Malicious Behavior Analysis for Android Applications

Quan Qian, Jing Cai, Mengbo Xie, Rui Zhang
(Corresponding author: Quan Qian)

School of Computer Engineering & Science, Shanghai University
99 Shangda Rd., Baoshan District, Shanghai, China
(Email: qqian@shu.edu.cn)

(Received Oct.31, 2013; revised and accepted Jan. 10 & Mar. 4, 2015)

Abstract

Android, as a modern popular open source mobile platform, makes its security issues more prominent, especially in user privacy leakage. In this paper, we proposed a two-step model which combines static and dynamic analysis approaches. During the static analysis, permission combination matrix is used to determine whether an application has potential risks. For those suspicious applications, based on the reverse engineering, embed monitoring Smali code for those sensitive APIs such as sending SMS, accessing user location, device ID, phone number, etc. From experiments, it shows that almost 26% applications in Android market have privacy leakage risks. And our proposed method is feasible and effective for monitoring these kind of malicious behavior.

Keywords: Android security, malicious behavior monitoring, permissions filtering, privacy leakage

1 Introduction

With the 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 [6]. 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 [5]. 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 malware 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 [18].

The main purpose of this paper is to analyze the Android applications accurately and comprehensively based on combining static and dynamic method to reveal the malicious behaviors of applications leaking user's privacy data. Privacy leakage mentioned in this paper refers to Android applications using sensitive permissions granted by user during the installation to collect user's privacy data, including user's device ID, IMEI, phone number, contacts, call records, location information, etc., and send user's privacy data via SMS or network.

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 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. Concerning about the dynamic method, there are traditional sandbox and dynamic taint tracking technology. Sandbox technology is a kind of isolated operating mechanism, is currently widely used in software testing, virus detection and other software security related areas [2]. Some background research on Android security is briefly introduced as follows.

Kui Luo proposed an 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 malcode [1]. Although this method can analyze the sample malcode effectively, it is unsatisfactory when the target program has been obfuscated. Enck implemented a Dalvik decompiler, DED, by using the static analysis package tool. The tool use Fortify SCA (a kind of white-box source code security testing software) to analyze the application's control flow, data flow, structure and semantics [19]. Qian et al also depends on Dalvik decompiling and gives a basic two-step-famework for Android malware behavior monitoring [22]. ComDroid analyzes the DEX byte code disassembled by Dedexer, and checks the Intent creation and transmission to identify the program broadcast hijacking vulnerabilities [7]. ScanDroid extracts the security specification from configuration files of Android application and checks the consistency between the application data flow and the specification [10]. ScanDroid is based on the WALA analysis framework, can only evaluate the open source applications.

Static analysis method can help to identify Android applications that applied unnecessary extra permissions or opened some interfaces for outer access without any protection. However, this method is easily confused by a variety of technologies, for instance obfuscation. While dynamic technology can make up for this. Enck proposed TaintDroid [8], which is a tracking framework to detect privacy leakage using dynamic taint. It modified the DVM layer of Android to complete the function of tainting data, and add Hooks to API interface of tainted sources to achieve infecting the private data accessed by applications, and finally got private data leakage through monitoring the socket of the network interface. In [3], it places a LKM module (Loadable Kernel Module) in the Android simulator, builds a sandbox system, intercepts and records all the underlying system calls from the kernel layer. However, modifying Linux kernel will lead to the Android emulator running extremely unstable, and the paper only uses automatic tools to simulate user interactions, not used in actual environment. Isohara uses a kernel based behavior analysis which depends on a log collector in the Linux layer of Android device and logs analysis application deployed on remote server [16]. However, the behavior understandability based on kernel level log is not good. Also, the server side uses regular expressions based signatures to detect the malware, but the signature maintenance and definition are difficult. Similarly, Peng et al. analyzes Binder IPC data from the server side to identify behaviors of different applications. By calculating the TP (threat point) value of different applications to evaluate whether an application is a malware or not [21]. However, Binder IPC based monitoring is a kernel based method and the TP threshold is hard to pre-defined. Asaf Shabtai designs SELinux [24] and deploys in Android system, which makes up for the defects of high level process and the experiments in HTC G1 show the feasibility of running SELinux in Android. However, SELinux policy maintenance is relatively cumbersome, and mobile phones with limited computing and storage capacity are not very suitable for deploying this kind of system.

