BERZIET UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

A framework for developing Secure Android Apps

Prepared by

Khaled Rimawi

Ahmad Al-Khatib

Muath Kefayah

Supervised by

Ahmad Afaneh

A graduation Project submitted to the Department of Electrical
and Computer Engineering in partial fulfillment of the requirements
for the degree of B.Sc. in Computer Engineering

BIRZEIT

Abstract

Android Operating system has many security vulnerabilities which make the extraction of applications' source codes possible using reverse engineering tools. Extracted source code can be reused for malicious or profitability purposes. Android OS suffers from vulnerabilities such as permission Escalation attack that allows malicious application to access critical resources without user permission [1]. In addition, Dangerous Permissions are another vulnerability that gives the attacker an access to the user's confidential data. In general, the idea for android being open source is a source of vulnerabilities; attackers can analyze each line of code to determine its weaknesses. In addition, it is possible to install the applications from unknown sources, which can contain Malware apps [2]. In this project, we aim to design a development framework that make Android applications more immune against reverse engineering and tampering. In our design, we will implement anti-debugging methods for anti-analysis purpose, and root detection techniques to detect if device has administration permission, as root access can be used for harmful purposes. Furthermore, we will apply many code obfuscation techniques for more security and anti-analysis purposes. Moreover, we will use client server methodology that executes the stored part of code on the server or it can be later downloaded partially when needed then removed. This method distinguishes our framework from other frameworks. Finally, this design doesn't make reverse engineering impossible, it makes it much harder.

المستخلص

يحتوي نظام تشغيل الاندرويد على ثغرات أمنية عديدة تجعل استخلاص كود التطبيق الاصلي ممكنا باستخدام ادوات الهندسة العكسية. الكود المستخلص ممكن أن يتم اعادة استخدامه لاغراض ربحية أو خبيثة. نظام تشغيل الاندرويد يعاني من عيوب مثل تصعيد هجوم الإذن الذي يسمح للتطبيقات الخبيثة بأن تولَج إلى المصادر الهامة بدون اذن المستخدم. بالاضافة إلى ذلك ،الأذونات الخطيرة تعد عيبا آخرا، حيث تعطي للمهاجم ولوجا إلى البيانات السرية . بشكل عام ، فكرة نظام الاندرويد بأنه مصدر مفتوح يعد مصدر عيوب. بالاضافة إلى ذلك ، من الممكن تنزيل التطبيقات من مصادر مجهولة وغير موثوقة والتي من الممكن أن تحتوي على تطبيقات خبيثة . و في هذا المشروع ، نحن نهدف لتصميم إطار برمجي يجعل تطبيقات الاندرويد أكثر تحصينا ضد الهندسة العكسية والتلاعب. و في تصميمنا هذا ، سنقوم ببناء طرق ضد تتبع البرنامج من أجل منع هدف تحليل وظيفة التطبيق. بالإضافة إلى طرق تكشف الروت والذي من خلاله يتم الوصول إلى ملفات النظام والذي يعد أمرا خطيرا. أيضا سنقوم بتطبيق طرق تقوم بتشويش الجمل البرمجية لهدف منع التحليل. علاوة على ذلك , نحن سوف نستخدم منهجية الخادم والعميل التي تقوم بتنفيذ جزء من الأوامر البرمجية المخزنة على الخادم أو ليتم تنزيلها لاحقا أثناء تنفيذ التطبيق ومن ثم يتم حذفها. هذا المنهج يميز إطار عملنا عن الأطر الأخرى .
أخيرا , هذا التصميم لا يجعل الهندسة العكسية أمرا مستحيلا ولكنه يصعبها قدر الإمكان.

TABLE OF CONTENTS **page**

TABLE OF FIGURES page

TABLE OF TABLES page

Acronyms and Abbreviations

SRE	Software Reverse Engineering
iOS	iPhone OS
APK	Android Package Kit
JAR	Java Archive
MDM	Mobile Device Management
SU	Super User
API	Application Programming Interface
DEX	Dalvik Executable
JWS	JSON Web Signature
JDWP	Java Platform Debugger Architecture
JVM	Java Virtual Machine
IDE	Integrated Development Environment
MVC	Model View Controller
JSON	JavaScript Object Notation

Chapter 1 Introduction

Software Reverse Engineering (SRE) is the practice of analyzing a software system, either in whole or in part, to extract design and implementation information. SRE may have a protection goals or malicious goals. Reverse Engineering skills are used to detect and neutralize viruses and malware, it is also can be used for software theft and reuse. [3]

1.1 Motivation

Android Operating system has many security vulnerabilities, which make the extraction of applications source code possible using reverse engineering tools. The extracted code can be manipulated and tampered after analysis, and then it can be reused for malicious purposes after adding harmful code. After creating the malware application, it can be downloaded as the original application, which is not. Moreover, some application functions can be vulnerable to theft and thus can be used in other applications. Therefore, we aim to design and implement a development framework to be used by Android applications developers to make their applications more immune to reverse engineering and license tampering.

1.2 Why Android and not iOS

Protecting a software from reverse engineering is hard in general. Protecting Android applications from not being reversed is harder than iOS due to many reasons, the fundamental reason is that Android is an open source, anyone can look what's inside it and try to analyze and understand it, unlike iOS, which is not. This makes Android more vulnerable to malware attacks. Other reason is that the Java bytecode is especially easy to reverse engineer. For example, the source code can be recovered by converting the APK to JAR file, then a java de-compiler can be used to generate the code for the application. Moreover, in Android, other than google play store, it is possible to install the applications from unknown sources. But, in iOS, the apps can be only installed from AppStore. It is one of the major security breaches in Android. Due to various security breaches in Android, attackers already regard smartphone as the target to steal personal information using various malware. [4]

1.3 The Structure of the Report

The next chapters talk about background and related work, in addition to design and future work. In background and related work chapter, , after that we will talk about the anti-reverse Engineering techniques that are used such as root and debug detection. In addition, we will talk briefly about obfuscation and the most famous techniques for it. In the next section, we will clarify the most popular framework for android applications such as ProGuard and DexGuard. In design chapter, we will present a design of our framework structure then describe its features and options it offers and how it will be implemented. In final chapter, we will talk about future work and conclude our report.

1.4 Android Security Issues and attacks

In order to strengthen Android against reverse engineering, we must understand it's security issues first and understand the attacks it is experiencing.

