IDENTIFICATION OF MALICIOUS ANDROID APPLICATIONS USING KERNEL LEVEL SYSTEM CALLS

DHRUV JARIWALA

A THESIS

IN

THE CONCORDIA INSTITUTE FOR INFORMATION SYSTEMS ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF APPLIED SCIENCE IN INFORMATION SYSTEMS SECURITY

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

OCTOBER 2014

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Dhruv Jariwala**

Entitled: **Identification of Malicious Android Applications Using Kernel Level System Calls**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science in Information Systems Security

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. N. Bouguila _____ Chair

Dr. J. Bentahar _____ Examiner

Dr. K. Galal _____ Examiner

Dr. A. Youssef _____ Examiner

Dr. A. Youssef _____ Supervisor

_____ Co-supervisor

Approved Dr. J Bentahar

Chair of Department or Graduate Program Director

_____ 20 _____ _____

Dr. Amir Asif, Dean

Faculty of Engineering and Computer Science

ABSTRACT

Identification of Malicious Android Applications Using Kernel Level System Calls

Dhruv Jariwala

With the advancement of technology, smartphones are gaining popularity by increasing their computational power and incorporating a large variety of new sensors and features that can be utilized by application developers in order to improve the user experience. On the other hand, this widespread use of smartphones and their increased capabilities have also attracted the attention of malware writers who shifted their focus from the desktop environment and started creating malware applications dedicated to smartphones. With about 1.5 million Android device activations per day and billions of application installation from the official Android market (Google Play), Android is becoming one of the most widely used operating systems for smartphones and tablets. Most of the threats for Android come from applications installed from third-party markets which lack proper mechanisms to detect malicious applications that can leak users' private information, send SMS to premium numbers, or get root access to the system.

In this thesis, our work is divided into two main components. In the first one, we provide a framework to perform off-line analysis of Android applications using static and dynamic

analysis approaches. In the static analysis phase, we perform de-compilation of the analyzed application and extract the permissions from its '*AndroidManifest*' file. Whereas in dynamic analysis, we execute the target application on an Android emulator where the 'strace' tool is used to hook the system calls on the 'zygote' process and record all the calls invoked by the application. The extracted features from both the static and dynamic analysis modules are then used to classify the tested applications using a variety of classification algorithms.

In the second part, our aim is to provide realtime monitoring for the behavior of Android application and alert users to these applications that violate a predefined security policy by trying to access private information such as GPS locations and SMS related information. In order to achieve this, we use a loadable kernel module for tracking the kernel level system calls.

The effectiveness of the developed prototypes is confirmed by testing them on popular applications collected from F-Droid, and malware samples obtained from a third party and the Android Malware Genome Project dataset.

Acknowledgments

First of all, I would like to express my deepest gratitude to my supervisor, Dr. Amr Youssef, for his constant support, heartily guidance and enduring patience during my graduate studies. This thesis would not have been possible without his help. His attitude and enthusiasm for scientific and academic research will always be my role model.

I also wish to express my appreciation to all the faculty and people at Concordia Institute for Information Systems Engineering (CIISE) for having such a warm and cosy working environment. To each of my professors, I owe a great debt of gratitude for the wonderful teaching, which has helped me in reaching this stage.

I am thankful to my parents, Mukesh Jariwala and Suchi Jariwala and my loving brother Chaitanya Jariwala for supporting me through every stage of my life and helping me when I need the most. I would not be able to achieve such heights without them.

Lastly, I thank my friends and collegues at Concordia University, specially, Jeet Jariwala, Parth Jariwala, Jaspreet Singh, Saurabh Oberoi, Song Weilong and Maryam Asgariazad for all the fun in Montreal and studies at the University memorable and pleasurable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Android is an open source Linux-based operating system led by Google. Many companies such as Sony, HTC and Samsung have deployed the Android operating system on their smartphones and tablets. On July 2013, the Google Play store [15] officially reached about 1.3 million published applications (apps) and over 50 billion downloads.

As a result of the increased capabilities and widespread use of smartphones, the abrupt increase in the number of application downloads, and the open source nature of Android, attackers and malware writers who are seeking for easy and rapid financial gain have shifted their focus and started creating malware applications dedicated to smartphones which are arguably easier targets compared to desktop computers. By injecting malicious code into legitimate applications, hackers can breach the privacy of users by accessing their address book and GPS coordinates, stealing personal information such as passwords, credit cards information, bank account numbers, IMSI (International Mobile Subscriber Identity) and IMEI (International Mobile station Equipment Identity) numbers. Some of these malicious applications can send premium SMS and calls and get root access privileges.

Android applications can be installed from Google Play Store (https://play.google.com/), Amazon (http://www.amazon.com) as well as from plenty of open source sites such as 4shared (www.4shared.com) and filecrop (www.filecrop.com). This helps attackers to easily spread their malicious applications by downloading clean applications, injecting their malicious code and then uploading them to third-party markets which lack proper mechanisms to detect malicious applications.

One of the important security aspects of Android is privilege separation, known as Application Sandboxing. In general, an Android application cannot harm or get access to other installed applications. However, at the time of installation, the application requests approval from the user to grant the required permissions. Once these permissions are granted, the application may have the ability to do other malicious activities that can be performed with these permissions. For example, an application with access rights to confidential data and eligible to expose data to public can easily steal the user's personal information.

Android applications are developed using Java programming language and get executed on the top of Dalvik virtual machine (dvm) middleware, whereas JNI (Java Native Interface) invokes the lower level native code. The higher level Android based behavior can be well understood from the Dalvik level semantics whereas JNI based behavior can be understood from OS-level, i.e., kernel level semantics such as reading or accessing files and executing a service or a process. Some analysts use Android's debug monitor to collect information about an application but it fails to record each and every event occurring in the application. In addition, some malware writers avoid logging information in the system to eliminate any traces of malicious activities.

Analyzing lower level system calls, in order to examine the behavior of applications and detect malware, seems to be a very useful approach. Whenever the application performs

any task, it issues a specific system call related to that particular action. For instance, in order to access GPS locations, socket() system call is invoked. Therefore, throughout this work, we investigate the use of kernel level system calls for the purpose of identifying malicious Android applications.

1.2 Objectives

With the increased growth of cyber attacks on the Android platform, we aim to design a security mechanism on the top of the existing Android architecture. By collecting and analyzing lower level system calls that are invoked during the execution of the applications, we aim to identify malicious activities and hence reduce the attacks on Android mobile devices.

The main objectives of our work can be summarized as follows:

- Develop a simulated environment for analyzing Android applications. This environment should allow monitoring the kernel level system calls invoked during the execution of applications.
- Perform off-line analysis based on the collected kernel level system calls and permissions, and investigate features that can be used to identify malicious activities/applications.
- Develop a real time monitoring that can be used to notify users for any malicious event occurring at the runtime of the applications.

1.3 Contributions

In Android operating system, all high-level Android-specific behaviors are indeed achieved via system call invocations. Android applications interact with the system via well-defined system call-initiated IPC (Inter-process Communication) and RPC (Remote Procedure Call) invocations to carry out their respective tasks. In order to track system calls such as read () and write (), we use loadable kernel module for hijacking the system calls invoked. This helps to trace all the Android applications for any malicious activities. Following are the main contributions of our work:

- The design and implementation of a prototype framework that can be used to perform off-line analysis of Android applications using static and dynamic analysis approaches. In the static analysis phase, we perform de-compilation of the analyzed application and extract the permissions from its '*AndroidManifest*' file. Whereas in dynamic analysis, we execute the tested application on an Android emulator where the 'strace' tool is used to hook the system calls on the 'zygote' process and record all the calls invoked by the application. The extracted features from both the static and dynamic analysis modules are then used to classify the tested applications using a variety of classification algorithms.

- The design and implementation of a prototype that can be used to provide realtime monitoring for the behavior of Android applications and alert users to these applications that violate a predefined security policy, for example by trying to access private information such as GPS locations and SMS related information. In order to achieve this, we use a loadable kernel module for tracking the kernel level system calls.

The effectiveness of the developed prototypes is confirmed by testing them on popular applications collected from F-Droid, and malware samples obtained from a third party and the Android Malware Genome Project dataset.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Android security, malware attacks on Android and related works are reviewed in the next chapter. In Chapter 3, we present the design and implementation of our off-line analysis and real-time monitoring frameworks. In Chapter 4, we present our experimentation results. Finally, in Chapter 5, we present conclusions of our research and some suggestions for future work.

Chapter 2

Android Security

2.1 Android Overview

Android is an open source operating system typically used for mobile devices. The main blocks of the Android platform are as follows [4]:

1. *Device Hardware*: The Android platform is capable of running on a wide range of hardware devices including mobile phones and tablets. Android is processor-agnostic as it utilizes the benefits specific to the underlying hardware device.

2. *Android OS*: The base operating system is built on top of a Linux-based kernel. All the hardware resources such as camera, GPS, WI-Fi, Bluetooth, telephone features and network operation are allowed through the Android operating system.

3. *Android Application Runtime*: The applications in Android are written in the Java programming language and executes in a Dalvik virtual machine. All Android applications execute within the security environment, enclosed within the application sandbox. Also, applications have some part of the filesystem where they can store their private application related data.

Android applications add extra features to the base Android platform. These applications can be classified into:

1. *Pre-installed Android applications*: Android has a set of pre-installed applications such as phone, message, web browser, maps and contacts. These are the basic functionalities of the Android operating system. These applications can also be used by other applications.

2. *User installed Android applications*: Google provides an open source development environment. This can help users install applications from the Android application market named Google Play store or from other third-part markets which contain thousands of applications in various categories.

The following series of cloud-based services are available from Google:

1. *Google Play*: It offers a variety of applications that users can install for free or purchase using their Android powered devices. Google Play provides a good place for developers to reach a large number of customers. Besides applications, it also offers services such as community review, license verification, and application security scanning.

2. *Android Updates*: The main function of this service is to notify users about the Android updates and new capabilities added to current version of the Android operating system.

3. *Application Services*: This provide a framework for developers to make use of cloud services such as cloud to device messaging for push notification.

2.1.1 Android Architecture

Figure 1 shows the software stack of the Android architecture [2]. As depicted in the figure, the Linux kernel is situated at the bottom of the system and is used to resolve system calls related to hardware resources.

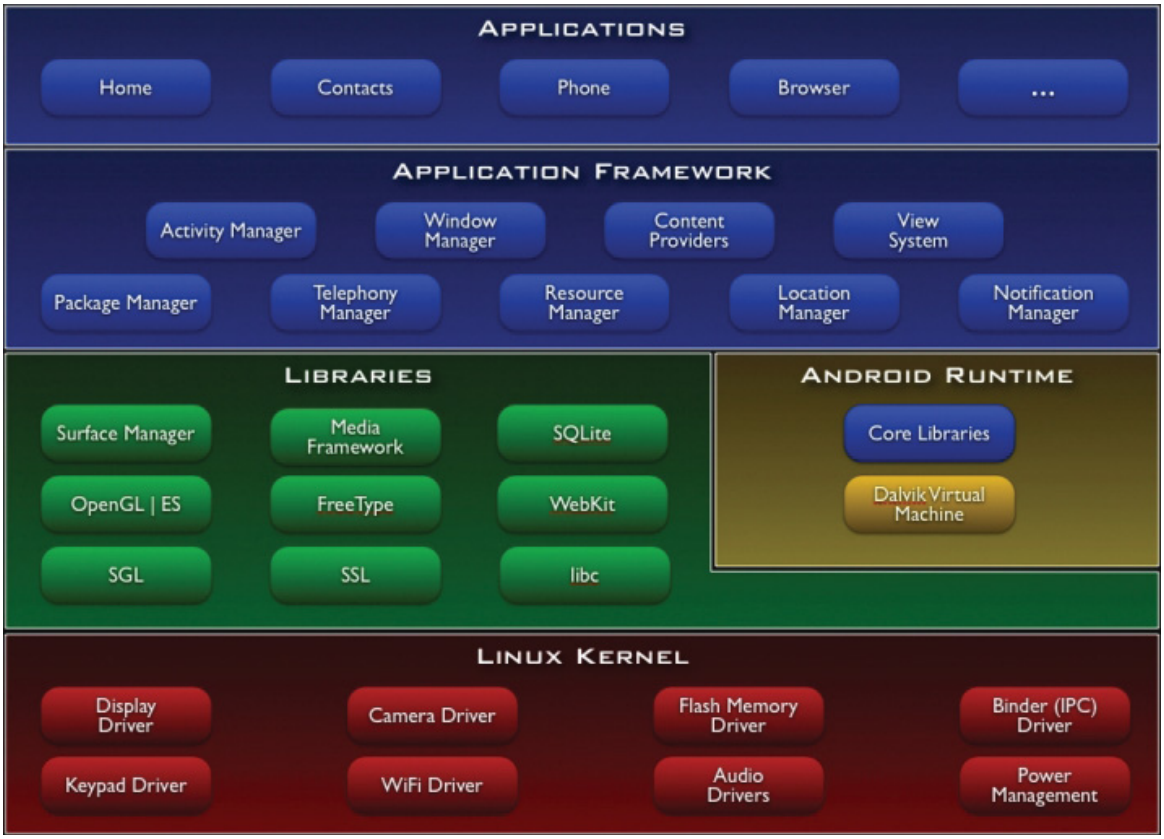


Figure 1: Android platform architecture [2]

The native libraries are used on the top of the Linux kernel. It includes various native libraries from surface manager to libc programmed in multiple languages.

Android runtime execution is based on DVM for handling process management. The JNI is not used because it requires heavy computational power, which degrades the system performance. In order to resolve this issue, DVM is used for the Android platform.

The application framework of Android consists of the system service and developer APIs. The system service includes various kinds of lower level functions. The developer APIs make use of lower level functions and create a unique feature that is made available to developers. Android also has a list of static permissions which is defined at this API level.

