ALMA MATER STUDIORUM – UNIVERSITA DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA E SCIENZE INFORMATICHE

Hooking Java methods and native functions to enhance Android applications security

Tesi in
Sicurezza delle Reti

Relatore:
Gabriele D’Angelo

Correlatore:
Bruno Crispo

Controrelatore:
Andrea Omicini

Presentata da:
Filippo Alberto Brandolini

Sessione II
Anno Accademico 2015/2016

Preface

This dissertation is the result of my personal elaboration of the work accomplished at IKS TN, Rovereto, Italy, during my work experience. Therefore, this is not to be considered an IKS TN official project, nor is IKS TN responsible for this work.

Acknowledgements

I would like to thank my thesis supervisor Prof. Gabriele D'Angelo of the University of Bologna for his assistance and precious collaboration.

I would also like to thank my assistant supervisor Prof. Bruno Crispo of the University of Trento for his support during my work experience, which inspired the realization of this work.

My sincere thanks also go to Prof. Andrea Omicini for being examiner of this thesis.

Finally, I would like to thank Dr. Maqsood Ahmad, who was involved in the analysis phase of this research project. His passionate participation and input have been essential.

Abstract

Mobile devices are becoming the main end-user platform to access the Internet. Therefore, hackers' interest for fraudulent mobile applications is now higher than ever. Most of the times, static analysis is not enough to detect the application hidden malicious code. For this reason, we design and implement a security library for Android applications exploiting the hooking of Java and native functions to enable runtime analysis. The library verifies if the application shows compliance to some of the most important security protocols and it tries to detect unwanted activities. Testing of the library shows that it successfully intercepts the targeted functions, thus allowing to block the application malicious behaviour. We also assess the feasibility of an automatic tool that uses reverse engineering to decompile the application, inject our library and recompile the security-enhanced application.

I dispositivi *mobile* rappresentano ormai per gli utenti finali la principale piattaforma di accesso alla rete. Di conseguenza, l'interesse degli hacker a sviluppare applicazioni mobile fraudolente è più forte che mai. Il più delle volte, l'analisi statica non è sufficiente a rilevare tracce di codice ostile. Per questo motivo, progettiamo e implementiamo una libreria di sicurezza per applicazioni Android che sfrutta l'*hooking* di funzioni Java e native per effettuare un'analisi dinamica del codice. La libreria verifica che l'applicazione sia conforme ad alcuni dei principali protocolli di sicurezza e tenta di rilevare tracce di attività indesiderate. La fase di *testing* mostra che la libreria intercetta con successo le funzioni bersaglio, consentendo di bloccare il comportamento malevolo dell'applicazione. Valutiamo altresì la fattibilità di un programma che in modo automatico sfrutti tecniche di *reverse engineering* per decompilare un'applicazione, inserire al suo interno la libreria e ricompilare l'applicazione messa in sicurezza.

Categories and Subject Descriptors: C.2.0 [**Computer-Communication Networks**]: Security and protection; D.4.6. [**Processor Architectures**]: Cellular architecture; D.2.7 [**Software Engineering**]: Restructuring, reverse engineering, and reengineering.

Keywords: android, security, monitoring, hooking, reverse engineering.

Contents

List of Abbreviations

ADBI	Android Dynamic Binary Instrumentation
API	Application Programming Interface
App	Application
APK	Android Package
ARM	Acorn/Advanced RISC Machine
ART	Android Runtime
BID	Base station Identifier
C&C	Command & Control
CDMA	Code Division Multiple Access
CERT	Computer Emergency Response Team
CID	Cell Identifier
DCL	Dynamic Class Loading
DEX	Dalvik Executable
DDoS	Distributed Denial of Service
GID	Group Identifier
GMS	Google Mobile Services
GPS	Global Positioning System
GSM	Global System for Mobile Communications (originally Groupe Spécial Mobile)
HAL	Hardware Abstraction Layer
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
ICCID	Integrated Circuit Card Identifier

JNI	Java Native Interface
IMEI	International Mobile Equipment Identity
IMSI	International Mobile Subscriber Identity
IPv4	Internet Protocol version 4
LAC	Location Area Code
LOC	Lines Of Code
NID	Network Identifier
OWASP	Open Web Application Security Project
RISC	Reduced Instruction Set Computing
SD	Secure Digital
SID	System Identifier
SIM	Subscriber Identity Module
TLS	Transport Layer Security
UID	User Identifier
UMTS	Universal Mobile Telecommunications System
URL	Uniform Resource Locator
VM	Virtual Machine

List of Figures

List of Tables

Chapter 1

Introduction

Over the past few years, mobile devices have overtaken laptops and desktop computers as the most popular technology for web browsing [2]. This led malicious hackers to focus on mobile devices and to implement *malwares*[1] especially created for this type of technology. Hiding malicious code inside an application is extremely common nowadays, and it represents the major threat in mobile security. The goal of this project is to provide a shield against such threat by developing a library that can be packaged inside Android applications to allow runtime analysis and to verify the application security level. The library, that we will address as *hooking* library, must investigate several scenarios and check if the application tries to perform some malicious activity or if it contains one or more vulnerabilities which could be exploited by adversaries for malicious purposes. To ensure this achievement, we perform an accurate analysis of the main mobile weaknesses and the malwares exploiting those weaknesses, in order to identify potential risks. Then, we examine the state of the art of Android security measures and we select the best strategy to implement our monitoring library. Lastly, as a case study, we implement a test application containing malicious code to

[1]Malware: abbreviation for malicious software. *"A piece of software (computer program) written by someone with mischievous and/or criminal intent"*. A malware usually try to *"spread itself by some means, and to do some sort of damage or theft"* [3].

experiment with the hooking library functionalities.

Malicious activities and malwares can be classified into a few major groups:

– *Personal Information Stealing*: applications (hereinafter referred to as *apps*) could try to access the users' personal information without their consent. Personal information may include photos stored in the phone memory, SMS messages, contacts, banking details, etc. The app can retrieve all the data and leak it to the outside world using telephony and radio networks or the Internet. According to Check Point[2] Threat Intelligence Research Team [5], *HummingBad* is the most common malware used to attack mobile devices in 2016. This Android malware *"establishes a persistent rootkit on the device, installs fraudulent applications and enables additional malicious activity such as installing a key-logger, stealing credentials and bypassing encrypted email containers used by enterprises"* [5];

– *Ransomware*: this type of malicious app usually encrypts users' data as soon as it gets installed on the device: adversaries then ask users to pay a ransom if they want to get their data back. *Trojan:Android/Koler* is the major example of ransomware for Android: *"on installation, the app sends the device International Mobile Subscriber Identity (IMEI) number to a remote server. It then opens a browser page that displays a fake notice over the Home screen saying the device has been locked due to security violations and all files have been encrypted"* [6], thus demanding payment of a fine to decrypt the files;

– *Dialer*: once installed on the user's device, a dialer app sends SMS messages to premium numbers, or call them. These operations result in the user unknowingly losing credit;

[2]Check Point Software Technologies Ltd. is one of the major security vendors protecting customers from cyber attacks and malwares, both on enterprises' networks and mobile devices [4].

- *Privilege Escalation*: privilege escalation attacks occur when apps with minimum permissions try to gain root access to take full control of the user's device. Although every Android app runs in a separate environment and its permissions are granted by the user during installation, Dengre and Kaushal [7] explain how *"privileged permissions can be obtained by malicious apps by launching privilege escalation attacks. Through these attacks, an application may gain permission to perform a privileged task which it is not authorized"*;

- *Botnets*: a special kind of malicious app starts communicating with a *Command & Control (C&C)* server as soon as it gets installed on the user's device and performs malicious activities as directed by the C&C server: stealing information, installing or uninstalling other apps, taking part in *Distributed Denial of Service (DDoS)*[3] attacks, etc. The C&C server connects many other compromised devices, forming a botnet[4]. Communications between the compromised devices and the C&C server usually occur through SMS messages or the Internet;

- *Deprecated Methods Exploits*: *"As programming languages evolve, functions occasionally become obsolete"*, hence deprecated, *"due to an improved understanding of how operations should be performed effectively and securely"* [9]. Therefore, *"the use of deprecated functions may indicate neglected code"* and a potential risk for the app security.

1.1 System Requirements

Based on our malwares analysis, we believe a monitoring app should first check if the target app attempts to access the user's private information. A user's phone is in fact home to a lot of sensitive and private information,

[3]A DDoS is a cyber attack where the perpetrator exploits more than one machine to flood the target infrastructure or network resource with superfluous requests, in order to make the targeted resource unavailable to its legitimate users [8].

[4]Botnet: combination of the words robot and network.

Action	*Threat*	*Malware*
Accessing SD Card	Information Stealing	AndroRATIntern [10]
Reading device information	Information Stealing	DroidDream [11]
Reading SMS messages	Information Stealing	Spitmo [12]
Detecting user location	Information Stealing	DroidDream [11]

Table 1.1: Personal Information Stealing attempts

such as login credentials, banking details, private conversations and Global Positioning System (GPS) locations. This data is usually stored in the phone internal memory or in Secure Digital (SD) and Subscriber Identity Module (SIM) cards. The Android framework provides methods which can be used to access this information. Table 1.1 provides a list of information stealing attempts, with examples of malwares exploiting those actions.

After collecting private information, a malicious app would try to leak that information to some C&C Server or just to the outside world using one or more communication channels listed in Table 1.2. The hooking library should scan the app code for possible Internet connections or telephony network communications.

Action	*Threat*	*Malware*
Calling phone numbers	$ Loss	BaseBridge [13]
Sending SMS/MMS messages	Privacy Leakage / $ Loss	Fakenotify [14]
Accessing the network	Privacy Leakage / Botnet	AnserverBot [15]

Table 1.2: Possible communication channels for private information leakage

Moreover, due to reasons such as negligence or incompetence, some programming practices, like using deprecated methods or using HyperText Transfer Protocol (HTTP) instead of HyperText Transfer Protocol Secure (HTTPS) might leave the app vulnerable to attacks. Also, critical information should always be encrypted in the phone memory; storing such important information in world-readable locations is extremely dangerous. The app should always communicate through secure channels and avoid deprecated methods and classes. Our library should check the app code for such weak programming practices or vulnerabilities. These weaknesses are listed in Table 1.3.

Action	*Threat*
Using deprecated methods	Privacy Leakage / Corrupt application
Using HTTP instead of HTTPS	Privacy Leakage / Corrupt application

Table 1.3: Weak programming practices resulting in security threats

All the potentially malicious actions listed in tables 1.1, 1.2 and 1.3 can be executed or are represented by specific Java methods or native functions, that we will call *pivotal functions*. The first idea would be to use static analysis to check if such functions are included in the app code. However, as explained by Ahmad and Crispo [16], recent dynamic techniques such as Reflection and Dynamic Class Loading (DCL) allow an app to change its behaviour at runtime. These techniques are mainly used in Android apps for extensibility. Nevertheless, malware developers take advantage of these techniques to bypass static analysis tools and they empower malicious apps to reveal their hidden code only when they are running on the user's device. Therefore, we focus on runtime analysis to prevent this threat. The goal of this work is to identify and intercept the pivotal functions, access their data dynamically and block their execution if illegitimate or insecure activities are found. To achieve this, a good knowledge of the Android framework is required.

