

BERZIET UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

**Secure framework for developing**

**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 code 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 itself is a problem; 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 for analyzing and tampering. From 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 part of the application on the server; this method distinguishes our framework from other frameworks. Finally, this design doesn’t make reverse engineering impossible, it makes it much harder.

**TABLE OF CONTENTS page**

[**TABLE OF FIGURES** iii](#_Toc503134915)

[**TABLE OF TABLES** iv](#_Toc503134916)

[**Acronyms and Abbreviations** v](#_Toc503134917)

[**Chapter 1 Introduction** 1](#_Toc503134918)

[**1.1 What is the Problem** 1](#_Toc503134919)

[**1.2 The Aim of The Project** 1](#_Toc503134920)

[**1.3 Why Android and not iOS** 1](#_Toc503134921)

[**Chapter 2 Background and Related Work** 3](#_Toc503134922)

[**2.1 Android Security Issues and attacks** 3](#_Toc503134923)

[**2.1.1 General Security Issues** 3](#_Toc503134924)

[**2.1.2 Permission Escalation Attack** 3](#_Toc503134925)

[**2.1.3 Dangerous Permissions** 5](#_Toc503134926)

[**2.2 Techniques used for Anti-Reverse Engineering** 6](#_Toc503134927)

[**2.2.1 Root Detection** 6](#_Toc503134928)

[**2.2.2 Anti-debugging** 11](#_Toc503134929)

[**2.2.3 Code obfuscation** 13](#_Toc503134930)

[**2.3 Related Frameworks** 17](#_Toc503134931)

[**2.3.1 ProGuard** 17](#_Toc503134932)

[**2.3.2 DexGuard** 17](#_Toc503134933)

[**2.3.3 ProGuard vs DexGuard** 18](#_Toc503134934)

[**Chapter 3 Design** 20](#_Toc503134936)

[**3.1 Main Structure** 20](#_Toc503134937)

[**3.2 Features and Options** 21](#_Toc503134938)

[**3.3 How the Design will be Implemented** 22](#_Toc503134939)

[**3.3.1 Instructions for using the software** 22](#_Toc503134940)

[**3.3.2 Front-End Software implementation** 23](#_Toc503134941)

[**3.3.3 Back-End Software implementation** 23](#_Toc503134942)

[**3.3.4 Database Implementation** 25](#_Toc503134943)

[**Chapter 4 Future Work and Conclusion** 26](#_Toc503134944)

[**References** 27](#_Toc503134945)

**TABLE OF FIGURES page**

[Figure ‎2‑1: Confused deputy attack 4](file:///C:\Users\sawadeh\Desktop\Introduction_Report_progress-2.docx#_Toc503116632)

[Figure ‎2‑2: Android File System Structure [7] 6](#_Toc503116633)

[Figure ‎2‑3:Methods of software protection [15] 14](#_Toc503116634)

[Figure ‎2‑4: Name obfuscation example 14](#_Toc503116635)

[Figure ‎2‑5: Control flow obfuscation [17] 15](#_Toc503116636)

[Figure ‎2‑6: static watermarking [18] 15](#_Toc503116637)

[Figure ‎2‑7: dynamic watermarking [18] 16](#_Toc503116638)

[Figure ‎2‑8:ProGuard process 17](#_Toc503116639)

[Figure ‎3‑1: Secure Framework Design 20](#_Toc503116640)

**TABLE OF TABLES page**

[Table 2‑1: Dangerous permissions Description [7] 5](#_Toc503134649)

[Table 2‑2:ProGuard vs DexGuard 18](#_Toc503134649)

[Table 4‑1: Future Work 26](#_Toc503134650)

# **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

# **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 What is the Problem**

Android Operating system has many security vulnerabilities which make the extraction of applications source code possible using reverse engineering tools. Extracted source code can be reused for malicious or profitability purposes. The extracted code can be manipulated and tampered after analysis.

# **The Aim of The Project**

The aim of the project is 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.

