Detecting Malicious Behavior of Android Applications

Dewashish Upadhyay
Rajiv Gandhi College of Engineering & research, Nagpur
R.T.M.N University, Nagpur, India

Sharvari Dharbey
Rajiv Gandhi College of Engineering & research, Nagpur
R.T.M.N University, Nagpur, India

Mitalee Munghate
Rajiv Gandhi College of Engineering & research, Nagpur
R.T.M.N University, Nagpur, India

Anjali Bondre
Rajiv Gandhi College of Engineering & research, Nagpur
R.T.M.N University, Nagpur, India

Kalyani Barapatre
Rajiv Gandhi College of Engineering & research, Nagpur R.T.M.N University, Nagpur, India

Abstract

Android, is an open source platform for mobile devices and covers almost 82% of all smart phone devices in the world. This makes its security issues more prominent, especially in user privacy leakage. We developed an Application for Android-based mobile devices which will check the permissions of all the applications present on the device & simultaneously trace the API calls to detect the Malicious behavior of Android application. This tool has the capability of generating log files of API calls made by an application running on a mobile device. In addition, the dynamic monitoring feature of this application will generate an action when the highly sensitive API will be called by any malicious application. The log file may be effectively utilized in analyzing app’s behavior and intentions, which will prove to be useful in several application domains. For example, this tool may be used in developing malware detection analysis based upon the API call, identifying resource usages in different app components, developing the tool for analyzing the application’s behavior, and constructing efficient mobile anti-malware tools, etc. We proposed a twostep model which combines static and dynamic analysis approaches. During the static analysis, pre-defined permission sets are used to determine whether an application has potential risks & in dynamic analysis for those suspicious applications we use reverse engineering method to embed monitoring Smali code for those highly sensitive APIs such as sending LOCATION, SMS accessin

Keywords: Android, API tracing, mobile security, mobile forensics

I. INTRODUCTION

Android on mobile devices has become a popular mobile operating system for various devices including smartphones. Android is the most widely used mobile operating system, with 81% of smartphones and 37% of tablets worldwide running this Google-made OS. With rapid development of network technology, the mobile Internet has been the development trend of the information age. According to the market research company Canalys released data, the global intelligent mobile phone shipments in 2011 has outpaced PC, reached 487,700,000 [7]. About the proportional share of the smartphone, Android OS has been in a rising trend. The first quarter shipments report of smartphone from Canalys showed that Android OS reached 75:6%, and there has been some increase compared with 69:2% of the previous quarter [3]. With the popularity and rapid development of Android OS, its security issues are also increasingly prominent. For instance, the security report from NetQin Company shows that they detected more than 65,227 new malwares in 2012, a 263% increase over 2011. And the vast majority of malicious software is designed to attack Android and Symbian devices. Moreover, Android devices accounted for the number of devices being attacked 94:8%, and software for the purpose of stealing user's privacy data reached as high as 28%, ranking first in all types of malicious behavior [11] Even though Android is built based on Linux, the API tracing techniques developed for desktops may not be applied. Android has different system architecture. At a lower level each application is encapsulated into a separate process and it is run by using the services provided by Linux kernel. Within each application, a virtual machine, known as a Dalvik Virtual Machine (DVM) provides a run-time environment for the Java components included in the application (app). All apps contain both Java and native components. Native components are simply shared libraries that are dynamically loaded at runtime. In the Dalvik virtual machine (DVM), a shared library named libdvm.so is then used to provide a Java-level abstraction for the app’s Java components. Application developers heavily make use of the objects and methods provided by the Java Library included in the DVM. To understand or grasp the intentions of the apps, it will be more beneficial to trace/utilize the Java-level semantics that comprehend the behaviors of the Java components in the app rather than focusing on system call histories captured at the lower level. Currently, the method for detecting user privacy data leakage in intelligent mobile phone platform mainly has two categories, static and dynamic. Static analysis methods mainly focused on the control flow, data flow and structural analysis [15]. But Android application mostly written with Java, the program will inevitably exist a large number

All rights reserved by www.ijste.org
of implicit function calls, and the static analysis methods cannot effectively handle it. At the same time, static method can obtain the concrete execution path of the application without executing the source code, but it does not determine whether the path will actually be performed, which can only be verified by dynamic method.

