# MIGDroid: Detecting APP-Repackaging Android Malware via Method Invocation Graph

Wenjun Hu†, Jing Tao†*, Xiaobo Ma†‡, Wenyu Zhou§, Shuang Zhao¶, Ting Han†
†MOE KLINNS Lab, Xi'an Jiaotong University, Xi'an, 710049, China
‡Xian Jiaotong University Suzhou Academy, Suzhou, 215123, China
§ShaanXi Banking Regulatory Commission Office, Xi'an, 710049, China
¶Institute of Information Engineering, CAS, Beijing, 10000, China
Email: {wjhu, jtao, xbma}@sei.xjtu.edu.cn, wcf1987@gmail.com, zhaoshuang@iie.ac.cn, than@sei.xjtu.edu.cn

***Abstract*—With the increasing popularity of Android platform, Android malware, especially APP-Repackaging malware wherein the malicious code is injected into legitimate Android applications, is spreading rapidly. This paper proposes a new system named MIGDroid, which leverages *method invocation graph* based static analysis to detect APP-Repackaging Android malware. The method invocation graph reflects the "interaction" connections between different methods. Such graph can be naturally exploited to detect APP-Repackaging malware because the connections between injected malicious code and legitimate applications are expected to be weak. Specifically, MIGDroid first constructs method invocation graph on the smali code level, and then divides the method invocation graph into weakly connected sub-graphs. To determine which sub-graph corresponds to the injected malicious code, the threat score is calculated for each sub-graph based on the invoked sensitive APIs, and the sub-graphs with higher scores will be more likely to be malicious. Experiment results based on 1,260 Android malware samples in the real world demonstrate the specialty of our system in detecting APP-Repackaging Android malware, thereby well complementing existing static analysis systems (e.g., Androguard) that do not focus on APP-Repackaging Android malware.**

***Keywords*-Android; malware; static analysis, method invocation graph**

## I. Introduction

With the rapid development of smartphones, Android OS-based system becomes the most popular platform for mobile devices. According to the data from IDC [1], In the third quarter of 2013, Android accounted 81.01% of the smart-phones OS market with 139.9 million shipments. The popularity is also partially propelled with the large number of feature-rich Android applications in various markets. Until July 2013, the official Android market Google Play held more than 1,000,000 applications and hit 50 billion downloads [2]. In addition to Google Play, a number of third-party alternative markets (e.g., Amazon Appstore, SlideMe), further boost the popularity of Android.

Despite its popularity, the Android platform is a prime target for attackers due to the openness of Android markets. As stated in [3], Android malware has dominated the mobile threat landscape with an exponential growth. A recent research shows that malicious applications exist in both the official and unofficial marketplaces with a rate of 0.02% and 0.2% respectively [4]. Currently, except Google's Bouncer [5], which provides automated scanning of Google Play for potentially malicious applications, other alternative markets take no effective security vetting processes. Even for Google's Bouncer, the attackers still have a chance to bypass the security vetting processes in practice [6] .

The absence of effective and strict security vetting processes renders it feasible for attackers to upload malicious apps to various markets. To make the malicious apps less suspicious and meanwhile easy to spread, the attackers usually trick users into downloading malware disguised as popular apps that have a large number of interested users. To be specific, the attackers first repackage popular apps by inject the malicious code into them, and then upload these APP-Repackaging malware to various markets. In fact, it's technically easy to repackage apps with malicious code. As reported in [7], 100% of top paid Android applications have been hacked and attackers successfully inject malicious code. W. Zhou's study [8] shows a worrisome fact that 5% to 13% of applications hosted on the alternative markets are APP-Repackaging malware. For example, *Geinimi*, a bot-like malware, which was publicly reported in January 2011, just repackages itself into legitimate applications like Monkey Jump2 [9], Sex Positions Social and spreads by uploading to alternative markets and app-sharing forums.

This paper proposes a new system named MIGDroid, which leverages *method invocation graph* based static analysis to detect APP-Repackaging Android malware. The method invocation graph reflects the "interaction" connections between different methods. Such graph can be naturally exploited to detect APP-Repackaging malware because the connections between injected malicious code and legitimate applications are expected to be weak. In order to guarantee the legitimate application's integrity, most APP-Repackaging Android malware is realized by injecting independent components (e.g., a Broadcast Receiver) rather than new methods into the legitimate application. These injected components can listen for a specific event to initiate and run the malicious code upon the event occurs. For example, attackers can inject a

* Corresponding Author. The research presented in this paper is supported in part by the National Natural Science Foundation (61103240, 61103241, 61221063, 91118005), the Application Foundation Research Program of SuZhou (SYG201311) and Huawei Cooperation Research Program(YB2013110125).