1.4.1 General Security Issues

In the latter part of 2010 and early 2011, a vulnerability issue was discovered in Android versions 2.2 and 2.3, respectively. The vulnerability is that an attacker can copy any file that is stored on the device's SD Card without granting a permission or even without a visible cue that this is happening. Moreover, the idea for android being open source itself is a problem; attackers can analyze each line of code to determine its weaknesses. In addition, Google play store is a bit of concern because of the relative ease of getting apps approved for sale. Malware apps can squeak through. In Android, it is possible to install the applications from unknown sources, like third-party android stores. It is one of the major security breaches in Android. [2] [5]

1.4.2 Permission Escalation Attack

Android's application-level security framework is based on permission labels, where permission label is simply a unique text string that can be defined by both the OS and third-party developers. Permission is granted to the application by the user at the installation time, it gives the app a specific resource access. Permission escalation attack allows a malicious application to collaborate with other applications to access critical resources without requesting for corresponding permissions explicitly [1]. The permission escalation attack is classified into two categories:

- **Collision Attack**

  Collision attack is a technique wherein two or more application share the same user ID so that they can access the permissions, which are granted to each other. For example. If application A has permissions to READ_CONTACTS, READ_PHONE_STATUS and B has permissions to READ_MESSAGES, LOCATION_ACCESS, if both the applications use the same user id SHAREDUSERID, then it is possible for application A to use the permissions granted to itself and the permissions granted to B. Similarly, it is possible for application B to use the permissions granted to itself and the permissions granted to A. Every Android application has unique ID that is its package name. Android supports shared User ID. It is an attribute in AndroidManifest.xml file. If this attribute assigned with the same value in two or more applications, then they can access permissions granted to each other. [1]

- **Confused Deputy Attack**

  Confused deputy attack and collusion attack. The confused deputy attack exploits the vulnerabilities in unprotected interfaces of privileged benign. As shown in Fig 1-1, the application A1 is not granted with the permission P1; C1, which is a component of A1, cannot directly access system resource R1 protected by permission P1. However, C1, can access R1 transitively if application A2 is granted with permission P1 and one of A2 's component, C2 does not require any permission to be accessed. As a result, C1 can access R1 through C2 and C3. [6]

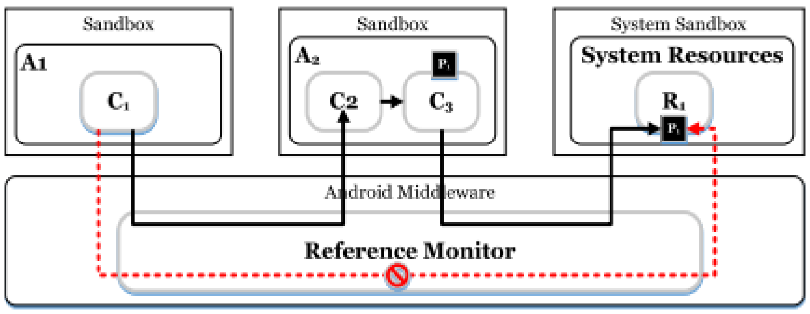


Figure 1-1: Confused deputy attack

1.4.3 Dangerous Permissions

Dangerous Permissions can access critical resources of the mobile. Dangerous permissions can give the app access to the user's confidential data. If app lists a normal permission in its manifest, the system grants the permission automatically. If app list a dangerous permission, the user has to explicitly give approval for the app for the successful installation of the app [1]. Table 2-1 describes all dangerous permissions.

Table 10-1: Dangerous permissions Description [7]

dangerous permissions	Description
READ_CONTACTS	Allows an application to read the user's contacts data
WRITE_CONTACTS	Allows an application to write (but not read) the user's call log data
GET_ACCOUNTS	Allows access to the list of accounts in the Accounts Service
ACCESS_FINE_LOCATION	Allows an app to access precise location
ACCESS_COARSE_LOCATION	Allows an app to access approximate location
SEND_SMS	Allows an application to send SMS messages
RECEIVE_SMS	Allows an application to receive SMS messages
READ_SMS	Allows an application to read SMS messages
RECEIVE_WAP_PUSH	Allows an application to receive WAP push messages
RECEIVE_MMS	Allows an application to monitor incoming MMS messages
READ_EXTERNAL_STORAGE	Allows an application to read from external storage.
WRITE_EXTERNAL_STORAGE	Allows an application to write to external storage.

Chapter 2 Background and Related Work

2.1 Techniques used for Anti-Reverse Engineering

2.1.1 Root Detection

Android File System and Directory Structure

Windows file systems as a file layout are familiar for most people and they are happy navigating it. for each physical drive or partition Windows uses a letter e.g. D: drive. A physical drive can have at least one partition. This applies on both Windows and Android. In windows, each disk partition has a root directory , e.g. the root directory of partition C is C:/.

Linux file system structure is used in android and it only has a single root. The (figure 2-1) below shows the structure: [8]

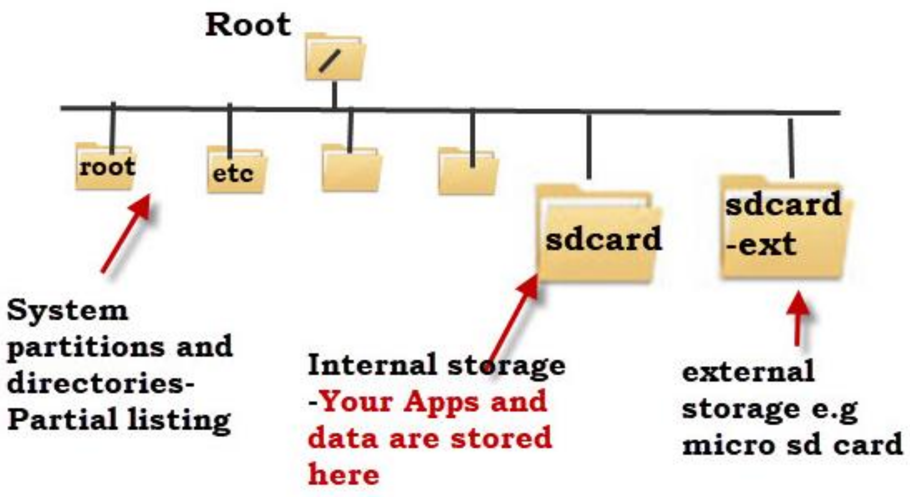

Root
/
root
etc
sdcard
sdcard -ext
System partitions and directories- Partial listing
Internal storage -Your Apps and data are stored here
external storage e.g micro sd card

Figure 2-1: Android File System Structure [8]

Unless your device is rooted (will be clarified later), the system files are protected and there is no access to these files. Disks and partitions are directories under the root, and unlike windows , they do not have a drive letter.