1.2 The Android Stack

To support code reuse[5] and to allow other developers to include our work in their projects, we aim at implementing a shared library containing all the monitoring code and we discourage developers from writing monitoring code directly inside their apps core logic. To identify the best strategy for the library implementation, we analyze the Android architecture.
Android is an open source, Linux-based mobile operative system consisting of six major components (Fig. 1.1):

- *Linux Kernel*: the foundation of the Android platform is the Linux kernel, which handles the most low-level system functionalities;

- *Hardware Abstraction Layer*: *"the Hardware Abstraction Layer (HAL) provides standard interfaces that expose device hardware capabilities to the higher-level Java framework"* [1]. HAL consists of multiple libraries, each one implementing an interface for a specific hardware component (e.g. Camera, Bluetooth);

- *Android Runtime*: Since Android version 5.0 — Application Programming Interface (API) level 21 — *"each app runs in its own process and with its own instance of the Android Runtime (ART)"* [1]. ART runs multiple virtual machines — one for each application — by executing Dalvik Executable (DEX) files, a bytecode format optimized for minimal memory footprint;

- *Native C/C++ Libraries*: Android system components require native libraries written in C and C++;

- *Java API Framework*: this layer exposes the native libraries to the higher-level apps;

[5]Code reuse consists in exploiting existing software or knowledge to implement new programs, following the reusability principles [17].

– *System Apps*: Android comes with a set of core apps (e.g. email, SMS messaging, internet browsing) which provide key functionalities that developers can include in their own apps.

Java is a high-level programming language, and Android reflects this trait. High-level programming languages provide a great amount of abstraction from machine language, which mainly implies a focus on usability and simplicity over optimal program efficiency and access to the system architecture. Not by chance, most of Java Android methods usually call basic functions from the native libraries (e.g. `socket`, `read`, `write`), executing code that cannot be changed or accessed from the Java level. For this reason, if we want to intercept all malicious activities, we need to access the Native Libraries and to program both in Java and C, which is possible thanks to Android NDK [18]. Once we have identified all the critical methods that require monitoring, we can group them into two main categories:

1. Java methods with direct access to the information we need;
2. Java methods calling native functions that handle the information we need;

The list of methods we want to intercept[6] consists of all the Java methods from group 1, plus all the native functions underlying the methods from group 2. In the next section, we evaluate several possible solutions to select the best candidate for the hooking library implementation.

1.3 Possible Solutions

"Securing Android devices often requires modifying their write-protected underlying system components files" [19]. The simplest way to do this is to *root* target devices to obtain full access to the system architecture. Rooting is the process of granting the device user privileged control (known as root

[6]We draw up this list in Chapter 2.

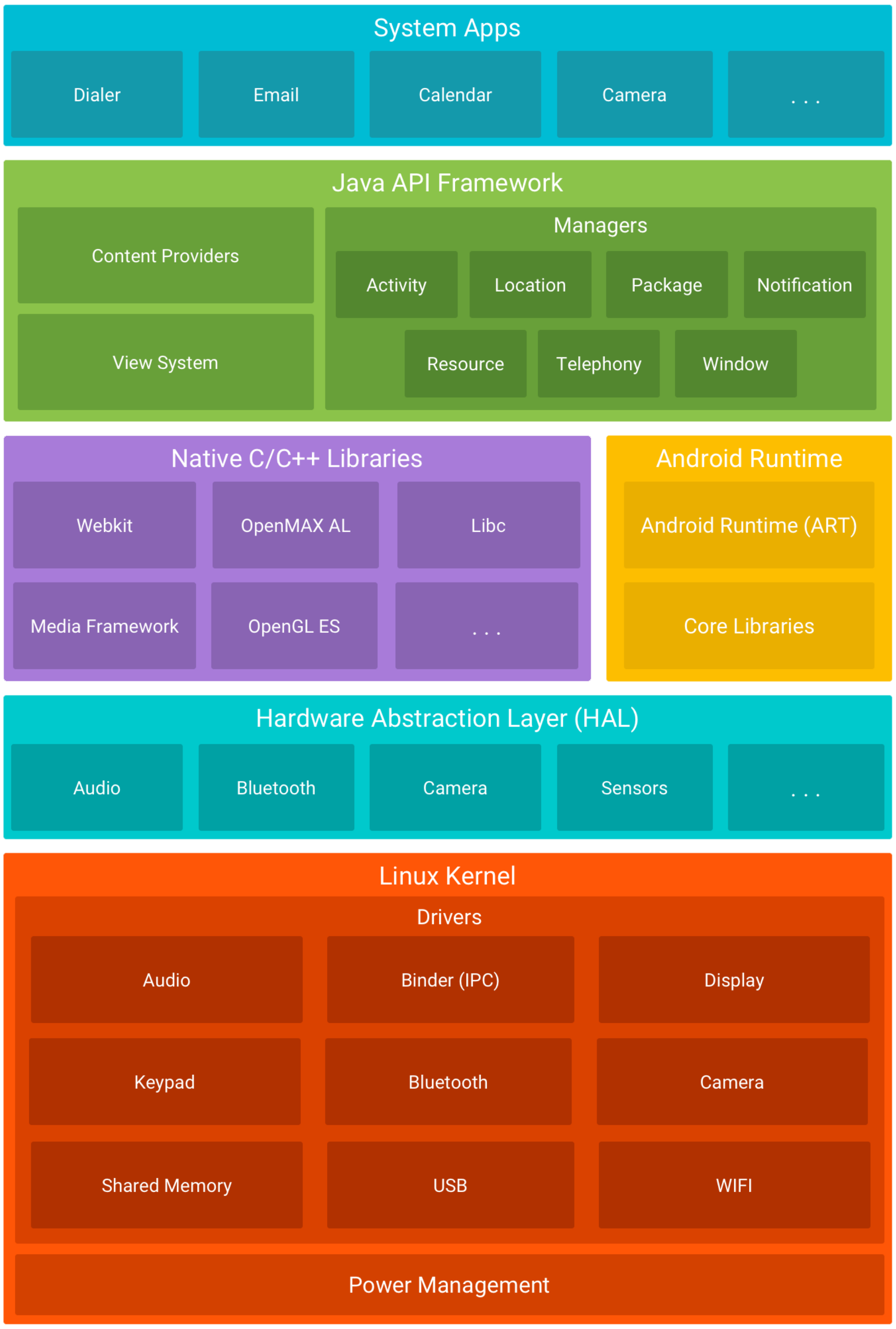


Figure 1.1: The Android software stack [1]

access) over the system. This process is often performed in order to bypass limitations set by carriers and hardware manufacturers. Therefore, rooting gives permissions to alter system components and settings that are otherwise inaccessible. In a rooted environment, we could provide a custom Android framework with monitoring features. However, forcing users to root their phones is not an acceptable solution. First, because the rooting process invalidates the device warranty, but also because the vast majority of users do not know how to do it, or they expect — with good reason — to have a perfectly secure device without having to do anything more than buying it. The first possibility is to introduce a reset procedure, triggered when launching the target application, that starts a new execution environment in which the reference to the underlying system components is redirected to a security-enhanced alternative. This technique is called *reference hijacking* [19]: in the new environment, *"the target application can load system class libraries and native libraries from any place instead of the default folders"*, allowing to introduce extra security features on unrooted devices. The environment reset is achieved by calling a native `exec` function, which completely replaces the current process with the security-enhanced program. The problem with this solution is that many vendors actually customize their firmwares instead of using Android stock version. Therefore, the process may be different for different versions of Android, depending upon how processes are started. Besides, the reset procedure is quite invasive and it introduces a noticeable overhead.

Another possible solution is to use `ptrace`, a system call provided by Linux to trace all system calls made by the targeted process. It is possible to deploy an app having two processes where the first one (the *"tracer"*) observes and controls the execution of the other (the *"tracee"*) [20]. In our case, the hooking library could run as the tracer process, attach `ptrace` to the target app and monitor its system calls. The tracer process could then intercept, extend or block the system calls, thus enforcing the desired security policies. Unfortunately, `ptrace` can only be used when tracer and tracee process have the same User Identifier (UID) or Group Identifier (GID), or when the tracer process has the `CAP_SYS_PTRACE` capability [21]. Before Android 4.4, Zygote[7] had this capability by default, therefore it could trace every process. However, from Android 4.4, Zygote loses this ability, so the only way to allow *"ptracing"* is to have both the processes with the same UID or GID. A possible strategy is to create a sandbox where the target app runs inside a *"container"* process, which forks on startup to spawn a child process, thus sharing its UID [22]. This way, `ptrace` can work for newer versions of Android as well. However, this solution is quite invasive, part of the target app must be changed and the container app requires different structures for different Android versions.

A third option found in the literature is to exploit *Virtual Memory Tampering*: every Android app has a list of all the virtual methods and their references in the app virtual memory. ART uses reflection or native methods to retrieve virtual methods references and invoke them. Tampering the virtual methods table is a way to direct a method call to custom monitoring code and then re-direct it back to the original code. This way, we can intercept critical methods and detect malicious activities. This solution requires root privileges to inject the hooking library in the virtual memory of the app [23]. However, we identify a couple of workarounds to use this strategy for unrooted devices as well:

1. the library can be packaged inside the app;

2. reverse engineering can be used to decompile the target app, inject the library and repackage the app into a new security-enhanced Android

[7]Like many other Linux-based systems, Android provides a startup bootloader which loads the kernel and starts the `init` process. This, in turn, launches all the daemons handling the hardware interfaces, and then executes the Zygote process. Zygote basically loads every Java class and resource used by the framework and the other applications, and then it starts listening on a socket (`/dev/socket/zygote`) for application launch requests. For every request, Zygote forks and spawns a new Virtual Machine (VM) in which the specific application will be executed. This means that every Android application runs in a separate process, and Zygote is the parent of every application process.

package (APK).

We believe that Virtual Memory Tampering is the best solution to reach our project achievement. Therefore, the hooking library will be implemented by following this strategy. To summarize, this work is organized as follows:

- we discuss the implementation of the hooking library in chapter 2;
- we evaluate the feasibility of an automatic tool that uses reverse engineering to decompile the application, inject our library and recompile the security-enhanced application in chapter 3;
- we discuss the project related works in chapter 4;
- we examine the project limitations and open challenges in chapter 5.