Broadcast Receiver which listens for the BOOT_COMPLETE broadcast into a legitimate application; when the compromised device rebooted, the Broadcast Receiver is triggered and attackers definitely can perform malicious behaviors in this Broadcast Receiver component. Due to such kind of APP-Repackaging implementation, the part of injected malicious code has weak connections with the legitimate application from the perspective of method invocation graph.

To exploit the method invocation graph for detecting APP-Repackaging Android malware, we first disassemble Android applications and construct method invocation graph on the smali code level. Then, the method invocation graph is further divided into weakly connected sub-graphs. Since many sensitive APIs such as *sendTextMessage* are invoked in Android malware to perform malicious behaviors, we can determine which sub-graph contains malicious code by calculating its threat score based on the invoked sensitive APIs. The sub-graph whose threat score exceeds the given pre-defined threshold will be labeled as malicious.

We summarize our contributions as follows:

- We propose a novel approach to effectively detect APP-Repackaging Android malware. To the best of our knowledge, MIGDroid is the first system that utilizes method invocation graph to detect malware on Android. The proposed system falls into the category of static analysis techniques, and thus is lightweight compared with dynamic analysis techniques.
- Experiment results based on 1,260 Android malware samples in the real world demonstrate the specialty of our system in detecting APP-Repackaging Android malware, thereby well complementing existing static analysis systems (e.g., Androguard) that do not focus on APP-Repackaging Android malware.

The rest of the paper is organized as follows. We first provide a brief background about Android and related stuffs in Section II. We then give the motivation of our work in Section III, followed by system design in Section IV. Section V presents the evaluation of our system and present a real-world case study in Section VI. After that, we provide some related work in Section VII. Finally, we conclude our work in Section VIII.

## II. Background

In this section, we give an overview of the basic structure of an Android application [10].

An Android application exists as an APK file which is actually an *zip* archive. It contains a *dex* file and a set of resources such as graphs, user interface layouts, media files, etc. There is also an important configuration file called *AndroidManifest.xml* which consists of some meta-information about the application such as package name, version name, version code, permissions required and other attributes. An APK file must be digitally signed with the developer's own signing key or Android system will not accept the installation requirement. According to the Android design, the developer's signing certificate can be self-signed and is included in the folder *META-INF*. Fig. 1 presents an overview of the APK structure.

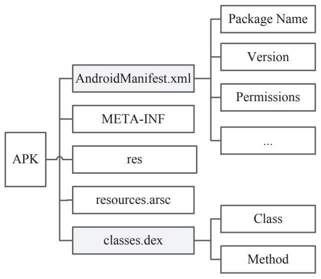


Fig. 1. The APK Structure

The functionality of an application is achieved by components which are logical application building blocks. There are four types of components defined in Android system:

- ***Activity*** component provides interaction with end users. They are often presented to the user as full-screen windows, floating windows or other forms. Each application may need one main activity which starts first when launches the application. This kind of activity should be explicitly declared in the AndroidManifest.xml under intent node with attribute value android.intent.action.MAIN. Activities are started with Intents, and can return data back to their invoking components upon completion.
- ***Service*** component runs in the background and doesn't interact with end users compared with activity. A service can perform a longer-running operation or supply functionality for other applications to use. Android malware usually performs malicious behaviors in the background as services in order to hide themselves from users' sight. Take *Geinimi* [11] for example, it will launch its own service prior to the launch of the main activity. Then the service allows the Trojan to start while the host application appears to function normally.
- ***Broadcast Receiver*** component can receive broadcast messages from other components or applications. There are two major classes of broadcasts that can be received, which are normal broadcasts and ordered broadcasts. Broadcast receivers can be either dynamically registered or statically declared in the AndroidManifest.xml.
- ***Content Provider*** component encapsulates data and provides itself to applications through the ContentResolver interface. It's database addressable by their application-defined URIs. Content provider allows data sharing between multiple applications.

## III. Motivation

In this section, we present an example to better demonstrate the method invocation graph based approach to detect APP-Repackaging Android malware.

### A. A Motivating Example: The Geinimi Trojan

Geinimi is a notorious Trojan that targets Android system. This Trojan can compromise a significant amount of personal privacy data on a user's Android device and then send it to remote servers. It can receive commands from a remote server that allow the owner of that server to control the device.