Root Access

Rooting in android is attaining privileged control (root access) over the subsystems, rooting Android devices gives superuser permissions as in Linux. It is done to exceed the limitations that hardware manufacturers put on the devices. Therefore, rooting gives the permission to modify system settings and applications, also it is done for running some specialized apps that require administration permission or perform operations that a normal Android user cannot do. In Android, rooting can be used for removing and replacing the device's OS (operating system). Root access user has the ability to use the permission escalation attack. [9]

Root Detection

Most Mobile Device Management (MDM) use similar solutions to detect rooted Android devices. Usually, this is done by looking for specific files and packages, directory permissions, and executing specific commands [10]

- **Default Files & Configurations**
  Non-rooted devices have default files and configurations that should present. Therefore, the first root detection checks them. Note that these files and configurations may present in rooted devices with non-custom ROMs.

1. **Check for the BUILD Tag for Test-keys**.
   If release-keys are not present and the test-keys are, this mean that the Android build is unofficial Google build or a developer build.

```
root@android:/ # cat /system/build.prop | grep ro.build.tags
ro.build.tags=release-keys
```

2. **Checking the Over the Air (OTA) Certifications**.
   By default, Google updates Android OTA using public certifications. If there is no certifications on the device, then probably a custom ROM is installed.

```
root@android:/ # ls -l /etc/security/otacerts.zip
ls -l /etc/security/otacerts.zip
-rw-r--r-- root     root         1733 2008-08-01 07:00 otacerts.zip
```

- **Installed Packages & Files**

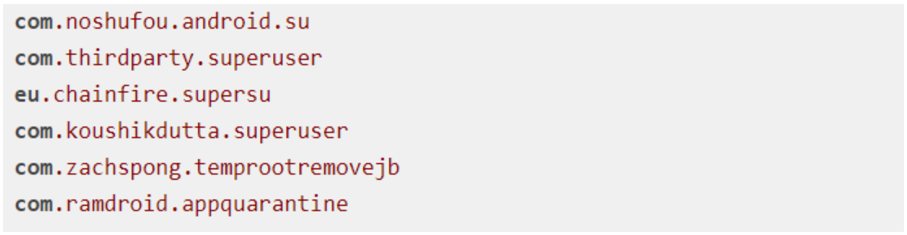
  MDMs look for many files and packages for detecting if there is a root access. The following list of files and packages ensures that the device is rooted.

  1. **Superuser.apk**

     This package is the most popular when looking for root access on the devices. Superuser allows the applications to run as root on the devices.

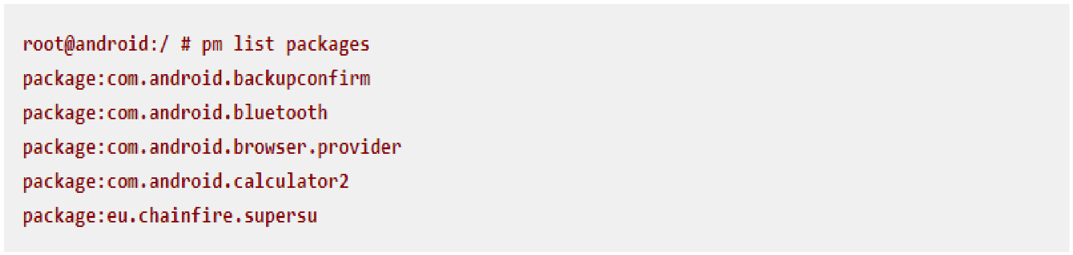
  2. **Other Packages**

     These packages are often looked for when detecting root access. The last two packages hides the su binary.



```
com.noshufou.android.su
com.thirdparty.superuser
eu.chainfire.supersu
com.koushikdutta.superuser
com.zachspong.temprootremovejb
com.ramdroid.appquarantine
```

  3. The currently installed packages on a rooted device are listed using ‘pm list packages’ command.



```
root@android:/ # pm list packages
package:com.android.backupconfirm
package:com.android.bluetooth
package:com.android.browser.provider
package:com.android.calculator2
package:eu.chainfire.supersu
```

  4. **Cyanogenmod.superuser**.

     When the Cyanogenmod ROM is installed, the com.android.settings package will contain the cyanogenmod.superuser activity. We can detect this by listing the activities within com.android.settings.

5. **Su Binaries**.

   Following directories often contain the SU binaries , so we check for them.

```
/system/bin/su
/system/xbin/su
/sbin/su
/system/su
/system/bin/.ext/.su
/system/usr/we-need-root/su-backup
/system/xbin/mu
```

- **Directory Permissions**

  Rooting Android device can change the permissions of common directories.

  1. **Checking for the following directories, if they are writable**

     /data
     /system
     /proc
     /dev
     /system/xbin
     /vendor/bin
     /system/bin
     /sys
     /system/sbin
     /sbin
     /etc

  2. **Checking the readability of /data directory**

     All installed applications are in the /data directory, by default this directory is not readable.

- **Commands**

  Some MDMs run the following commands to detect the root access.

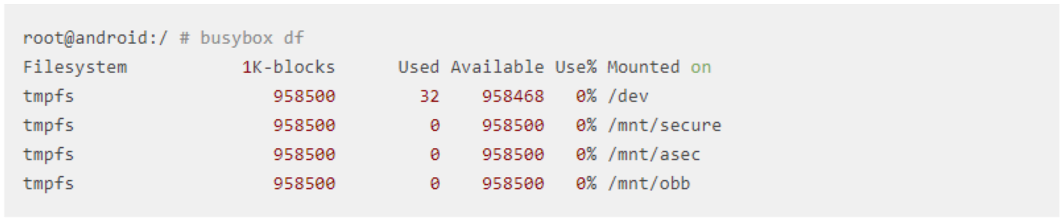
  1. **Su**

     If the device is rooted then uid will be 0 after running su command followed by id.

```
shell@android:/ $ su
shell@android:/ # id
uid=0(root) gid=0(root)
```

2. **Busybox**.

   Most likely, Busybox has not been installed as well on rooted android device. Busybox stores many Linux commands, so running it is a good method to detect root access.



```
root@android:/ # busybox df
Filesystem           1K-blocks      Used Available Use% Mounted on
tmpfs                   958500        32    958468   0% /dev
tmpfs                   958500         0    958500   0% /mnt/secure
tmpfs                   958500         0    958500   0% /mnt/asec
tmpfs                   958500         0    958500   0% /mnt/obb
```

- **SafetyNet**

  It is an Android API that creates a profile of the device using software and hardware information. This profile is then compared against a list of white-listed device models that have passed Android compatibility testing. SafetyNet is not well documented, and may change at any time: When you call this API, the service downloads a binary package containing the device validation code from Google, which is then dynamically executed using reflection. An analysis showed that the checks performed by SafetyNet also attempt to detect whether the device is rooted, although it is unclear how exactly this is determined.