Chapter 2

The hooking library

The foundations of the hooking library are Android Dynamic Binary Instrumentation (ADBI) [24] and Legend [25]. ADBI is a tool written in C which injects code in the memory of an Android app and it implements virtual memory tampering for functions hooking. It exploits in-line hooking to perform redirection: by modifying the entry point of a function, the tool makes the function jump to the address of a custom code, which returns the control after performing the required processing. Legend is a hooking framework for Dalvik and ART Android applications allowing to hook Java methods without root privileges. It uses Java Native Interface (JNI) to call Java methods from the native program implementing the hooking core. It can run on Android versions 4.2—6.0.1. Both ADBI and Legend are compatible with ARM-32[1] architectures *(armeabi-v7a)*. The hooking library imports ADBI and Legend and it implements the security verifications on top of it.

2.1 Reading device information

Some malicious apps will try to retrieve information about the device in which they are running. Android provides some legitimate methods that

[1]ARM: originally Acorn RISC Machine, later Advanced RISC Machine. RISC stands for Reduced Instruction Set Computing.

could be exploited by adversaries to achieve this goal. We consider them pivotal functions and we show them in Table 2.2. All of the methods require `READ_PHONE_STATE` permissions.

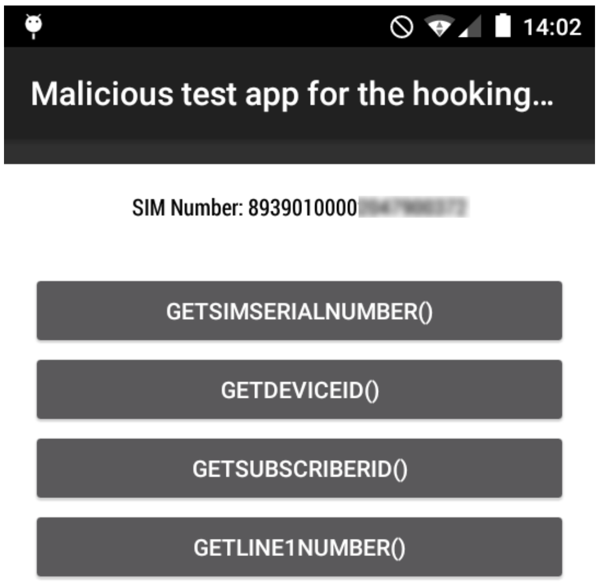
Method	*Class*	*Information retrieved*
`getSimSerialNumber()`	TelephonyManager	ICCID
`getDeviceId()`	TelephonyManager	IMEI
`getSubscriberId()`	TelephonyManager	IMSI
`getLine1Number()`	TelephonyManager	Telephone Number

Table 2.1: List of methods that access device information

The Integrated Circuit Card Identifier (ICCID), or SIM Serial Number, is a global identifier for unique SIM cards. Adversaries might be interested in stealing ICCID codes to do illegal activities while pretending to be someone else, thus avoiding to be tracked. A SIM card ICCID can be retrieved using the `getSimSerialNumber()` method. The following are hooking method for `getSimSerialNumber()` and its test code:

```
@Hook("android.telephony.TelephonyManager::
    getSimSerialNumber")
public static String TelephonyManager_getSimSerialNumber(
    TelephonyManager tm) {
    if (!ALLOW_GETSIMSERIALNUMBER){
        return "POTENTIAL INFORMATION LEAKAGE DETECTED:
            getSimSerialNumber() hooked and blocked! The
            app just tried to retrieve your SIM number!";
    } else {
        return ""+HookManager.getDefault().callSuper(tm);
    }
}
```

```
enableGetSimSerialNumberCheckBox.setOnCheckedChangeListener
    (new CompoundButton.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton buttonView,
        boolean isChecked) {
```



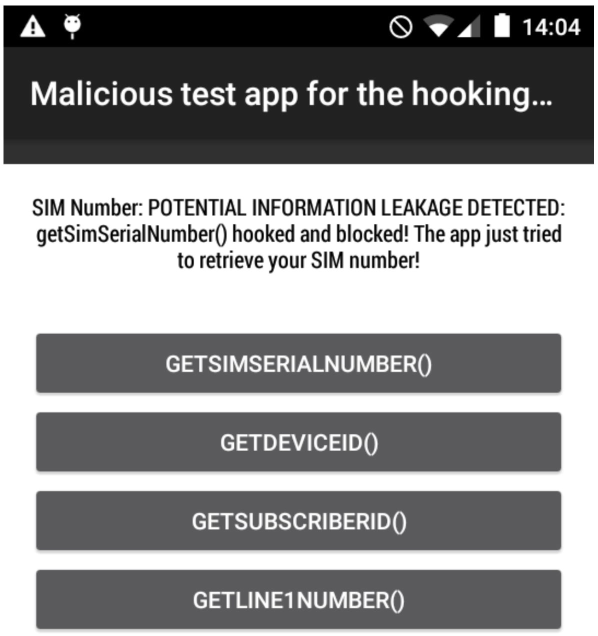
SIM Number: 8939010000[redacted]

Figure 2.1: Malicious behaviour: getSimSerialNumber()

```java
            App.ALLOW_GETSIMSERIALNUMBER = isChecked;
        }
});

button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        TelephonyManager telephonyManager = (TelephonyManager) getSystemService(TELEPHONY_SERVICE);
        telephonyManager.getSimSerialNumber();
        String result = telephonyManager.getSimSerialNumber();
        String tv = ("SIM Number: " + result);
        textView.setText(tv);
    }
});
```

Our test app behaviour with `getSimSerialNumber()` enabled is showed in Figure 2.1, while hooking behaviour is showed in Figure 2.2.



SIM Number: POTENTIAL INFORMATION LEAKAGE DETECTED: getSimSerialNumber() hooked and blocked! The app just tried to retrieve your SIM number!

Figure 2.2: Malicious behaviour: getSimSerialNumber(), blocked

The International Mobile Equipment Identity (IMEI) is a global identifier for unique devices. IMEI code is sent during the handshake process when connecting to a network, and the carrier can use it to identify the device model. The main reasons why adversaries want their victims' IMEI are the following:

- they can pretend to possess their victims' devices;
- they acquire more knowledge about their victims' devices;

In the first place, let us consider the Samsung Case in September 2016, when the South Korean company had to recall 2.5 million Galaxy Note 7s due to exploding batteries [26]. The manufacturer offered a replacement process for which if customers presented their device IMEI, the company would verify if they were eligible for a free replacement or refund. Although, if their device IMEI had been stolen, someone else could claim the free device in place of the legitimate customers. In this kind of scenario, the company typically asks every customer to provide a proof of purchase. However, adversaries try to take advantage of the circumstances (e.g. telling the device was a gift, threatening to make official complaints about customer dissatisfaction) so that replacements can be granted more easily.

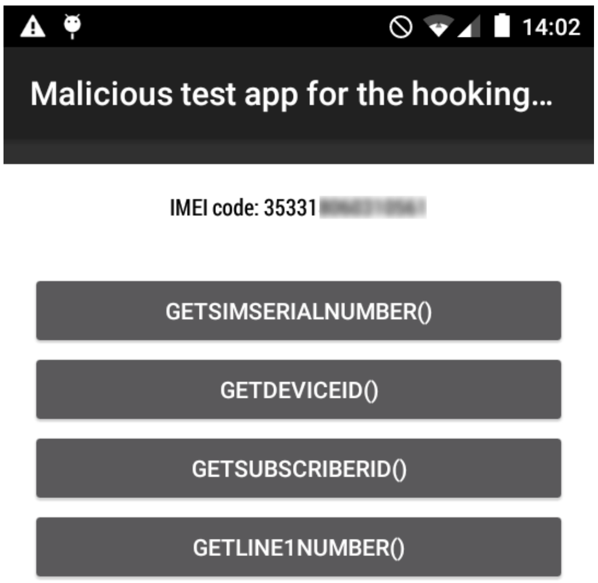
Secondly, an adversary can gather even more information about the victim's device by using an IMEI analyzer tool or web service.

IMEI codes can be retrieved by using the `getDeviceId()` method. The following are hooking method for `getDeviceId()` and its test code:

```java
@Hook("android.telephony.TelephonyManager::getDeviceId")
public static String TelephonyManager_getDeviceId(
    TelephonyManager tm){
     if (!ALLOW_GETDEVICEID) {
          return "POTENTIAL INFORMATION LEAKAGE DETECTED:
              getDeviceId() hooked and blocked! The app just
              tried to retrieve your IMEI!";
     } else {
          return ""+HookManager.getDefault().callSuper(tm);
     }
}
```

```java
enableGetDeviceIdCheckBox.setOnCheckedChangeListener(new
    CompoundButton.OnCheckedChangeListener() {
     @Override
     public void onCheckedChanged(CompoundButton buttonView,
          boolean isChecked) {
          App.ALLOW_GETDEVICEID = isChecked;
     }
});

button5.setOnClickListener(new View.OnClickListener() {
     @Override
     public void onClick(View v) {
          TelephonyManager telephonyManager = (
              TelephonyManager) getSystemService(
              TELEPHONY_SERVICE);
          String IMEI = telephonyManager.getDeviceId();
          String tv = ("IMEI code: " + IMEI);
          textView.setText(tv);
```



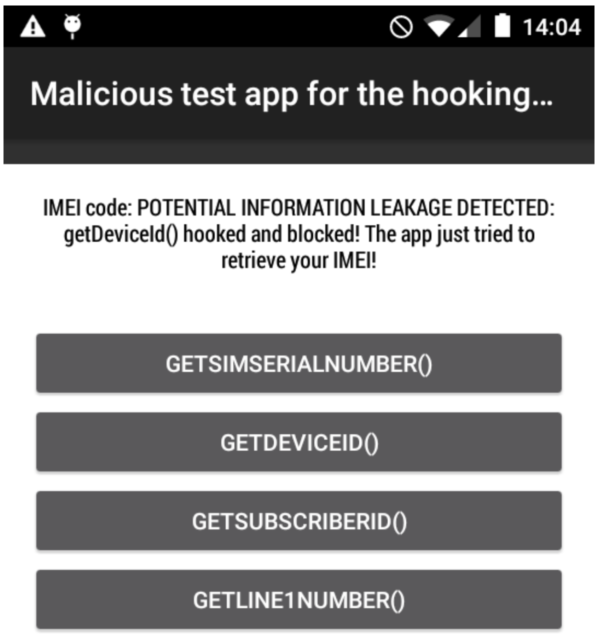
IMEI code: 35331[redacted]

Figure 2.3: Malicious behaviour: getDeviceId()

```
    }
});
```

Our test app behaviour with `getDeviceId()` enabled is showed in Figure 2.3, while hooking behaviour is showed in Figure 2.4.

