/android/MainService;->execMark(Ljava/lang/String;[Ljava/lang/String;)V
   ......
.end method
```

Figure 4: The Code Snippet of *execTask* (this function calls different malicious payloads – *execPush*, *execSoft* or *execMask* – based on the command it receives from the hard-coded C&C server – *push*, *soft*, or *mask*)

3.2 False Negative Measurement

While the above experiments focus on the understanding of overall repackaged apps in current third-party app marketplaces, they also show the effectiveness of our system in having a small low false positives. Next, we measure the false negative rates of our system. Because there are no public list of known repackaged apps available for us to use, we prepare such a set by ourselves. Specifically, we first collect those app pairs which have identical or similar package names, but are signed by different developer certificates. After that, we manually identify and confirm 150 repackaged apps as a test set to evaluate our system. As a result, DroidMOSS successfully reports 134 of them as repackaged, but misses 16 of them, implying a false negative rate of 10.7%. By examining those missed cases, we found two main reasons. (1) The first reason is that some repackager may add a large chunk of code into the original app. When the ratio between the added code and the original one is larger than certain threshold, the calculated fingerprints may differ a lot, leading to a small similarity score and causing a false negative. (2) The second reason is due to the fact our white-list is incomplete, which means some shared (ad) libraries are still contained in the sequence as noise. This added noise could result in considerable difference in the final fingerprints, thus leading to a miss in DroidMOSS.

To summarize, our experimental results show an alarming repackaging rate (ranging from 5% to 13%) among our current third-party marketplaces. The repackaged apps are mainly used to replace existing in-app advertisements or embed new ones to "steal" or re-route ad revenues. We also identified a few cases with planted backdoors or malicious payloads among repackaged apps. The results call for the need of a rigorous vetting process for better regulation of third-party smartphone app marketplaces.

4. DISCUSSION

Our evaluation results show that our prototype can effectively detect repackaged apps. In this section, we further examine possible limitations in our system and explore ways for future improvement.

First, our current prototype assumes that the official Android Market contains legitimate (and original) apps. However, this may not be the case in practice. For example, it has been reported that even in the official Android Market, there may exist malicious apps [29] repackaged from other legitimate apps. Also, it is possible that an app (from a third-party marketplace) might be an original one and the corresponding app from the official Android Market is actually repackaged. In either case, DroidMOSS is still helpful in distinguishing repackaged apps and answering the key questions that motivate this work.

reputation by providing benign, high-quality apps so that other users will trust them more when the time arrives for them to publish some bad or malicious apps.

Second, to discern any repackaged app, DroidMOSS depends on the existence of the corresponding original app in our data set. Due to various reasons, our current database is far from complete. For example, our current collection is comprised of those free apps and do not include paid apps in the official Android Market. As a result, we may miss some repackaged apps. Because of that, we have the reason to believe the overall repackaging rate is higher than we report in this paper. From another perspective, this also indicates the need of continuously expanding our current data set with more comprehensive samples.

Finally, our prototype still experiences difficulties due to the use of shared libraries or ad SDKs for repackaged app detection. Specifically, our current approach uses a white-list approach that may not detect possible malicious changes to the ad SDKs or shared libraries. A systematic method to automatically identify shared libraries and detect abnormal changes could greatly improve our prototype.

5. RELATED WORK

Software similarity measurement The first category of related work includes prior efforts in measuring the similarity of software or documents in general. Among the most noted, MOSS [39] is designed to measure software similarity (at the source level) and has been widely used to detect plagiarism in college classes. Our system differs from it in two key aspects: First, DroidMOSS directly works at the Dalvik bytecode level without the source code access, which is required by MOSS. Second, both systems require the use of a sliding window to generate the fingerprint. However, MOSS uses it to generate a k-gram to directly compose the fingerprint, while DroidMOSS calculates the hash value to compare against a reset point to further localize repackaged changes. Our such design is needed and tailored to meet the scalability requirement (Section 2).

Our fuzzy hashing is due to Andrew Tridgell for the spamsum algorithm [2]. The original algorithm is proposed to detect spams and has been later extended by others for different purposes. For example, Payne *et al.* [21] applies it to expand the capability of his forensic tool. Kornblum *et al.* [33] materializes a similar concept for digital forensic analysis by identifying similar text documents. Our system instead applies it for repackaged app detection. To the best of our knowledge, we are the first to apply it for this purpose. Meanwhile, we notice an independent work from the App Genome project [27] that looks into the Android apps from two alternative China-based marketplaces and reports that nearly 11% of their apps also available on the Android Market were found to be repackaged. However, the study does not disclose any methodology as well as technical details behind their findings. Based on their summary-style description, we observe that the results are *only* applied to those apps also available on the official Android Market. Our work instead does not have this limitation by investigating apps we collected from six alternative geographically-scattered marketplaces. Moreover, our study further looks into possible motivations behind repackaged apps and leads to unique insights (e.g., stealing or re-routing ad revenues – Section 3), which have not been reported by others.