# **1.3 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]

The next chapters talk about background and Related work, in addition to design and future work. In background and related work chapter, we will explain the main security vulnerabilities that android suffers from, 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 make 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.

# **Chapter 2 Background and Related Work**

# **2.1 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.

# **2.1.1 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. [5]
* The idea for android being open source itself is a problem; attackers can analyze each line of code to determine its weaknesses. [2]
* Google play store is a bit of concern because of the relative ease of getting apps approved for sale. Malware apps can squeak through. [2]
* 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]

# **2.1.2 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**

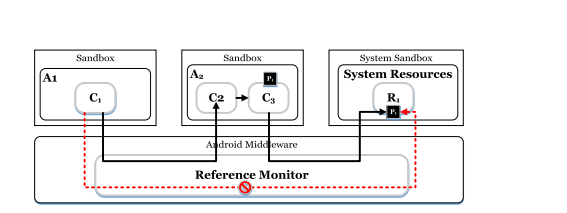
Confused deputy attack and collusion attack. The confused deputy attack exploits the vulnerabilities in unprotected interfaces of privileged benign. As shown in Fig 1-1, the application A1 is not granted with the permission P1; C1, which is a component of A1, cannot directly access system resource R1 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 C2. [6]

Figure 2‑: Confused deputy attack

# **2.1.3 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 2‑: 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. |

# **2.2 Techniques used for Anti-Reverse Engineering**

# **2.2.1 Root Detection**

* **Android File System and Directory Structure**

Most people are familiar with the file layout on Windows and are happy navigating the

Windows file System. Windows uses a drive letter for each physical drive or partition e.g. C: drive. A Physical drive will have at least 1 partition but can have more than one. This is true for both Windows and Android. Each disk partition has a root directory which contains files and folders (directories). The root of the C drive is C:\ and of the F drive is F:\ etc.

Android uses the **Linux file** system structure which has a single root. The (figure 2-1) below

shows the Structure: [8]

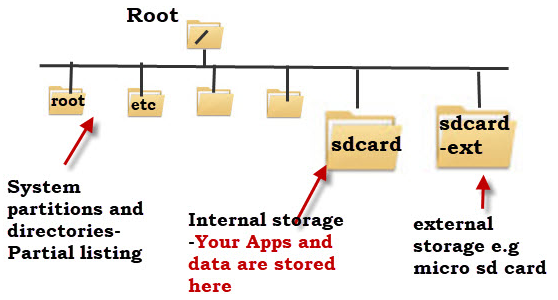


Figure 2‑2: Android File System Structure [8]

The system partitions and directories are protected and unless your device is rooted you don’t normally have access to these although some file managers will display them. Physical disks and partitions appear under the root as directories, and do not have a drive letter as in Windows. Android doesn’t normally come with a default file manager, and so you will need to install a file manager App like Astro file manager, to locate and manage files and folders. Partitions in android are directories under the root. In windows you can reach partition using the drive name followed by ‘:’ (e.g. c:/), in android: (/drive-name).

* **Root Access**

Rooting is the process of allowing users of smartphones, tablets and other devices running the Android mobile operating system to attain privileged control (known as root access) over various Android subsystems. As Android uses the Linux kernel, rooting an Android device gives similar access to administrative (superuser) permissions as on Linux or any other Unix-like operating system such as FreeBSD or macOS. Rooting is often performed with the goal of overcoming limitations that carriers and hardware manufacturers put on some devices. Thus, rooting gives the ability (or permission) to alter or replace system applications and settings, run specialized applications ("apps") that require administrator-level permissions, or perform other operations that are otherwise inaccessible to a normal Android user. On Android, rooting can also facilitate the complete removal and replacement of the device's operating system, usually with a more recent release of its current operating system. Root access user has the ability to use the permission escalation attack. [9]

* **Root Detection**

After taking a look at a lot of Mobile Device Management (MDM) solutions lately to figure out how they are detecting rooted Android devices. Many of MDM solutions use similar methods to detect rooted devices. This usually involves looking for specific packages and files, directory permissions, and running certain commands. Disclosing which MDMs use which methods will not provide, but a list of packages, files, folders, and commands that are found to be used in root detection will provide. [10]