The International Mobile Subscriber Identity (IMSI) is a global identifier for unique subscribers to the wireless communications network. It has the format `MCC-MNC-MSIN`, where `MCC` (the first 3 digits) is the Mobile Country Code (e.g. 222 for Italy), `MNC` (2 or 3 digits) is the Mobile Network Code (e.g. 410 for AT&T) and `MSIN` is the Mobile Subscription Identification Number. All communications through the Global System for Mobile Communications (GSM) and the Universal Mobile Telecommunications System (UMTS) networks use IMSI as the primary identifier for every subscriber. Adversaries target IMSI codes to enable interception and traffic analysis on the victims' calls with the help of devices such as IMSI catchers and fake station generators [27]. IMSI codes can be retrieved by using the `getSubscriberId()` method. The following are hooking method for `getSubscriberId()` and its test code.

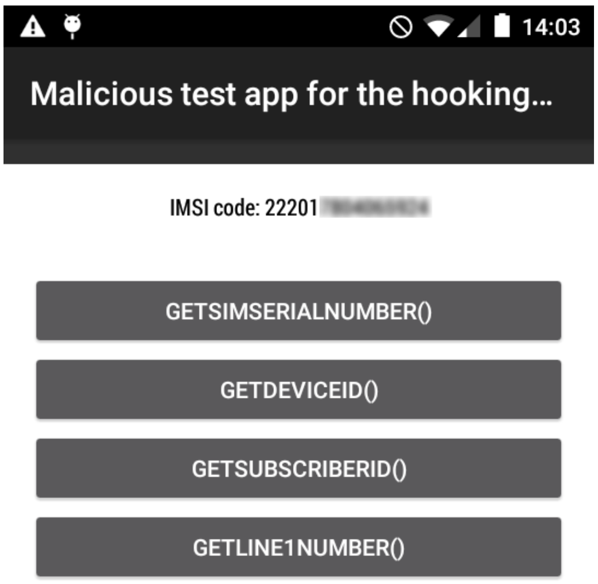


IMEI code: POTENTIAL INFORMATION LEAKAGE DETECTED: getDeviceId() hooked and blocked! The app just tried to retrieve your IMEI!

Figure 2.4: Malicious behaviour: getDeviceId(), blocked

```java
@Hook("android.telephony.TelephonyManager::getSubscriberId"
    )
public static String TelephonyManager_getSubscriberId(
    TelephonyManager tm){
     if (!ALLOW_GETSUBSCRIBERID){
          return "POTENTIAL INFORMATION LEAKAGE DETECTED:
              getSubscriberId() hooked and blocked! The app
              just tried to retrieve your IMSI!";
     } else {
          return ""+HookManager.getDefault().callSuper(tm);
     }
}
```

```java
enableGetSubscriberIdCheckBox.setOnCheckedChangeListener(
    new CompoundButton.OnCheckedChangeListener() {
     @Override
     public void onCheckedChanged(CompoundButton buttonView,
          boolean isChecked) {
          App.ALLOW_GETSUBSCRIBERID = isChecked;
     }
```



IMSI code: 22201[redacted]

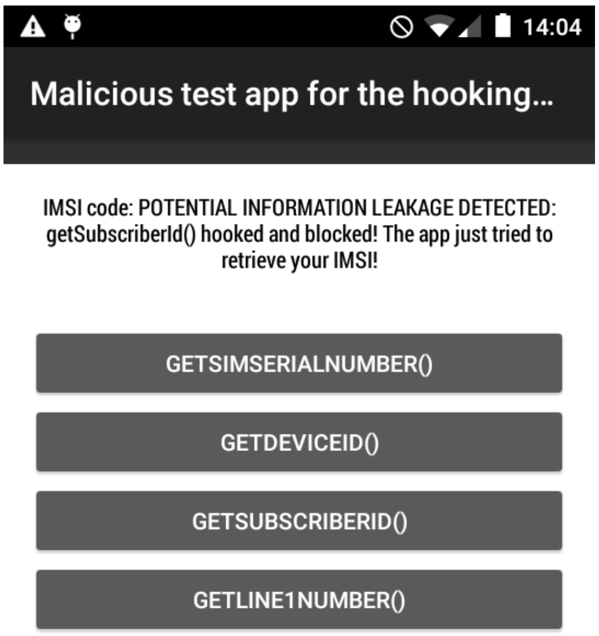
Figure 2.5: Malicious behaviour: getSubscriberId()

```java
});

button4.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        TelephonyManager telephonyManager = (
            TelephonyManager) getSystemService(
            TELEPHONY_SERVICE);
        String IMSI = telephonyManager.getSubscriberId();
        String tv = ("IMSI code: " + IMSI);
        textView.setText(tv);
    }
});
```

Our test app behaviour with `getSubscriberId()` enabled is showed in Figure 2.5, while hooking behaviour is showed in Figure 2.6.

Lastly, adversaries can try to retrieve the telephone number of the victims to send them spam messages and phishing attacks via SMS, or even to eavesdrop on calls and read texts. The user telephone number can be retrieved by calling the `getLine1Number()` method. The following are hooking method


14:04

Malicious test app for the hooking...

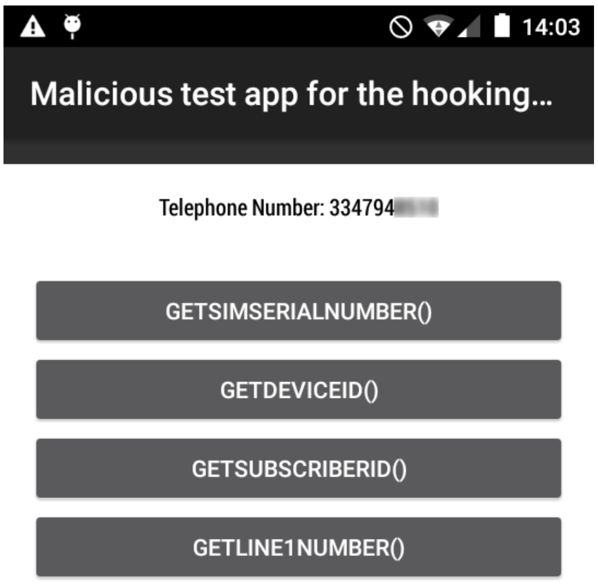
IMSI code: POTENTIAL INFORMATION LEAKAGE DETECTED: getSubscriberId() hooked and blocked! The app just tried to retrieve your IMSI!

Figure 2.6: Malicious behaviour: getSubscriberId(), blocked

for `getLine1Number()` and its test code:

```java
@Hook("android.telephony.TelephonyManager::getLine1Number")
public static String TelephonyManager_getLine1Number(
    TelephonyManager tm){
    if (!ALLOW_GETLINE1NUMBER){
        return "POTENTIAL INFORMATION LEAKAGE DETECTED:
            getLine1Number() hooked and blocked! The app
            just tried to retrieve your telephone number!";
    } else {
        return ""+HookManager.getDefault().callSuper(tm);
    }
}
```

```java
enableGetLine1NumberCheckBox.setOnCheckedChangeListener(new
     CompoundButton.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton buttonView,
         boolean isChecked) {
        App.ALLOW_GETLINE1NUMBER = isChecked;
```


Malicious test app for the hooking...

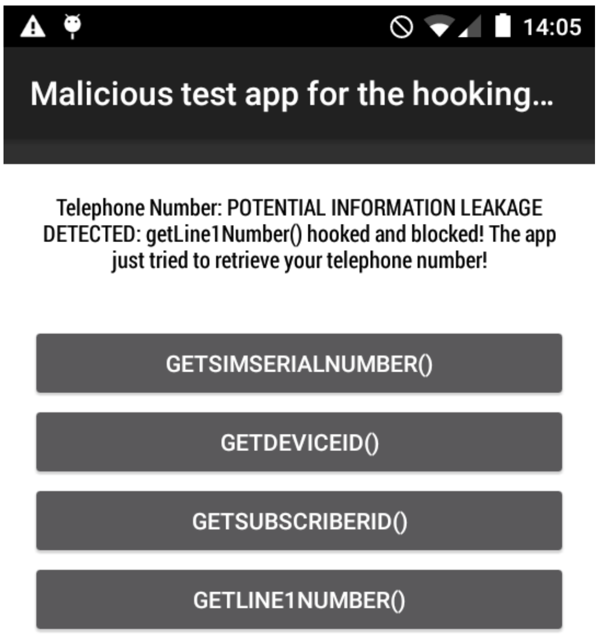
Telephone Number: 334794[redacted]

Figure 2.7: Malicious behaviour: getLine1Number()

```java
    }
});

buttonGetLine1Number.setOnClickListener(new View.
    OnClickListener() {
    @Override
    public void onClick(View v) {
        TelephonyManager telephonyManager = (
            TelephonyManager) getSystemService(
            TELEPHONY_SERVICE);
        String TelephoneNumber = telephonyManager.
            getLine1Number();
        String tv = ("Telephone Number: " + TelephoneNumber
            );
        textView.setText(tv);
    }
});
```

Our test app behaviour with `getLine1Number()` enabled is showed in Figure 2.7, while hooking behaviour is showed in Figure 2.8.


14:05

Malicious test app for the hooking...

Telephone Number: POTENTIAL INFORMATION LEAKAGE DETECTED: getLine1Number() hooked and blocked! The app just tried to retrieve your telephone number!

Figure 2.8: Malicious behaviour: getLine1Number(), blocked

2.2 Detecting user location

A malicious app could try to detect the user location. There are two ways to do it:

- exploiting GPS and Internet data;
- exploiting GSM and Code Division Multiple Access (CDMA) Telephony data.

GPS consists of 24 satellites orbiting around the Earth. These satellites, owned by the United States government, can locate world objects with an average precision of 6-12 metres [28]. GPS receivers catch the satellites signals to determine the location. Conversely, GSM technology determines an object location by using signal triangulation from base stations.

In this work, we focus on Telephony network exploits. User location detection via Telephony network depends on whether the device uses GSM or CDMA network. In the first case, which is the most common scenario, the adversary must obtain the device MCC and MNC codes plus the Location Area Code (LAC) and the Cell Identifier (CID). MCC and MNC can be retrieved by using `getSubscriberId()` to obtain the device IMSI, as explained in p. 18. As for LAC and CID, they can be retrieved by calling their respective *getter* methods — `getLac()` and `getCid()` — on `GsmCellLocation`. A `GsmCellLocation` instance with the device current location data can be obtained by calling `getCellLocation()` on a `TelephonyManager` instance with running on a GSM device. Pivotal functions for GSM user location detection are listed in Table 2.3. `getLac()` and `getCid()` are sub-pivots of

Method	*Class*	*Information retrieved*
`getCellLocation()`	TelephonyManager	GsmCellLocation
↪ `getLac()`	GsmCellLocation	LAC
↪ `getCid()`	GsmCellLocation	CID
`getSubscriberId()`	TelephonyManager	IMSI

Table 2.2: List of methods that access GSM device location data

`getCellLocation()`.