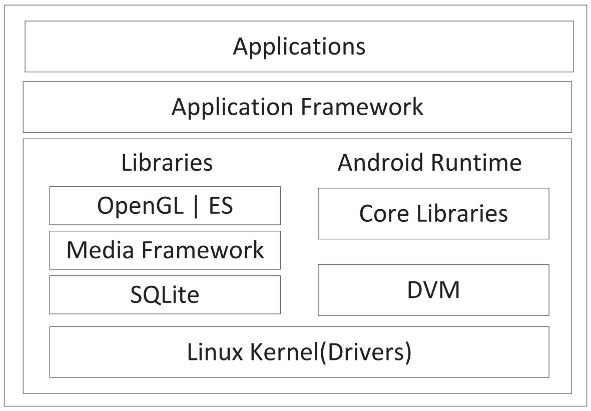


Figure 1: A brief architecture of Android system

The organization of the paper are as follows: Section 2 is about the Android basic framework and its security issues. Section 3 is the security analysis mechanism we proposed for android applications. Experiments are shown in Section 4. Section 5 summarizes the whole paper including the contributions and some future work.

2 Android Security Related Issues

2.1 Android Basic Architecture

Android, as a mobile operating system platform based on Linux kernel, was developed by the Google Open Handset Alliance [12]. Android has a layered architecture, including the Linux kernel layer, middleware layer and application layer, which can provide a uniform service for the upper layer, masks the differences of the current lay and lower layer [12]. The core functions of a smartphone are provided by the middleware layer, implemented by Java or C/C++. Applications running on Android are written in Java, and then multiple *.class* files are converted to *.dex* format by the Android DX tool. Each Android application is as a separate instance to run in DVM, and has a unique process identification number. Figure 1 gives a brief architecture of an Android system.

Among different components of Android, DVM [4], is the core part of Android platform. It can support Java applications, which are converted to *.dex* (Dalvik Executable) format. The *.dex* format is designed for a compressed format of Dalvik, suitable for memory and processor speed limited system. Dalvik is responsible for process isolation and threads management. Each Android application corresponds to a separate instance of Dalvik virtual machine, and can be executed in a virtual machine under its interpretation.

2.2 Android Security Mechanisms

Android security mechanisms are similar to Linux [13]. Android itself provides a series of mechanisms for the protection of privacy data. The core of Android security mechanism mainly includes the sandbox, application signature and permission mechanism. The permission mechanism limits applications to access user's privacy data (i.e. telephone numbers, contacts etc.), resources (i.e. log files) and system interface (i.e. Internet, GPS etc.). In permission mechanism, the phone's resources are organized by different categories, and each category corresponds to one kind of accessed resource. If an application requires access to certain resources, it needs to have the corresponding permissions. Android permission mechanism is coarse-grained and belongs to a kind of stated permissions before installing. Although this mechanism is simple, it also has some defects that cannot protect the user's privacy information adequately. Early in the conference of the ACSAC in 2009, Ontang et al questioned Android security model, and pointed out that the current Android permissions model cannot meet certain security requirements [20]. Enck proposed Kirin [9], a detection tool, to enhance existing Android permissions model. Based on a set of policy, Kirin is used to determine whether to grant the requested permissions to applications, and through the analysis of the Android application's Manifest file to ensure the granted permission in accordance with system strategy. Android permissions mechanism is coarse-grained and inflexible [13]. The application required permissions must be granted all before installed and cannot be changed after installation. This permission model leads to certain potential security threats. On the one hand, permissions to access private data will be decided by users. For those non-security awareness users, the permission granting process is casual and blind. During the installation phase, if the program obtains permissions to access privacy information, then can be arbitrary abuse of user privacy sensitive data at any time; On the other hand, the mechanism cannot effectively prevent permission elevation attacks. Applications can take advantage of a combination of permissions to steal the user's sensitive data.

In order to reveal Android apps leaking user privacy information behavior, according to the Android OS security mechanism, this paper proposed a malicious behavior analysis model combining the dynamic and static method, which will be discussed in detail in the next sections.