Geinimi effectively utilizes repackage technique to inject the malicious code into legitimate applications. Take a sample (MD5:08E4A73F0F352C3ACCC03EA9D4E9467F) as an example, from the AndroidManifest.xml file, we can figure out that the package name of the legitimate application is named com.splGUI. The author of Geinimi Trojan adds a launcher activity named com.geinimi.custom.Ad3072_30720001 which can perform privacy collection when the application starts. Also, a receiver named com.geinimi.AdServiceReceiver is added to listen to the BOOT_COMPLETED broadcast. Whenever the device reboots, this receiver is triggered to start working. Since Android adopts the permission system to manage resource access, some sensitive permissions such as ACCESS_FINE_LOCATION, SEND_SMS are added in the AndroidManifest.xml to guarantee that Geinimi can work properly.

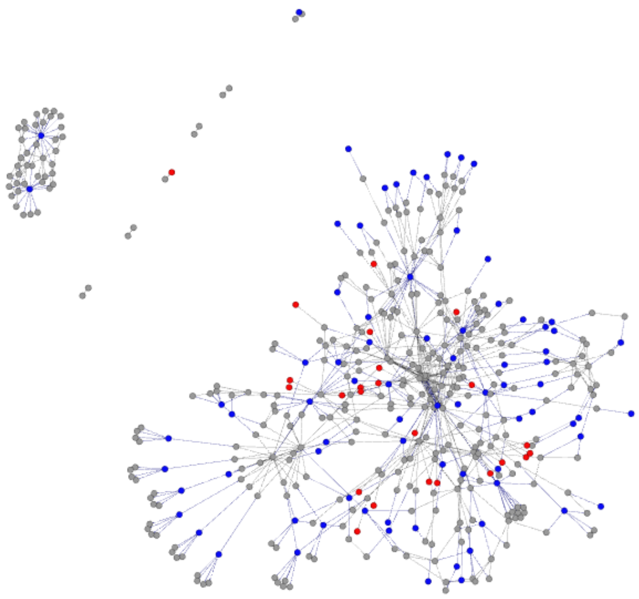


Fig. 2. Method Invocation Graph of Geinimi

Fig. 2 shows the method invocation graph of the Geinimi sample. Each node represents a method and the directed-edge between nodes shows the invocation relationship. The top-left bunch of nodes are from the package com.splGUI, while nodes at the bottom-right are methods implemented by Geinimi. Sensitive APIs (refer to Section V) invoked which can access constrained resources are labeled in red. From the method invocation graph we can find that in the legitimate part of the sample, there are no sensitive APIs invoked. While Geinimi invokes a bundle of sensitive APIs to perform malicious behaviors. It is obvious that this sample is mainly divided into two parts, namely, legitimate application and Geinimi Trojan. Meanwhile, there are no connections between the Geinimi Trojan and the legitimate application.

## B. *Overview of Our Detection Approach*

Our approach mainly includes two stages: a method invocation graph construction stage and a malware identification stage. The method invocation graph construction stage is responsible for generating corresponding graph structure from smali code. After that, in the malware identification stage, we can determine if there exists malicious code based on the generated method invocation graph.

In the method invocation graph construction stage, we first convert the *classes.dex* to *.smali* files using baksmali tool [12]. We traverse all the smali files and extracts the method call sequences. Since the *overload* and *override* techniques are frequently used in Object Oriented language like Java, we identify each method by its name, return value type and parameter type. Then the extracted method invocation sequences are aggregated to generate a method invocation graph. In the stage of method invocation graph construction, there is a need to remove noises such as code fragments and advertisement modules. We then divide the method invocation graph into sub-graphs.

In the malware identification stage, the threat score is calculated for each sub-graph. We extract sensitive APIs and each API is given a specific threat score based on the statistics on a collection set of known Android malware in the wild collected from [13]. The procedure of calculating the threat score of each sub-graph is: (1) Add each sensitive API's threat score in the sub-graph to get the whole threat score; (2) Divide the whole threat score by the total number of all invoked methods in the sub-graph to get the final threat score. If the final threat score of a sub-graph exceeds a given threshold, then it's labeled as malicious.

## IV. SYSTEM DESIGN

MIGDroid's architecture and workflow is depicted in Fig. 3. The system consists of two key modules: the Method Invocation Graph Construction module and the Malware Detection module. The Method Invocation Graph Construction module constructs the method invocation graph from a given Android application. The Malware Detection module can determine whether the given Android application contains malicious code based on the method invocation graph.

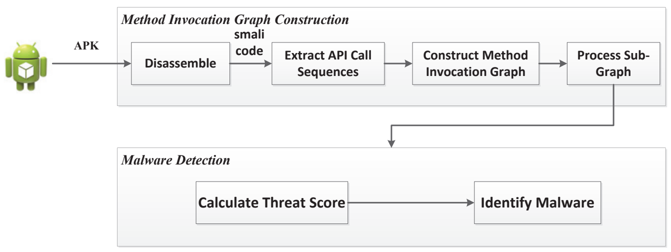


Fig. 3. The Architecture and Workflow of *MIGDroid*

## A. *Method Invocation Graph Construction*

Four steps (i.e., disassembling the APK, extracting API call sequences, constructing method invocation graph and processing sub-graphs) will be elaborated on the following to illustrate how the method invocation graph is constructed.