When the hacked device uses CDMA network (e.g. every tablet device without SIM card[2]), the adversary must combine the MCC code with the cell System Identifier (SID), Network Identifier (NID) and Base station Identifier (BID) codes. The MCC code can be retrieved by using `getSubscriberId()` to obtain the device IMSI, as explained in p. 18. The other identifiers can be obtained by calling their respective *getter* methods — `getSystemId()`, `getNetworkid()` and `getBaseStationId()` — on `CdmaCellLocation`. A `CdmaCellLocation` instance with the device current location data can be obtained by calling `getCellLocation()` on a `TelephonyManager` instance running on a CDMA device. New pivotal functions for CMDA user location detection are listed in Table 2.4. `getSystemId()`, `getNetworkId()` and `getBaseSystemId()` are sub-pivots of `getCellLocation()`.

[2]For example, the Samsung Galaxy Tab CDMA P100.

Method	*Class*	*Information retrieved*
`getCellLocation()`	TelephonyManager	CdmaCellLocation
↪ `getSystemId()`	CdmaCellLocation	SID
↪ `getNetworkid()`	CdmaCellLocation	NID
↪ `getBaseStationId()`	CdmaCellLocation	BID
`getSubscriberId()`	TelephonyManager	IMSI

Table 2.3: List of methods that access CDMA device location data

We decide to hook the macro function `getCellLocation()` in place of its sub-pivots for maximum security[3]. The following are hooking method for `getCellLocation()` and its test code, which checks if the device uses GSM or CDMA network and it executes its malicious code, accordingly:

```
buttonGetCellLocation.setOnClickListener(new View.
    OnClickListener() {
     @Override
     public void onClick(View v) {
          TelephonyManager telephonyManager = (
              TelephonyManager) getSystemService(
              TELEPHONY_SERVICE);
          String tvloc = "";
          if (telephonyManager.getPhoneType()==
              TelephonyManager.PHONE_TYPE_GSM){
               GsmCellLocation location = (GsmCellLocation)
                   telephonyManager.getCellLocation();
               int cid = location.getCid();
               int lac = location.getLac();
               tvloc = ("CID: " + cid + ", LAC: " + lac);
          } else if (telephonyManager.getPhoneType()==
              TelephonyManager.PHONE_TYPE_CDMA){
               CdmaCellLocation location = (CdmaCellLocation)
                   telephonyManager.getCellLocation();
               int sid = location.getSystemId();
```

[3]In a trade-off analysis, hooking the sub-pivots in place of their macro function could reduce the number of false positives.

```
                int nid = location.getNetworkId();
                int bid = location.getBaseStationId();
                tvloc = ("SID: " + sid + ", NID: " + nid + ",
                    BID: " + bid);
            } else {
                //
            }
            textView.setText(tvloc);
        }
});
```

2.3 Using deprecated methods

The majority of methods deprecations have no security ramifications, that is why blindly flagging all deprecated methods would produce many false positives. For example, `FontMetrics.getMaxDecent` method was deprecated because of a spelling error:

```
As of JDK version 1.1.1, replaced by getMaxDescent().
```

In fact, according to The Open Web Application Security Project (OWASP), *"not all functions are deprecated or replaced because they pose a security risk"* [9]. However, deprecated functions may indicate that part of the code is in a state of disrepair, which *"raises the probability that there are security problems lurking nearby"* [9]. At the Computer Emergency Response Team (CERT) of Carnegie Mellon University, experts say *"never use deprecated fields, methods, or classes in new code"* [29], because it can lead to erroneous behaviour that might become a threat for security in a second moment. The point of deprecating a method is to let developers know that there is now a better way to do what that method did, and that the deprecated code is likely to be removed in a future release. Our goal is to draw up a list of methods, classes and constants that have been deprecated for security reasons, so that we can try to hook them from the target application. If a hook succeeds, it means that the corresponding method is used by the application.

2.3.1 getRecentTasks(), getRunningTasks()

One of the changes in Android API 21, *LOLLIPOP*, is the deprecation of ActivityManager methods getRecentTasks() and getRunningTasks(), and the inclusion of their replacement ActivityManager.getAppTasks(). The main reason behind this change is the introduction of document-centric *recents*, which make getRecentTasks() and getRunningTasks() exploitable by adversaries for personal information leakage. Android say that these methods *"should never be used for core logic in an application"* [30]. Some banking Trojans use a technique that invokes getRunningTasks() to determine which process is currently running in the foreground. If the running process is a banking app, the malware can push itself to the foreground to steal information [31]. The following are hooking methods for getRecentTasks() and getRunningTasks(), plus their test codes and outputs:

```java
@Hook("android.app.ActivityManager::getRecentTasks@int#int"
    )
public static List<ActivityManager.RecentTaskInfo>
    ActivityManager_getRecentTasks (ActivityManager
    activityManager, int maxNum, int flags) {
     if (!ENABLE_GETRUNNINGANDRECENTTASKS) {
          System.out.println("getRecentTasks() hooked! This
              method has been deprecated in API 21 for
              security reasons and it should not be used!");
     } else {
          return HookManager.getDefault().callSuper(
              activityManager, maxNum, flags);
     }
     return null;
}
```

```java
buttonGetRecentTasks.setOnClickListener(new View.
    OnClickListener() {
     @Override
     public void onClick(View v) {
          ActivityManager am = (ActivityManager)
              getSystemService(ACTIVITY_SERVICE);
```

```
        List<ActivityManager.RecentTaskInfo> tasksInfo = am
            .getRecentTasks(1,1);
    }
});
```

```
D/Legend-Log: [+++] ActivityManager_getRecentTasks hooked.
I/System.out: getRecentTasks() hooked! This method has
    been deprecated in API 21 for security reasons and it
    should not be used!
```

```
@Hook("android.app.ActivityManager::getRunningTasks@int")
public static List<ActivityManager.RunningTaskInfo>
    ActivityManager_getRunningTasks (ActivityManager
    activityManager, int maxNum) {
    if (!ENABLE_GETRUNNINGTASKS) {
        System.out.println("getRunningTasks() hooked! This
            method has been deprecated in API 21 for
            security reasons and it should not be used!");
    } else {
        return HookManager.getDefault().callSuper(
            activityManager, maxNum);
    }
    return null;
}
```

```
button2.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        ActivityManager am = (ActivityManager)
            getSystemService(ACTIVITY_SERVICE);
        List<ActivityManager.RunningTaskInfo> tasksInfo =
            am.getRunningTasks(1);
    }
});
```

```
D/Legend-Log: [+++] ActivityManager_getRunningTasks hooked.
I/System.out: getRunningTasks() hooked! This method has
    been deprecated in API 21 for security reasons and it
    should not be used!
```

2.3.2 `MODE_WORLD_READABLE, MODE_WORLD_WRITABLE`

`MODE_WORLD_READABLE` and `MODE_WORLD_WRITABLE` are two constants for file creation allowing all other applications to have read access and write access, respectively, to the created file. These constants were deprecated in Android API 17 (*JELLY_BEAN_MR1*). *"Creating world-readable"* — and world-writable — *"files is very dangerous, and likely to cause security holes in applications"* [32]. Instead, applications should use `ContentProvider`, `BroadcastReceiver`, or `Service` for interactions. *"As of N, attempting to use this mode will throw a* `SecurityException`*"* [32]. To search for `MODE_WORLD_READABLE` and `MODE_WORLD_WRITABLE` occurrences at runtime, we can hook the `openFileOutput` method, which is used to *"open a private file for writing"* or to *"create the file if it doesn't already exist"* [33]. The second parameter passed to this method is an integer indicating the file creation mode. If our hook detect an `openFileOutput()`, it can access its mode and check if `Context.MODE_WORLD_READABLE` or `Context.MODE_WORLD_WRITABLE` are used.

2.3.3 Deprecated methods overview

The complete list of deprecated methods, classes and constants marked as threats is shown in Table 2.5.

Element	*Type*	*Risk*	*Deprecated in*
`MODE_WORLD_READABLE`	Constant	High	API 17
`MODE_WORLD_WRITABLE`	Constant	High	API 17
`getRecentTasks()`	Method	Medium	API 21
`getRunningTasks()`	Method	Medium	API 21

Table 2.4: List of methods, classes and constants that have been deprecated for security reasons

2.4 Using HTTP instead of HTTPS

The Internet resources can be accessed via HTTP, the basic Internet protocol, or HTTPS, which initializes encrypted connections to allow authentication of the requested resource and protection of the integrity of the exchanged data between client and server. HTTPS should always be preferred to HTTP, especially when the app manages user private information. To check if an app opens secure connections when accessing the Internet, we need to identify the main Java class used to open connections, analyze all of its dependencies until we find the most low-level functions called by the Java class, and access their data in search of information about the protocol and the connection integrity.
This is how a basic HTTPS connection is created in Android [34]:

```
URL url = new URL("https://wikipedia.org");
URLConnection urlConnection = url.openConnection();
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

`URLConnection` is the abstract class for every class that acts as communication link between the app and a Uniform Resource Locator (URL). In general, creating a connection to a URL is a multistep process [35]: the connection is initialized by invoking the `openConnection()` method on a specific URL, and then it is completed by the `connect()` method, which makes the actual connection to the remote object and allows to access its content. The connection process sub-methods are also listed in Table 2.1. We identify `URLConnection` as the main Java class used to open connections, and we acknowledge that `connect()` is the most low-level native function called by the Java class.

Follows `connect()` declaration in `/bionic/libc/bionic/connect.cpp`:

```
#include "private/NetdClientDispatch.h"
#include <sys/socket.h>
int connect(int sockfd, const sockaddr* addr, socklen_t
    addrlen) {
```

openConnection()	Manipulates parameters that affect the connection to the remote resource. High abstraction level.
connect()	Interacts with the resource; queries header fields and contents. Low abstraction level.

Table 2.5: URLConnection sub-methods

```
    return __netdClientDispatch.connect(sockfd, addr,
       addrlen);
}
```

This function takes `sockaddr*` as a parameter, which is a struct storing an array of characters, `char sa_data[14]`.