* **Default Files & Configurations**

The first root detection checks are for default files and configurations that should be present on a non-rooted device. These may also be present in rooted devices with non-custom roms. [10]

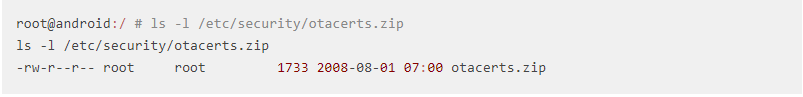
1. **Checking the BUILD tag for test-keys**.

By default, stock Android ROMs from Google are built with release-keys tags. If test-keys are present, this can mean that the Android build on the device is either a developer build or an unofficial Google build. This is why my build tags show release-keys. [10]



1. **Checking for Over the Air (OTA) certs**.

By default, Android is updated OTA using public certs from Google. If the certs are not there, this usually means that there is a custom ROM installed which is updated through other means. Updating my device however, will probably break root.



* **Installed Files & Packages**

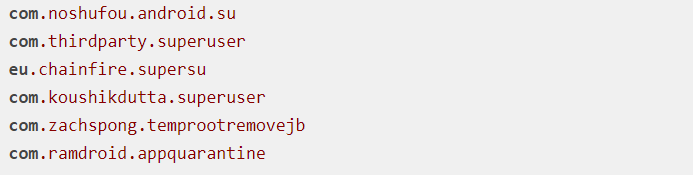
There are many files and packages that MDMs look for when detecting if a device is rooted. A list of ones is compiled to know for sure are being detected. [10]

1. **Superuser.apk**

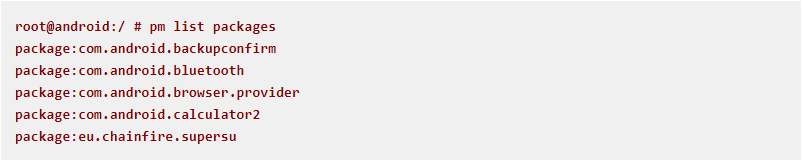
This package is most often looked for on rooted devices. Superuser allows the user to authorize applications to run as root on the device.

1. **Other packages**

The following list of packages are often looked for as well. The last two facilitate in temporarily hiding the su binary and disabling installed applications.

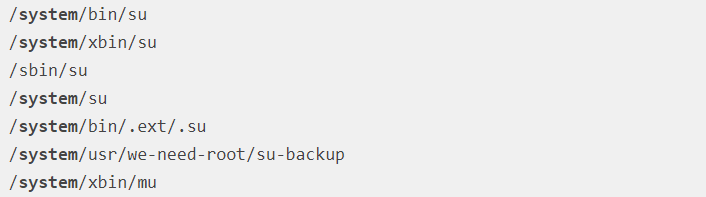


1. The following command lists packages that are currently installed on rooted device.



One of MDMs looks for any package that is developed by chainfire. The most notable one being SuperSU.

1. **Cyanogenmod.superuser**. If the Cyanogenmod ROM is installed, the cyanogenmod.superuser activity may be in the com.android.settings package. This can be detected by listing the activities within com.android.settings.
2. **Su Binaries**. The following list of Su binaries are often looked for on rooted devices.



* **Directory Permissions**

Sometimes when a device has root, the permissions are changed on common directories. [10]

1. **Are the following directories writable?**

/data

/system

/system/bin

/system/sbin

/system/xbin

/vendor/bin

/sys

/sbin

/etc

/proc

/dev

1. **Can we read files in /data?**

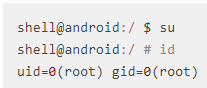
The /data directory contains all the installed application files. By default, /data is not readable.