To use the API, an application may use the **SafetyNetApi.attest()** method which returns a **JWS message** (it is information, along with proof that the device information hasn't changed since being signed) with the Attestation Result, and then check the following fields: [11]

- ctsProfileMatch: Of "true", the device profile matches one of Google's listed devices that have passed Android compatibility testing.
- basicIntegrity: Of "true", The device running the app likely wasn't tampered with.

The attestation result looks as follows.

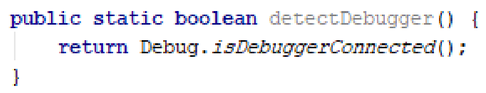
```
{
  "nonce": "R2Rra24fVm5xa2Mg",
  "timestampMs": 9860437986543,
  "apkPackageName": "com.package.name.of.requesting.app",
  "apkCertificateDigestSha256": ["base64 encoded, SHA-256 hash of the
                          certificate used to sign requesting app"],
  "apkDigestSha256": "base64 encoded, SHA-256 hash of the app's APK",
  "ctsProfileMatch": true,
  "basicIntegrity": true,
}
```

2.1.2 Anti-debugging

Java Platform Debugger Architecture (JDWP) is a protocol for communication between the debugger and the Java Virtual Machine (JVM) that it debugs. JDWP is a standard debugging protocol that's supported by all command line tools and Java IDEs, including JDB, JEB, IntelliJ, and Eclipse, A JDWP debugger allows you to step through Java code, set breakpoints on Java methods, and inspect and modify local and instance variables. JDWP debugger used most of the time when debug "normal" Android apps. The Android application package file, APK file, can be easily decompiled using Android reverse engineering tools. Thus, general apps can be easily transformed into malicious application through reverse engineering and analysis. These repacked apps could be uploaded in general android app market. To prevent theses malicious behaviors such as malicious code injection or code falsifications, many techniques and tools were developed. However, these techniques also can be analyzed using debuggers. Also, analyzed apps can be tampered easily. For example, when applying anti-analysis techniques to android apps using DexGuard, it can be seen that these techniques can also be analyzed using debugger. so, to protect the android app from the attack using debugger, we propose anti-debugging techniques for code and managed code debugging of android apps. [12]

- **Static Method**

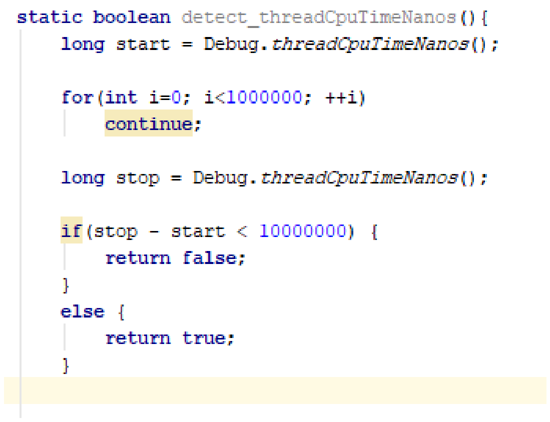
  The Android Debug system class offers a static method for checking whether a debugger is currently connected, we can use a class as shown below:



```
public static boolean detectDebugger() {
    return Debug.isDebuggerConnected();
}
```

- **Timer Checks**

  The "Debug.threadCpuTimeNanos" indicates the amount of time that the current thread has spent executing code. As debugging slows down execution of the process, the difference in execution time can be used to make an educated guess on whether a debugger is attached.



```
static boolean detect_threadCpuTimeNanos(){
    long start = Debug.threadCpuTimeNanos();

    for(int i=0; i<1000000; ++i)
        continue;

    long stop = Debug.threadCpuTimeNanos();

    if(stop - start < 10000000) {
        return false;
    }
    else {
        return true;
    }
```

- **Checking TracerPid**

  On Linux, the **"ptrace** ()" system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers, A straightforward way of using the ptrace system call for anti-debugging is forking a single child, and then calling **ptrace*(parent_pid)*** to attach to the parent. [13]

```
void anti_debug()  {

          child_pid =  fork();

          if (child_pid == 0)
          {
               int ppid =  getppid();
               int status;

               if(ptrace(PTRACE_ATTACH, ppid , NULL, NULL) == 0)
               {
                 waitpid(ppid, &status ,0);
                 ptrace(PTRACE_CONT , ppid ,NULL ,NULL);

                 while(waitpid(ppid, &status ,0)) {

                       if(WIFSTOPPED(status)) {
                          ptrace(PTRACE_CONT , pidd , NULL , Null);
                       }else {
                             // Process has exited for some reason
                             _exit(0);
                       }
                 }
               }
           }
    }
```

If implemented as above, the child will keep tracing the parent process until the parent exits, causing future attempts to attach a debugger to the parent to fail. We can verify this by compiling the code into a JNI function and packing it into an app we run on the device.

2.1.3 Code obfuscation

- **Definition of Obfuscation**

  Given a code, how can we make it hard to reverse-engineer it? This is one of major open problems concerning computer practice. Code obfuscation is the most viable method for preventing reverse-engineering [14] is a set of program transformations that make program code and/or program execution difficult to analyze, and it can hide certain properties such as a software fingerprint or a watermark, In the case of obfuscation, the 'key' can specify which transformations were performed, in what order, and on which section of the code. This key allows the software owner to reconstruct the code, these keys can be kept in a database until required for maintenance or analysis of bug reports.