*1) Disassembling The APK:* Before constructing the method invocation graph, we must inspect all the methods used in a given Android application. In order to extract the information of methods invoked, we disassemble the application. Considering the accuracy of existing "Dalvik bytecode-to-Java bytecode" translators, we prefer operating and analyzing directly on the Dalvik bytecode. The smali code is an intermediate representation of the Dalvik byte-code. So we choose to do static analysis on the smali code level. Meanwhile, it is easy to get the smali code of an APK with off-the-shelf tools.

*2) Extracting API Call Sequences:* With disassembled smali files, we extract all the API call sequences. To this end, we exhaustively search all possible API call sequences by starting from every method. The return value type and parameter type of methods are also extracted. Note that indirect calls and event-driven calls are ubiquitous in an Android applications and thus should not be ignored. However, these calls can not be identified by exhaustively searching. To solve this, we use the methods proposed by Woodpecker [14], which uses a conservative method for indirect function calls and adds links by semantics for event-driven calls. Finally, we can obtain all the API call sequences.

*3) Constructing Method Invocation Graph:* Considering that code in an Android application is organized by *class*, we aggregate all the extracted API call sequences based on class to generate the method invocation graph. In Android system, calling sensitive APIs requires corresponding permissions. So we can identify the position of sensitive invocation in smali code according to high sensitive permissions. Finally, we can outline the method invocation graph.

*4) Processing Sub-Graphs:* We adopt the depth-first traversal algorithm to generate sub-graphs. After the process of graph traversal, we can cluster all the methods into different sub-graphs. The methods within each sub-graph are closely connected, while the methods between two distinct sub-graphs are weakly connected. To further identify the sub-graph that contains malicious code, there exist two challenges. One is the code fragment existing in the Android application. The other is the existence of advertisement modules. To be specific, from the view of the method invocation graph, there exist fragments which have no practical functionality. In Fig. 2, nodes scattering between com.splGUI and Geinimi belong to this condition. Since the number of such fragments is really considerable and most of them are useless, we argue that labeling these fragments is one of the most effective way to exclude the impact on malicious code identification. As far as we know, there is no labeling mechanism for Android code fragments currently. We present a simple yet effective method to identify these fragments.

1) Label a sub-graph with only one class which has no sensitive method invocation and no more than 4 methods as a fragment sub-graph.
2) Label a sub-graph with empty method body as a fragment sub-graph.

Most Android developers tend to insert advertisements in the application with the help of advertisement SDKs. Developers can import these advertisement SDKs without modifying the application's own code due to the flexible implementation of advertisement modules. The implementation mechanism of advertisement module makes the methods of advertisement code an independent sub-graph. On the other hand, most advertisement SDKs will invoke many sensitive APIs so as to collect users' privacy information for accurate advertisement recommendation. Fig. 4 shows the method invocation graph that contains advertisement module. The top left sub-graph represents advertisement module called Wooboo [15]. We can observe that the advertisement module contains sensitive APIs. Therefore, the advertisement modules are similar to APP-Repackaging malware, which might result in false positives.

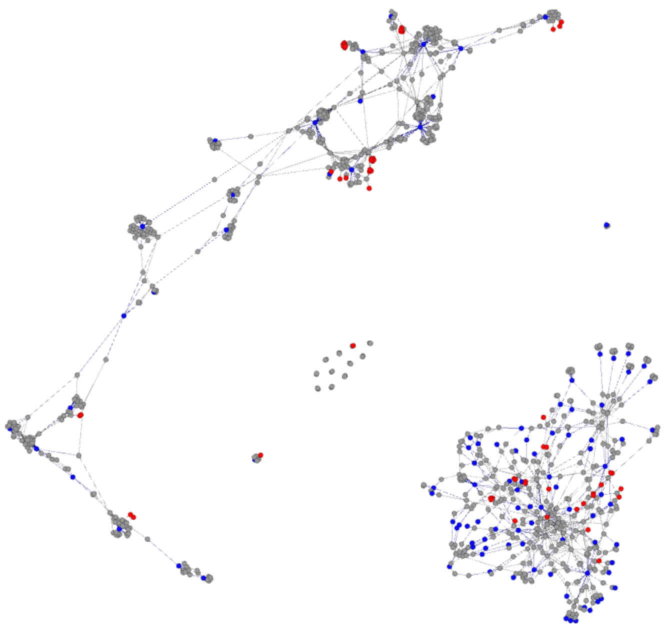


Fig. 4. Method Invocation Graph of Application Containing Advertisement Module

To distinguish the advertisement modules from APP-Repackaging malware, we need to label these advertisement modules like the way we process fragment code. Each advertisement module has its own package name which can be used as a feature to detect the existence of the advertisement. We build a whitelist based on the package name of advertisement modules. Table. I shows several examples of the advertisement modules' package name. We can figure out the advertisement module if the package name exists in the whitelist to avoid further analysis.