* **Commands**

A few MDMs execute common commands to detect if a device is rooted. [10]

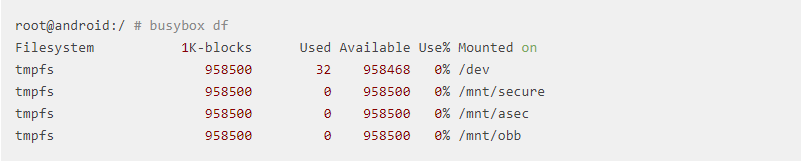
1. **Su**

Execute su and then id to check if the current user has a uid of 0 or if it contains root.



1. **Busybox**.

If a device has been rooted, more often than not Busybox has been installed as well. Busybox is a binary that provides many common Linux commands. Running Busybox is a good indication that a device has been rooted.



* **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 app may the **SafetyNetApi.attest()** method with returns a **JWS message** (it is a way to authenticate (but not necessarily encrypt) information in a highly serializable, machine-readable format. That means that it is information, along with proof that the information hasn't changed since being signed. It can be used for sending information from one web site to another, and is especially aimed at communications on the web. It even contains a compact form optimized for applications like URI query parameters) 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,**

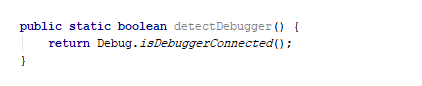
}

# **2.2.2 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]

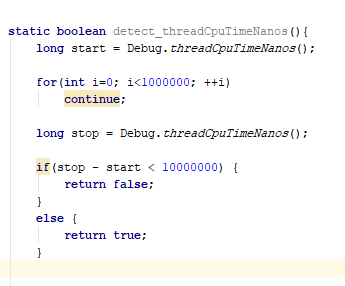
* **static method**

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:



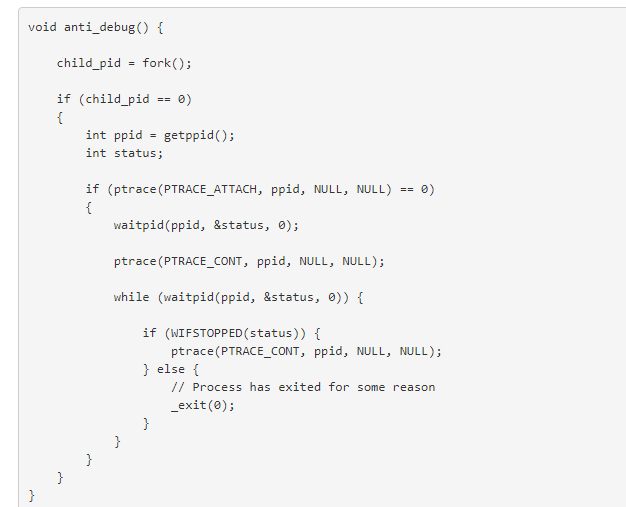
* **Timer Checks**

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](https://slides.night-labs.de/AndroidREnDefenses201305.pdf).



* **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]



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.

# **2.2.3 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.

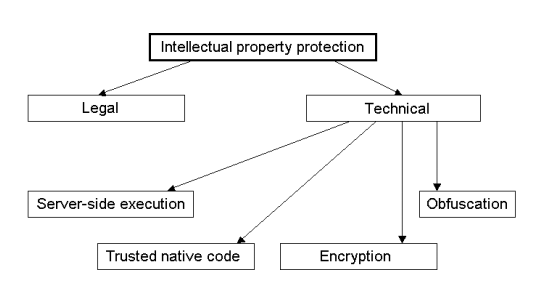


Figure 2‑3: Methods of software protection [15]

* **Algorithms of Obfuscation**

**1. Name obfuscation**

It is the process to replace de 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]