The Android Application layer of the architecture consists of all the applications installed on the system. Programmers can develop a variety of applications using these developer APIs. By default, Android comes with some basic functionality applications such as phone calls, message, contacts and calendar. There is some extra storage for users to install other applications, and data such as downloaded music and photos.

2.1.2 Components of Android Applications

An android application can have different types of components. In what follows, we briefly overview the use of each of these application components [1].

Activities

Activities [1] provide the user interface of Android applications. An activity displays the screen with which the user interacts with the application. Also, an activity has the responsibility of keeping track of ongoing operations and responding to actions performed by the users.

An activity has various life cycles. It starts with onCreate() and stops with onDestroy(). After an activity is created, it can be available to the user only when onStart() is called. When an activity is interrupted or placed in the background, it is in onPause() state. When an activity is brought back alive from a previous state, onResume() state is called. When an activity is in onStop() state, it remains alive but it cannot be resumed and gets disconnected

from the application manager window.

There can be more than one activity for an Android application, but the activity started at the time of launching an application is known as the main activity. Also, one activity can start another activity. However, once the new activity is started, the previous one runs in the background. The previous activity can be resumed by calling onResume or if the user performs back navigation.

Services

Services work in a similar fashion as activities [1]. However, services always execute in the background and keep running for a relatively longer time. Because services run in the background, they do not have a user interface. There are two ways for starting a service. The first one is by calling the startService() method. This is used when a particular task is to be performed and the service quits after that task is performed. The second approach is through application binding. This bound service is linked to an application where the application can start the service and kill it when needed.

Content Providers

Content providers act as data storage for applications [1]. The data stored in content providers can be private or made sharable across different applications. There is a default content provider available to any application by Google. In order to store data in a content provider, the developer needs to mention the data label using a valid Uniform Resource Identifier (URI).

Broadcast Receivers

Broadcast receivers [1] are capable of listening a specific state of the operating system or the application. They are usually used for activating a service at a specific state. Suppose we want some application to start after the phone is up and running. For such case, we have to first register a broadcast receiver that checks the state of the phone and notifies once the booting is finished. After that, the application can be signaled to launch.

In Android, the notification used for this purpose is termed Intent. The notification message consists of data along with the functionality to be preformed on it. Intent filters are used to get rid of unwanted Intents and help users get notification for important ones only.

2.1.3 Configuration of Android Applications

All the configurations of Android applications are stored in a configuration file named *AndroidMainifest.xml* [1]. It consists of all the application components used along with the external libraries. It also includes the list of requested Android permissions.

2.2 Levels of Android Security

2.2.1 Overview

Android has been designed with security as one of its cornerstone principles. As shown in Figure 2, Android utilizes a multi-layer security defense mechanism for protecting the user information. The presented defense strategy makes sure that processes do not collect more resources/information without having the required permissions.

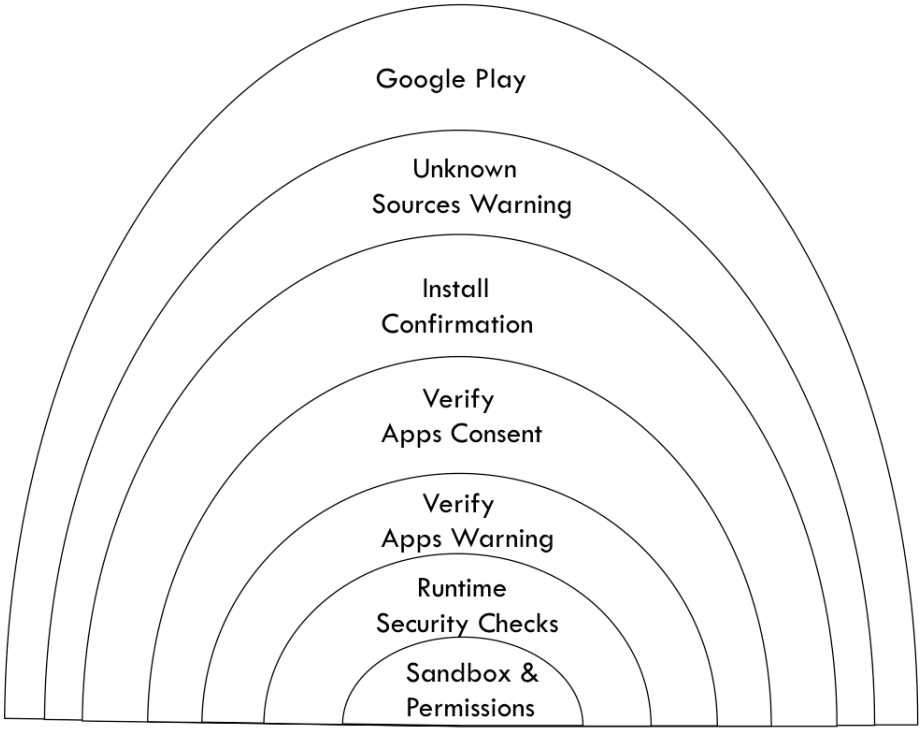


Figure 2: Layers of Android security [28].

In order to install an application, it has to pass through various security checks. Initially, upon the user's consent to install the application, it passes through Google Play and "Unknown Source" warning. If successful, it has to go through the Google "Verify Apps" security feature, where Google checks the APK (Application Package Kit) file against its own database of known malwares before installing the application. At last, the application is sandboxed and restricted to the granted permissions. Android security can be divided into the following levels [4]:

- Secure Android operating system development
- System and kernel level security
- Application level security

2.2.2 Secure Android Operating System Development

For making and keeping an operating system secure, it is always necessary to think about security from the very first steps of the operating system development cycle. According to Google, the key steps in the development process of a secure Android operating system are summarized below:

- **Design Review**: The security process starts from the very first step in the operating system life cycle. Every important feature of the Android platform is examined by many security professionals. The development process of the Android operating system goes to the next steps only if the design passes all the security aspects of an operating system.

- **Penetration Testing and Code Review**: At the time of developing the Android platform, this task is performed by the Google information security team and other security consultants. Their main task is to do penetration testing to identify any vulnerabilities in the system before its release.

- **Open Source and Community Review**: Google allows the Android operating system to be reviewed by external security professionals, security developers, code reviewers and secure code reviewing communities. Google Play also provides a blog for users to report any kind of problems, including security related ones, in the application.

- **Incident Response**: Even after deploying all the above precautions, there is still a possibility of security problems happening after shipping the operating system. For this reason, Android has a special response team to deal with vulnerabilities that are discovered once the operating system is live. This full-time security team monitors the security issues or any general discussion about potential vulnerabilities in the system. If any vulnerability is found, the Android security team has a strong response

process, which helps to get rid of that vulnerability, and ensures the minimum potential risk to all Android users. These responses may include cloud based responses for updating the Android platform and removing infected applications from the Google Play store.

2.2.3 System and Kernel Level Security

Android is a Linux-based operating system, customized for mobile devices. Android applications are written in Java and compiled to Dalvik executable (Dex) bytecode. Application related activities such as making a phone call or sending SMS are executed by Linux kernel level system calls. Also, all the high level operations of Android are eventually performed by the low level kernel system calls. The Linux kernel consists of about 370 system calls which can be categorized into 4 categories [38] based on their functionalities, namely file system, network, inter-process or virtual machine, and miscellaneous. Examples of functions related to file system calls are read() and write(), which assign a file descriptor for reading and writing a file, for sending/receiving network configuration or reading/changing the system configuration with the help of file '/proc'. The virtual filesystem calls are categorized depending upon the assigned file descriptor. System calls related to file system are also used to access data saved on the SD card and flash drive of the mobile device. In addition to the read() and write() system calls, the open() system call is also associated with file system. Network related system calls include mainly sendto() and recvfrom() calls on file descriptors related to network connection via connect() system call. IPC and virtual machine related system calls include some operations associated with processes such as scheduling, idling, process communication and timing. Examples of common system calls related to IPC and virtual machine are clock_getime(), getid(), getpid(), ioctl(), and futex(). The remaining system calls are related to GPS and other features on the Android device and are categorized as miscellaneous.

The Android platform performs security checks at the operating system level. The security aspects that covers the operating system level security of Android platform can be summarized as follows [4]:

- **Linux Security:** The base foundation of Android platform is the Linux kernel. Android uses all the security features of the widely used Linux kernel. The stability of the Linux kernel has undergone a constant process of research, and has improved by developers from all around the world. As a result, the underlying Linux kernel is considered to be secure and trusted by companies and security professionals. Some of the Linux kernel security features include:
  - A user-based permission model
  - Process isolation
  - Extended mechanism for secure IPC
  - Deleting unwanted and unsecured modules of the kernel.
- **Application Sandbox:**The Android platform makes use of Linux user based protection mechanisms for identifying and isolating application related data. The Android system allocates a unique user ID (UID) to each Android application and runs it as a unique separate process. This is termed as kernel level application sandboxing. The security between the application and operating system is enforced at the process level. Every application is assigned a group and user IDs. An application cannot communicate with other applications and it has a restricted access to the operating system. For instance, if an application tries to access another application's data or send message without having the required permission, then the operating system prevents these operations. As the application sandbox exists in the kernel, this security feature expands to the native code layer and the operating system application layer.

Since all the system libraries, application framework, application runtime environment, and installed applications run inside the application sandbox, memory corruption errors do not compromise the security of the device because only the application will be affected and not the complete device.

- **System Partition and Safe Mode:** In Android, the system partition consists of various components such as Linux kernel, native core operating system libraries, application framework, application runtime and the installed applications. This part of memory, i.e., the system partition, has read-only privilege. Hence, whenever the user boots the device in safe mode, only system related pre-installed Android applications are present. This enables users to boot their devices in a secure environment that has no third party applications installed in it.

- **Filesystem Permissions:** In Linux-based operating system such as Android, file system permissions guarantee that no user or application can change, modify or read another user's or application's data. This is because, in Android, every application has a unique UID and it can only access other application's data if that application explicitly declares that these files can be accessed by other applications also.

- **Security-Enhanced Linux:** Android makes use of Security-Enhance Linux [6] (SELinux) which enables mandatory access control (MAC) over all the processes. This also includes the processes running as a root or as a superuser. SELinux helps in making Android secure. With the help of SELinux, Android protects the system services, controls access to applications' resources and system logs, mitigates the effectiveness of malicious applications and protects users from any flaws in the device.

  From Android 4.3 and later versions, the discretionary access control (DAC) policy has been replaced by a MAC policy. MAC does not allow an application with root privileges to write or modify data exterior to raw block device. Hence, overcoming

the limitation of the DAC policy.

- **Cryptography:** Android offers a set of cryptographic APIs which can be used by applications to keep the sensitive data, such as credentials and banking information, secure from any external or network intrusions. These techniques use cryptographic primitives such AES, SHA, RSA and DSA.

- **Memory Management Security Enhancements:** Many common exploits are possible through memory corruption.

  The following enhancements were made to the memory management in the context of security:

  - **Android 1.5:**
    - Extensions to OpenBSD dlmalloc to prevent double free() vulnerabilities and to prevent exploits against heap corruption
    - OpenBSD calloc to prevent integer overflows during memory allocation
    - ProPolice to prevent stack buffer overruns
    - safe_iop to reduce integer overflows
  - **Android 2.3:**
    - Hardware-based No eXecute (NX) to prevent code execution on the stack and heap
    - Linux mmap_min_addr to mitigate null pointer dereference privilege escalation
    - Format string vulnerability protections
  - **Android 4.0:**
    - Address Space Layout Randomization to randomize key locations in memory

  - **Android 4.1:**
    - Avoid leaking kernel addresses
    - Position Independent Executable support
  - **Android 4.2:**
    - FORTIFY_SOURCE for system code
  - **Android 4.3 [5]:**
    - ADB Authentication
    - Android sandbox reinforced with SELinux
  - **Android 4.4 [5]:**
    - Per User VPN
    - Device Monitoring Warnings
    - Certificate Pinning

- **Rooting of Devices:** There are various methods to escalate from a normal user to gain root access. One of them is by unlocking the bootloader and installing a completely new operating system that allows root access. Another way of gaining root privilege is by installing a new operating system by connecting a physical device through a USB cable.

  Any application that gains root access is considered a dangerous security threat. In Android OS, some processes such as kernel and some core applications need root privilege for running. If any other application, other than the core applications, gains root access, then it may modify the working of operating system, kernel or any installed application's data. Encryption would not help against such threats because the encryption keys are also stored on the device. One of the solutions against such threats is that applications must add a layer of protecting data by encrypting with

the key stored on their respective servers rather than on the device. The other solution for protecting data is to make use of the hardware resources. For instance, the equipment manufacturer may limit the access to specific data such as DRM (Digital Rights Management) for video playback or the NFC (Near Field Communication) related trusted storage for Google Wallet.

- **Filesystem Encryption:** From Android 3.0 and onward, the filesystem encryption security feature is enabled. In this, all the user data is encrypted with dmcypt execution of AES128 with CBC and ESSIV:SHA256. The encryption key is derived from the user password and is protected by AES128. Additionally, the password is combined with a random salt and is then hashed repeatedly with SHA1. This provides resistance against password guessing attacks. The wide range of characters used in the password also helps protecting against dictionary-based password guessing attacks.

- **Password Protection:** Android can enable the use of password before providing access to the device. This helps to secure the device from unauthorized use. Also, it protects the cryptographic keys stored on to the device. There are six different ways of locking an Android device which include swipe, face unlock, face and voice, pattern, PIN and password.

- **Device Administration:** The Device Administration API is present in Android 2.2 and later versions. It enables a user with admin privileges on the device. An application can be granted admin privileges but it has to be activated by the user. Upon successful activation, this allows the user to erase data, change the password, lock the screen, set lock screen password expiration, monitor screen unlock patterns, set password rules, set storage encryption and disable camera. It can also wipe all the data from the device and restore it to a factory setting.