```
struct sockaddr {
  sa_family_t sa_family;
  char sa_data[14];
};
```

This array of characters contains information about the opening connection port number, which could be our first classifier for HTTP and HTTPS connections, but unfortunately it comes as raw data. However, in Network Programming, whenever we have a function taking a `sockaddr*` struct as a parameter, we can play with the struct `sockaddr_in` instead, and cast it to `sockaddr*` type with safety. The struct `sockaddr_in` is the basic Internet Protocol version 4 (IPv4) structure used for all system calls and functions dealing with Internet addresses, and it is of the same memory size of the struct `sockaddr`, so we can freely cast the pointer of one type to the other without any risk [36]. The struct `sockaddr_in` stores an address family in `sin_family`, a port in `sin_port`, and an IPv4 address in `sin_addr`:

```
#include <netinet/in.h>
// IPv4 AF_INET sockets:
struct sockaddr_in {
    short sin_family; // e.g. AF_INET
    unsigned short sin_port; // e.g. htons(3490)
    struct in_addr sin_addr;
    char sin_zero[8];
};
```

This trick will grant us access to the connection port number every time we hook the `connect()`, which becomes our first pivotal function. Follows the hooking method for `connect()`:

```
int my_connect(int sockfd, struct sockaddr* addr, socklen_t
    addrlen)
{
    int (*orig_connect)(int sockfd, struct sockaddr* addr,
       socklen_t addrlen);
    orig_connect = (void*)eph.orig;
    struct sockaddr_in *addr_in = (struct sockaddr_in*)addr
       ;
    log("my_connect() called!");
    log("Port: %d", ntohs(addr_in->sin_port));
    hook_precall(&eph);
    int res = orig_connect(sockfd, addr, addrlen);
    hook_postcall(&eph);
    return res;
}
```

The `ntohs` function [37] converts the port number from network byte order (big-endian[4]) to host byte order (little-endian[5] on Intel and many ARM processors[6]).

To verify the hooking function efficacy, we want our test app to open a basic HTTP connection. To do that, we create the class `NetworkTask.java` and we call its execution from the `MainActivity`.

```
package disi.unitn.test.adbitest;
import android.os.AsyncTask;
```

[4]Parameters are always sent most significant byte first.

[5]Parameters are always sent least significant byte first.

[6]The ARM architecture was purely little-endian before version 3, when it became bi-endian. Bi-endianness allows for switchable endianness in data and instruction fetches. A bi-endian machine can compute or send data in either endian format. The vast majority of architectures use little-endian as host byte order, anyway.

```java
import android.util.Log;
import java.net.HttpURLConnection;
import java.net.URL;

public class NetworkTask extends AsyncTask<String, Void,
    Void> {
     private Exception exception;

     @Override
     protected Void doInBackground(String... urls) {
         try {
             URL url = new URL(urls[0]);
             HttpURLConnection conn = null;
             conn = (HttpURLConnection) url.openConnection()
                 ;
             conn.connect();
             Log.d("NetworkTask:", "After Connect!");
         } catch (Exception e) {
             this.exception = e;
             return null;
         }
         return null;
     }
}
```

```java
String link = "http://www.google.com";
Log.d("Return fr. N HTTP Task", new NetworkTask().execute(
    link).toString());
```

The following is the output of the program: the hooking code tampers the virtual method reference, invokes the custom `connect()`, which retrieves the port number, and then it resumes the original `connect()` execution.

```
I/HOOKLIB: name: connect 11c99
I/HOOKLIB: hooking: connect = 0xb6d7ec99
I/HOOKLIB: THUMB using 0xb03ad659
I/HOOKLIB: my_connect() called!
I/HOOKLIB: Port: 80
D/NetworkTask:: After Connect!
```

The port detected is 80, which is in line with the type of protocol requested. If we open a secure `HttpsURLConnection()` instead of the HTTP based connection, the output will change accordingly.

```java
package disi.unitn.test.adbitest;
import android.os.AsyncTask;
import android.util.Log;
import java.net.HttpURLConnection;
import java.net.URL;
import javax.net.ssl.HttpsURLConnection;

public class NetworkHTTPSTask extends AsyncTask<String,
    Void, Void> {
    private Exception exception;

    @Override
    protected Void doInBackground(String... urls) {
        try {
            /* HTTPS connection test */
            URL url = new URL("https://wikipedia.org");
            URLConnection urlConnection = url.
                openConnection();
            InputStream in = urlConnection.getInputStream()
                ;
            copyInputStreamToOutputStream(in, System.out);
            Log.d("NetworkTask:", "After HTTPS Connection!"
                );
        } catch (Exception e) {
            this.exception = e;
            return null;
        }
        return null;
    }
}
```

```java
String link = "https://www.google.com";
Log.d("Return fr. N HTTPS Task", new NetworkHTTPSTask().
    execute(link).toString());
```

```
I/HOOKLIB: name: connect 11c99
I/HOOKLIB: hooking:   connect = 0xb6d7ec99
I/HOOKLIB: THUMB using 0xb03b3659
I/HOOKLIB: my_connect() called!
I/HOOKLIB: Port: 443
D/NetworkTask:: After HTTPS connection!
```

In the majority of cases, this verification will detect insecure connections of the malicious application. However, a skilled adversary could force an insecure connection on port 443, thus avoiding to be spotted. Therefore, checking the connection port number is not enough to tell if the connection is truly secure. To go deeper into the analysis, we examine other native libraries, such as `libssl` and `libcrypto`, in search of functions which could be useful to extract data related to the Transport Layer Security (TLS) handshake. For example, we could retrieve information about certificates, cipher suites and their validity.

However, hooking native functions included in the `libssl` or `libcrypto` libraries is much more complicated than hooking `libc` functions. Initially, we thought that these libraries were public, meaning that we could access their functions directly from our program. On the contrary, we found out that `libssl` and `libcrypto` are not public libraries, but platform private libraries. Private libraries can not be accessed from the program unless we manually include the library header files in the project, to override access to the library. Moreover, from API 24 (*N*[7]), Android imposes stricter restrictions on the type of libraries that can be loaded. More specifically, the dynamic linker does not load private libraries anymore. Consequently, apps do not access `libssl` and `libcrypto` directly. Instead, they use Google Mobile Services (GMS) Security Provider, when required. This implies that access to `libssl` and `libcrypto` libraries from our native code program might not be possible. Therefore, we conclude that our solution is not compatible with Android Nougat, which limits the hooking library compatibility to Android API 23

[7]Nougat.

(M[8]).

In order to hook `libssl` and `libcrypto` functions from our native code program, we build them locally by including their header files in the project[9]. We identify `SSL_connect` as the first pivotal function from the `libssl` library. This function takes the `SSL*` struct as a parameter, and it access lots of TLS information and values. The following are attempt of hooking function for `SSL_connect` and its output:

```
int my_SSL_connect(SSL* ssl){
    int (*orig_SSL_connect)(SSL*);
    orig_SSL_connect = (void*)eph.orig;
    log("my_SSL_connect() called!");
    log("cipher_list: %d", ssl->cipher_list->ciphers);
    log("client_CA: %d", ssl->client_CA);
    log("cert: %d", ssl->cert);
    log("ctx: %d", ssl->ctx);
    log("enc_method: %d", ssl->enc_method);
    log("handshake_func: %d", ssl->handshake_func);
    log("param: %d", ssl->param);
    hook_precall(&eph);
    int *res = orig_SSL_connect(ssl);
    hook_postcall(&eph);
    return res;
}
```

```
I/HOOKLIB: hooking: SSL_connect = 0xb5d0ff21
I/HOOKLIB: my_SSL_connect() called!
I/HOOKLIB: cipher_list: 197398
I/HOOKLIB: client_CA: 0
I/HOOKLIB: cert: 0
I/HOOKLIB: ctx: 0
I/HOOKLIB: enc_method: -1219687888
I/HOOKLIB: handshake_func: 1
I/HOOKLIB: param: 0
D/NetworkTask:: After HTTPS connection!
```

[8]Marshmallow.
[9]Header files for `libssl` and `libcrypto` can be downloaded from OpenSSL repository.

We are successful in hooking the function from `libssl`, but we access raw data instead of human readable information. We address this problem in Chapter 5, where we highlight TLS analysis as one of the major future works to be done in this research.

2.5 The hooking library manually imported in new projects

In this section we show how we set up the configuration files for Android NDK correct usage. If developers want to include our project in their apps, they can simply import the hooking library source code, containing `Android.mk` and `Application.mk` *makefiles* configured as follows:
`Android.mk`

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := base
#LOCAL_SRC_FILES := base/obj/local/armeabi-v7a/libbase.a
#LOCAL_SRC_FILES += base/obj/local/x86/libbase.a
LOCAL_SRC_FILES := base/obj/local/armeabi/libbase.a
LOCAL_EXPORT_C_INCLUDES := base
include $(PREBUILT_STATIC_LIBRARY)
include $(CLEAR_VARS)
LOCAL_MODULE    := myjni
LOCAL_SRC_FILES := HookLibJNI.c  epoll_arm.c.arm
LOCAL_CFLAGS := -g
LOCAL_SHARED_LIBRARIES := dl
LOCAL_STATIC_LIBRARIES := base
#LOCAL_SHARED_LIBRARIES += base
include $(BUILD_SHARED_LIBRARY)
####
#LOCAL_PATH := $(call my-dir)
#include $(CLEAR_VARS)
#LOCAL_MODULE    := myjni
#LOCAL_SRC_FILES := HookLibJNI.c
```

```
#LOCAL_SRC_FILES += elf_hook.c
#LOCAL_SRC_FILES += libtest2.c
#LOCAL_LDLIBS := -L$(SYSROOT)/usr/lib -llog
#include $(BUILD_SHARED_LIBRARY)
#LOCAL_SHARED_LIBRARIES += myjni
#include $(CLEAR_VARS)
#LOCAL_MODULE    = libtest2
#LOCAL_SRC_FILES = libtest2.c
#LOCAL_LDLIBS := -L$(SYSROOT)/usr/lib -llog
#include $(BUILD_SHARED_LIBRARY)
#LOCAL_SHARED_LIBRARIES += libtest2
```

`Application.mk`

```
#APP_ABI := armeabi-v7a
#TARGET_ARCH_ABI := armeabi-v7a
APP_ABI := armeabi
#APP_PLATFORM := android-16
```

Alternatively, the hooking library project can be exported as a shared object (`.so`) and consequently imported in new projects as a static library, as follows:

```java
public class MainActivity extends Activity {
        static {
        System.loadLibrary("myjni"); // .dll in Windows
                                     // .so in Unix
        }
        ...
```

2.6 Test

In this section we show the results of our testing of the hooking library on different devices. Testing is done on the following devices:

1. a physical rooted Motorola Moto E running custom Android 5.1.1;
2. a physical unrooted Xiaomi Redmi Note 3 running custom Android 5.0.2;

3. a Google Nexus 5 emulator running stock Android 6.0.1.

The hooking library is able to initialize the hooks in 92% of cases[10]. 82% of those hooks are successful in intercepting pivotal functions and performing monitoring, while 18% of them are successful in function redirection but they do not manage to retrieve enough data to verify the presence of malicious activity. Lastly, two hooks, accounting for 8% of the cases, failed due to segmentation faults. We consider this a good result, but we expect to achieve a better outcome in the near future. A complete list of the library hooks is shown in Table 2.6.