Instruction sequence-based security applications As DroidMOSS uses instruction sequences to generate a distinguishing feature to characterize Android apps, we also consider – as the second category of related work – recent security applications that are based on instruction sequences. For example, software birthmarks can be generated based on the k-gram of instructions (e.g., [36]). SigFree [41] applies the notion to network traffic stream by attempting to extract instruction sequence from incoming network requests or packets. By applying code abstraction analysis, SigFree can effectively test whether the packets contain executable instructions or not. Also closely related, recent work apply instruction instructions or even system call sequences in different ways to help detect known or unknown malware (e..g, [9]). DroidMOSS differs from them with a different focus on the app similarity measurement problem and applies instruction sequence in a different way. Specifically, based on the instruction sequence, DroidMOSS further applies fuzzy hashing to condense it to a shorter fingerprint for app similarity measurement.

Smartphone platform and app security The third category include various systems [6,8,13–16,18,19,37,38,43,44] to improve the smartphone platform and app security. For example, TaintDroid [13] applies dynamic taint analysis to monitor apps and detect runtime privacy infringement behaviors in Android apps. PiOS [12] develops a static analysis tool to spot possible information leaks in iOS applications. ScanDroid [18] aims to automatically extract data flow policy from the app manifest, and then check whether data flows in the apps are consistent with the extracted specification. Kirin [15] proposes to enhance the install process in Android to block possibly unsafe apps that request dangerous permission combinations. A follow-up work of Kirin [14] reports a series of systematic findings in Android application security from the study of 1,100 popular free Android applications. Apex [37], MockDroid [6], and TISSA [44] enhance the Android infrastructure so that users can better control the access to specific resources or permission at runtime themselves. Stowaway [16] studies the over-privilege problem of a set of 940 apps and finds that about one-third are not following the least privilege principle. DroidRanger [43] leverages both static and dynamic analysis to detect malicious apps in existing Android marketplaces. Woodpecker [19] statically analyzes pre-loaded apps in smartphone firmware to uncover possible capability leaks. All these tools use either static or dynamic analysis techniques to infer relevant security properties from individual smartphone apps. In contrast, DroidMOSS measures the similarity of two apps (as a pair) by distilling their instruction sequences into corresponding fingerprints.

More recently, researchers also look into the interaction between different Android apps. For example, Saint [38] examines the interfaces one application exported to other and extends the Android framework to enforce inter-application security policy (at install and runtime). ComDroid [8] uncovers possible unintended consequences of exposing certain app components. A relevant work [17] goes further to study both unintentional and intentional exporting of internal components, which can cause apps with permissions to perform privileged task for apps without permissions. This paper also proposes IPC Inspection to address these vulnerabilities. Quire [11] similarly addresses the permission delegation problem by proposing an IPC call chain tracking mechanism to identify the provenance of these IPC requests and enforce certain policy. Stratus [4] explores the security of multi-market app ecosystem and proposes a new app installation model to retain the original single-market security semantics (e.g., kill switches or developer name consistency). Barrera *et al.* [5] uses a self-organization map to analyze 1,100 Android apps and identifies related usage patterns about android app permissions. DroidMOSS differs from them by systematically studying the app repackaging situation in our current third-party marketplaces by measuring the similarity of app pairs.

6. CONCLUSION

In this paper, we examine the problem of repackaged smartphone

applications in current third-party marketplaces, and have accordingly developed a prototype system called DroidMOSS to detect them. Our system adopts a fuzzy hashing technique to effectively localize and detect possible changes from app repackaging. We have applied our system to detect repackaged apps in six third-party Android marketplaces and found that 5% to 13% of apps hosted in them are repackaged. Furthermore, we manually analyze those repackaged apps and our results show that apps are mainly repackaged to replace existing in-app advertisements (or embed new ones) to "steal" or re-route ad revenues, or even more seriously plant backdoors and malicious payloads. The results call for the need of a rigorous vetting process for better regulation of third-party smartphone app marketplaces.

Acknowledgements The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. Special thanks go to Simon Zou, Jack Chiang, and Yang Cao at NQ Mobile for their constructive suggestions and feedbacks. 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] Smali - An Assembler/Disassembler for Android's dex Format. http://code.google.com/p/smali/. Online; accessed at May 17, 2011.