This paper firstly introduces the de-compilation tool named as apktool that decompiles APKs and gets the source code in smali, and defines a series of sensitive APIs to build a sensitive API library. Then to detect the sensitive APIs in source code by keywords searching [2], and finally view the code relevant to sensitive APIs to analyze the application and determine whether there is sensitive data leakage.

II. RELATED WORK

Some background research on Android security is briefly introduced as follows Kui Luo proposed a byte code converter for malicious code of leakage privacy, converting DVM (Dalvik Virtual Machine) byte code into Java code, and putting the Java code into the Indus (a static analysis of Java code and slice tool) to analyze [17]. Leonid Batyuk proposed a method by decompiling sample applications, in the premise of not affecting the program core function, through modifying the binary code to separate the malicious code [2].

In signature-based malware detection systems malwares are detected by utilizing the sets of rules or policies [1]-[6]. If an attack shows a signature exactly matching one of the known signatures, then it can be easily detected. However, this mechanism may not be effective against new malwares with unknown signatures.

In anomaly detection approaches machine learning algorithms are first used to obtain classifiers from the known malware behaviors [12]. Then, the classifier will be used at run-time to detect malwares. Although anomaly detection is able to detect new or evolved malwares more effectively compared to the signature-based approaches, it sometimes causes high false positive. Then, the classifier will be used at run-time to detect malwares. Although anomaly detection is able to detect new or evolved malwares more effectively compared to the signature-based approaches, it sometimes causes high false positive. Malware detection techniques may also be classified into two different categories, static vs. dynamic. In static approaches the classifier or signatures will be obtained only from the apps codes, which remove the necessity to collect the data by running the apps [3] these approaches have limitations. Metamorphic and polymorphic techniques may be applied to generate new signatures for the same virus. If polymorphic techniques are used, due to the encryption, it is difficult to generate the signatures; if metamorphic techniques are applied, all the codes will be obfuscated. In dynamic approaches, they obtain classifiers or signatures only based upon data obtained at run-time [21]. They don’t have limitations as in the static approaches, but it becomes critical how to design and conduct app running experiments to capture their behaviors or signatures in a comprehensive manner.

III. METHODOLOGY

![Malware behavior analysis framework](image-url)

Fig. 1: Malware behavior analysis framework
Generally speaking, methods for malware analysis mainly include static and dynamic approach. Static analysis is a kind of method based on program's source code. It has the advantages of being wide coverage and can analyze the source code comprehensively. However, static method is based on source code. And if we cannot get the target source code, through decompiling or reverse engineering, it is hard to analyze the program accurately, especially in the occasion that the target program has been obfuscated. Dynamic analysis refers to the tracking and monitoring its run-time behavior through running the program. This kind of method is more accurate for capturing the actual malicious program behavior. Meanwhile, the dynamic method has its own disadvantages because of its limited execution coverage, that is to say we cannot guarantee all of the running paths have been triggered during the test.

In this paper, we present a combination of static and dynamic security analysis model that can make up for their shortcomings with each other, enable the analysis of malicious behavior more comprehensively and accurately. Figure 1 shows the whole steps. Before analyzing the Android application, APK (Android application package) needs to be statically decompiled to get the corresponding configuration and Smali [14] files. Among them, the configuration file with the format of AndroidManifest.xml is mainly used for permissions filtering stage, and the Smali files are mainly applied to dynamic monitoring module. First of all, we choose those suspicious applications with great potential to leak user's privacy. Then if a program is suspicious, enter into the dynamic monitoring module, where input the target Smali codes, embed some tracking code, repackage and re-sign the APK. In future, once the APK is running, we can dynamically monitor the behavior of privacy leakage and give immediate alarm for users. And those alerts or logs can be used for further detailed analysis manually or automatically. Next, we will discuss the three core components of the framework: APK Static De-compiler, Permission Filtering Module and Dynamic Monitoring Module.