3 Android Malicious 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 trigged during the test.

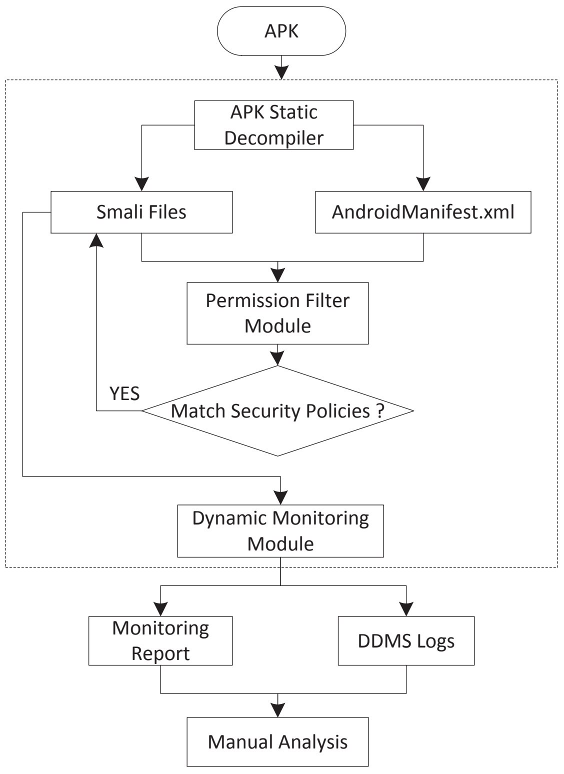


Figure 2: Malicious behavior analysis framework for android App

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 2 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 Decompiler, Permission Filtering Module and Dynamic Monitoring Module.

3.1 APK Static Decompiler

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 [23] for decompiling. The file structure of Android application after decompiled is shown in Table 1.

Table 1: The file structure after APK decompiled

Directory/File	Description
res	Application's resource file, including pictures, sound, video and etc.
smali	Dalvik register bytecode files of APK
AndroidManifest.xml	The global configuration file of APK including the package name, permissions, referenced libraries and other related information of the application.
Apktool.yml	The configuration file of Apktool

3.2 Permission Filtering Module

Some permissions may not exist risks by itself, but if there are some permissions combined there may exist a security risk. For example, an application applies for permissions to read phone state and sending messages, then there may exist the threat of sending the phone number or IMEI out. Permissions filtering module is based on a set of security policies to determine whether an application has some special risk permission combinations. For Android permissions, there are four different security levels. Those are $Normal, Dangerous, SignatureandSignatureOrSystem$.

- **Normal** lower-risk permissions that present minimal risk to Android apps and will be granted automatically by the Android platform without asking for user's explicit approval.
- **Dangerous** higher-risk permissions that would give access to the user's personal sensitive data and even control over the phone device that can negatively impact the user. Applications requesting dangerous permissions can only be granted if the user approves the permission explicitly.
- **Signature** permissions that the system grants only if the requesting application is signed with the same certificate as the application declared the permission. Signature permissions are automatically grant without user explicit approval if the certificates match.
- **SignatureOrSystem** permissions are only granted to applications that are in the Android system image or are signed with the same certificate as the application that declared the permission. Permissions in this category are used for certain special situation where multiple vendors have applications built into a system image and need to share specific features explicitly because they are being built together.

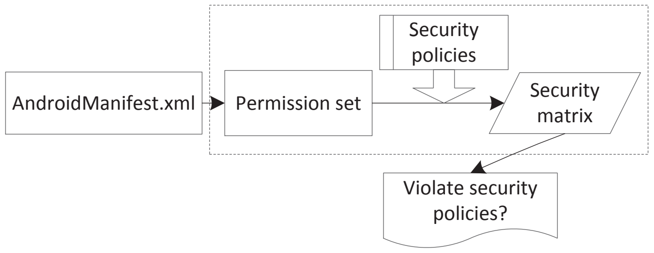


Figure 3: The procedure of permissions filtering module

From the above four permission levels, we mainly concern the *Dangerous* level which has great potential risks for leaking user privacy data. Moreover, 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. At present, we are only focused on two leakage ways, one is SMS (Short Message Service), and the other is network transmission, namely $android.permission.SEND\_SMS$ and $android.permission.INTERNET$. Figure 3 shows the procedure of permission filtering module.