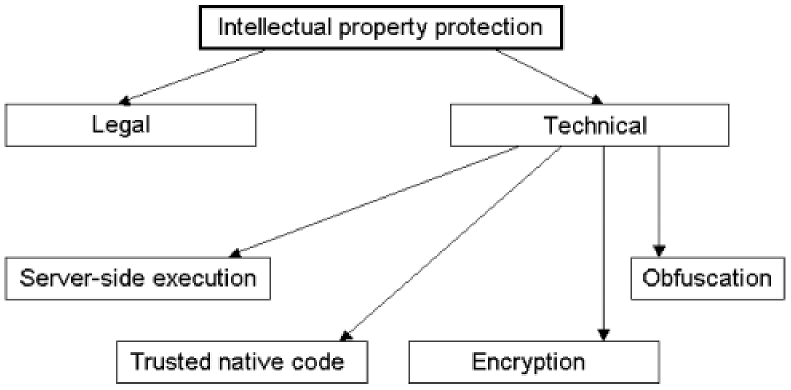
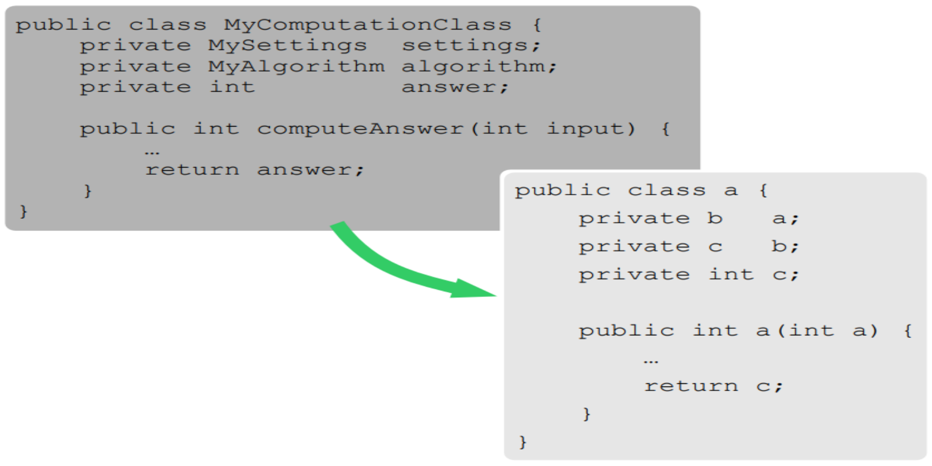


Figure 2-3: Methods of software protection [15]

- **Algorithms of Obfuscation**

1. Name Obfuscation

It is the process to replace identifiers with meaningless strings as new identifiers. Usually, the identifiers have meaning for a better recognition of the source code structures like classes, methods, variables and so forth. Once an identifier is renamed, it is mandatory to provide consistency across the entire application through replacements of the old names by the new identifiers. [16]



```
public class MyComputationClass {
    private MySettings   settings;
    private MyAlgorithm  algorithm;
    private int          answer;

    public int computeAnswer(int input) {
        …
        return answer;
    }
}
```

```
public class a {
    private b   a;
    private c   b;
    private int c;

    public int a(int a) {
        …
        return c;
    }
}
```

Figure 2-4: Name obfuscation example

2. Code Flow Obfuscation

It is the process to change the control flow in a software application. The changed control flow must lead to the same results as the initial one, but the software application is more difficult to be analyzed.

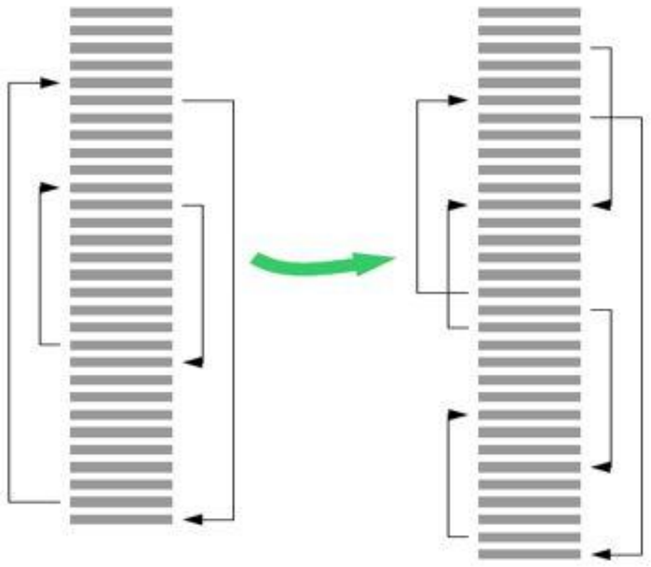


Figure 2-5: Control flow obfuscation [17]

3. Watermarking

It is the process to embed information in the software application. This information is used to prevent unauthorized software disclosure.

- **Static Watermarking**

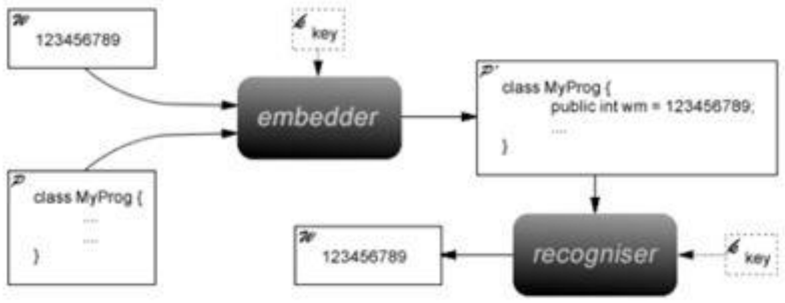


Figure 2-6: static watermarking [18]

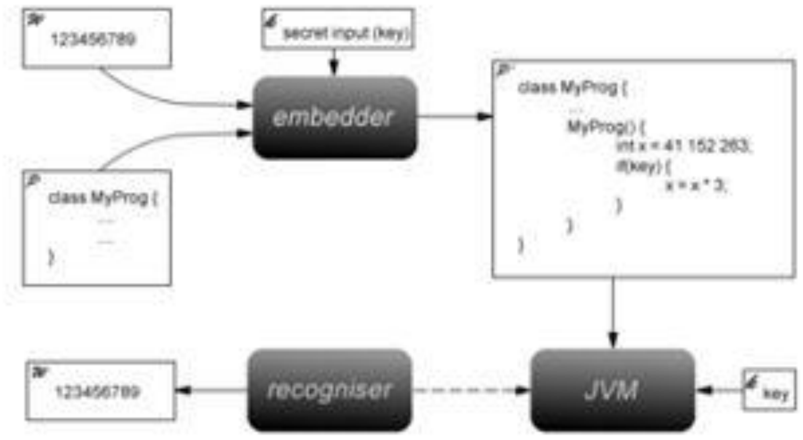
It’s a technique that adds obfuscation to program code.

- Adding redundant syntax to the program in which makes it more complex to understand (spaghetti code), but doesn’t change the logic, in addition, an ambiguous part of code could be added to prove the ownership of the author (it’s like a puzzle, solving it leads to a surprise).
- “Essential” parts of the program are steganographically encoded into the media. If the watermarked image is attacked, the embedded code will crash.