### A. APK Static Decompiling:

Before permission filtering and dynamic monitoring, we need to extract the Android application's AndroidManifest.xml file and Smali files corresponding to the target APK. The Android application is an installation package ended with suffix .APK (an acronym for Android Package). APK is similar to .exe executable file in PC, after installed can be executed in Android OS immediately. APK is actually a compressed file compliance with the ZIP format, which can be extracted by popular .zip compatible decompression tools. In addition, it must be noted that most applications are code-obfuscated, and the unzipped file is not able to analyze directly. It should be decompiled to extract its resource, permissions, the intermediate representation files. In this paper, we use the apktool [1] for decompiling. The file structure of Android application after decompiled is shown in Table 1.

### B. Permission Filtering:

Through analyzing the malware samples, we find the process of privacy leakage has two steps: read the privacy information and send out. Accordingly, the potential causing privacy leakage permissions are also divided into two categories. One is mainly used to read the privacy data, such as android.permission.READ PHONE STATE, which allows to read phone state, such as SIM card, phone numbers, phone's IMEI (International Mobile Equipment Identity) and some others. The other is mainly used to send out privacy information. Figure 2 shows the procedure of permission filtering module.

<table>
<thead>
<tr>
<th>Directory/File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>res</td>
<td>Application's resource file, including pictures, sound, video and etc.</td>
</tr>
<tr>
<td>smali</td>
<td>Dalvik register bytecode files of APK</td>
</tr>
<tr>
<td>AndroidManifest.xml</td>
<td>The global configuration file of APK including the package name, permissions, referenced libraries and other related information of the application.</td>
</tr>
<tr>
<td>Apktool.yml</td>
<td>The configuration file of Apktool</td>
</tr>
</tbody>
</table>
In Figure 2, the security policy is the core part, where each security policy is a cross combination of the above two kinds of permission set. The first one is READ_P and the second one SEND_P. After the first step of static decompile, we can extract the application permissions set APP_P from the App's configuration file AndroidManifest.xml. During the static decompiling phase, we extract the permissions set APP_P form AndroidManifest.xml, and then classify APP_P into two categories, read and send defined above. We assume that if APP_P set on matrix has a valid value, that is to say, the APK requested permissions have the higher-risk, and then the app can be regarded as suspicious. [1]

C. Dynamic Monitoring:

This module is to monitor the call information of sensitive APIs in APK. We implement dynamic real-time monitoring by inserting monitoring code to the decompiled APK. Smali and baksmali are an assembler and disassembler respectively for the .dex format used by the DVM [5]. In this paper in order to avoid the differences between JVM and DVM, we try to directly rewrite Dalvik bytecode, insert the monitoring Smali bytecode into the decompiled Smali files. The procedure of dynamic monitoring module is shown in Fig.3

D. Smali Bytecode:

Smali is an Intermediate Representation(IR) for Dalvik Bytecode. Smali code is a kind register based language which can shield the source code level differences. For instance, malware sometimes use source code obfuscation to avoid detection. But in Smali code, the core sensitive APIs are inevitably exposed. So, we can monitor these sensitive APIs to track the behavior of those suspicious programs.

E. Smali Bytecode Library for Sensitive Apis:

The Smali bytecode library stores sensitive APIs and their corresponding Smali bytecode. The main function of the library is to locate the detailed position of sensitive APIs in Smali files after the target APK was decompiled. According to the typical sensitive APIs that malwares often used for leaking Android user's privacy data, we choose five and their class name, function name, and Smali bytecode are indicated in Table 2.

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive Types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>Void</td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>Boolean</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Byte</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>Sbyte</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Char</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>Long (64 bit)</td>
<td></td>
</tr>
</tbody>
</table>
F. Monitoring bytecode library for Sensitive APIs:

The monitoring bytecode library is to store the sensitive APIs calling information when the APK is running. For different APIs, monitoring information to be recorded is different. Such as SMS sending text messages, we need to record the message recipients as well as the content of the message. The unique part of each API is its input and output. According to API's function prototypes and register naming principles in Smali syntax, we can obtain the Smali register number of each API parameters. According to Smali syntax, there are two ways to determine a method that how many registers are available, which can be shown in Table 3.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Function Name</th>
<th>Description</th>
<th>Smali Bytecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>android.telephony.SmsManager</td>
<td>sendTextMessage(String, String, String, String, PendingIntent, PendingIntent)</td>
<td>Send messages</td>
<td>android/telephony/SmsManager/sendTextMessage(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/app/PendingIntent;Ljava/app/PendingIntent;)</td>
</tr>
<tr>
<td>android.location.LocationManager</td>
<td>getLastKnownLocation(String)</td>
<td>Get location</td>
<td>android/location/LocationManager/getLastKnownLocation(Ljava/lang/String)</td>
</tr>
<tr>
<td>android.telephony.TelephonyManager</td>
<td>getDeviceId()</td>
<td>Get ID, IMEI of phone</td>
<td>android/telephony/TelephonyManager/getDeviceId(Ljava/lang/String)</td>
</tr>
<tr>
<td>android.location.LocationManager</td>
<td>getSimSerialNumber()</td>
<td>Get SIM serial Number</td>
<td>android/telephony/TelephonyManager/getSimSerialNumber()</td>
</tr>
<tr>
<td>android.telephony.TelephonyManager</td>
<td>getLine1Number()</td>
<td>Get phone Number</td>
<td>android/telephony/TelephonyManager/getLine1Number()</td>
</tr>
</tbody>
</table>

IV. EXPERIMENT

Before experiment, some necessary tools such as Eclipse, JDK6, JRE6, Android SDK, Python2.7, and other tools will be installed. APK static de-compiler and permissions filtering module were implemented with Java. Among them, APK static compiling module is to call apktool [27]. Permissions filtering module mainly implements security policy settings; extract permissions feature from Androidmanifest.xml that generated by static decompiling module. Dynamic monitoring module is to scan the Smali files generated by APK static decompiling module, embed monitoring bytecode, repackage and re-sign the Smali files to generate a new APK. Then we run the new APK in Android emulator, it will generate some running logs or monitoring report to show what has happened. To illustrate Android users facing the growing threat of information leakage, we choose 642 popular applications to conduct experiments in the Permissions filtering module. We handled 642 APK samples by permissions filtering module and found that almost 26% apps have security risks for leakage user’s sensitive data. The security policy violated by most of those 642 apps is about IMEI permission combinations, namely, the most common information leaked is IMEI. The reason may be the IMEI can determine phone type and device parameters, and can provide accurate user identity information for developers and advertisers. The next is phone Number, Contacts, and Location. If these sensitive information is used illegally, it will possibly bring huge losses to users. To verify effectiveness and feasibility of dynamic monitoring module, we did experiments on the Android emulator in Windows7. We chose an APK, SendSMS_example.apk, that will automatically send text message in the background. The prototype system successfully detected the leakage behavior.

<table>
<thead>
<tr>
<th>Leakage corresponding to Policies</th>
<th>App Amount</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATION</td>
<td>27</td>
<td>17.59</td>
</tr>
<tr>
<td>SMS_TEXT</td>
<td>14</td>
<td>-</td>
</tr>
<tr>
<td>CONTACTS</td>
<td>32</td>
<td>44%</td>
</tr>
<tr>
<td>PHONE NUMBER</td>
<td>77</td>
<td>9.50%</td>
</tr>
<tr>
<td>IMEI</td>
<td>273</td>
<td>33.07%</td>
</tr>
</tbody>
</table>

V. CONCLUSION

Our experiments’ goal was to develop an efficient API call tracing tool for static & dynamic app behavior analysis. This tool may be applied to a wide range of applications including designing effective malware detection techniques on mobile devices.
The paper focuses on detecting sensitive API calls and studying obfuscated smali source code so as to identify the potential privacy theft and determine whether there is sensitive data leakage. As a conclusion, it shows some advantages of our approach:

1) Using Smali bytecode, it is based on intermediate language, which shows some advantages over Java source code method and it possesses anti-obfuscation to a certain degree.

2) The method is simple, just insert some monitoring Smali bytecode, and the performance influence can be ignored.

3) This method can be used in a wide scale, which can deploy remotely and provide monitoring service automatically.

Future scope of the paper will be adding a dynamic module to the current application which will generate an action log that will trace the behavior of each and every application through sensitive API calls.

REFERENCES