Pivotal function	*Language*	*HookDev1*	*HookDev2*	*HookDev3*
`connect()`	C	Success	Success	Success
`getSimSerialNumber()`	Java	Success	Success	NA
`getDeviceId()`	Java	Success	Success	NA
`getSubscriberId()`	Java	Success	Success	NA
`getLine1Number()`	Java	Success	Success	NA
`getRecentTasks()`	Java	Success	Success	Success
`getRunningTasks()`	Java	Success	Success	Success
`getCellLocation()`	Java	Success	Failed	NA
`sendTextMessage()`	Java	Initialized	Initialized	NA
`SSL_connect()`	C	Initialized	Initialized	Failed

Table 2.6: List of hooks and their test outcomes

[10]We exclude from the analysis every scenario in which a method that requires a working telephony network and/or a SIM card is monitored inside the emulator (Not Applicable — NA in Table 2.6).

Chapter 3

Automatic Tool Feasibility

The hooking library can be used in the following ways:

– developers embed the hooking library inside their applications during project phase;

– developers or users inject the hooking library inside target finished applications.

In this chapter, we evaluate the feasibility of an automatic tool that allows to permanently inject the hooking library inside target finished applications. We analyze some reverse engineering tools and strategies to verify the practicability of this idea.

3.1 Permanent injection of the hooking library inside target apk

Apktool is *"a tool for reverse engineering 3rd party, closed, binary Android apps"* [38]. It allows to decode Android APKs into source code, thus granting access to the application structure and functions.
The following shell command decompiles an APK into its DEX classes, eXtensible Markup Language (XML) files and global resources:

```
apktool d app.apk
```

where `d` stands for *decode* and `app.apk` is the target Android application. Apktool decoding and recompiling processes are based on *Smali* and its counterpart *baksmali* [39], which are assembler and disassembler, respectively, for the dex format used by Dalvik, Android's Java VM. Based on Jasmin [40] syntax, they allow to modify Android applications structure and behaviour. Apktool exploits *smali/baksmali* to decompile the application package into editable intermediate files and then it recompiles them into a new signable application. Working at this intermediate level allows to edit the application logic without altering its functionality. It is thus possible to deploy a recompiled application that basically executes the same code as the original one, but also with new features on top of it. The automatic tool should:

1. decompile the target apk;
2. copy the hooking library `.so` or package inside the decompiled app source folders;
3. edit the app source code to add the import line;
4. recompile the app;
5. generate a keystore (optional);
6. resign the app with the generated keystore or an existing one.

All of these actions can be automated by a script that exploits apktool, a shell text editor, *keytool* and *jarsigner*. The first one enables reverse engineering, while keytool and jarsigner are tools for creating keystores and signing apks, respectively. A keystore can be generated with the following command:

```
keytool -genkey -v -keystore hooklib.keystore -alias
   hooklib -keyalg RSA -keysize 2048 -validity 10000
```

After creating a keystore, the new apk can be signed with jarsigner:

```
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -
   keystore hooklib.keystore app.apk hooklib
```

The automatic tool should also allow to use an already existing keystore: the keystore path could be specified as a parameter of the process command. For example:

```
hooklibtool -keystore user.keystore targetapp.apk
```

We do not go any further with this analysis, but we conclude that the idea of an automatic tool is practicable, and we leave it as an open challenge for future works.

Chapter 4

Related Work

To the best of our knowledge, the hooking library is the first library for Android projects that dynamically verifies if the app shows compliance to the security standards discussed in chapter 1. Nonetheless, we acknowledge that this library is not sufficient to ensure full protection from adversaries' malicious intent. There exist many other researches and tools with background and goals similar to our project's. Developers should always consider multiple state of the art options and possibly combine them, adding extra layers of security to their apps.

AndroTotal [41] allows to scan Android apps against an arbitrary set of malware detectors, and it is publicly available as a web service. TraceDroid [42] is another tool for automated analysis which emulates a few actions, *"such as user interaction, incoming calls and SMS messages"* to trigger the app's malicious behaviour. Androwarn [43], similar to the hooking library in functionalities, tries to detect geolocation information leakage, telephony services abuse, external memory operations and many other malicious activities. Androguard [44] can disassemble Dalvik bytecode back to Java source code, detect repackaged apps or known malwares and retrieve useful information about the app integrity from the app *manifest*[1]. However, all of these

[1] *"The app manifest file provides essential information about your app to the Android system, which the system must have before it can run any of the app's code"* [45].

projects have different approaches to security. For example, Androwarn and Androguard use static analysis to scan the app data flow and they generate a security report. Differently, the hooking library is a dynamic analyzer operating at runtime. As explained in Chapter 1 (p. 5), dynamic analysis have the advantage of detecting hidden malicious behaviour loaded with Reflection & DCL.

Moreover, these related projects do not address the risk of using methods that have been deprecated for security reasons: they just focus on intentionally malicious code. Instead, the goal of this work was also to search for weak programming practices and vulnerabilities inside the app. The only tool we found in the literature that deals with code vulnerabilities is Quick Android Review Kit (QARK) by LinkedIn. QARK is a tool designed to *"look for several security related Android application vulnerabilities, either in source code or packaged APKs"* [46]. It is based on Python and it uses static analysis to generate reports on the app flaws. Among the vulnerabilities that can be detected by QARK: improper `x.509` certificate validation, private keys hardcoded in the source and, just like our hooking library but more exhaustively, creation of world-readable or world-writeable files and use of outdated API with known vulnerabilities.

We have analyzed tools with dynamic support and tools that aim at detecting weak programming practices, but we found very few works that accomplished both goals. For example, Hooker can dynamically intercept and modify API calls made by the target application. However, it is based on Cydia Substrate [47], which means that it can work on rooted devices only, thus violating our design restrictions. Besides, since Cydia Substrate is not compatible with Android 4.4 and higher versions, Hooker compatibility is limited to Android 4.3 as well. Frida [48] is a multi-platform hooking tool supporting many architectures, such as x86 and ARM (both 32 and 64 bit versions). It uses a JavaScript runtime framework as an interface for the underlying hooking engine written in C. Frida consists of approximately 250.000 Lines Of Code (LOC). Including such a huge framework inside a mobile project necessarily adds a noticeable overhead. Besides, Frida offers many functions that are not strictly related to hooking Java methods and system calls, which means that most of its code would be superfluous. To make a comparison, the hooking library is below 5.000 LOC, including ADBI and Legend. Some Android security solutions combine Client side and Server side analysis to reduce performance overhead on the user device and to reach maximum protection. Examples of analysis that can be performed on the server side are:

– maintaining an updated blacklist of known malicious or vulnerable libraries, and flagging the app as dangerous if it contains any library from this list[2] [49];

– if the app code is not obfuscated, disassembling the app and inspecting calls to libraries in its bytecode[3];

– checking the entropy of class names, method names and variable names to obtain information about obfuscation quality[4];

– checking the presence of executables such as `.sh`, `.exe`, `.elf` and `.so` inside the app package. A malicious app can execute code from such files.

Delosières and García [50] propose a security infrastructure that combines static and dynamic analysis: static analysis is provided by Androguard, while dynamic analysis is provided by DroidBox [51]. *"Both analyzers work jointly in order to extract as many Android characteristics as possible"* [50]. Static and dynamic analysis combination is also investigated by Spreitzenbarth et

[2]Alternatively, the server could keep an updated whitelist of libraries that can be used by the app. This strategy is more suitable for ad-hoc projects, since it would be impossible to draw up a global list of all legitimate libraries.

[3]For obfuscated code, we must rely on the Client side runtime analysis.

[4]The primary motivation of using obfuscation is to protect the app from being reverse engineered by adversaries, which may want to attack the app after understanding its weaknesses, or to plagiarize the app. A weak obfuscation may result in the app being exploited by adversaries.

al. [52]: a static and dynamic analyzer is merged with machine learning techniques to support malware analysts in detecting malicious behaviour.

The strength of the hooking library is to provide a way to perform runtime analysis also on areas where static analysis is generally used. For this reason, we believe the hooking library *does* enhance the app security by adding an extra layer of monitoring, and it represents a good choice for Android security, especially when combined with other static and dynamic analysis tools.

Chapter 5

Conclusions & Future Work

In this work, we have designed and implemented a security library for Android applications exploiting the hooking of Java and native functions to enable runtime analysis. The library verifies if the application contains malicious code or weak programming practices that might threaten the user privacy. Testing of the library showed that it successfully intercepts the targeted functions and it blocks the application malicious behaviour in the majority of cases. We have also assessed the feasibility of an automatic tool that uses reverse engineering to decompile the application, inject our library and recompile the security-enhanced application. We have identified a possible strategy for its implementation, and we have concluded that the idea is perfectly practicable.

However, our hooking library does not come without limitations or open challenges. First, it is compatible with ARM-32 architectures only. A significant improvement would be porting the project to ARM-64 architectures. At the moment, ARM-32 is more popular than ARM-64, but the latter is recently gaining more and more importance among manufacturers, and it is likely to become very common in the near future. Since ADBI and Legend run on ARM-32 architectures only, we should rewrite part of their C code, as well as the assembly code to be injected into the app memory for function redirection. To achieve this, a detailed study on ARM-32 and ARM-64 architectures instruction sets would be essential. We should understand the way registers are used and how exactly memory is accessed. We have briefly investigated this and we have found that the major difference between the two architectures is in the size and in the number of registers. The use of the stack and the way memory is accessed are different. However, most of the assembly instructions are similar. Therefore, porting the hooking library to ARM-64 while keeping ADBI and Legend as the core for function redirection is possible, but very expensive in terms of *know-how* and time. A possible alternative is to replace ADBI and Legend with tools already set up for ARM-64 compatibility. Samsung has realized its own version of ADBI [53] and it seems that both ARM architectures are supported. ElfHook [54] is another tool that could enable function hooking for ARM-64. Even though more analysis is required, we believe that realizing the hooking library for ARM-64 architectures is a concrete possibility, and we consider it the next major work in this research.

Secondly, TLS analysis should be investigated further. Advanced hackers can force insecure connections on port 443 to bypass our main protocol verification based on the connection port number. The goal here is to retrieve certificates, cipher suites and encryption algorithms data, and to check if they show anomalies that could suggest malicious intent. Weak or outdated encryption API should be avoided as well.

Also, our list of methods that have been deprecated for security reasons is limited and it should be extended. It is not a simple task and it could require a lot of research, but it is definitely something that should be realized in the future. LinkedIn QARK source code could provide some useful information to simplify the analysis work.

Moreover, we should find a way to extend the project compatibility to Android N. The hooking core must probably be reengineered because of N stricter restrictions on the type of libraries that can be loaded, but we can work on it and, eventually, find a solution.

Finally, implementing the reverse engineering tool *hooklibtool* for which we provide a basic analysis in Chapter 3 would be the perfect way to bring this research to its final stage.