TABLE I
EXAMPLES OF THE ADVERTISEMENT MODULE WHITELIST

| **Advertisement Vendor** | **Package Name** |
|---|---|
| Admob [16] | com.google.ads |
| Millennial Media [17] | com.millennialmedia.android |
| InMobi [18] | com.inmobi.andoridsdk |
| Chartboost [19] | com.chartboost.sdk |
| AppBrain AppLift [20] | com.appbrain.AppBrain |
| Mopub [21] | com.mopub.mobileads |

### B. Malware Detection

After generating method invocation graph and processing sub-graphs, we need to determine if the Android application contains malicious code. The method we use to detect malicious code includes two steps: calculating the threat score of each sub-graph and identifying the sub-graph that contains malicious code.

*1) Calculating The Threat Score:* Android malware performs malicious behaviors by calling sensitive APIs which can access restricted resources such as reading files, accessing network, collecting device information. We make a statistics based on 1,000 Android malware samples collected from [13] and generate a sensitive API list. Fig. 5 shows the top 10 most used APIs in these Android malware samples plus the number of each API's invocation. Each sensitive API is assigned a threat score based on its potential damage and the invocation count by malware samples.

For each sub-graph, we calculate the whole threat score based on the invoked sensitive APIs (each invocation of the same sensitive API will also be added to the whole threat score).

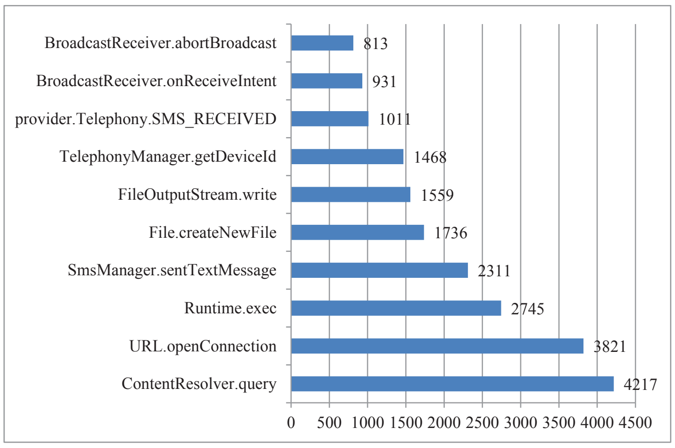


Fig. 5. Top 10 Most Used APIs Invoked

*2) Identifying The Malware:* After calculating threat score for sub-graphs of 100 Android malware samples, we find that sub-graphs containing malicious code have higher threat score and smaller number of method invocation, while legitimate sub-graphs have lower threat score and larger number of method invocation. So we can determine whether the sub-graph contains malicious code based on the *sub-graph score*, which equals the sum of the *sensitive* API calls' scores divided by the *total* number of methods in the sub-graph.

If the higher the *sub-graph score* is, the more likely the sub-graph contains malicious code. When the threshold of *sub-graph score* is set higher than 0.4, there exists false negatives when detecting sub-graphs containing malicious code. That is to say, sub-graphs containing malicious code are recognized as legitimate sub-graphs. While if the threshold is set lower than 0.4, there exists false positives when detecting legitimate sub-graphs. In this paper, we set the threshold to 0.4. When the *sub-graph score* is higher than 0.4, then this sub-graph will be identified as malicious.

## V. Evaluation

In this section, we evaluate the detection efficacy of MIGDroid as compared with Androguard based on 1,260 Android malware samples. Table II shows the detection results for APP-Repackaging Android malware samples while Table III presents the detection results for Non-APP-Repackaging ones. Since Androguard utilizes a signature based mechanism to detect Android malware, we label those malware samples which don not exist in Androguard's signature database with an asterisk.