- **Dynamic Watermarking**

  Aims to:

  - Embeds code to generate a watermark at run-time
  - Recognizers uses a debugger to extract watermark
  - Should be resilient to semantics-preserving transformation



```
class MyProg {
    ...
    ...
}
```

```
class MyProg {
    ...
    MyProg() {
        int x = 41 152 263;
        if(key) {
            x = x * 3;
        }
    }
}
```

Figure 2-7: dynamic watermarking [18]

The most popular type of watermarking is **Easter Egg Watermarks** [19]**:** When the special input sequence is entered, it performs some action that is immediately perceptible by the user. Typically, a copy right message or an unexpected image is displayed.

2.2 Related Frameworks

2.2.1 ProGuard

ProGuard is a Java class file optimizer, obfuscator, shrinker and preverifier. It removes the unused classes in the shrinking step, and fields, methods, and attributes. The obfuscation step gives the classes, methods, and fields meaningless names. In the optimization step, it optimizes the bytecode. All of These steps make the code harder for analysis. The final preverification step adds preverification information to the classes. All of these features are optional i.e. some of them may not be used. Fig. 2-7 shows the ProGuard process.

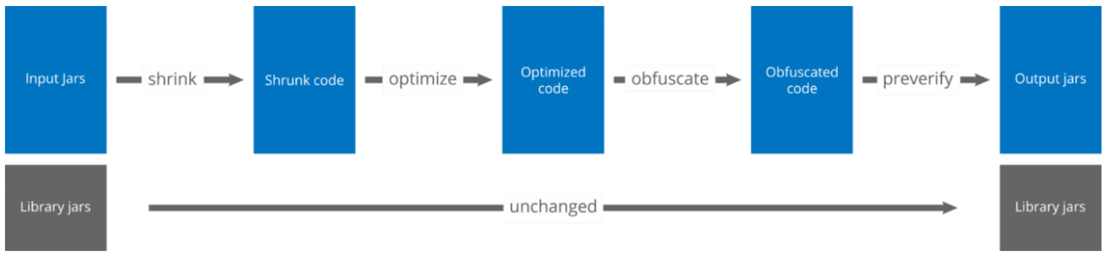


Figure 2-8: ProGuard process

At the beginning, ProGuard reads the **jars** (apks, zip files). Then it shrinks, optimizes, obfuscates, and preverifies them. ProGuard writes the results to one or more output jars.

2.2.2 DexGuard

DexGuard is a commercial optimizer and obfuscator tool written by Eric Lafortune (who developed ProGuard), it is based on ProGuard and It is used in the place of ProGuard. Rather than targeting Java, DexGuard is specialized for Android resources and Dalvik bytecode.
DexGuard shields applications from both static and dynamic analysis by using runtime security mechanisms that check the integrity of the application and of the environment in which it is running and enables the application to react whenever suspicious activity is detected. It also Offers name obfuscation, arithmetic and logical expressions in the code and the control flow of the code inside methods, also it offers encryption of strings and classes and adds reflection to access-sensitive API's.

2.2.3 ProGuard vs DexGuard

Regarding android, it is more secure to use DexGuard, given that it provides additional security features than ProGuard. Table 2-2 shows the differences between both. [21]

Table 2-2: ProGuard vs DexGuard

	ProGuard	DexGuard
Optimization	It is a versatile optimizer for Java bytecode, it enables you to shrink, optimize and obfuscate desktop applications, embedded applications and mobile applications (Android).	It is specifically designed to protect and optimize Android applications, It offers functionality that helps you to make optimal use of the Android platform. It comes with a tuned configuration for the Android runtime and for common libraries (Google Play Services, Dagger, Realm, SQLCipher etc.) and automatically splits Dex files that exceed the size limits imposed by the format (MultiDex).
Code protection	It offers basic protection against static analysis only by harden the source code of the application using a multitude of obfuscation and encryption techniques	It shields applications from both static and dynamic analysis by using runtime security mechanisms that check the integrity of the application and of the environment in which it is running and enables the application to react whenever suspicious activity is detected.
Obfuscation	It offers basic protection in the form of name obfuscation by obfuscate names of classes, fields and methods.	It Offers name obfuscation, arithmetic and logical expressions in the code and the control flow of the code inside methods, also it offers encryption of strings and classes and adds reflection to access-sensitive APIs

Table 2-2, continued

Protection area	ProGuard's action is restricted to the bytecode of Java applications	It provides 360-degree protection. Besides the Dalvik bytecode, it optimizes, obfuscates and encrypts manifest files, native libraries, resources, resource files and asset files.
Price	Free, open source tool.	commercial, enterprise-grade product

Chapter 3 Design

An anti-reverse engineering software will be programmed and deployed on a Web server, this software applies all the methods that make it harder for the android application to be reversed.

3.1 Design Flow

The software is a website where users upload a zipped android source code file and download APK with the framework library integrated in. After uploading the source code, the software will apply its features that will be explained in the next section to produce APK as output; these features are shown in Fig 3-1.