[2] Tridgell Andrew. Spamsum README. http://samba.org/ftp/unpacked/junkcode/spamsum/README. Online; accessed at May 17, 2011.

[3] AndroLib. Android Market Statistics from AndroLib. http://www.androlib.com/appstats.aspx. Online; accessed at June 1, 2011.

[4] 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 May 17, 2011.

[5] 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.

[6] Alastair Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, 2011.

[7] 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 May 17, 2011.

[8] 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.

[9] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining Specifications of Malicious Behavior. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC-FSE '07, pages 5–14. ACM, 2007.

[10] Cydia. Cydia App Store. http://cydia.saurik.com/. Online; accessed at May 17, 2011.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] William Enck, Machigar Ongtang, and Patrick McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[16] 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.

[17] 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.

[18] 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 June 1, 2011.

[19] 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.

[20] James Hunt and McIlroy Douglas. An Algorithm for Differential File Comparison. Technical report, Computing Science Technical Report, Bell Laboratories, 1976.

[21] Dustin Hurlbut. Fuzzy Hashing for Digital Forensic Investigators. Technical report, Access Data Inc., http://accessdata.com/downloads/media/

Fuzzy_Hashing_for_Investigators.pdf. Online; accessed at May 17, 2011.

[22] Amazon.com Inc. Amazon AppStore for Android. http://www.amazon.com/mobile-apps/b?ie=UTF8&node=2350149011. Online; accessed at May 17, 2011.

[23] Apple Inc. Apple App Store for IPhone. http://www.apple.com/iphone/apps-for-iphone/. Online; accessed at May 17, 2011.

[24] Google Inc. Admob for Android Developers. http://developer.admob.com/wiki/Android. Online; accessed at May 17, 2011.

[25] Google Inc. Android Dvelopment Guide: Signing Your Applications. http://developer.android.com/guide/publishing/app-signing.html. Online; accessed at May 17, 2011.

[26] Google Inc. Android Market. https://market.android.com/. Online; accessed at May 17, 2011.

[27] Lookout Inc. App Genome Report: February 2011. https://www.mylookout.com/appgenome/. Online; accessed at May 17, 2011.

[28] Lookout Inc. Security Alert: Geinimi, Sophisticated New Android Trojan Found in Wild. http://blog.mylookout.com/2010/12/geinimi_trojan/. Online; accessed at May 17, 2011.

[29] 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 May 17, 2011.

[30] Symantec Inc. Android Threats Getting Steamy. http://www.symantec.com/connect/blogs/android-threats-getting-steamy. Online; accessed at May 17, 2011.

[31] Wooboo Inc. How to add Wooboo advertisement SDK into Android. http://admin.wooboo.com.cn:9001/cbFiles/down/1272545843644.swf. Online; accessed at May 17, 2011.

[32] Xuxian Jiang. Security Alert: New Sophisticated Android Malware DroidKungFu Found in Alternative Chinese App Markets. http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html. Online; accessed at Sep 17, 2011.

[33] Jesse Kornblum. Identifying Almost Identical Files Using Context Triggered Piecewise Hashing. DFRWS '06, 2006.

[34] Greg Kumparak. TechCrunch: Android Now Seeing 550,000 Activations Per Day. http://techcrunch.com/2011/07/14/android-now-seeing-550000-activations-per-day/. Online; accessed at Sep 15, 2011.

[35] Kaspersky Lab. First SMS Trojan Detected for Smartphones Running Android. http://www.kaspersky.com/about/news/virus/2010/First_SMS_Trojan_detected_for_smartphones_running_Android. Online; Accessed at May 17, 2011.

[36] Ginger Myles and Christian Collberg. K-gram Based Software Birthmarks. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, pages 314–318, New York, NY, USA, 2005. ACM.

[37] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, 2010.

[38] 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.

[39] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 76–85, New York, NY, USA, 2003. ACM.

[40] New York Times. Angry Birds, Flocking to Cellphones Everywhere. http://www.nytimes.com/2010/12/12/technology/12birds.htm. Online; accessed at May 17, 2011.

[41] Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Zhu. SigFree: a Signature-Free Buffer Overflow Attack Blocker. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, Berkeley, CA, USA, 2006. USENIX Association.

[42] Youmi.net. Wiki - Youmi Android Banner Version 2.1. http://wiki.youmi.net/wiki/Youmi_Android_Banner_Version_2.1. Online; accessed at May 17, 2011.

[43] 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.

[44] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vince Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proceeding of the 4th International Conference on Trust and Trustworthy Computing*, TRUST '11, 2011.