2.2.4 Application Level Security

- **Android Permission Model:** As explained earlier, each Android application runs in an application sandbox. There exists a set of APIs for accessing various system resources. These include:
  - Bluetooth functions.
  - Network or Data connections
  - Camera functions.
  - Telephony functions.
  - SMS or MMS functions.
  - Location data.

  In order to make use of these APIs, the application must request the associated permissions by specifying them in its *AndroidManifest* file. When the user installs an application, the system shows a dialog box describing the permissions requested by the application and asks for the user consent. Figure 3 shows an example of the dialog box displayed at the time of installing an application (Google Maps). Upon approval, the system installs the application. It is important to note that in order to install any application, the user has to agree upon all the permissions included by the developer. The user cannot deny any of the permission and continue installing the application. There are some system functionalities that can be disabled from the device settings such as GPS and WiFi. If the application requested GPS permission at the time of installation, the user can still restrict its use by disabling the GPS from the device settings.

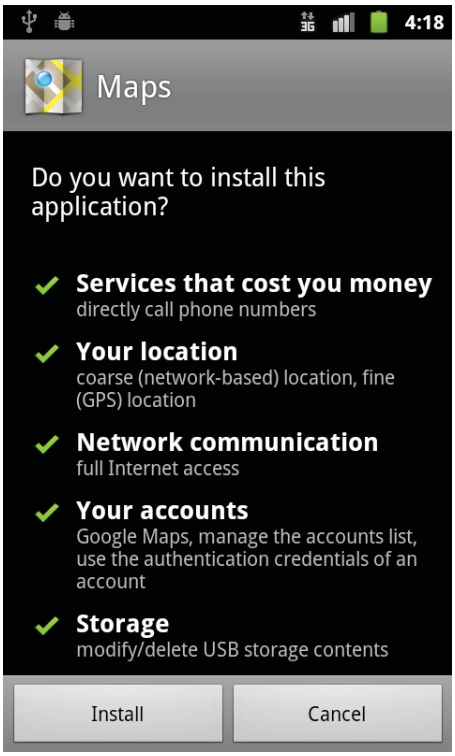


Figure 3: Example of the dialog box displayed at the time of installing Google Maps

- **Cost-Sensitive APIs:** In Android, various functionalities are triggered by different APIs. A cost sensitive API is one that, upon execution, has a cost associated with it. These APIs are located among the set of protected APIs managed by the operating system. Upon installation, the user has to grant permissions to applications that require using these APIs. The following APIs have cost associated with them:
  - *Telephony:* Android applications use this API to make calls or access telephone directories. If the application is malicious, it may call to a premium number to make money.
  - *NFC Access:* Android applications make use of this API for Near Field Communication (NFC) related functionalities. For instance, the Google Wallet may use NFC for payments.
  - *In-App Purchase:* Some Android application are available for free. Others are

for trials only. In order to continue using these applications, the user has to purchase the application or any associated product from the application itself. In this case, it is necessary to make an in-app purchase. This is possible by issuing APIs related to In-App purchases.

- *SMS or MMS:* An Android application with malicious intent may send messages to premium numbers.
- *Network or Data:* Some applications constantly download and upload data, causing user charge for using the Internet. In most of the cases, the user is unaware about the activity running in background. For security reasons, Android 4.2 and later notifies the user for using APIs related to SMS. It informs the user that there might be a charge upon sending the SMS so that the user can decide to cancel it if it is not a legitimate one.
- **Personal Information:** Some personal information is stored in system applications such as contacts, calendar and photos. In order for a third party applications to access such information, this application has to be granted a permission from the user by indicating it in its *AndroidManifest* file.
- **Sensitive Input Devices:** There are various sensitive data related devices such as the camera, Bluetooth, WiFi and NFC. The use of these hardware components is restricted by the Android operating system. In order to make use of such devices by a third party application, the developer needs to explicitly indicate this in the application *AndroidManifest* file. This will alert the user about the requested sensitive use of these devices. If approved, the requested device is then granted to the application.
- **Device Metadata:** Android has a feature of restricting access to sensitive data. However, other seemingly non sensitive data can indirectly identify unique features of the mobile device and then compromise the user privacy. The device

metadata security feature restricts the access to such data that may indirectly reveal information about the user characteristics and preferences. For instance, unless a permission is requested at installation time, an Android application is restricted from accessing the operating system logs, phone number, serial number, hardware number, browser history and network information.

– **Application Signing:** Every developer has to sign the application code in order to uniquely identify the authors of applications at the Google Play store. In case of security threats, developers can be identified easily based on their signatures and actions can be taken accordingly. In Android, every application runs in an application Sandbox environment where each application has a unique user ID UID and this helps in maintaining different users on the device through Inter-process communication. If there exists two applications from the same developer, then it is possible to access the data of one another. At the time of installing the Android application, the Android Package Manager verifies that the APK is actually signed with the certificate existing in the APK. If the certificate is already used by another application installed on the device, then it can share the same User ID.

– **Application Verification:** The Application verification feature is available in Android 4.2 and later versions. This feature enables Google to examine applications before actually installing them. Users have the option of enabling or disabling the "Verify Apps" option at the time of installing the application. If it is enabled, it alerts the user if the application being installed is malicious and consequently blocks malware applications from being installed.

2.3 Evolution of Android Security

In what follows, we briefly review the advancements and changes made in improving the security of the Android platform.

1. **Android Updates:** Android releases OS updates for security enhancements and also for additional features on the mobile device. This helps to fix any security flaw in the previous version and keep the device secure. If a security flaw is found inside the organization, then it has to protect it from being disclosed to the public and fix the issue as soon as possible. On the other hand, if the vulnerability is not known inside the organization and is posted on some security blogs, for example, then Google will start creating a patch for the system and once tested, it becomes available to the users. In general, if a vulnerability is discovered, the following measures are taken by the Google to fix it:
   - Google will notify the organizations that have signed under non-disclosure agreement (NDA) about the flaw and start working on fixing the problem.
   - The respective department where the flaw is found fixes the code.
   - The Android security team will fix the security related issues.
   - The patch is made available to NDA signed companies.
   - The Android team will post the patch on the Android Open Source Project.
   - The respective carrier pushes the update to their customers.

   There are two methods for releasing patches and/or updates to the system: over-the-air (OTA), and side loaded updates. When using OTA, the update is pushed onto all compatible devices and is also made available to download manually for a specific interval of time. When using the side loaded approach, the user can either download the update/patch from Google on the actual device or download it on a desktop computer and then install it on the device.

2. **Bouncer:** Google Bouncer [3] is a malware scanner released in 2012 by Google and is compatible with Android 4.2 and later versions. It is used to scan the Google Play store applications, mark suspicious applications and alert users about them before installation. Google Bouncer was initially designed to scan the applications from Google Play store only. However, later on, it became capable of scanning third party applications installed on the mobile devices. It also notifies the user when any application tries to send SMS to premium numbers and blocks them from being sent until the user approves this action.

3. **Android Device Manager:** In August 2013, Google released its official administration control manager named "Android Device Manager" (ADM) [3]. ADM allows users to have admin privileges over the mobile device. It is capable of remotely wiping, tracking and locating stolen devices. It can also lock and ring the physical device in case the device is misplaced. It is compatible from Android version 2.2 and higher. In a 4 months span, Google also released an application on Google Play store having the same functionalities.

4. **Security by Third Party Applications:** Apart from the security enforced by Google, there exists many third party applications that can be used to add more layers of security to Android. For instance, MacAfee, AVG Technologies, MobileIron and Lookout mobile security are some security applications available on the Google Play store. Their main focus is to protect sensitive information such as email, documents, applications and browser history, and settlings such as network configurations and WiFi passwords that are present on mobile devices. The following are some of the security measures taken by these applications [24]:

   - *OS Integrity:* This feature notifies the user if the integrity of the Android OS is threatened, which happens, for example, in the case when the device

is rooted.

- *Policy Enforcement:* The security policy is enforced upon the range of characters used in selecting a password. This policy enforces a selection of strong passwords for locking the device, lost device protection and filesystem encryption.
- *Data Containerization:* This helps to prevent data access by unauthorized applications. Enforcing the authentication and encryption policy protects the sensitive information.
- *Posture and Trust:* The trust level is calculated for the device by continuous monitoring. This feature is used to assess the security feature of the device and calculate the level of trust. Depending on the level of trust, it decides whether the access to the sensitive information is blocked or not.
- *Access Control:* If the device fails to comply with any of the security features, all its access is revoked or blocked.
- *Privacy Controls:* Some actions and controls that affect the user privacy on the device are constantly monitored.
- *Identity:* The user identity and the device associated with it help determine the information access over network.

2.4 Android Malware Attacks

In this section, we briefly review some of the malware attacks on the Android platform [34].

- **iBanking:** iBanking [27] is a very powerful mobile banking Trojan which was distributed through HTML website injection on banking website pages. It deceives the target users by imitating itself as an anti-virus security application

for Android. It was able to bypass two way authentication factors used by some banking websites. Once installed on the victim's device, iBanking can be controlled by the attacker either through HTTP or SMS control. The attacker can perform the following tasks using iBanking C&C server:

- Steal sensitive information
- Intercept, forward SMSs and phone calls to a server
- Upload contacts, GPS location and recorded audio to a server
- Redirect a call to the attackerŠs number
- Prevent its removal or uninstallation
- Restore device to factory defaults

According to ESET Security Software, when iBanking was removed from the Google Play store, a new version known as Android/Spy.Agent.AF which targeted Facebook users was released. It implemented webinject which injected a Java script into Facebook web pages, and created a fake Facebook verification page.

– **Android Pjapps:** The Android.Pjapps [7] is another famous malware code injected into legitimate applications and distributed via third party marketplaces. It is used to establish a back door between the Android device and the Bot server. The main aim was to build a large botnet controlled by a large number of Command and Control (C&C) servers. Apart from these, it was able to remotely access web browsers' history, send text messages, online Internet browsing and install applications. This malware runs as a service in the background without the user's consent. It constantly checks the C&C server for executing the following commands:

  - note: This command is used in order to send messages to premium rate numbers.

- push: This command is used for blocking SMS messages from being sent.
- soft: This command allows the application to be installed on the target devices.
- window: This command is used for browsing the Internet on the web browser.
- mark: This command helps to add website as a bookmark on web browser.
- xbox: This command is used for parsing code on the Android device.

– **Android.Genimi:** The trojan horse named Gemini [37] was injected into a large number of original legitimate games available on the Google Play store. These injected applications were later published on Chinese third party markets and were used to remotely access users' personal information and route it to the C&C server. When an application injected with Gemini [22] runs on a mobile device, a back door is created to collect sensitive information related to the user such as device serial number, hardware number, location of device, and phone information (e.g., IMEI, IMSI). Every 5 minutes, it tries to connect to the bot server and send the collected information to it. Also, it was capable of changing the list of remote servers as guided by the C&C server. Once the connection is strong and active, the trojan can perform the following actions:

- Call or send SMS to a specific number.
- Display a map or web page.
- Change list of remote C&C servers.
- Send contact information.
- Send SMS data to the server.
- Send phone identity information such as IMEI, IMSI and SIM Number.
- Install or uninstall an application

- Change the wallpaper.
- Create a notification in the notification center.

– **DroidDream:** The DroidDream [18] malware not only infected application on third party markets but also managed to infect more than 50 applications on the Google Play store. The authors of this malware injected the code in the legitimate applications in such a manner that it cannot be detected. The main aim of DroidDream was to steal phone information by injecting malicious code inside the legitimate application and to obfuscate the code in order to prevent its detection. When an injected application runs on the device, it sends following sensitive information to its remote C&C server:

  - SDK version of the device
  - Device model name and unique ID
  - IMEI
  - IMSI

  Once the connection is established successfully, DroidDream [21] sends sensitive information and performs internal check of whether the device was infected by previous versions of DroidDream to avoid multiple infection for the same device. DroidDream used two methods to inject the device. One is 'exploid' and the other is 'rageagainstthecage' for getting past through the Android security container. Once the device is rooted, it searches for a package named 'com.android.providers.downloadsmanager'. If the package is found, it means that DroidDream already infected the device. Some variants of DroidDream are 'Bowling Time' and 'Falling Down'.

– **Android.Bgserv:** The trojan named "Android.Bgserv" [36] [35] creates a back door in the background service and sends sensitive information to a remote C&C server. The infection is performed by changing the original code and

generating a repacked application. The information leaked to the bot server includes SMS messages, operating system version, phone number, IMEI and IMSI. All these information is uploaded to the remote server via HTTP POST method. The server controls the device by issuing a reply to the POST command.

– **GGTracker:** This Android trojan was identified by Lookout security firm [20]. GGtracker automatically sends a number of premium messages to private numbers. It was installed on the user's device by clicking on in-app advertisements. After GGTracker is installed and run on the device, it registers the user for premium subscription services. The trojan contacts another server in the background. This server is responsible for malicious behavior on the device and for intercepting the crucial approval of the data being sent. For instance, the service has to type the phone number and pin code received via SMS for successful registration. This was preformed in the background without user's consent. The charges was up to $9.99 for each SMS.

– **DroidKungFu:** *DroidKungFu* [19] was discovered in June 2011 by researchers at NC state University. Its later versions were released in July and August. These researchers were also able to identify the fourth, fifth and sixth versions of *DroidKungFu* codenamed *DroidKungFu4*, *DroidKungFuSapp* and *DroidKungFuUpdate*, respectively. The mode of infection was through repackaged applications. This trojan was recognized as the most sophisticated family of Android malware attacks. It consists of shadow payloads, root exploits, C&C server and code obfuscation. *DroidKungFu* was among the malware family that made use of encrypted root exploits, which prevents signature based detection of the malware. The encryption keys of encrypted root exploits kept on changing with the malware version. *DroidKungFu* also has the capability to steal

sensitive information by remote C&C server and can control the mobile device. The attackers made many changes in order to avoid detection by researchers and security professional. They used encrypted server address with their own encryption algorithm. Another important component of *DroidKungFu* is shadow payload. It consists of embedded application with malicious payload within the *DroidKungFu* package. *DroidKungFu* installs the embedded application upon gaining root access by root exploit. The reason for installing two applications is that if the user uninstalls the first repackaged application, the other embedded application can still be effective.