In Figure 3, 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 $READ\_P = \{a\_1, a\_2, ..., a\_n\}, n \in N$ and the second one $SEND\_P =$

$\{b\_1, b\_2, ..., b\_n\}, n \in N$. The security policy is $S\_i = \{a\_i, b\_i\}, i \in N$. The set of all security policies are $SECURITY\_P = \{S\_1, S\_2, ..., S\_n\}, n \in N$. After the first step of static decompile, we can extract the application permissions set $APP\_P = \{p\_1, p\_2, ..., p\_n\}, n \in N$ from the App's configuration file *AndroidManifest.xml*. We define a permission matrix, the column for accessing privacy data permissions and the row for sending permissions. Through the values of matrix we can determine whether some risky combination of two permissions exists. Matrix model can represent a combination of permissions, not only can reflect the presence or absence of permissions, but also can demonstrate the relationship between permissions in detail. Matrix model is shown in Table 2.

Table 2: An example of permissions matrix model

Read Permission	**Send Permission**	
	SEND_SMS	INTERNET
ACCESS_FINE_LOCATION	1	0
READ_CALENDAR	0	0
READ_PHONE_STATE	0	1
READ_OWNER_DATA	_	_
READ_SMS	_	_
...	...	...

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 (here is 1), that is to say, the APK requested permissions have the higher-risk, and then the app can be regarded as suspicious. For example, the second row of table 2, "0" means that the application did not violate the security policy of leaking user calendar data risk because there are no permission combinations of $(READ\_CALENDAR, SEND\_SMS)$ and $(READ\_CALENDAR, SEND\_INTERNET)$. Conversely, "1" indicates that the APP_P set of an application holds the risk of phone state and user location leakage.

3.3 Dynamic Monitoring Module

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. The Android developers write the application in Java, compiles it into Java bytecode, and finally transfers to the Dalvik bytecode which can be executed in DVM. So it is straightforward to do the monitoring reversely by converting the Dalvik bytecode to Java bytecode, then rewrite the Java bytecode, and finally convert the rewritten Java bytecode back to Dalvik bytecode. However, this kind of approach sometimes does not work. First of all, there are several important differences between JVM(Java Virtual Machine) and DVM. The most obvious one is that JVM is based on stack whereas DVM is register based. Several tools, such as *dex2jar* [14] and *ded* [19], attempt to convert Dalvik bytecode back to Java bytecode. However this is not a lossless converting, that is to say, some information from the Java bytecode is lost when being converted to Dalvik. These tools try to infer the missing details based on the context, but sometimes the inference is unreliable (as described by Reynaud et al. [23]). Even though these errors may not prevent static analysis on the converted Java bytecode, in our experience they often lead to invalid Java bytecode or later invalid Dalvik bytecode. In other words, after we converted an application's Dalvik bytecode to Java bytecode (e.g. *dex2jar*) and then back to Dalvik bytecode, the resulting application sometimes failed to run. So the feasible way is to directly use the Dalvik bytecode.

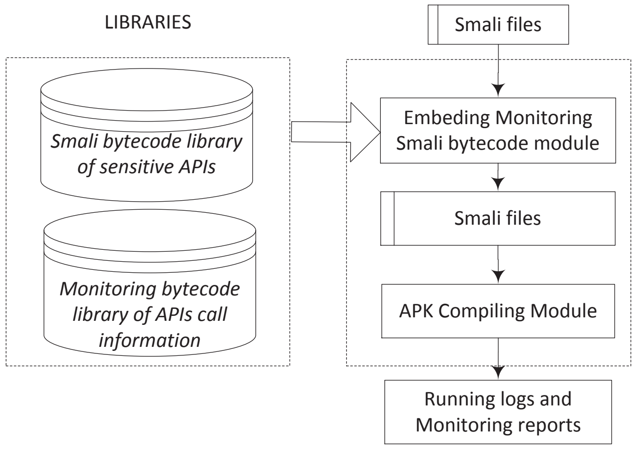

LIBRARIES
Smali bytecode library of sensitive APIs
Monitoring bytecode library of APIs call information
Smali files
Embeding Monitoring Smali bytecode module
Smali files
APK Compiling Module
Running logs and Monitoring reports

Figure 4: The procedure of dynamic monitoring module

Smali and baksmali are an assembler and disassembler respectively for the dex format used by the DVM. Its syntax is loosely based on Jasmin's syntax [11]. Smali is an intermediate representation of Dalvik bytecode. Smali can fully realized all the features of dex format (annotations, debug information, thread information, etc.). Moreover, dex and Smali can convert lossless between each other. So, 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 Figure 4.