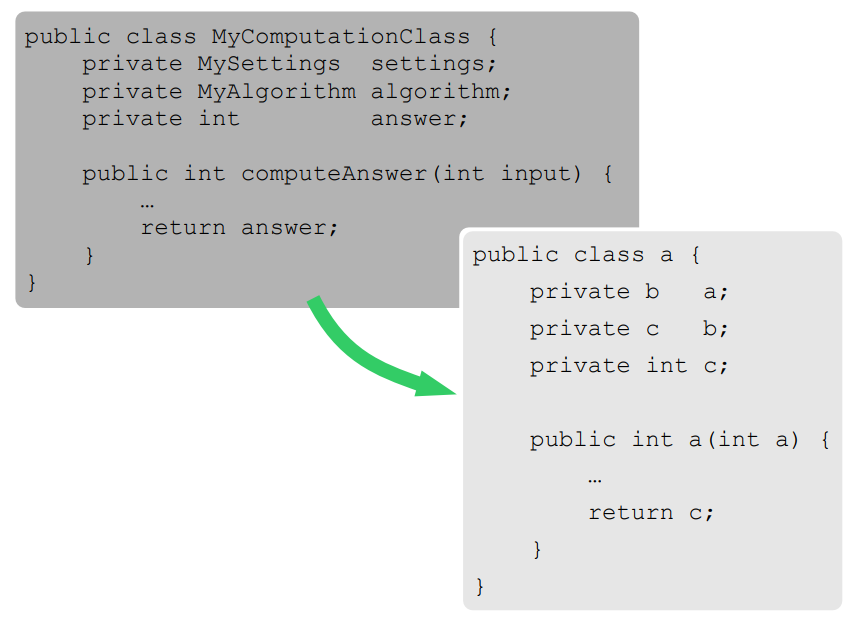


Figure 2‑4: Name obfuscation example

**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.

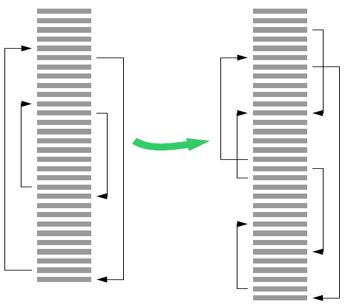


Figure 2‑5: Control flow obfuscation [17]

**3. Watermarking**

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

* **Static watermarking**

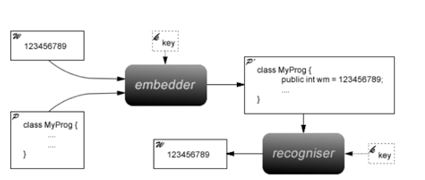


Figure 2‑6: static watermarking [18]

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

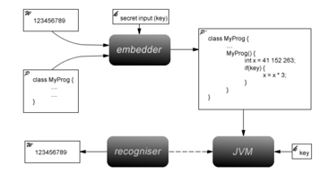


Figure 2‑7: dynamic watermarking [18]

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.

# **2.3 Related Frameworks**

# **2.3.1 ProGuard**

**ProGuard** is a Java class file shrinker, optimizer, obfuscator, and preverifier. In the shrinking step it removes the unused classes, methods, fields and attributes. In the optimization step it optimizes the bytecode of the methods. The obfuscation step renames the remaining classes, fields, and methods using meaningless names. All of These steps make the code harder to reverse engineer. The final preverification step adds preverification information to the classes, which is required for Java Micro Edition and for Java 6 and higher. Each of these steps are optional, they may not all be used. Furthermore, ProGuard can be used to list dead code in an application. Fig. 2-7 shows the ProGuard process.