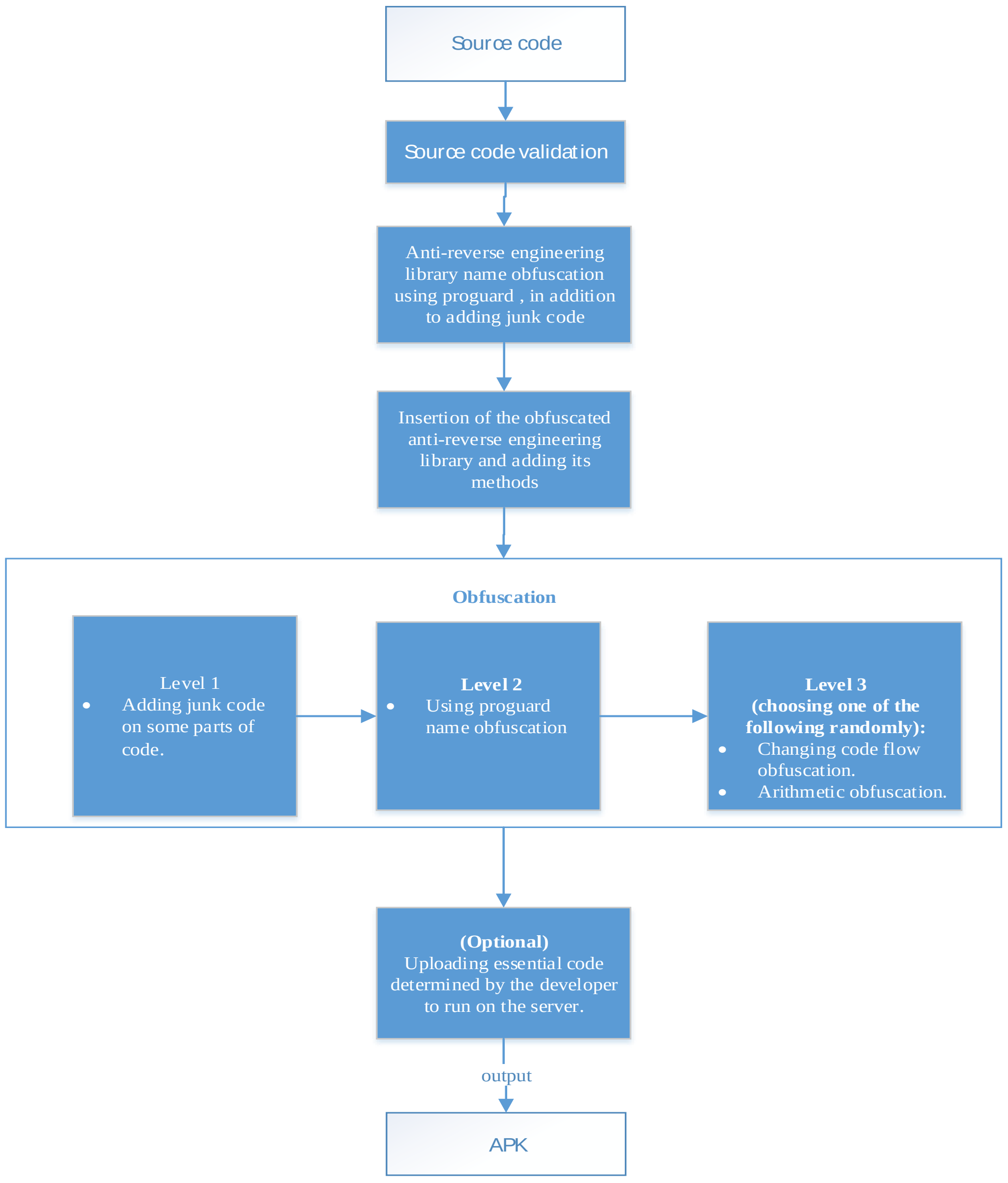

Source code
Source code validation
Anti-reverse engineering library name obfuscation using proguard , in addition to adding junk code
Insertion of the obfuscated anti-reverse engineering library and adding its methods
Obfuscation
Level 1
• Adding junk code on some parts of code.
Level 2
• Using proguard name obfuscation
Level 3 (choosing one of the following randomly):
• Changing code flow obfuscation.
• Arithmetic obfuscation.
(Optional) Uploading essential code determined by the developer to run on the server.
output
APK

Figure 3-1: Framework Design

3.2 Features and Options

Insertion of the Anti-Reverse Engineering Library

An Android library, which contains anti-debugging, root detection methods will be written and obfuscated using ProGuard name obfuscation, after that it will be stored on the server side. Before uploading the zipped source code to the server, the software gives the user an option whether to insert both the root detection and anti-debugging methods to the code or one of them or none. For the anti-debugging option, the user must decide the location of where to insert the methods in the code (by adding special comment). After extracting the zipped file, the software will add this library to the app and import it in the code.

Anti-debugging methods are used to check whether a debugger is currently connected or not. If it turns out that there is a connected debugger the application must terminate.

Root detection methods are added in the main activity to check if there is root access on the device. If it turns out there is a root access, the application must terminate to avoid attacks like permission escalation attack.

Uploading Part of Code to the Server

Generally, most applications contain a part of code that has the most important functionality of the total app. Instead of installing it on the client side, there are two options, it could be executed on the server, or it can be stored so the code can be downloaded partially when needed then removed. This method distinguishes our framework from other frameworks; this prevents it from being visible to the client. Therefore, it is difficult for anyone that decompiles the code to understand it.

This method is optional, as some applications do not work on the internet. An option of whether to use this feature or not is displayed on the webpage before uploading the zipped source code. The developer must surround each function that he wishes to upload to the server by special comment. The software then extracts these functions from the code and store them in a database. Next, it replaces them with a code that connects the client side with the server side. The client side will send the essential data for the server-side code to be executed.

Obfuscation

In this step, the obfuscation techniques that were described in chapter 2 will be applied on the source code. Three levels of obfuscation:

- ✓ **level one**, in order to make the code harder for the analysis, junk code is added to predefined locations, this junk code does not affect the functionality or logic of the original code, it is just for anti-analysis purpose.

- ✓ **level two**, name obfuscation technique that ProGuard offers can be used, ProGuard (as an official plugin supported in Android studio) offers basic code protection in the form of name obfuscation by obfuscating names of classes, fields and methods.

- ✓ **level three**, one of the two other techniques of obfuscation can be applied randomly. First, code flow obfuscation, the code can be divided into finite number of blocks and a relation among them is found, the flow of the relation will be changed in a way that does not change the original code logic. Second, Arithmetic obfuscation, this is done by changing arithmetic operations to a more complex form with the same functionality of the original ones, in order to make the code harder to understand.

3.3 Software Components

Laravel (MVC PHP Framework) will be used for the implementing the back-end of the software. The server contains stored folders and files, which are the anti-reverse engineering library and files with different junk codes.

User Interface

This component is built using JavaScript language and bootstrap (CSS Framework). It will contain the instructions of using the software, options, Form for uploading the source code and displayed messages.

ZIP File Extractor

This component extracts the uploaded input ZIP file and store the extracted folder in a certain directory.

Source Code Validator

The source code validator will build the uploaded source code, if errors while building the APK occur, then there is a problem with the code. A message will be displayed on the user interface, which illustrates the building error.

For building the APK using cmd (command prompt), the gradle wrapper command line tool that is available in every Android project is used. It is available as a shell script for Linux and Mac (gradlew.sh), and as a batch file for Windows (gradlew.bat). It is accessible from the root of the project. It can be used as the following: First, change the directory in cmd to one of Android projects folder, then type the following command: gradlew assembleDebug. If no errors in building occur the resulting APK will be in App_Name\app\build\outputs\apk folder. If errors while building occur, they are displayed.[22]

Root Detection Methods Insertor

The root detection methods insertor adds the root detection methods in the main activity to check if there is root access on the device. The root detection methods terminate the application on the client side if it turns out there is root access.

Anti-Debugging Methods Insertor

The user should add a comment at the location, which the debugging is not allowed. The form of the comment should be as the following:

/*1*/, which "1" represents the anti-debugging feature.

This component reads '/*1*/' comments from the code and replace them with one of the ant-debugging methods. The anti-debugging methods terminate the application on the client side, if there is a debugging.