In Figure 4, we can obtain Smali files from the static decompiling. Then locate the concrete position of the sensitive API, and embed monitoring Smali bytecode to each different sensitive API. After that we use apktool to repackage the modified Smali bytecode to create a new APK and use the signature tool to re-sign it. Running the new APK on Android emulator, we can use logcat to view the runtime logs. It can generate a log on SD card which records the detailed call information of those

Table 3: Descriptoin of Smali syntax

Type		**Syntax**	**Meaning**
Primitive Types		V	void
		Z	boolean
		B	byte
		S	short
		C	char
		I	int
		J	long (64 bits)
		F	float
		D	double (64 bits)
Reference types	Object	Lpackage/name/ObjectName	Package.name.ObjectName
	Array	[primitive type signature	[I, represents a array of int, like int[] in java
	Array of Objects	Lpackage/name/ObjectName	[Ljava/lang/String, represents a array of String Objects

sensitive APIs.

1) **Smali Bytecode.** Smali [11] is an Intermediate Representation(IR) for Dalvik Bytecode. Smali code is a kind of 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.

   In DVM, although all the register is 32 bits, it can support any data types. In order to represent a 64 bits type (Long/Double), it uses two registers. Dalvik bytecode has two types, primitive types and reference types. Reference types are only Objects and Arrays. The Smali syntax is indicated briefly in Table 3.

   Objects take the form Lpackage/name/ObjectName, where the leading L indicates that it is an Object type, package/name is the package that the object is in, ObjectName is the name of the object. For example, Ljava/lang/String; is equivalent to java.lang.String. Arrays take the form $[I$, i.e. int[] in java. Methods take the form as below:

   Lpackage/name/ObjectName;→
   MethodsName(III)Z.

   In this example, *MethodsName* is obviously the name of the method. *(III)Z* is the signature of the method. *III* are the parameters (in this case, three integers), and Z is the return type . For example, $method(I;\ [[II;\ Ljava/lang/String;\ [Ljava/lang/Object;)\ Ljava/lang/String;$ is equivalent to a string $method(int,\ int[\cdot][\cdot],\ String,\ Object[\cdot])$ in java.

2) **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 4.

3) **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 are 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 5 .

   When a method is invoked, its parameters will be placed in the last N available registers. For example, supposing a method has two parameters and five available registers ($v0 \backsim v4$), then the parameters will be placed in the last two registers ($v3$ and $v4$). Moreover, the first argument of the non-static method is always the object which call the method, and for static methods except there is no implicit this parameter, others are the same. For example, the method for sending text messages is as follows:

   public $sendTextMessage$ (String destinationAddress, String scAddress, String text, PendingIntent sentIntent, PendingIntent deliveryIntent).

   The above method has 5 parameters, defined as "*public*" which means it is a non-static method and the first register $v0$ is the object used to call the

Table 4: Smali bytecode library for sensitive API

Class Name	Function Name	Description	Smali Bytecode
android. telephony. SmsManager	sendTextMessage (String, String, String, PendingIntent, PendingIntent)	Send messages	Landroid/telephony/SmsManager; →sendTextMessage(Ljava/lang/String; Ljava/lang/String;Ljava/lang/String; Landroid/app/PendingIntent; $Landroid/app/PendingIntent;)V$
android.location. LocationManager	getLastKnownLocation (String)	Get location	Landroid/location/LocationManager; → getLastKnownLocation(Ljava/lang/String)
android. telephony. TelephonyManager	getDeviceId()	Get ID, IMEI of phone	Landroid/telephony/TelephonyManager; → getDeviceId()Ljava/lang/String
android.location. LocationManager	getSimSerialNumber()	Get SIM serial Number	Landroid/telephony/TelephonyManager; → getSimSerialNumber () Ljava/lang/String
android.telephony. TelephonyManager	getLine1Number()	Get phone Number	Landroid/telephony/TelephonyManager; → getLine1Number () Ljava/lang/String

method, namely, it stores "*this*" parameter. Meanwhile, for *sendTextMessage* we need to record the SMS destination address (corresponding to the destinationAddress parameter) and the SMS content (text parameter). According to the principle of register allocation in Smali syntax, the first parameter destinationAddress stored in registers $v1$ and the third parameter text stored in the register $v3$.

For other sensitive APIs, sometimes we need to record the return value of the method. And the instruction for the return value is the last instruction in the method. The basic bytecode for return instruction is *return* and there are four return instructions in total, which are shown in Table 6.