2.5 Android Security Analysis and Monitoring Tools

In this section we briefly review some of the work related to analyzing Android applications for malicious activities.

Saint framework [26] is a modified version of Android application installer. This custom installer, also known as AppPolicy provider, ensures that, at the time of installation, only applications that do not violate predefined policies in the AppPolicy provider can be installed. The Saint framework uses three policies: 'Install-time Policy Enforcement', 'Run-Time Policy Enforcement' and 'Administrative Policy'. Saint uses install time policy to control permissions granted by the application. The run time policy helps to control the hardware and software interactions with the Android middleware layer. The last administrative policy restricts the application to change its policy and assigns mandatory access control. The authors of Saint framework have gone to a great extent of checking existing applications' permissions for any suspicious permission requests. The central components of Saint mostly work on assigning permissions statically to third party applications, thereby deciding which operations an application may or may not perform at runtime.

DroidScope [40] is a visualization based malware analysis of Android applications. DroidScope performs analysis by exporting 3-tiered APIs, namely: hardware, OS, and Dalvik Virtual Machine. DroidScope also collects API related activities and tracks sensitive information leakage and Dalvik level instructions. DroidScope is powerful enough to analyze the Java level and API calls used in the Android application. It has four different analysis tools: native instruction tracer, Dalvik instruction tracer, API tracer and taint tracer. For identifying the detailed information about the application's execution, the native and Dalvik instruction tracer is used. For more high-level information about the API calls made by the application, the API tracer is used. The taint tracker is used track information leakage through both the Java and native components using taint analysis.

Apex [25] is a policy enforcement framework which uses a fine-grained permission granting model for installing Android applications. It allows users to select the permissions granted to the application and can also restrict access to resources. Apex uses extended package installer for installing applications through a simple user interface. This installer allows users to specify security policies which specify the rules stating the condition to grant a specific permission. These policies are stored in a repository. Whenever any application requests a permission, Apex searches the repository for the security policy and grants or denies this permission accordingly. Another important thing to notice is that Apex has backward compatibility, which means that it can revert back to original Android security mechanism.

CopperDroid [30] is an out-of-the-box approach for dynamic analysis of Android applications. CopperDroid is built on the top of Android QEMU Architecture. It performs analysis based on tracking system level calls, such as opening, reading, writing a file or executing a program, and Android specific high-level behavior, such as sending a SMS, and making a phone call. CopperDroid is capable of performing seemingly in the Android

platform. It was used to perform analysis on 1200 malware samples, and successfully identified the behavior of these samples. CopperDroid also has a web interface for users to perform dynamic analysis of their samples for detecting any malicious behavior. CopperDroid has three stages. The first one is system call invocation which uses VM Interpreter to track system calls. The second stage is binder analysis, where it dissects the communication such as IPC to understand the high level Android related behavior. The last one is the path coverage, where the application is traversed through every path possible for inspecting any malicious behavior.

System Call Sequence Droid (SCSDroid) [17] is one of the techniques used for detecting malicious repackaged applications. SCSDroid makes use of the thread-grained sequence of system calls to identify the malicious behavior. It first prepares the most common subsequence of malicious repackaged applications system calls belonging to same family. There are three claimed advantages for SCSDroid. First, it can detect malicious activities even if the malicious application sample is obfuscated or encrypted. Second, SCSDroid uses thread based system call tracking as opposed to process-based tracking. Finally, SCSDroid does not require any original benign application to prepare a training set. SCSDroid reached a detection accuracy of 95.97% among the 149 analyzed applications.

CrowDroid [11] is another tool for detecting malware in the Android platform through dynamic analysis. The complete design of CrowDroid is integrated with the Android architecture so that it can collect number of traces from real world users by crowdsourcing. All the data is collected on the central remote server and is divided into two sets, one contains the data collected from known malware and the other contains the artificial malware made for test cases. CrowDroid is mainly divided into three components. One is the data acquisition, which collects application data from real users using CrowDroid application. It uses '*strace*' to track system calls. The second component is data manipulation, which parses

the collected information received from *strace* system logs. It produces a feature vector for clustering data. The third component is the malware analysis and detection, for clustering and analyzing the feature vectors obtained from the previous step. It uses the k-means clustering algorithm for analyzing the data. The reported results show that CrowDroid is able to detect malwares from the pjApps family with 100% accuracy and from the HongToutou family with 82% accuracy.

YAASE [31] (Yet Another Android Security Extension) aims to protect information from being leaked by any malicious application. It provides an extension to the Android framework and uses *Taintdroid* architecture as a base for tainting and tracking data. On the top of *Taintdroid*, YAASE supports user defined labels by enforcing filtering policy on the collected tainted data. YAASE has policies defined by the user stating all the applications authorized for labeling data. If any application tries to access data labels whose access is not granted in the defined policies, then its access to that data is denied.

AASandbox [9] performs static and dynamic analysis of Android applications. Static analysis engages searching malicious code in the application and it compares the scanned application with a know database of malware samples by matching the signatures of the application being analyzed. AASandbox installs a kernel module for intercepting system calls during the dynamic analysis process. When the application runs, Android 'monkey' is started to perform random gestures on the application. The final kernel log is used to analyze the malicious behavior of the application. AASandbox can also be deployed on the cloud.

Stowaway [14] is a another tool that applies static analysis on the analyzed applications. After that, the tool applies mapping of permission with each operation. The aim of Stowaway is to investigate excess privileged permissions. API mappings were built inside the

Android emulator to determine the permissions required to interact with system APIs.

Static analysis of Android applications was also performed using an extended version of Julia [29], which is static analysis tool for Java bytecode. Since Android applications are written in Java and shipped in Dalvik bytecode, they need to be converted into Java bytecode before applying this tool. This static analyzer tool keeps track of commonly used Java statements and generates an abstract overview of the application.

Bugiel *et al.* proposed a security framework named XManDroid [10] which monitors real-time communication between applications and verifies the inter-process communication against a set of predefined security policies. The main aim behind this framework is to prevent privilege escalation. XManDroid maintains the system state of all the installed applications and the communication links requested by the applications. It also monitors the Inter-Component Communication traffic and checks it for any privilege escalation. XManDroid has the limitation that it cannot control communications related to the Internet.

In what follows, we briefly describe some techniques and tools that can be used for real time monitoring of Android applications.

Taintdroid [13] is one of the most feature-complete and well-documented security enhancements made for Android. It is an extensive modification for the entire Android stack to track the flow of sensitive personal data from third party applications at runtime. The modification allows *Taintdroid* to detect when sensitive data is leaked in various ways such as by sending an email or SMS containing personal data or by uploading a file directly. *Taintdroid* "taints" sensitive data to keep track of its use throughout the system. It logs the information about the application that transmits the data and its destination address. This realtime monitoring helps the users know about the intent of the application and to get rid of malicious ones. *Taintdroid* goes to the extent of controlling what may happen

with sensitive data at runtime. It has a high level of instrumentation which results in up to 27% runtime overhead. *Taintdroid* made some changes in the Android's architecture by modifying 4 modules: variable level, method level, message level and file level. The VM interpreter performs the variable-level tracking of the application code and provides meaningful information about the data and not the code. The message-level tracking between the application helps to track messages exchanged during IPC. *Taintdroid* uses method-level tracking for running the native code without instrumentation of taint propagation. It also uses file-level tracking, which guarantees the tainted information is persistent.

Isohara *et al.* proposed a kernel based behavior analysis [16] for inspecting Android malware. They used a behavior monitoring framework for auditing all security related activities performed by the applications. The system mainly consists of a log collector and a log analyzer. The log collector collects all the system calls related to an application and filters out events associated with the target application. Using regular expressions, the log analyzer searches the activities and the associated signatures of any suspicious behavior. This framework targets any information leakage related to smartphone identification such as phone number, Google accounts, SIM serial and IMEI numbers. Out of the 230 analyzed applications, it was found that 37 applications leaked such private information.

FireDroid [32] provides an efficient security mechanism by executing fine-grained security policies without modifying any Android source code or the underlying Linux OS. FireDroid monitors the execution of all the processes in a secure environment. This makes it efficient as compared to other approaches. It can enforce strict security policy to stop attacks by exploiting the vulnerability of the application or the Android system services. FireDroid uses interposition of system calls produced from applications to enforce the security policy. It also acts as a network firewall controlling the interactions made by system

calls. FireDroid uses the *ptrace* command for getting system calls interactions where processes are monitored and a policy is enforced for collecting required information.

Min *et al.* [23] proposed a runtime based dynamic analysis for detecting Android malware. The complete design is based on monitoring the logs collected by executing the tested application in a customized environment capable of hooking system calls. Firstly, they performed static analysis of the application and obtained information such as list of all permissions, activities, broadcast receivers and services. The analyzed application is then executed in a controlled emulator with a modified (loadable) kernel modules to track the flow of sensitive data. This system uses Robotium for creating random events on the screen and performs monitoring of system logs for any malicious flow of information. Out of the 350 analyzed applications, 82 applications were responsible for data leakage and 8 applications sent SMS to premium numbers.

Bauer *et al.* [8] proposed a dynamic security analysis technique based on runtime verification. The proposed system is capable of showing that the system is working securely and satisfying a specific security policy. It monitors the Android system passively and alerts the user in case of any breach in the specified policy. The user can specify sequences of events that are considered to be suspicious. For example, if an application checks the user's location via GPS and then sends it via the Internet, then the system can be configured to alert the user by sounding an alarm when such sequence of events happen.

Chapter 3

System Design

3.1 Introduction

In this chapter, we describe the main contributions of our work which can be divided into two main components. The first one, as depicted in Figure 4, is a framework that can be used to perform off-line analysis of Android applications using static and dynamic analysis approaches. In the static analysis phase, we perform de-compilation of the analyzed application and extract the requested permissions from its '*AndroidManifest*' file. Whereas in dynamic analysis, we execute the tested application on an Android emulator where the *strace* tool is used to hook the system calls on the 'zygote' process and record all the calls invoked by the application. The extracted features from both the static and dynamic analysis modules are then used to classify the tested applications, as malicious or benign, using a variety of classification algorithms.

The second part is a realtime monitoring system that can be used for monitoring the behavior of Android applications and alerting users of applications that violate a predefined security policy by trying to access private information such as GPS locations and SMS related information. In order to achieve this, we use a loadable kernel module for tracking the kernel level system calls. As shown in Figure 5, the proposed monitoring system is

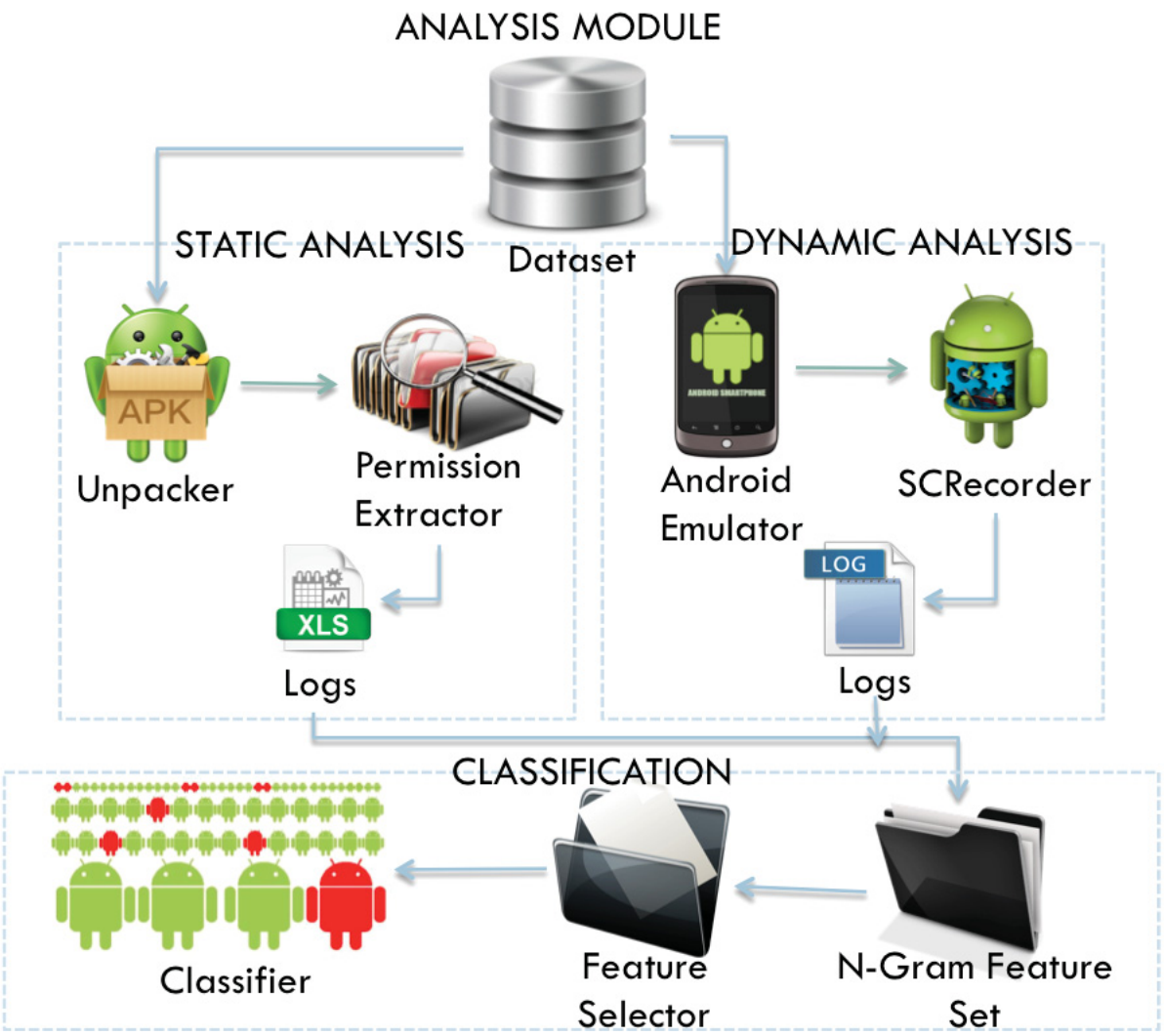


Figure 4: Main components of the off-line analysis system

composed of three modules. The first module is a configuration module which allows the Android device to be configured for enabling the monitoring of system calls. The second one is a kernel tracker which hooks the system calls on the Android device. Finally, the notifier module provides real-time notifications to the users in case of suspected malicious activities.