The experiment result shows that MIGDroid has a much higher detection rate than Androguard on APP-Repackaging Android malware. From Table II and III, we can find that Androguard can not detect any of the samples without corresponding signature. While our system can detect Android malware without any signature except the extracted sensitive APIs. Since most Android malware must invoke sensitive APIs to perform malicious behaviors and these APIs are steady to a certain extent, the method invocation graph based approach is much more stable and can detect both known and unknown Android malware. Androguard detects none of the samples whose signature is not in the database, while even for some Android malware samples like DroidDreamLight, Plankton and DroidKungFu2 which exist in Androguard signature database, Androguard does not show a better detection rate than MIGDroid.

TABLE II
DETECTION EFFECTIVENESS BETWEEN *MIGDroid* AND ANDROGUARD FOR *APP-REPACKAGING SAMPLES*

| **Malware Family** | ***MIGDroid*** | **Androgurad** | **Samples** |
|---|---|---|---|
| ADRD | 100.00% | 59.09% | 22 |
| AnserverBot | 100.00% | 0.00%* | 187 |
| BeanBot | 100.00% | 0.00%* | 8 |
| Bgserv | 100.00% | 0.00%* | 9 |
| DroidDream | 100.00% | 93.75% | 16 |
| DroidDreamLight | 100.00% | 28.26% | 46 |
| DroidKungFu1 | 97.06% | 0.00% | 34 |
| DroidKungFu3 | 99.35% | 0.00% | 309 |
| DroidKungFu4 | 72.92% | 0.00%* | 96 |
| Geinimi | 100.00% | 97.10% | 69 |
| GoldDream | 100.00% | 0.00% | 47 |
| jSMSHider | 100.00% | 0.00% | 16 |
| Pjapps | 72.41% | 41.38% | 58 |
| Plankton | 100.00% | 18.18% | 11 |
| YZHC | 100.00% | 95.45% | 22 |
| Zsone | 100.00% | 100.00% | 12 |
| Others | 89.29% | 21.43% | 56 |
| Average | 95.94% | 32.63% | 1018 |

We manually analyze some of the samples on which our method does not perform well and detail the reasons.

1) *SndApps*: SndApps actually has no serious security threat to Android devices since it only injects a number of advertisement modules into legitimate applications, while our method removes all the advertisement modules before performing malware detection as stated in IV. So it is obvious that MIGDroid detects none of such samples like SndApps.
2) *zHash*: zHash has different variants by adding garbage code and it realizes functionality which is useful for end users. So, the parts which perform malicious behaviors

TABLE III
DETECTION EFFECTIVENESS BETWEEN *MIGDroid* AND ANDROGUARD FOR *NON-APP-REPACKAGING SAMPLES*

| Malware Family | *MIGDroid* | Androgurad | Samples |
|---|---|---|---|
| Asroot | 50.00% | 0.00%* | 8 |
| BaseBridge | 44.26% | 0.00% | 122 |
| DroidKungFu2 | 60.00% | 0.00% | 30 |
| KMin | 21.15% | 78.86% | 52 |
| RogueSPPush | 55.56% | 100.00% | 9 |
| SndApps | 0.00% | 100.00% | 10 |
| zHash | 0.00% | 0.00%* | 11 |
| Average | 33.00% | 39.84% | 242 |

have strong connections with the legitimate ones. This kind of malware implementation requires high cost and definitely increases the difficulty to identify correctly.

3) *Asroot*: Asroot has the capacity to acquire root privilege of the device. Some samples of this malware family does not inject the malicious code into legitimate applications. Rather, they perform malicious behaviors by implementing a complete Android application by themselves. Like zHash, MIGDroid cannot detect this kind of Android malware correctly.

To demonstrate the false positive rate, we randomly choose 1,000 Android applications which are collected during October 2013 and contain no malicious code from Google Play. From Table IV, we can conclude that MIGDroid labels 89 samples as containing malicious code. So, the false positive rate is *8.9%*.

While after investigation, we found that most of the packages labeled as malicious are advertisement or third party modules. For example, *com.smaato.SOMA* is from an advertisement module called SMAATO [22]. the package *com.phonegap* comes from the third party module named PhoneGap [23] which provides APIs to easily create apps using the web technologies such as HTML, CSS and JavaScript. Hence, we can definitely decrease the false positive rate by extend the whitelist.