Code Splitter

The user can only upload a function to the server. The user must add a comment before the function as the following:

```
//3
Function(){
}
```

If there is the ‘upload to server’ comment (//3 function () {}), the code splitter extracts the function and store it in a java file on the server and store its path with application ID and function name in the data base. Then, the function is replaced with a code that connects the client with the server using its IP. This function sends Java Script Object Notation with the app ID and function name at the execution on the client side. As the following example:

Function	JSON object
`Function (int x,int y){`	`{`
`}`	`"appID":application ID,`
	`"funName": function name,`
	`"parameters": [`
	`{"x":value of x,`
	` "y":value of y`
	`}`
	`],`
	`}`

Level 1 Obfuscator (Junk Code Insertor)

For junk code feature the user should add a comment at the beginning and end of the location that he/she wants to add as the following:

//2.1

Code

//2.2

This component reads the junk-code comment (//2.1 …. code …. //2.2), then it chooses one of the junk code files stored on the server randomly and insert its lines between the surrounded code as the following algorithm:

i. Calculate number of surrounded code lines (x).

ii. Read junk code file name, which represents number of code lines in it (y).

iii. If x>y, then result = x/y , for every ‘result’ lines of original code , insert one line of junk code.

iv. If y>x, then result = y/x, for every ‘result’ lines of junk code, insert one line of original code.

Level 2 Obfuscator (ProGuard)

This component will modify the build.gradle(Module:app) to enable ProGuard plugin name obfuscation as the following:

```
buildTypes {
    release {
        minifyEnabled false  => minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-
rules.pro'
    }
}
```

Level 3 Obfuscator (Code Flow and Arithmetic Obfuscation)

This component will use control flow obfuscation or arithmetic obfuscation algorithms, which not determined yet.

The Uploaded Functions Executor

This component can execute the uploaded functions on the server, and then return the results to the client. Or the code can be downloaded partially when needed then removed. The first option is more secure than the other as it keeps the functions on the server. The two options will be more explored in future work.

Database

we will use MYSQL database management system for the database. The database will contain one table (App_functions), it has 3 columns, which are: App_ID,Func_Name,Func_path. The superkey is (App_ID, Func_Name).

Example:

App_ID	Func_Name	Func_path
1	Func1	Path1
1	Func2	Path2
2	Func1	Path3

Chapter 4 Future Work and Conclusion

In the future, we will convert the design into real implementation, and test it by some users' feedback to improve it. The time period for the next semester is divided to 6 stages as shown in Table 4-1.

Table 4-1: Future Work

Task	**Start time**
Create the anti-reverse engineering android library	February,5th
Junk code obfuscation	February,15th
Code flow and arithmetic obfuscation	March,1st
"uploading part of code" to server and Data base creation	April,1st
Name obfuscation, executing function on the server and returning result to client	April,20th
Testing and improving	May,10th

As a conclusion, we designed a framework which provides anti-analyses features that make reverse engineering much harder in Android application. Consider that this framework doesn't provide full guarantee of protecting the applications against reverse engineering. This framework is distinguished from other frameworks that it provides the server-client methodology and multi-level obfuscation. Executing parts of code on the server can affect the performance of the application, as there could be several functions uploaded to the server.

References

[1] Binu, Sumitra, Android Security Issues and Solutions, Department of Computer Science, Christ University,2017

[2] Android General Security Issues https://www.informationweek.com/mobile/8-android-security-concerns-that-should-scare-it/d/d-id/1319412?image_number=1 Retrieved Oct. 2017

[3] Peter Stavroulakis, Mark Stamp (Eds.), Handbook of Information and Communication Security,2010

[4] Android Vs iOS https://www.quora.com/How-resistant-are-Android-apps-to-reverse-engineering Retrieved Nov. 2017

[5] Android Apps Security – Sheran A. gunasekra 2012

[6] Z. Fang, W. Han, and Y. Li, "Permission based Android security: Issues and countermeasures," Computers & Security, vol. 43, pp. 205–218, Jun. 2014.

[7] Manifest Permissions https://developer.android.com/reference/android/Manifest.permission.html Retrieved Dec. 2017

[8] Android File System and Directory Structure Explained http://www.stevesandroidguide.com/android-files/ Retrieved Nov. 2017.

[9] Rooting (Android) http://spyappsmobile.com/introduction-guide-on-how-to-root-your-android/ Retrieved Nov. 2017.

[10] Android Root Detection Techniques https://blog.netspi.com/android-root-detection-techniques/ Retrieved Nov. 2017.

[11] Testing Root Detection

https://github.com/OWASP/owasp-mstg/blob/master/Document/0x05j-Testing-Resiliency-Against-Reverse-Engineering.md#safetynet Retrieved Nov. 2017.

[12] Anti-debugging scheme for protecting mobile apps on android platform https://link.springer.com/article/10.1007/s11227-015-1559-9/ Retrieved Nov.2017

[13] Android Anti-Debugging Fun http://www.vantagepoint.sg/blog/89-more-android-anti-debugging-fun/ Retrieved Nov.2017

[14] Advances in Cryptology — ASIACRYPT 2000
6th International Conference on the Theory and Application of Cryptology and Information Security Kyoto, Japan, December 3–7, 2000 Proceedings

[15] Christian Collberg, Clark Thomborson, Douglas Low, A Taxonomy of Obfuscating Transformations, Technical Report #148, Department of Computer Science, The University of Auckland, 1997

[16] Journal of Mobile, Embedded and Distributed Systems, vol. III, no. 4, 2011 ISSN 2067 – 4074 205 Techniques of Program Code Obfuscation for Secure Software

[17] Mariem Graa, Nora Cuppens-Boulahia, Frédéric Cuppens, Ana Cavalli, rotection against Code Obfuscation Attacks based on control dependencies in Android Systems, International Workshop on Trustworthy Computing, Jun 2014, San Francisco, United States

[18] James, Hamilton. PhD student. Software watermarking, Goldsmith university of London Feb 26, 2011 https://www.slideshare.net/j_ham3/static-software-watermark Retrieved Nov.2017

[19] Christian S, Collberg. Clark, Thomborson. Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection-, University of Arizona Computer Science Technical Report 2000-03, February 10, 2000

[20] ProGuard Manual
https://www.guardsquare.com/en/proguard/manual/introduction Retrieved Oct. 2017

[21] DexGuard VS ProGuard
https://www.guardsquare.com/en/blog/dexguard-vs-proguard Retrieved Oct. 2017

[22] Build Your App from the Command Line
https://developer.android.com/studio/build/building-cmdline.html Retrieve Jan.2017