References

[1] Android.
https://developer.android.com/guide/platform/index.html.
Accessed: 2016-11-12.

[2] StatCounter, “Mobile and tablet internet usage exceeds desktop for first time worldwide.”
http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide.
Accessed: 2016-12-01.

[3] University of Cambridge, “Computer viruses and other malware: what you need to know.”
http://www.ucs.cam.ac.uk/security/malware.
Accessed: 2016-12-01.

[4] Check Point Software Technologies Ltd.
https://www.checkpoint.com/about-us/facts-a-glance/.
Accessed: 2016-10-04.

[5] Check Point Software Technologies Ltd., “September’s ‘most wanted’ malware list.”
http://blog.checkpoint.com/2016/10/21/septembers-top-wanted-malware-list-ransomware-top-3-first-time/, 2016.
Accessed: 2016-10-04.

[6] F-Secure, "Trojan:Android/Koler Threat Description."
https://www.f-secure.com/v-descs/trojan_android_koler.shtml.
Accessed: 2016-10-04.

[7] S. Dengre and R. Kaushal, "Privilege Escalation Attacks in Android: Their Approaches, Detection and Defense Techniques."
http://cerc.iiitd.ac.in/spsymp15/papers/25.pdf.
Accessed: 2016-09-11.

[8] United States Computer Emergency Readiness Team (US-CERT), "Security Tip (ST04-015): Understanding Denial-of-Service Attacks."
https://www.us-cert.gov/ncas/tips/ST04-015.
Accessed: 2016-12-01.

[9] OWASP, "Use of Obsolete Methods."
https://www.owasp.org/index.php/Use_of_Obsolete_Methods.
Accessed: 2016-09-10.

[10] Lookout, Inc., "AndroRATIntern: A Japanese Mobile Threat With Global Implications for Mobile Data Security."
https://info.lookout.com/rs/051-ESQ-475/images/Lookout_AndroRATIntern_Whitepaper_v2.1_10-31-2016.pdf, 2015.
Accessed: 2016-11-12.

[11] T. Strazzere, Lookout, Inc., "Update: Android Malware DroidDream: How It Works."
https://blog.lookout.com/blog/2011/03/02/android-malware-droiddream-how-it-works/, 2011.
Accessed: 2016-11-12.

[12] Symantec Corporation, "Android.Spitmo."
https://www.symantec.com/security_response/writeup.jsp?docid=2011-091407-1435-99.
Accessed: 2016-07-29.

[13] F-Secure, "Trojan:Android/BaseBridge.A Threat Description."
`https://www.f-secure.com/v-descs/trojan_android_basebridge.shtml`.
Accessed: 2016-09-10.

[14] Trend Micro Incorporated, "AndroidOS_FakeNotify.A."
`http://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/androidos_fakenotify.a`.
Accessed: 2016-09-10.

[15] Y. Zhou and X. Jiang, "An Analysis of the AnserverBot Trojan."
`https://www.csc.ncsu.edu/faculty/jiang/pubs/AnserverBot_Analysis.pdf`.
Accessed: 2016-11-12.

[16] M. Ahmad, B. Crispo, and T. Gebremichael, "Empirical Analysis on the Use of Dynamic Code Updates in Android and Its Security Implications," in *Nordic Conference on Secure IT Systems*, pp. 119–134, 2016.

[17] W.B. Frakes and Kyo Kang, "Software Reuse Research: Status and Future," in *IEEE Transactions on Software Engineering*, vol. 31, p. 529–536, 7 2005.

[18] Android, "Android NDK."
`https://developer.android.com/ndk/index.html`.
Accessed: 2016-11-12.

[19] W. You, B. Liang, W. Shi, S. Zhu, P. Wang, and S. Xie, "Reference Hijacking: Patching, Protecting and Analyzing on Unmodified and Non-Rooted Android Devices," in *ICSE '16 Proceedings of the 38th International Conference on Software Engineering*, pp. 959–970, 2016.

[20] M. Kerrisk, "Linux Programmer's Manual - ptrace."
`http://man7.org/linux/man-pages/man2/ptrace.2.html`.
Accessed: 2016-11-12.

[21] N. Kralevich, Android Platform Security Engineering Lead/Manager, “Permission Changes for ptrace in Kitkat.”
`http://android-security-discuss.narkive.com/mttOQjz2/permission-changes-for-ptrace-in-kitkat`.
Accessed: 2016-11-12.

[22] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna, “NJAS: Sandboxing Unmodified Applications in non-rooted Devices Running stock Android,” in *5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2015.

[23] V. Costamagna and C. Zheng, “ARTDroid: a virtual-method hooking framework on Android ART runtime,” in *Proceedings of the Workshop on Innovations in Mobile Privacy and Security IMPS at ESSoS’16* (D. Aspinall, L. Cavallaro, M. N. Seghir, M. Volkamer, ed.), 2016.

[24] C. Mulliner, “ADBI.”
`https://github.com/crmulliner/adbi`.
Accessed: 2016-11-22.

[25] *asLody*, “Legend.”
`https://github.com/asLody/legend`.
Accessed: 2016-11-22.

[26] Paul Mozur and Su-Hyun Lee for *The New York Times*, “Samsung to Recall 2.5 Million Galaxy Note 7s Over Battery Fires.”
`http://www.nytimes.com/2016/09/03/business/samsung-galaxy-note-battery.html`.
Accessed: 2016-11-22.

[27] A. Dabrowski, N. Pianta, T. Klepp and M. Mulazzani and E. Weippl, “IMSI-Catch Me If You Can: IMSI-Catcher-Catchers,” in *ACSAC ’14 Proceedings of the 30th Annual Computer Security Applications Conference*, pp. 246–255, 12 2014.

[28] US Department of Transportation, "Global Positioning System (GPS) Civil Monitoring Performance Specification."
`http://www.gps.gov/technical/ps/2009-civil-monitoring-performance-specification.pdf`.
Accessed: 2016-11-24.

[29] Carnegie Mellon University, Software Engineering Institute (SEI) CERT, "MET02-J. Do not use deprecated or obsolete classes or methods."
`https://www.securecoding.cert.org/confluence/display/java/MET02-J.+Do+not+use+deprecated+or+obsolete+classes+or+methods`.
Accessed: 2016-11-15.

[30] Android, "ActivityManager, getRecentTasks()."
`https://developer.android.com/reference/android/app/ActivityManager.html#getRecentTasks(int,int)`.
Accessed: 2016-11-25.

[31] Symantec Corporation, "Malware may abuse Android's accessibility service to bypass security enhancements."
`https://www.symantec.com/connect/blogs/malware-may-abuse-android-s-accessibility-service-bypass-security-enhancements`. Accessed: 2016-11-15.

[32] Android, "Context."
`https://developer.android.com/reference/android/content/Context.html#MODE_WORLD_READABLE`.
Accessed: 2016-11-20.

[33] Android, "Context, openFileOutput()."
`https://developer.android.com/reference/android/content/Context.html#openFileOutput(java.lang.String,int)`.
Accessed: 2016-11-30.

[34] Android, “Security with HTTPS and SSL.”
https://developer.android.com/training/articles/security-ssl.html.
Accessed: 2016-11-14.

[35] Android, “URLConnection.”
https://developer.android.com/reference/java/net/URLConnection.html.
Accessed: 2016-11-14.

[36] B. Hall, “Beej’s Guide to Network Programming, Using Internet Sockets.”
http://beej.us/guide/bgnet/output/html/multipage/sockaddr_inman.html.
Accessed: 2016-07-22.

[37] Microsoft, “ntohs function.”
https://msdn.microsoft.com/it-it/library/windows/desktop/ms740075(v=vs.85).aspx.
Accessed: 2016-11-16.

[38] C. Tumbleson, “Apktool.”
http://ibotpeaches.github.io/Apktool/.
Accessed: 2016-07-22.

[39] J. Freke, “smali/baksmali.”
https://github.com/JesusFreke/smali.
Accessed: 2016-07-22.

[40] J. Meyer and D. Reynaud, “Jasmin.”
http://jasmin.sourceforge.net/.
Accessed: 2016-07-13.

[41] F. Maggi, A. Valdi, and S. Zanero, “AndroTotal: A Flexible, Scalable Toolbox and Service for Testing Mobile Malware Detectors,” in *3rd*

Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM), 2013.
`http://www.syssec-project.eu/m/page-media/3/spsm07s-maggi.pdf`.

[42] V. Van Der Veen, "TraceDroid: A Fast and Complete Android Method Tracer."
`https://www.owasp.org/images/7/7c/TraceDroid.pdf`.

[43] T. Debize, "Androwarn, Yet another static code analyzer for malicious Android applications."
`https://github.com/maaaaz/androwarn`.
Accessed: 2016-07-28.

[44] A. Desnos and G. Gueguen, "Android: From Reversing to Decompilation," in *Proceedings of the Black Hat Conference, Operational Cryptology and Virology Laboratory*, 7 2011.

[45] Android, "App Manifest."
`https://developer.android.com/guide/topics/manifest/manifest-intro.html`.
Accessed: 2016-12-02.

[46] LinkedIn, "Quick Android Review Kit (QARK)."
`https://github.com/linkedin/qark/`.
Accessed: 2016-11-28.

[47] SaurikIT, LLC, "Cydia Substrate, The powerful code modification platform behind Cydia."
`http://www.cydiasubstrate.com/`.
Accessed: 2016-12-02.

[48] NowSecure, "Frida: Inject JavaScript to explore native apps on Windows, Mac, Linux, iOS, Android, and QNX."

http://www.frida.re/.
Accessed: 2016-12-02.

[49] J.-H. Hoepman and S. Katzenbeisser, "ICT Systems Security and Privacy Protection," in *31st IFIP TC 11 International Conference, SEC 2016*, (Ghent, Belgium), 6 2016.

[50] L. Delosières and D. Garciía, "Infrastructure for Detecting Android Malware," in *Information Sciences and Systems 2013, Proceedings of the 28th International Symposium on Computer and Information Sciences*, pp. 389–398, Springer International Publishing, 2013.

[51] P. Lantz and A. Desnos, "Droidbox."
https://github.com/pjlantz/droidbox.
Accessed: 2016-12-02.

[52] M. Spreitzenbarth, T. Schreck, F. Echtler, D. Arp and J. Hoffmann, "Mobile-Sandbox: combining static and dynamic analysis with machine-learning techniques," in *Int. J. Inf. Secur.*, pp. 141–153, Springer Berlin Heidelberg, 4 2015. doi:10.1007/s10207-014-0250-0.

[53] Samsung, "ADBI."
https://github.com/Samsung/ADBI.
Accessed: 2016-11-22.

[54] *asLody*, "ElfHook."
https://github.com/asLody/ElfHook.
Accessed: 2016-11-22.