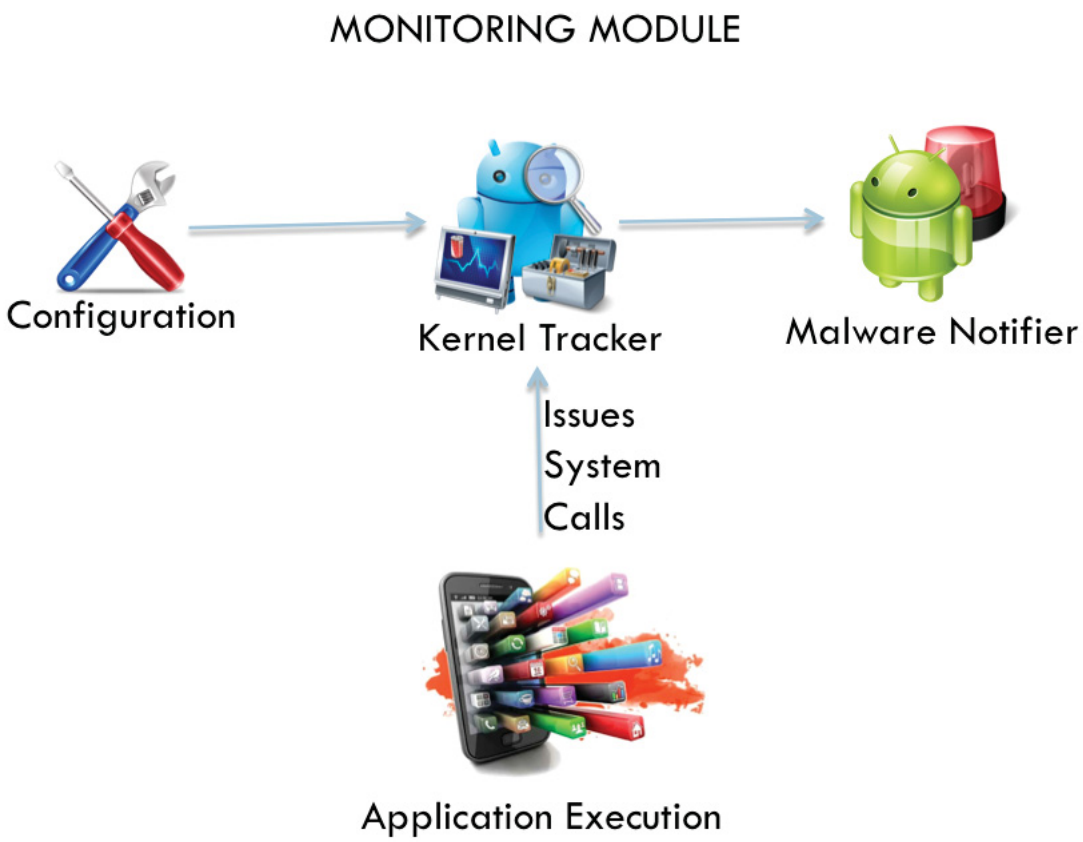


Figure 5: Main components of the monitoring system

3.2 System Analysis

As mentioned in the previous section, our off-line Android analysis tool is composed of three modules:

1. Static analysis module
2. Dynamic analysis module
3. Classification module

The static analysis module and dynamic analysis module generate different logs that are used to extract the features used by the classification module for determining whether the

tested application is malicious or benign. The complete working of each of these modules is explained in the following sections.

3.2.1 Static Analysis Module

Static analysis proved to be a very useful tool in identifying potential malicious behavior of Android applications. It aims to examine the Android applications without executing them, which makes this approach an attractive option in many malware analysis scenarios. The main rational of the static analysis of Android applications is that, in order to access any system related resources on the Android device, the Android application needs to specify all associated permissions in its *AndroidManifest* file. Examining these permission, such as BLUETOOH, SEND_SMS, RECEIVE_SMS, CALL_PHONE, READ_CONTACTS, WIPE_DATA, BIND_DEVICE_ADMIN and CAMERA, implicitly reveals some of the intended functionality of the application. For instance, a typical gaming application should have no interest in requesting CAMERA permission.

As shown in Figure 6, all the applications to be analyzed are stored in a database. Just as there exists MSI packages in Windows, Android applications are packed in Application Package Kit (APK) files for installing them on the Android devices. The APK is similar to a ZIP compressed package consisting of compiled Linux libraries, application resources (such as icons, images , fonts, sound clips and configuration files), bytecode in '*classes.dex*' file and an XML config file named 'AndroidManifest'. The *AndroidManifest* file contains all the application activities and permissions. While, the '*classes.dex*' file consists of the application bytecode. Any modification or change in these two files can lead to altering the behavior of the application. When performing the static analysis, the application to be analyzed is automatically fetched from this dataset. Then, the application APK is unpacked using the *apktool*, which is a third party reverse engineering tool used to decode

the resources of the APK package. After regenerating the *AndroidManifest* file included in the APK package file, the list of permissions, services, broadcast receivers and activities that are specified in it are extracted and stored for later processing and analysis by the classification module.

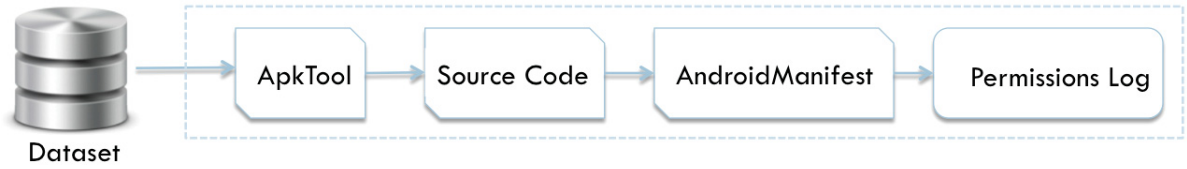


Figure 6: Static analysis module

3.2.2 Dynamic Analysis Module

The main advantage of the static analysis described above is that it can be performed relatively very efficiently without the need to execute the applications and hence avoids any risk associated with executing malicious applications. On the other hand, some malware writers use different obfuscation and cryptographic techniques that make it very hard for static analysis techniques to obtain useful information and hence, it is often essential to use dynamic analysis.

Dynamic analysis is most widely used to examine the behavior and interactions of an application with the operating system. The dynamic analysis process is carried out using a virtual controlled environment in order to reduce the risk associated with executing malware applications on actual Android devices. Throughout our dynamic analysis process, the application is installed and executed on the Android emulator, which is an application which provides a platform such as a virtual mobile phone for running Android applications. It constitute a full Android system software stack, down to the kernel. The emulator

also has a series of preinstalled system applications such as Dialer, Message and Browser, and comes with up to date versions of the Android system that can run by configuring the Android Virtual Device (AVD).

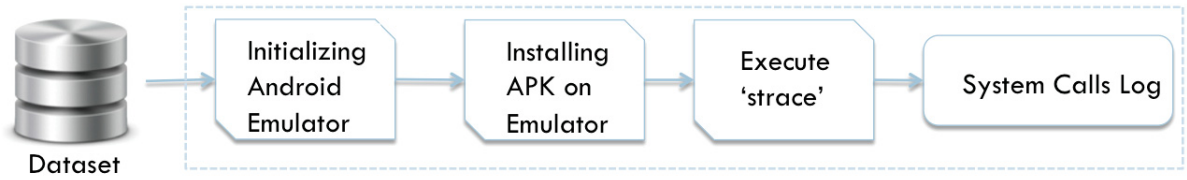


Figure 7: Dynamic analysis module

Figure 7 depicts our dynamic analysis process. First the application to be analyzed is fed to the Android emulator. Then the *strace* tool is started to hook system calls on the 'zygote' process. Now, the 'zygote' process is the root process in the Android operating system and hence it records all the system calls invoked by the application and saves them onto the system calls Log file for later processing by the analysis module. It should be noted that while Android has of tool called 'logcat' that can be used for tracking and debugging applications, for our purpose, logcat gives very little information about the application and also not all the application logs are captured by it. On the other hand, *strace* records all the application related system calls and any kind of signals it receives. Also, there is no need to modify the application in order to be able to support *strace*.

The steps involved in the process of our dynamic analysis are summarized as follows:

1. The application to be analyzed is pulled from the analysis dataset.

2. The emulator is launched with Android version 4.2.2 (API level 16).

3. The application is installed on the Android emulator using ADB (Android Debug Bridge).

4. The *strace* command is executed for hooking the system calls on the Android emulator.

5. The main activity of the application is launched for execution using the MonkeyRunner tool.

6. The *strace* command is kept running for a pre-specified interval of time (default is 1 minute for our analysis).

7. All the system calls invoked by the application during its execution is saved to system call logs.

Finally, the system call logs extracted by the dynamic analysis module along with the permissions logs extracted from the static analysis module are fed to the classification module for further analysis.

3.2.3 Classification Module

Various forms of n-gram analysis have been previously utilized for malware detection on Unix/Linux and Windows platforms. In our system, the n-gram analysis of the Android application is performed by using the system calls logs collected from the above mentioned modules. As there exists a large number of extracted n-gram and permission features, Classwise Document Frequency (CDF) is then used for reducing the feature space and selecting the most useful features that can be used to distinguish malware applications. After sorting the extracted features based on their CDF values, the top k features are used as input to different classifiers.

The effectiveness of the proposed approach is confirmed by testing it on popular applications collected from F-Droid, and malware samples obtained from a third party and the

Android Malware Genome Project dataset. The obtained classification results are presented in the next chapter.

3.3 Monitoring System

In the above off-line security analysis of Android applications, the kernel level system calls proved to be an important feature for detecting suspicious activities associated with malware applications. In this section, we extend our work and also using kernel level system calls, we present a realtime monitoring system that can be used for monitoring the behavior of Android applications and alerting users to applications that violate a predefined security policy by trying to access private information such as GPS locations and SMS related information. Our system traces kernel system calls such as open() and socket() related to accessing personal SMS and GPS location, respectively, as they can potentially leak private information. In order to achieve this, we use a loadable kernel module (LKM) for tracking kernel level system calls. In particular, we use a modified goldfish kernel for the Android device. The monitoring application, which also needs to be installed on the Android device, notifies the user with any such malicious activity. Figure 8 shows the complete flow of the developed monitoring system.

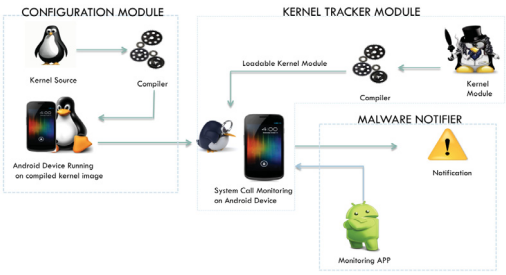


Figure 8: Monitoring system

The steps involved in the development and running of our monitoring system can be summarized as follows:

- The kernel source is obtained from Android Open Source Project (AOSP).
- The configuration file of the the kernel is edited and compiled for the ARM architecture.
- A ‘zImage’ kernel image is produced after successful compilation.
- The Android device is booted using this ‘zImage’ kernel.
- The kernel module program is configured to track specified system calls, namely open() and socket() in our case.
- The kernel module program is compiled to obtain loadable kernel module(LKM) which needs to run on the Android device.
- The monitoring application that checks the kernel logs is installed and run of the device.

In what follows, we explain the above steps in some more details. Android devices come with a precompiled kernel source and hence it is not possible to run loadable kernel module onto these devices. Hence, we need to make changes in the kernel source in order to fulfill our needs. There are many kernel sources available on Android Open Source Project. Some of the kernel sources are for actual mobile devices, but for testing purposes, we used ‘goldfish’ kernel.

Once the goldfish kernel is obtained, we need to change the default configuration of the kernel source in order to run loadable our kernel module tracker. Editing the config file present in the downloaded goldfish kernel source helps us achieve this. After changing the configuration file, the compilation has to be performed in order to reflect these changes. Since we want this kernel to run on Android devices and not on Linux operating system

and since the Android device also runs on ARM architecture, we have to compile using the "ARM EABI cross toolchain" compiler.

After successful compilation, a compiled kernel image "zImage" is produced. In addition, System.map file is generated. This file contains the address of system calls, i.e., mapping between symbolic names and the associated addresses in memory. Each symbolic name can be the name of a variable or the name of a function. We can use this for hooking system calls.

The loadable kernel module (LKM) is an object file that contains code to extend the running kernel, or so-called base kernel, of an operating system. LKMs are typically used to add support for new hardware and/or file systems, or for adding system calls. When the functionality provided by a LKM is no longer required, it can be unloaded in order to free memory and other resources. We use kernel module programs for tracking system calls. Our aim is to track system calls such as open() and socket() as they are responsible for accessing SMS and GPS location, respectively. Also, in order to track system calls, we need the address of the system call table, which is obtained at the time of compiling goldfish kernel as mentioned above. Hence, using we use the system call table address to point to our kernel module program.