Table 5: Dalvik bytecode for registers

Instruction	Description
.registers	Specifies the total number of registers in a method
.locals	Indicates the number of nonparameter register in a method, which appears in the first line of the method

After we have got the syntax of Smali, the monitoring code for sensitive APIs in Table 4 can be provided in Table 7.

So far, we have introduced the mechanisms of our static-dynamic analysis method. Next, detailed experiments will show how it works.

4 Experiments

Before experiment, some necessary tools such as Eclipse, JDK6, JRE6, Android SDK, Python2.7, and other tools will be installed. APK static decompiler and permissions

Table 6: Dalvik bytecode for return value

Instruction	Description
return-void	Return from a void method
return vAA	Return a 32-bit non-object value and the return value register is an 8-bit register, vAA
return-wide vAA	Return a 64-bit non-object value and the return value register is an 8-bit register, vAA
return-object vAA	Return a Object value and the return value register is an 8-bit register, vAA

Table 7: Monitoring code for sensitive APIs of Table 4

Return type	Function Name	Register
void	sendTextMessage(String destinationAddress, String scAddress, String text, PendingIntent sentIntent, PendingIntent deliveryIntent)	v1,v3
Location	getLastKnownLocation(String provider)	vAA
string	getDeviceId()	vAA
string	getSimSerialNumber()	vAA
string	getLine1Number()	vAA

filtering module were implemented with Java. Among them, APK static compiling module is to call apktool [23]. Permissions filtering module mainly implements security policy settings, extract permissions features 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.

4.1 Android Markets Sensitive API Analysis

To illustrate Android users facing the growing threat of information leakage, we choose 642 popular applications to conduct experiments in the permissions filtering module. These 642 applications mainly come from Android online markets such as shouji.com.cn, appchina.com, market.goapk.com, and eoemarket.com. APK samples chose in this paper are popular applications coming from different app markets, and they mostly have more than half of Android users.

Meanwhile, in these experiments we only concern user's privacy data including: Location, SMS, Contacts, Address book, phone Number, IMEI (International Mobile Equipment Identity) and ICCID (Integer Circuit Card Identity). Leakage ways includes sending messages and network. We handled 642 APK samples by permissions filtering module and found that almost 26% apps have security risks for leakage user's sensitive data. Here, revealing user's privacy information refers to the app handled by permissions filtering, and it is considered to be suspicious. The specific statistical data is shown in Table 8.

Table 8: Analysis of suspicious Apps

Market	App Number	Suspicious Number	Ratio (%)
shouji.com.cn	59	6	10.17
appchina.com	283	53	18.73
market.goapk.com	66	25	37.89
eoemarket.com	234	85	36.32
Total	642	169	26.32

Then we analyze the permissions requested by APK through permissions filtering module. According to the security policies matrix proposed in section 3, we count the number of applications corresponding to each type of privacy information. The specific statistical results are shown in Table 9.

From Table 9, the security policy violated by most of those 642 apps is about IMEI permission combinations. Namely, the most common information leakage is IMEI. The reason may be the IMEI can determine phone type

Table 9: Analysis of privacy information leakage

Leakage corresponding to security policies	App amounts	Ratio(%)
Location	15	2.34
SMS text	1	-
Contacts	17	2.64
PhoneNumber	61	9.50
IMEI	199	31.01
ICCID	7	1.09

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 are used illegally, it will possibly bring huge losses to users.

4.2 Sensitive API Monitoring

To verify effectiveness and feasibility of dynamic monitoring module, we did experiments on the Android emulator in Windows7. Android source code version is Android 2.3.4_$r1$, and the kernel is Linux kernel 2.6.29 goldfish. The monitoring report generated by dynamic monitoring module is TXT, and the output position is in Android emulator SD card. We designed an APK, *showLog.apk* (also can use message box or phone ringing), to show the monitoring log. We chose an APK, *SendSMS_example.apk*, that will automatically send text message in the background. *showLog.apk* and *SendSMS_example.apk* successfully installed in Android emulator are shown in Figure 5.

For different sensitive API, through context we need to find its relative registers, the return value and then call different log functions to record the information while the app runs. For send text message API, $sendTextMessage$, the Smali code is shown in Figure 6.