TABLE IV
DETECTION RESULTS ON 1,000 ANDROID APPLICATIONS FROM GOOGLE PLAY

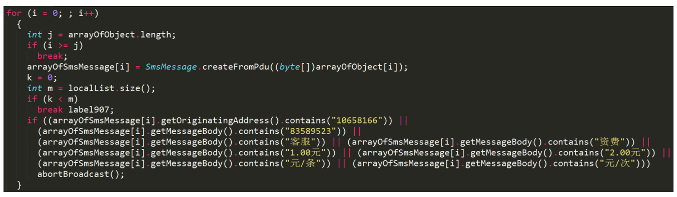
| Package Contains "Malicious" Code | *Samples* |
|---|---|
| com.phonegap | 20 |
| org.acra | 16 |
| com.airpush.android | 14 |
| com.facebook.android | 11 |
| com.smaato.SOMA | 10 |
| com.mobfox.sdk | 5 |
| com.mobclix.android.sdk | 4 |
| com.Leadbolt | 3 |
| com.wiz.bell_83199293g | 1 |
| com.jankroearing | 1 |
| com.ironsource.mobilcore | 1 |
| lu.luxair.android | 1 |
| com.jiubang.goscreenlock.theme.loveunlockr | 1 |
| com.fest.wall.thanksgiving | 1 |
| Total | 89 |

## VI. Real-World Case Study

As stated above, our approach does not depend on Android malware signature. So, MIGDroid has the capacity of detecting malware without any training sample or signatures. The example below shows that MIGDroid successfully detects a Trojan which has the "phone-home" behavior and listens to commands from a remote command and control (C&C) server. The MD5 of the sample is 8FC95259D0C648B7C087CC9CAED87DAB. MIGDroid identifies that this sample contains malicious code which is embedded in the classes such as *com.nl.MyService*, *com.nl.MyReceiver*. The detailed report can be accessed through our system's URL [24].

This sample comes with the package name com.android.XWLauncher. A bundle of high sensitive permissions declared in the AndroidMainfest.xml, including SEND_SMS, CALL_PHONE, INTERNET and INSTALL_PACKAGES etc. These high sensitive permissions indicates that this sample has the ability to send premium text messages and install packages without users' knowledge. A receiver named com.nl.MyReceiver is defined to intercept received SMS and listens to the boot complete event. In order to intercept received SMS in the first place, the author set the highest priority value 2147483647 to the intent filter. After the device rebooted, com.nl.MyReceiver will start a service which is also defined in the AndroidManifest.xml with the name com.nl.MyService. This service will try to download Android applications through http://www.nnetonline.com/mobile/softad, and then install the downloaded applications with root privilege utilizing the command *pm install*. Users are not aware of the applications' installation under such condition. In addition to downloading applications from the specified web server, this sample will also send the installed-application information of the compromised device back to the remote server.

As mentioned earlier, this sample has the ability to send SMS messages. After our analysis, we found that it will fetch the destination phone number and SMS message from http://www.nnetonline.com/mobile/sms2 before sending out a SMS message. When a SMS message is received, the sample will intercept the message which contains *83589523* and other key words. Any SMS message sent from the number *10658166*, which is actually a premium service number, will also be blocked. Fig. 6 represents the code snippet of blocking received SMS message according to the source phone number and SMS message content. It is obvious that this sample can send out fraudulent premium SMS messages from compromised devices for financial gains.



```
for (i = 0; ; i++)
  {
    int j = arrayOfObject.length;
    if (i >= j)
      break;
    arrayOfSmsMessage[i] = SmsMessage.createFromPdu((byte[])arrayOfObject[i]);
    k = 0;
    int m = localList.size();
    if (k < m)
      break label907;
    if ((arrayOfSmsMessage[i].getOriginatingAddress().contains("10658166")) ||
      (arrayOfSmsMessage[i].getMessageBody().contains("83589523")) ||
      (arrayOfSmsMessage[i].getMessageBody().contains("客服")) || (arrayOfSmsMessage[i].getMessageBody().contains("资费")) ||
      (arrayOfSmsMessage[i].getMessageBody().contains("1.00元")) || (arrayOfSmsMessage[i].getMessageBody().contains("2.00元")) ||
      (arrayOfSmsMessage[i].getMessageBody().contains("元/条")) || (arrayOfSmsMessage[i].getMessageBody().contains("元/次")))
      abortBroadcast();
  }
```

Fig. 6. Code snippet for blocking received SMS message

At the time of writing, the web server mentioned above is still accessible. We find there exists a a phishing URL that provides a registration entry http://www.nnetonline.com/U/Reg.aspx on the web server.

This registration process requires a lot of private information about users including telephone number, bank account, bank name, etc. We searched this sample on VirusTotal [25] and find it is labeled as *Fjco* Trojan by most of the anti-virus products. This result confirmed that MIGDroid has the power to detect Android malware without any training sample or signatures.