Again, we have to compile the kernel module program using "Cross Compiler arm eabi toolchain" because we want to run on Android device running on ARM architecture. After successful compilation, we get loadable kernel module. This LKM is now ready to be inserted in the device. For this, "insmod" command is used for installing loadable kernel module on the Android device.

After the above steps are performed, we have the Android device running with our loadable kernel module. We build an application that notifies the user in case any application

tries to access SMS or GPS location. For testing purposes, we performed this experiment on the Android emulator and installed our monitoring application using the ADB (Android Debug Bridge) command.

The service running in our monitoring application keeps track of any malicious activity such as accessing SMS or GPS locations. An Android service is an application component capable of performing operations for a long time. A service can be of two categories.

1. Started Service: A service is referred to as 'started' when any of the application component starts it by calling its method using startService(). This service continues to run in the background though the component responsible for starting the service is destroyed. The start service executes the operation and does not return back to the caller. For instance, consider a service that uploads or downloads some content from a server. It continues to perform the operations even though the user is playing music.

2. Bounded Service: A service is referred to as 'bounded' when any of the application component tries to bind the service by issuing a method named bindService(). The service provides a user with a graphical interface allowing that component to give commands to the service such as sending request, getting results and also communicating with the process through inter-process communication. The bounded service has a lifetime associated with it. As soon as the application component bounded to the service is destroyed, the service also gets destroyed along with it. More than one application component can be attached with the bound service but when any of those components unbind the service, the service also gets destroyed.

Since in our our monitoring application we want our service to run indefinitely, we make use of ‘started’ service.

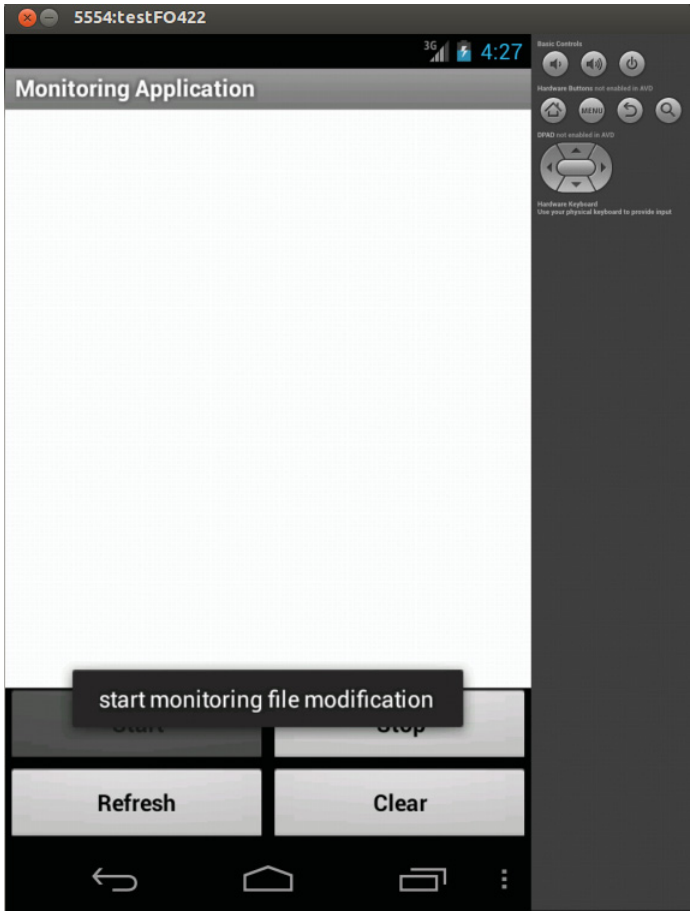
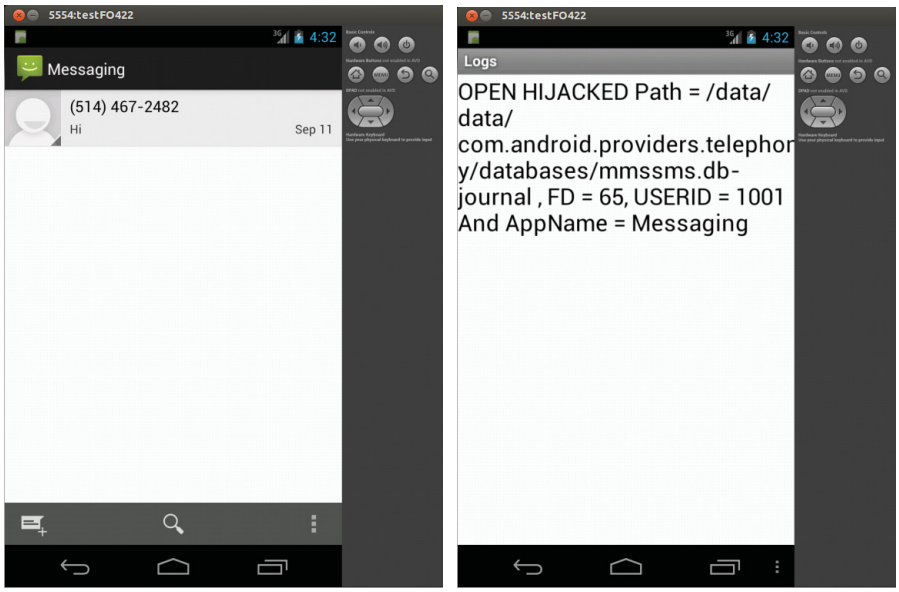


Figure 9: A snapshot of the monitoring application

Figure 9 shows the main GUI of our monitoring application. It has four options. First one is ‘start’. This will start the service of monitoring system calls. Second one is ‘stop’ which stops the service of monitoring system calls. The ‘refresh’ option updates the contents of the kernel logs (without clicking on the notification) and ‘clear’ clears all kernel logs of all previous activities.

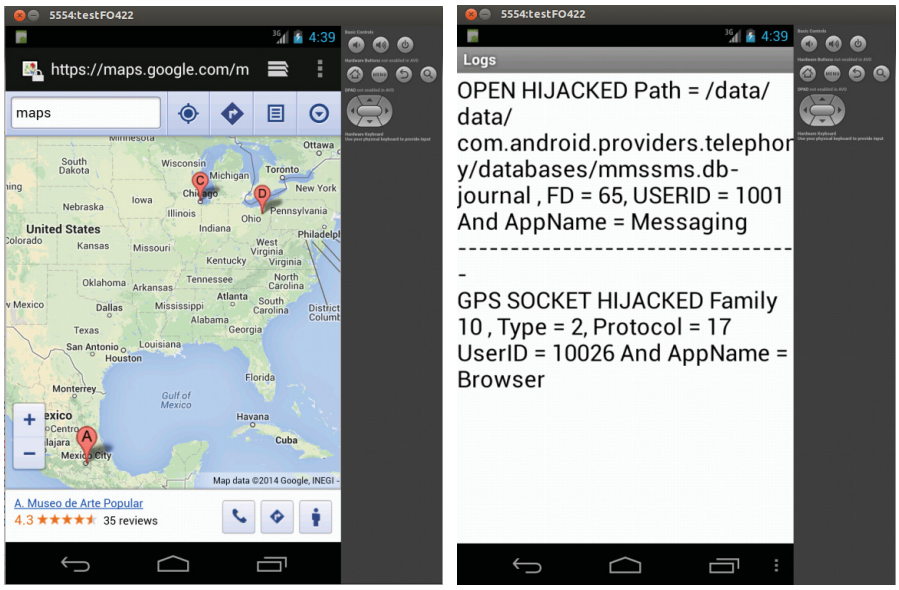


(a) Application accessing SMS

(b) Notification Log for SMS open() system call

Figure 10: A snapshot of monitoring application while accessing SMS Messages

Once the monitoring application is installed, it starts monitoring other applications. In case our security policy is breached, the monitoring application creates a notification alert. On viewing the notification, the user gets to know about the malicious activity. Figure 10 and Figure 11 show snapshots for for SMS open() system call, and GPS socket() system call notification logs.



(a) Application Accessing GPS Location (b) Notification Log for GPS socket() system call

Figure 11: A snapshot of monitoring application while accessing GPS Locations

Chapter 4

Experimental Results

The experimental results presented in this section are obtained by analyzing a total of 1932 Android applications, out of which 970 are benign and 962 are malicious. We collected the malicious samples from the Android Malware Genome Project and from another third party. The benign samples were obtained from F-Droid which is a Free and Open Source Software (FOSS) repository for Android applications. We also verified the applications collected from F-Droid are benign using VirusTotal.

4.1 Frequent System Calls and Permissions

Our first experiment focuses on identifying the frequent permissions and frequent kernel level system calls to check their applicability as distinguishing features amongst malicious and benign Android applications.

Figure 12 shows the top 15 permissions used by the analyzed malicious applications and their frequency as compared to the benign ones. From the figure, it is clear that some permissions are more often used by malicious applications. For instance, the READ_PHONE_STATE permission is used by approximately 86% of the malicious applications

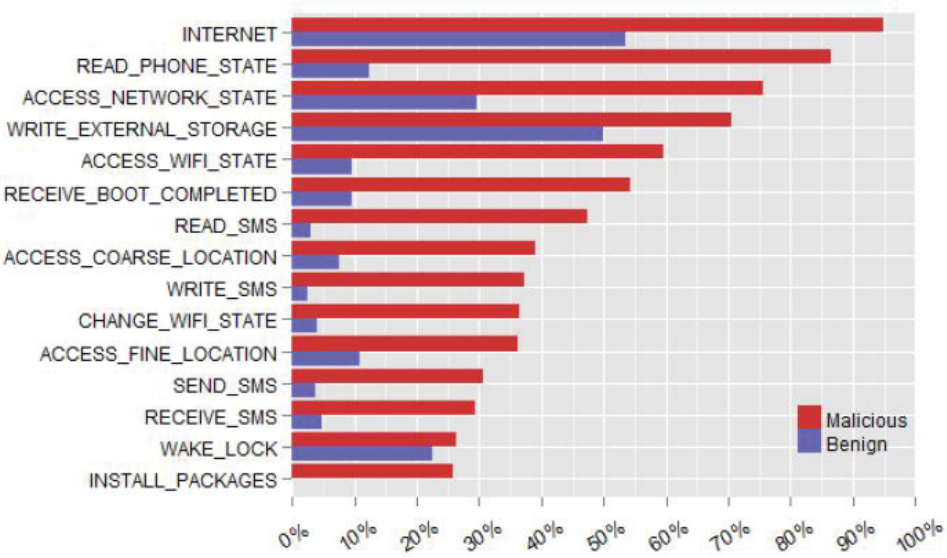


Figure 12: Top 15 permissions used by malicious applications

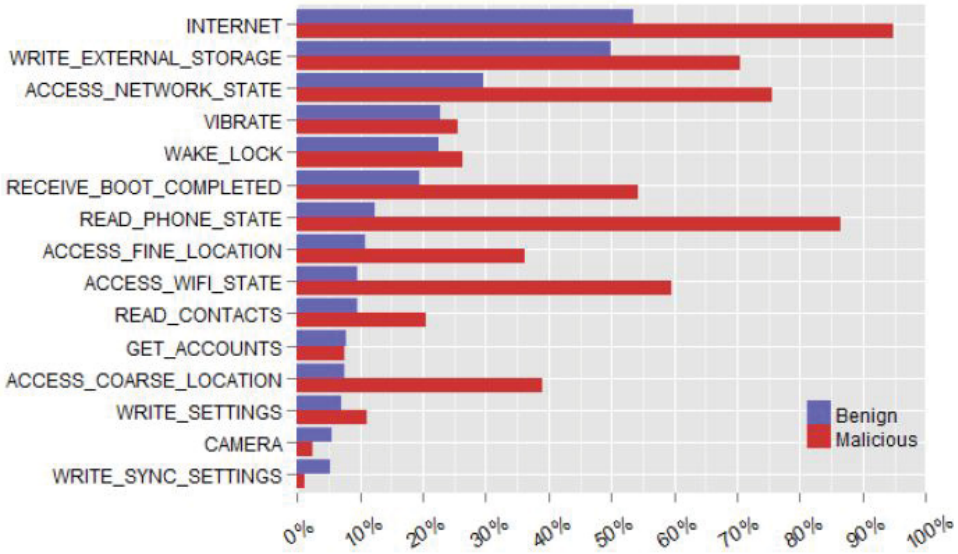


Figure 13: Top 15 permissions used by benign applications

as compared to about 12% of the benign applications. This permission is usually used to steal system related sensitive information such as the operating system version, phone make and model, SIM serial number, IMEI and IMSI. Other permissions which are used by a large portion of malicious applications include INTERNET, ACCESS_NETWORK_-STATE, ACCESS_WIFI_STATE, WRITE_SMS and READ_SMS. Figure 13 shows the top 15 permissions used by the analyzed benign applications and their frequency as compared to the analyzed malicious applications.

We also obtained the top system calls used by malicious and benign applications. Some system calls such as OPEN (related to opening a file), CLOSE (related to close a file), GETID (related to application ID) and GETPID (related to process ID) are very common and are very likely to be issued by all applications, whether they are malicious or benign. Hence, they do not prove to be useful in distinguishing between malicious and benign applications. On the other hand, system calls such as SOCKET (related to opening a socket for network connection), GETSOCKETOPT (related to getting socket options), CONNECT (related to connect via socket), BIND (related to bind the ip address and the connected server) and SENDTO (related to sending information to the server) are observed in a large percentage of malicious applications as depicted in Figure 14. These system calls aim to steal users' information and send to it via network connections. Figure 15 shows the corresponding top 10 system calls issued by the analyzed benign applications. Note that both Figure 14 and Figure 15 do not show the non-distinguishing system calls, i.e., the ones that appeared in all the analyzed malicious and benign applications, such as OPEN, CLOSE, GETID and GETPID.