From Figure 6, it shows the $sendTextMessage$ function has 5 parameters. Among them we only focus on the recipient number and the message contents. From the context, it is obvious that register $v1$ stores the recipient number and $v3$ stores the message contents. The other 3 registers $v2, v4, v5$ are null. So, we just need to pass the $v1$ and $v3$ to log function (Smali format) to record the SMS information while it runs. After embedding the log monitoring bytecode, we again used the apktool to repackage the modified Smali files, and then call signapk.jar to re-sign the new APK. The monitoring report was stored in external SD card. In order to view the log, we designed showLog.apk to show the recipient number, messages contents and timestamps. An example of detailed monitoring report on Android emulator is shown in Figure 7.

The above experiments show that the dynamic monitoring module was successfully embedded into the Smali files of original APK. The log records the detailed infor-

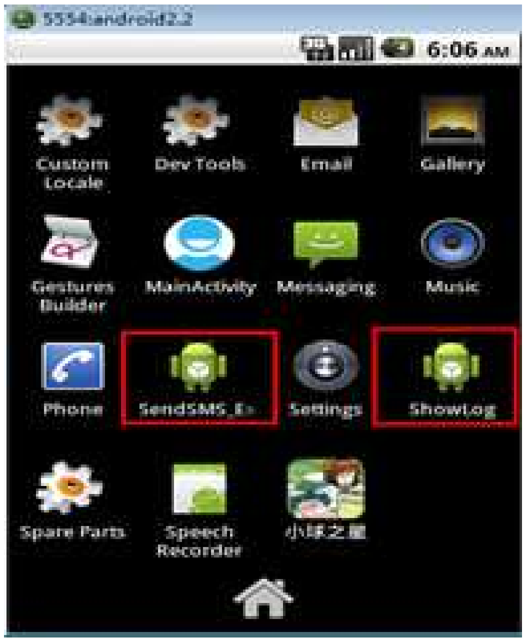
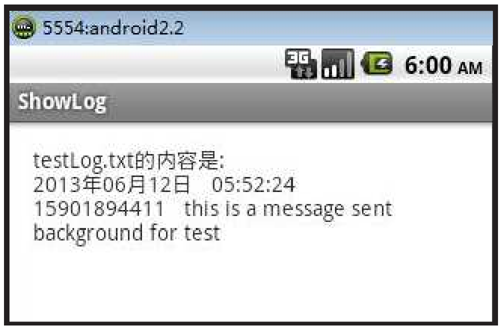


Figure 5: Running interface of installed *showLog.apk* and *SendSMS_example.apk*

```
iget-object v1, p0,
Lcom/example/sendsms_example/SendSMS_ExampleMainActivity;
->phoneNumber:Ljava/lang/String;
iget-object v3, p0,
Lcom/example/sendsms_example/SendSMS_ExampleMainActivity;
->SMSContext:Ljava/lang/String;
move-object v4, v2
move-object v5, v2
invoke-virtual/range {v0 .. v5}, Landroid/telephony/SmsManager;
->sendTextMessage(Ljava/lang/String;Ljava/lang/String;Ljava/lang/
String;Landroid/app/PendingIntent;Landroid/app/PendingIntent;)V
```

Figure 6: An example of Smali code for *sendTextMessage*



```
testLog.txt的内容是:
2013年06月12日   05:52:24
15901894411   this is a message sent
background for test
```

Figure 7: An Example of monitoring log for sendTextMessage

mation related to the sensitive API. Once we have found the suspicious behavior of an app, any further deep analysis, such as DDMS (Dalvik Debug Monitor Server), can analyze it more accurately and comprehensively.

5 Conclusion

A two-step malicious Android application detection method was proposed in this paper. First of all, we use permission combination matrix to discover those potential risk applications. And then those suspicious applications are further sent into the dynamic monitoring module to track the call information of the sensitive APIs while it is running. 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.

Further research directions include considering more sensitive APIs and provide a real App on Android market for fans to use. Also, we need integrate others malware detection method, such as dynamic taint analysis to conduct some cross-field deep research.

Acknowledgments

This study was supported by Shanghai Municipal Natural Science Foundation under grant No.13ZR1416100. The authors gratefully acknowledge the anonymous reviewers for their valuable comments.

References

[1] L. Batyuk, M. Herpich, S. A. Camtepe, and K. Raddatz, "Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications," in *Proceeding of the 6th International Conference on Malicious and Unwanted Software*, pp. 66–72, Fajardo, Oct. 2011.

[2] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: Trading privacy for application functionality on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pp. 49–54, Phoenix, USA, Mar. 2011.