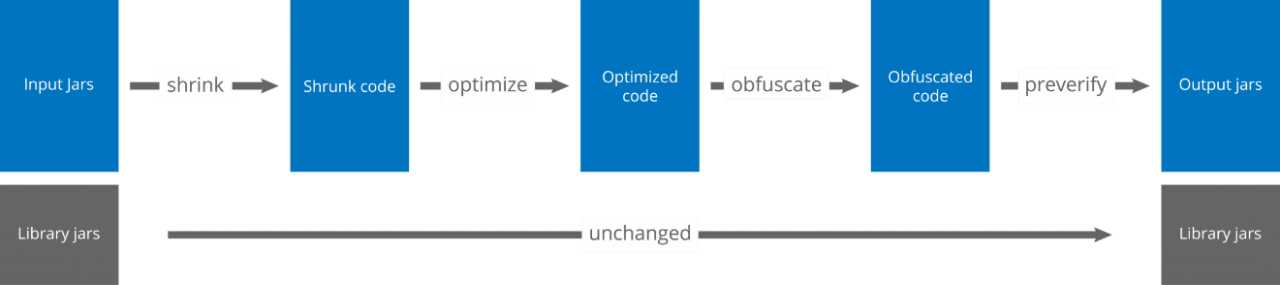


Figure 2‑8: ProGuard process

ProGuard first reads the **input jars** (or zips, apks, or directories). It then subsequently shrinks, optimizes, obfuscates, and preverifies them. ProGuard writes the processed results to one or more **output jars** (or zips, apks, or directories). The input may contain resource files, whose names and contents can optionally be updated to reflect the obfuscated class names. [20]

# **2.3.2 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.

# **2.3.3 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** | offers basic protection against static analysis only by harden the source code of the application using a multitude of obfuscation and encryption techniques | 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** | offers basic protection in the form of name obfuscation by obfuscate names of classes, fields and methods. | 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 Main Structure**

The software has one input, which is zipped Android source code file, the developer must upload it first; this is done by selecting it after browsing the website that belongs to the Web server. Later, the software will apply its Features that will be explained in the next section to produce a new zipped source code as output; these Features are shown in Fig 3-1.



Figure 3‑: Secure Framework Design

# **3.2 Features and Options**

* **ZIP File Extraction**

As a first step, the software must extract the uploaded input ZIP file programmatically and store the extracted folder in a certain directory.

* **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 permission escalation attack.

* **Uploading Part of Code to the Server**

Generally, most applications contain a part of code that has the most functionality of the total app. Instead of installing it on the client side, it could be executed on a server side; 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 with an application ID. 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. Two levels of obfuscation:

* **level one**, in order to make the code harder for the analysis, junk code is added to predefined location, 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 offer 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 How the Design will be Implemented**

# **3.3.1 Instructions for using the software**

* **Instructions for using anti-debugging feature**

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.

* **Instructions for using junk code feature**

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

//2.1

Code

//2.2

* **Instructions for using “uploading part of code to server” feature**

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

//3

Function(){

}

# **3.3.2 Front-End Software implementation**

We will use JavaScript language and bootstrap (CSS Framework) for building the WEB page.

The WEB page will contain the instructions of using the software and Form for uploading the

source code.

# **3.3.3 Back-End Software implementation**

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 include different junk codes (file name is number of code lines in it). The steps of the

software process is:

1. The software will insert the root detection methods from anti-reverse engineering library.
2. Read the source code as zip file. Also, read the options which were entered by the user from the front end.
3. Extract the zip file and determine the path of the java files.
4. Add the anti-reverse engineering library to the project folder.
5. Move in each java file and read if there is feature comments.
6. If there is debugging comment (/\*1\*/), replace it by one of the anti-debugging library methods randomly.
7. If there is the ‘upload to server’ comment (//3 function () {}), extract 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 replace the function with a code that connects the client with the server using its IP. Parameters needed for the function to be executed on the server are sent in JSON object with the app ID and function name.

example:

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

The executed function on the server will return the result in JSON object to the client, which will proceed then.

1. If there is junk-code comment (//2.1 …. code …. //2.2), then choose one of the junk codes randomly to insert its lines between the surrounded code. How is this done?
2. Calculate number of surrounded code lines (x).
3. Read junk code file name, which represents number of code lines in it (y).
4. If x>y, then result = x/y , for every ‘result’ lines of original code , insert one line of junk code.
5. If y>x, then result = y/x, for every ‘result’ lines of junk code, insert one line of original code.
6. The software 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'** }  
}

1. The software will use a control flow obfuscation and arithmetic obfuscation algorithms

, which not determined yet.

# **3.3.4 Database Implementation**

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‑: 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.

# **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](https://www.informationweek.com/mobile/8-android-security-concerns-that-should-scare-it/d/d-%20%20%20id/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.](https://github.com/OWASP/owasp-mstg/blob/master/Document/0x05j-Testing-Resiliency-%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20Against-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