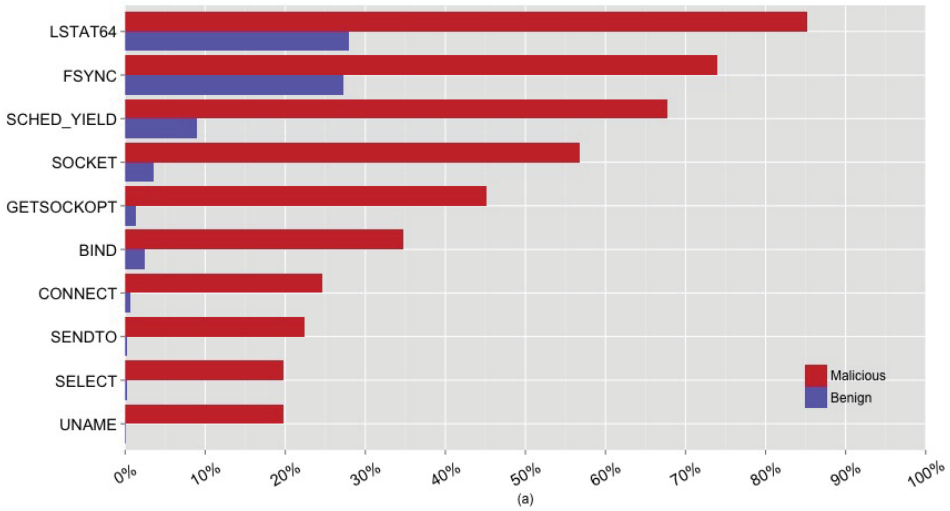


Figure 14: Top 10 distinguishing system calls observed in malicious applications

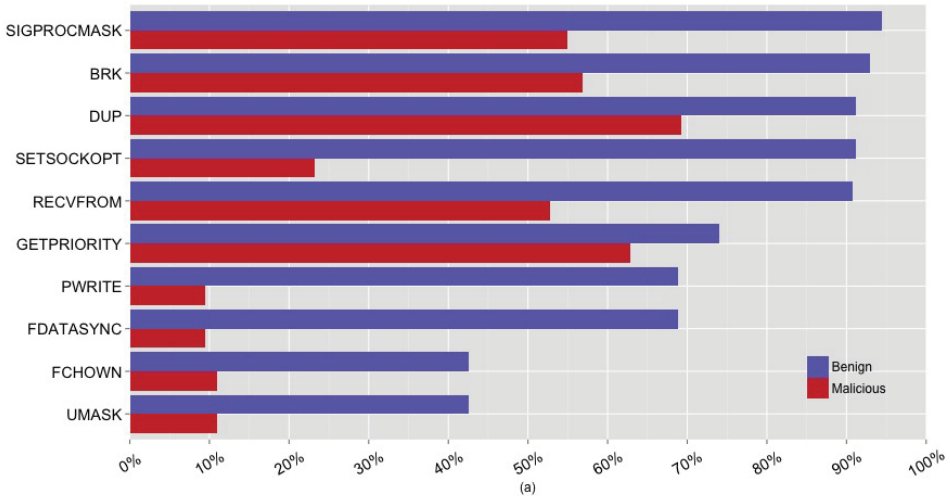


Figure 15: Top 10 distinguishing system calls observed in benign applications

4.2 Malware Detection

The experimental results presented in this section are obtained by feeding the features extracted from both the static and dynamic analysis modules described in Chapter 3 to a variety of classification algorithms supported by Weka for classification with 10-fold cross-validation (The Waikato Environment for Knowledge Analysis (Weka) is a popular suite of machine learning software). In what follows, we provide a brief overview of the classifiers used throughout our experiment. For further details on the theoretical foundations of these classifiers, the reader is referred to [39].

1. *Decorate*: DECORATE is a meta-learner for building diverse ensembles of classifiers by using specially constructed artificial training examples. Comprehensive experiments have demonstrated that this technique is consistently more accurate than the base classifier, Bagging and Random Forests. Decorate also obtains higher accuracy than Boosting on small training sets, and achieves comparable performance on larger training sets.

2. *SMO*: SMO is one type of implementation of the Sequential Minimal Optimization algorithm. It is used for training a support vector classifier and uses kernel functions such as polynomial or Gaussian kernels. In this algorithm, unfilled values are filled globally, nominal attributes are exchanged into binary attributes and all the attributes are normalized. In this, the output coefficients are based on this normalized data. Anyone can switch off the normalization or can set the input to zero mean and one variance. Also, in order to support multiclass, pairwise classification can be used.

3. *RotationForest*: RotationForest is one of the implementations of the rotation forest ensemble. This algorithm makes use of random subspaces and primary components in order to generate an ensemble of decisions trees. Weka's implementation allows performing classification using any base classifiers. The conversion of the primary

component is done using Weka's filter of that same implementation. It can also be used to configure different projections such as random projections or partial least squares. Other parameters instruct the algorithm for controlling the size of the subspaces and the number of input to the filter.

4. *DMNBtext*: DMNBtext is another type of Naive Bayes scheme. It gains the knowledge of multinomial Naive Bayes in both generative and discriminative way. Default parameters of a simple Bayesian network are the frequency counts calculated from the conditional probability tables obtained from the training set, therefore maximizing the likelihood of the data model. On the other hand, the parameters used for maximizing the accuracy are found in classification settings. DMNBtext inserts a different element into parameter learning by looking at the current classifier's predictions for a training data prior modifying the frequency counts. At the time of processing a single training instance, the frequency counts are increased by the value of one minus the predicted probability of its class value. It also allows us to indicate the number of iterations over the training sample and whether to consider word frequency information where it learns the simple Bayes model contrast to the multinomial one.

5. *LMT*: Logistic Model Tree(LMT) builds a model tree based on logistic. It can handle both binary and multiclass target parameters, numeric and nominal parameters and missing variables. While adjusting the logistic regression functions at a node by the LogitBoost algorithm, it makes use of cross validation for indicating the number of iterations to be performed for running just once and uses this information throughout the tree rather cross validating at each and every node. This can help improving the runtime considerably without any effect on the accuracy. There is a parameter for setting the number of boosting operations to be performed throughout the entire process of making a tree. In order to improve runtime, weigh trimming might be

used. LMT also deploys the minimal cost complexity mechanism for generating a compact tree model.

4.2.1 Classification using n-gram of system calls

In this experiment, we used total of 1000 applications, from which 500 were malicious and the other 500 were benign. During the dynamic analysis interval for these applications, 100 unique system calls were observed. Table 1 shows the classifications results obtained when using unigram (n=1) features consisting of individual system call frequencies, bigram (n=2) features consisting of pairs of system calls frequencies, and when using a combination both the unigram and bigram features. From the Table, the best accuracy, 91.3%, is obtained when using the Decorate classification algorithm. In the following section, we will examine the effect of adding features obtained from the static analysis module in order to improve the classification accuracy.

Algorithm Name	Accuracy in %		
	Unigram	Bigram	Unigram and Bigram
Decorate	84.9	91.3	91.3
SMO	78	90	90
RandomForest	85	90.2	90.2
DMNBtext	71.2	86.3	86.3
LMT	82.1	89.3	89.3

Table 1: System calls based classification results

4.2.2 Classification using n-gram of system calls, and static permissions

In this experiment, we utilized the static permission features along with the system call feature set. In particular, we added static permission features to the unigram and bigram features obtained from the 100 observed system calls. The obtained results are shown in

Algorithm Name	Accuracy in %		
	Unigram & Permissions	Bigram & Permissions	Three Combined
Decorate	94.6	95.7	96
SMO	94.4	95.6	95.6
RandomForest	95.4	94.9	95.6
DMNBtext	91.4	95.1	95.1
LMT	94.6	94.8	94.9

Table 2: Combined system calls and permissions based classification results

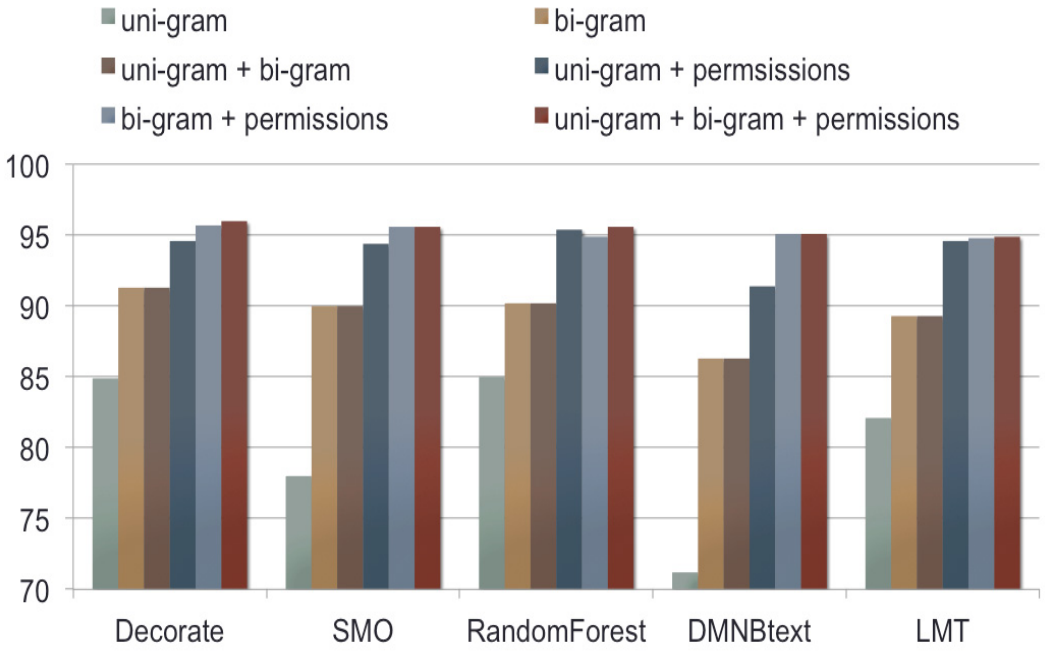


Figure 16: Malware Analysis Results

Table 2. As depicted in the table, the use of such extended feature vectors improved the accuracy to 96% and 95.6% using Decorate and SMO classification algorithms, respectively.

Figure 16 provides a pictorial illustration for the results presented in Table 1 and Table 2. It should be noted that the results obtained in this section should be interpreted with care. In particular, these results should not be used to favour one classification algorithm over another since the difference in the classification accuracy is not statiticaly significant and it might have resulted because of the limited characteristics of the specific analyzed samples.

One of the main disadvantages of dynamic analysis is that the analysis of each application takes a relatively long time (about 5-6 minutes including the time required to refresh the virtual machine environment), which is not practical when analyzing thousands/millions of applications. On the other hand, malware writers use obfuscation tools to limit the effectiveness of static signature based detection. In the following section, we present some preliminary results on automatic clustering of obfuscated applications. If we were able to achieve a good accuracy on this process, then one would perform the clustering process and then perform the detailed dynamic analysis only on some representative sample(s) of the application. Unfortunately, as will be illustrated by the results obtained in the next section, this does not seem to be an easy task.

4.3 Clustering of Obfuscated Applications

In this experiment, we created obfuscated version of 10 Android applications using a third party software named 'DexProtector'. DexProtector is designed for providing comprehensive protection of Android applications against reverse engineering and it comes with following main features:

- Encryption of strings
- Encryption of classes
- Encryption of resources and assets
- Hiding of method calls

Encryption of strings is usually used for hiding the string constants in the Android applications. DexProtector uses AES encryption with dynamic keys that are generated based on various parameters and they can only be extracted from the application code. Therefore,

if anyone decompiles the application code, all the contents of the string constants will be hidden and hence no one will be able to access it for any static analysis and decryption process.

Encryption of classes is one of the fastest and easiest methods of application protection. The classes protected by this function are encrypted and moved from classes.dex file, which is the common storage for all the classes present in the application. Thus, they are now hidden for any decompiling software such as apktool and dex2jar.

Encryption of resources/assets allows the protection of all the application resources from being modified or copied. In this case, the source code remains the same but the resources used by it is encrypted. This is important in the case where the resources need to be protected.

Hiding of method calls protects the most critical locations from where the application can be easily modified or analyzed. This method covers the calls of the library methods used in the application and application methods used with the variable functions structured in a special way. This mechanism has the disadvantage that it may lead to lowering the performance of the application.

Figure 17 depicts a snapshot of the DexProtector software. As it can be seen from the figure, DexProtector has four protection options available: string encryption, class encryption, resource encryption and hide access. Using these four parameters, we obtained 15 different obfuscated versions of each Android application. Thus for the 10 different applications we used in this experiment, we have a total of 160 applications, 10 corresponding to the original applications and 150 corresponding to the obfuscated ones.

For each one of these files, we extracted both unigram and bigram statistics from the corresponding classes.dex file which holds all of application bytecode. Then we used these

n-gram features to perform automatic clustering of these applications using several clustering algorithms including: SIB, Farthest First, Hierarchical Clusterer, EM and X-Means. Some basic information about these algorithms is given below.