[3] T. Blasing, L. Batyuk, A. D. Schmidt, S. A. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," in *Proceeding of the 5th International Conference on Malicious and Unwanted Software*, pp. 55–62, Nancy,France, Oct. 2010.

[4] D. Bornstein, *Dalvik VM Internals*, 2008. (https://sites.google.com/site/io/dalvik-vm-internals)

[5] Canalys, *Mobile Device Shipments Survey Report in the First Session of 2013*, May 2013. (http://cn.engadget.com/tag/canalys)

[6] Canalys, *Smartphone Shipments Survey Report in the Fourth Session of 2011*, May 2013. (http://cn.engadget.com/tag/canalys)

[7] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th International Conference on Mobile Systems, Applications and Service*, pp. 239–252, Washington, USA, June 2011.

[8] W. Enck, P. Gilbert, B. G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pp. 1–6, Vancouver, Canada, Oct. 2010.

[9] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pp. 235–245, Chicago, USA, Nov. 2009.

[10] A. P. Fuchs, A.Chaudhuri, and J. S. Foster, "Scandroid: automated security certification of android applications," *Technical Report of University of Maryland*, 2009. (http://www. cs. umd. edu/ ãvik/ projects/ scandroidascaa)

[11] Google, *Smali*, July 11, 2015. (http://code. google. com/ p/ smali/)

[12] Google, *Android Home Page*, 2009. (http://www. android. com)

[13] Google, *Android Security and Permissions*, 2013. (http://d.android.com/guide/topics /security /security.html)

[14] Google, *Dex2jar: Tools to Work with Android .dex and java .classfiles*, 2013. (http://code. google. com/ p/ dex2jar/)

[15] P. Hornyack, S. Han, J. Jung, S. schechter, and D. Wetherall, "These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pp. 639–652, Chicago, USA, Oct. 2011.

[16] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for android malware detection," in *Seventh International Conference on Computational Intelligence and Security*, pp. 1011–1015, Hainan, China, Dec. 2011.

[17] K. Luo, "Using static analysis on android applications to identify private information leaks," *Master Dissertation of Kansas State University*, 2011.

[18] Netqin, *Mobile Security Report in 2012*, May 2013. (http://cn.nq.com/anquanbobao)

[19] D. Octeau, W. Enck, and P. McDaniel, *The DED Decompiler*, 2011. (http://siis. cse. psu. edu/ ded/ papers/ NAS-TR-0140-2010.pdf)

[20] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in android," in *Proceedings of the 25th Annual Computer Security Applications Conference*, pp. 340–349, Honolulu, USA, Dec. 2009.

[21] G. Peng, Y. Shao, T. Wang, X. Zhan, and H. Zhang, "Research on android malware detection and interception based on behavior monitoring," *Wuhan University Journal of Natural Sciences*, vol. 17, no. 5, pp. 421–427, 2012.

[22] Q. Qian, J. Cai, and R. Zhang, "Android malicious behavior detection based on sensitive api monitoring," in *2nd International Workshop on Security*, pp. 54–57, Nov. 2013.

[23] D. Reynaud, D. Song, T. Magrino, E. Wu, and R. Shin, "Freemarket:shopping for free in android applications," in *19th Annual Network & Distributed System Security Symposium*, Hilton San Diego, USA, Feb. 2012.

[24] A. Shabtai, Y. Fledel, and Y. Elovici, "Securing android-powered mobile devices using selinux," *IEEE Security & Privacy*, vol. 8, no. 3, pp. 36–44, 2010.

Quan Qian is a Professor in Shanghai University, China. His main research interests concerns computer network and network security, especially in cloud computing, data privacy protection and wide scale distributed network environments. He received his computer science Ph.D. degree from University of Science and Technology of China (USTC) in 2003 and conducted postdoc research in USTC from 2003 to 2005. After that, he joined Shanghai University and now he is the lab director of network and information security.

Jing Cai is a master degree student in the school of computer science, Shanghai University. Her research interests include Android security and software security analysis.

Mengbo Xie is a master degree student in the school of computer science, Shanghai University. Her research interests include data privacy, privacy based data mining, computer and network security.

Rui Zhang received her B.E. and Ph.D. degree from Department of Electronic Engineering & Information Science, University of Science and Technology of China, in 2003 and 2008, respectively. After that, she joined the School of Computer Engineering and Science, Shanghai

University. Now, she is an associate professor and her main research interests include computer networks, network coding for wireless networks and wireless communication, etc.