## VII. Related Work

### A. Static Analysis

Several static analysis on detecting Android malware have been proposed. For example, Yajin Zhou et al. presented a systematic study for the detection of malicious applications on popular Android markets [4]. They propose a permission-based behavioral footprinting scheme to detect Android malware. With their heuristics-based detection engine, two zero-day malware is successfully detected. D.Wu et al. proposed a static feature-based mechanism for detecting the Android malware [26]. They first extract the information including required permissions, deployment of components, Intent messages, etc. Then, K-means algorithm is applied to enhance the malware modeling capability. To determine the number of clusters, they utilize Singular Value Decomposition method. After that, kNN algorithm is used to classify the application as benign or malicious. RiskRanker [27] proposed by Grace et al. aims to detect zero-day Android malware using a set of vulnerability specific-signatures combined with control flow and intra-method data flow analysis. They successfully uncovered 718 malware samples out of 118,318 total Android applications and 322 of them are zero-day. MIGDroid belongs to the static analysis category to detect APP-Repackaging Android malware by leveraging the method invocation graph.

### B. Dynamic Analysis

Unlike static analysis, dynamic analysis involves running the Android application in a controlled and isolated environment such as Android emulator in order to reveal its execution behaviors. CrowDroid [28] proposed by I.Burguera et al. performs clustering algorithms on activity and behavior data of running Android applications to classify them as benign or malicious. Rastogi et al. presented the AppsPlayground framework [29] which utilizes dynamic analysis to detect Android malware. AppsPlayground re-purposes the Android emulator and is built as a virtual machine environment. Several detection techniques such as taint tracing, sensitive API monitoring are adopted to reveal samples' running behaviors.

## VIII. Conclusions and Future Work

We presented a novel approach to detect APP-Repackaging Android malware. Our method first extracted API calling sequences from a given Android application and then constructed the method invocation graph. After generating sub-graphs based on the method invocation graph, we calculated each sub-graph's threat score according to the sensitive APIs invoked in the sub-graph. Finally, we labeled the sub-graph whose score exceeds the given pre-defined threshold as malicious. Our system evaluation showed that the approach we proposed has a good performance on APP-Repackaging Android malware. The real-world case study revealed the power of detecting Android malware in practical environment without any training sample or signature. The advantage of our approach is that we only utilize static analysis that is lightweight. In the future, we plan to consider analyzing native code, with which the attackers can invoke the sensitive APIs in C and C++ language. Java reflection is another challenge in constructing method invocation graph. We will resort to the symbolic execution methodology to address these challenges.

## References

[1] "Android pushes past 80% market share while windows phone shipments leap 156.0% year over year in the third quarter, according to idc," http://126.am/WmoUv0.

[2] "Google play," http://en.wikipedia.org/wiki/Google_Play.

[3] F-Secure, "Mobile threat report q3 2013," Tech. Rep., 2013.

[4] Y. Zhou, Z. Wang, W. Zhou, and X. 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*, 2012.

[5] "Bouncer," http://126.am/8RtSe4.

[6] "To hide android malware apps from google's 'bouncer', hackers learn its name, friends, and habits," http://126.am/rAVU54.

[7] Arxan, "State of security in the app economy: Mobile apps under attack," Tech. Rep., 2013.

[8] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, ser. CODASPY '12. New York, NY, USA: ACM, 2012, pp. 317–326.

[9] "Mokey jump2," https://play.google.com/store/apps/details?id=com.dseffects.MonkeyJump2.

[10] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *Security & Privacy, IEEE*, vol. 7, no. 1, pp. 50–57, 2009.

[11] "Security alert: Geinimi, sophisticated new android trojan found in wild," https://blog.lookout.com/blog/2010/12/29/geinimi_trojan/.

[12] "smali," http://code.google.com/p/smali/, 2013.

[13] "Andromalshare," http://202.117.54.231:8080/, 2013.

[14] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[15] "Wooboo," http://www.wooboo.com.cn/.

[16] "Admob," http://www.google.com/ads/admob/.

[17] "Millennialmedia," http://www.millennialmedia.com/.

[18] "Inmobi," http://www.inmobi.com/.

[19] "Chartboost," http://www.chartboost.com/support/sdk.

[20] "Appbrain," http://www.appbrain.com/info/sdk.

[21] "Mopub," http://www.mopub.com/.

[22] "Smaato," http://www.smaato.com/.

[23] "Phonegap," http://phonegap.com/.

[24] "Analysis report of com.android.xwlauncher," http://126.am/dvlrD4.

[25] "Virustotal report," http://126.am/sJc414.

[26] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "Droidmat: Android malware detection through manifest and api calls tracing," in *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*. IEEE, 2012, pp. 62–69.

[27] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012, pp. 281–294.

[28] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 15–26.

[29] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: Automatic security analysis of smartphone applications," in *Proceedings of the third ACM conference on Data and application security and privacy*. ACM, 2013, pp. 209–220.