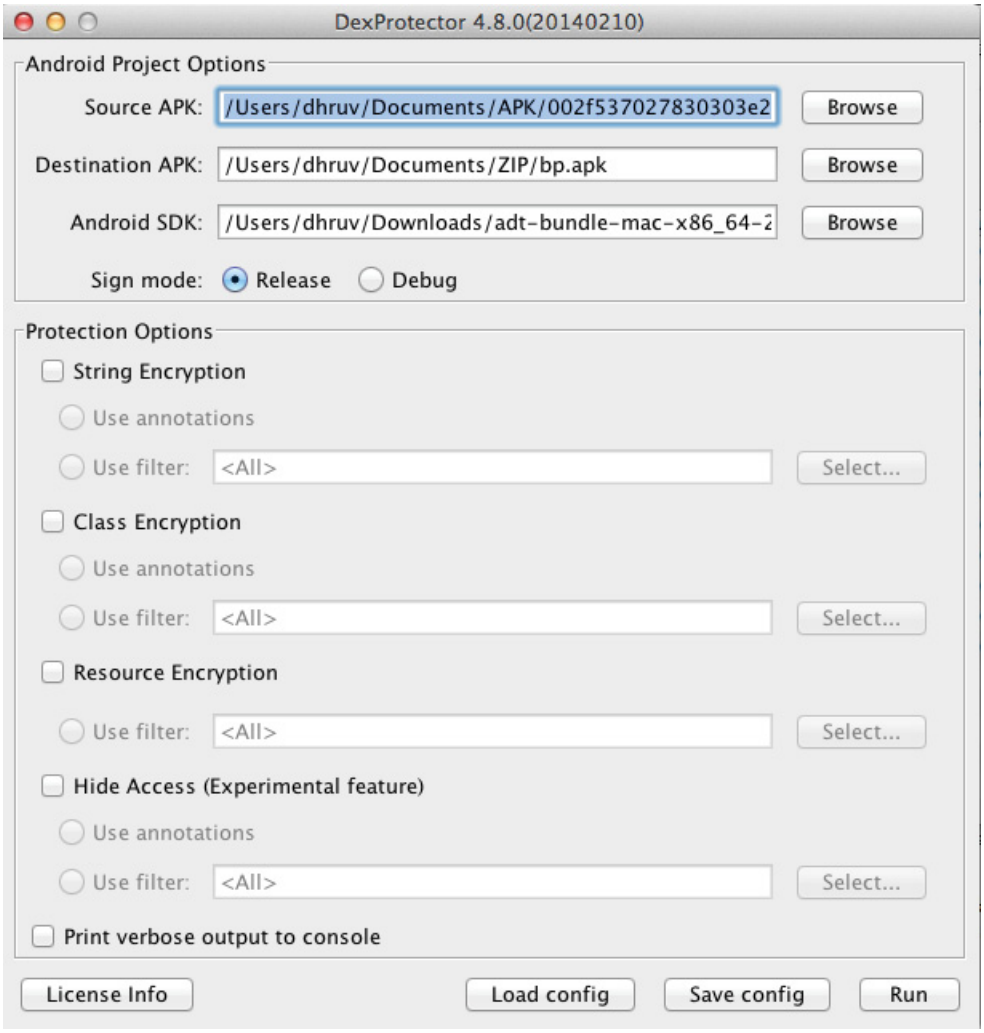


Figure 17: Snapshot of DexProtector

1. *sIB*: Clustering data using the sequential information bottleneck (sIB) algorithm [12] associates each instance with a cluster having the minimum distance to that instance. The main idea of this algorithm is to extract the important aspects of the data that was previously done using information bottleneck by clustering one instance, meanwhile storing information about the another instance.

2. *FarthestFirst*: FarthestFirst [39] is one of the clustering algorithms used very often. It is based on simple KMeans algorithm model. In FarthestFirst algorithm, position of each cluster center in turn is placed at the furthest location from the currently existing cluster. This point should lie in the data area. This helps speed up the clustering process as lower number of re-assignment and adjustment is required.

3. *HierarchicalClusterer*: In Hierarchical [39] clustering technique, a hierarchy of clusters is built. In order to decide which clusters can be combined or where the clusters can be splitted, a dissimilarity score between the sets of observations is necessary. In hierarchical clustering, this can be achieved by an appropriate use of metric and a linkage criterion. A metric is a measure of distance between the pairs of observations. A linkage criterion identifies the dissimilarity of sets as a function of pairwise distances of observations in the sets.

4. *EM*: Expectation-Maximization (EM) [33] is an iterative method for finding the maximum likelihood estimate of the parameter in statistical models. In EM, the iteration alternates from performing an Expectation (E step) for performing the expectation of the log likelihood calculated using the current value of the estimated parameters and Maximization (M step) for computing the parameters maximizing the expected log likelihood found on the previous E step. The values of the parameters obtained from the M step is again used in E step and continues until the distance between expected and calculated value of parameter is same.

5. *XMeans*:Cluster data using X-Means [12] algorithm. X-Means is an extension to K-Means algorithm by an Improve-Structure part. In this part of algorithm, the cluster centres are attempted to split in its appropriate region. The selection of the children of each center and itself is done based on comparison of the Bayesian Information Criteria (BIC) of the two structures. In addition, it also uses BIC for deciding the number

of clusters. Also, the user can select the distance function to be used, the minimum and maximum number of clusters to be considered and the maximum number of repetitions to be performed.

Algorithm Name	Accuracy in %
sIB	68.75
FarthestFirst	68.125
HierarchicalClusterer	68.125
EM	66.25
XMeans	64.375

Table 3: Clustering results

Table 3 shows the accuracy of correctly classified applications based on the bigram features of 160 applications. The best obtained accuracy, 68.75%, is obtained when using the sIB algorithm. Table 4 shows the corresponding confusion matrix. Clearly this poor accuracy (which corresponds to our best obtained results) shows that better features/approaches are needed in order to accurately perform the task of automatic clustering of obfuscated Android applications and hence for the time being, it seems that the lengthy dynamic analysis process has to be performed for each individual Android application.

	Cluster1	Cluster2	Cluster3	Cluster4	Cluster5	Cluster6	Cluster7	Cluster8	Cluster9	Cluster10
App1	2	-	-	2	-	-	-	-	6	6
App2	-	16	-	-	-	-	-	-	-	-
App3	-	-	16	-	-	-	-	-	-	-
App4	2	-	-	2	-	-	-	-	6	6
App5	-	-	-	-	16	-	-	-	-	-
App6	-	-	-	-	-	16	-	-	-	-
App7	-	-	-	-	-	-	16	-	-	-
App8	-	-	-	-	-	-	-	16	-	-
App9	2	-	-	2	-	-	-	-	6	6
App10	2	-	-	2	-	-	-	-	6	6

Table 4: Confusion matrix when using the sIB algorithm

Chapter 5

Conclusions and Future Work

5.1 Conclusion

The abrupt increase in the use of Android powered devices has led to the creation and dissemination of a huge number of Android malware. In this thesis, we proposed a framework to perform off-line analysis of Android applications using both static and dynamic analysis approaches. The effectiveness of the proposed approach was confirmed by testing it on popular applications collected from F-Droid, and malware samples obtained from a third party and the Android Malware Genome Project dataset. The obtained experimental results show that the static permissions requested by the application and the kernel level system calls are good features for distinguishing malicious applications from benign ones. We also developed a prototype that can be used to provide realtime monitoring for the behavior of Android applications and alert users to these applications that violate a predefined security policy. We also investigated the possibility of automatic clustering of obfuscated Android applications based on n-gram analysis of its classes.dex files which hold the applications' bytecode. The obtained experimental results confirmed the difficulty of this approach.

5.2 Future Work

There are few limitations with our current work that need to be addressed in our future research and development. In what follows, we summarize some of these weaknesses and limitations:

- Our off-line analysis system requires loadable kernel modules to be loaded inside the Android emulator and the modified kernel becomes somewhat overloaded with tracking system calls which slows down the overall system performance. Optimizing the speed of this system call tracking module is a very interesting and challenging implementation project.

- Throughout our dynamic analysis process, we do not interact with the applications for creating carefully chosen gestures. Consequently, there is no guarantee that we check complete paths that can be traversed by the application or even a good portion of it. For future work, one may try to perform random and carefully chosen events using some automated tools such as the monkeyrunner, which comes with the Android SDK.

- The usability of the developed realtime monitoring tool was not tested. Balancing security and usability for such tools requires a formal usability study among a large set of users.

- Our approach for automatic clustering of obfuscated Android applications resulted in a very poor accuracy. Investigating better features that can be used to accurately perform the task of automatic clustering of obfuscated Android applications without performing any dynamic analysis is a challenging research project.

Bibliography

[1] Android. Android Application Fundamentals. [Online]. Available: http://developer.android.com/guide/components/fundamentals.html.

[2] Android. Android Architecture Platform. [Online]. Available: http://androidmentor.com/system/24/android-platform-architecture.html.

[3] Android. Android Operating System. [Online]. Available: http://en.wikipedia.org/wiki/Android_(operating_system).

[4] Android. Android Security Overview. [Online]. Available: https://source.android.com/devices/tech/security/.

[5] Android. Security Enhancements. [Online]. Available: https://source.android.com/devices/tech/security/enhancements.html.

[6] Android. Validatin Security-Enhanced Linux in Android. [Online]. Available: https://source.android.com/devices/tech/security/se-linux.html.

[7] M. Ballano. Android threats getting steamy. [Online]. Available: http://www.symantec.com/connect/blogs/android-threats-getting-steamy.

[8] A. Bauer, J.-C. Küster, and G. Vegliach. Runtime verification meets Android security. In *NASA Formal Methods*, pages 174–180. Springer, 2012.

[9] T. Blasing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak. An Android application sandbox system for suspicious software detection. In *5th international conference on Malicious and unwanted software (MALWARE), 2010*, pages 55–62. IEEE, 2010.

[10] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new Android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011.

[11] 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*, pages 15–26. ACM, 2011.

[12] G. Chechik, A. Globerson, N. Tishby, and Y. Weiss. Information bottleneck for gaussian variables. In *Journal of Machine Learning Research*, pages 165–188, 2005.

[13] 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 real-time privacy monitoring on smartphones. *Communications of the ACM*, 57(3):99–106, 2014.

[14] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[15] Google. Google play. [Online]. Available: http://en.wikipedia.org/wiki/Google_Play.

[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 (CIS), 2011*, pages 1011–1015. IEEE, 2011.

[17] Y.-D. Lin, Y.-C. Lai, C.-H. Chen, and H.-C. Tsai. Identifying Android malicious repackaged applications by thread-grained system call sequences. *Computers & Security*, 39:340–350, 2013.

[18] Lookout. Android malware droiddream: How it works. [Online]. Available: https://blog.lookout.com/blog/2011/03/02/android-malware-droiddream-how-it-works/.

[19] Lookout. New malware found in alternative android markets: Droidkungfu. [Online]. Available: https://blog.lookout.com/blog/2011/06/06/security-alert-new-malware-found-in-alternative-android-markets-legacy/.

[20] Lookout. Security alert: Android trojan ggtracker charges premium rate sms messages. [Online]. Available: https://blog.lookout.com/blog/2011/06/20/security-alert-android-trojan-ggtracker-charges-victims-premium-rate-sms-messages/.

[21] Lookout. Security alert: Droiddream malware found in official Android market. [Online]. Available: https://blog.lookout.com/blog/2011/03/01/security-alert-malware-found-in-official-android-market-droiddream/.

[22] Lookout. Security alert: Geinimi, sophisticated new Android trojan found in wild. [Online]. Available: https://blog.lookout.com/blog/2010/12/29/geinimi_trojan/.

[23] L. X. Min and Q. H. Cao. Runtime-based behavior dynamic analysis system for Android malware detection. *Advanced Materials Research*, 756:2220–2225, 2013.

[24] MobileIron. Android Mobile Security - The Mobileiron Layered Security Model. [Online]. Available: http://www.mobileiron.com/en/smartwork-blog/mobile-security-mobileiron-layered-security-model.

[25] M. Nauman, S. Khan, and X. Zhang. Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM*

Symposium on Information, Computer and Communications Security, pages 328–332. ACM, 2010.

[26] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. *Security and Communication Networks*, 5(6):658–673, 2012.

[27] P. Paganini. ibanking mobile trojan exploitation on Android platform. [Online]. Available: http://securityaffairs.co/wordpress/24069/malware/ibanking-trojan-targets-facebook.html.

[28] S. M. Patterson. Contrary to what you've heard, Android is almost impenetrable to malware. [Online]. Available: http://qz.com/131436/contrary-to-what-youve-heard-android-is-almost-impenetrable-to-malware/.

[29] É. Payet and F. Spoto. Static analysis of Android programs. *Information and Software Technology*, 54(11):1192–1201, 2012.

[30] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct Android malware behaviors. *EuroSec, April*, 2013.

[31] G. Russello, B. Crispo, E. Fernandes, and Y. Zhauniarovich. Yaase: Yet another Android security extension. In *IEEE Third International conference on Privacy, security, risk and trust (PASSAT), 2011*, pages 1033–1040. IEEE, 2011.

[32] G. Russello, A. B. Jimenez, H. Naderi, and W. van der Mark. Poster: Firedroid: Hardening security in Android with system call interposition. In *IEEE Symposium on Security & Privacy*, 2013.

[33] N. Sharma, A. Bajpai, and M. R. Litoriya. Comparison the various clustering algorithms of weka tools. *facilities*, 4:7, 2012.

[34] R. Strohmeyer. Notorious malware attacks. [Online]. Available: http://www.informationweek.com/mobile/8-notorious-android-malware-attacks/d/d-id/1099385?

[35] Symantec. Android.bgserv. [Online]. Available: http://www.symantec.com/security_response/writeup.jsp?docid=2011-031005-2918-99&tabid=2.

[36] Symantec. Android.bgserv found on fake google security patch. [Online]. Available: http://www.symantec.com/connect/blogs/androidbgserv-found-fake-google-security-patch-part-ii.

[37] Symantec. Android.geinimi. [Online]. Available: http://www.symantec.com/security_response/writeup.jsp?docid=2011-010111-5403-99&tabid=2.

[38] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: multi-layer profiling of Android applications. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 137–148. ACM, 2012.

[39] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[40] L.-K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic Android malware analysis. In *USENIX Security Symposium*, pages 569–584, 2012.