A

Dissertation On

Android App Repackaging detection using signing certificate and permissions comparison

Submitted in Partial Fulfilment of the Requirement
For the Award of Degree of

Master of Technology

In

Software Technology

By

Niraj Kumar
University Roll No. 2K16/SWT/508

Under the Esteemed Guidance of
DR. MANOJ KUMAR

Associate Professor, Computer Science & Engineering, DTU

COMPUTER SCIENCE & ENGINEERING DEPARTMENT
DELHI TECHNOLOGICAL UNIVERSITY
DELHI – 110042, INDIA

STUDENT UNDERTAKING

Delhi Technological University

(Government of Delhi NCR)

Bawana Road, New Delhi-42

This is to certify that the thesis entitled **"Android Apk Repackaging detection using signing certificate and permissions comparison"** done by me for the Major project for the award of degree of **Master of Technology** Degree in **Software Engineering** in the **Department of Computer Science & Engineering**, Delhi Technological University, New Delhi is an authentic work carried out by me under the guidance of Dr. Manoj Kumar.

Signature:
Student Name
Niraj Kumar
2K16/SWT/508

Above Statement given by Student is Correct.

Project Guide:
Dr. Manoj Kumar, Associate Professor
Department of Computer Science & Engineering
Delhi Technological University, Delhi

ACKNOWLEDGEMENT

I take this opportunity to express my deep sense of gratitude and respect towards my guide **Dr. Manoj Kumar, Associate Professor, Department of Computer Science & Engineering.**

I am very much indebted to him for his generosity, expertise and guidance I have received from him while working on this project. Without his support and timely guidance, the completion of the project would have seemed a far-fetched dream. In this respect I find myself lucky to have my guide. He has guided not only with the subject matter, but also taught the proper style and techniques of documentation and presentation.

Besides my guide, I would like to thank entire teaching and non-teaching staff in the Department of Computer Engineering, DTU for all their help during my tenure at DTU. Kudos to all my friends at DTU for thought provoking discussion and making stay very pleasant.

Niraj Kumar
MTech, Software Engineering
2K16/SWT/508

ABSTRACT

Android eco system works in two steps as far as an application developer is concerned. In the first step, developers design and develop an Android app and subsequently publish it on Google Play Store either as paid apps or with some advertisements to earn monetary benefit or sometimes as free app just for building user base. Besides the bona fide Android ecosystem, there is a parallel dark world of malicious attackers who repackage other developer's application (Free apps available in Play Store or other app stores) and publish it with their own name. Alternatively, they add advertisements with their own bank accounts or even add malicious code and use it for their own benefit.

In a similar fashion, malicious attackers embed malware payload into the original applications so as to gain control of the mobile devices on which they are running to retrieve the private data of the user, stealthily read or send SMS messages to premium rate numbers, read banking credentials and so on. Although there are many identification methods which have been used traditionally to detect these repackaged applications existing in various Android app stores. However, it is not always effective to analyse any new application.

Repackaged apps are a serious vulnerability these days in Android phones. Various threat mitigation measures have been devised like watermarking in case of rooted device. But, a defence mechanism that prohibits repackaged apps from running on a user device (non-rooted device) is not common. Our repackage-proofing technique for Android apps is trustworthy and covert.

Repackaged apps present considerable challenge to the security and privacy of smartphone users. But fortunately such apps can be made to crash (randomly crash to confuse attackers) using keystore check as well as permissions added in repackaged code. Even other techniques like code obfuscation using ProGuard tool are helpful. It does not require any change at the Framework or System level. Private key used to sign the apk is with the original developer.

Any app which is installed in an Android system need to pass signature validation. PackageManager API reads an apk and extracts the app information. It then saves the information in three different files on device path /data/system. Out of these three files, the most important is packages.xml. It contains key information like names, code paths, public keys, permissions, etc.,

for all the installed apps. PackageManager API is used to retrieve Kr from the file. We split this Kr value into 8 equal parts and store it as constants in different classes. At runtime, we merge these parts to recreate Kr and compare it with Kr value returned by making use of PackageManager. For repackaged apps as signing key has changed, the two Kr values will be different.

TABLE OF CONTENTS

CERTIFICATE .... [i]
ACKNOWLEDGEMENT .... [ii]

ABSTRACT .... [1]
TABLE OF CONTENTS .... [3]
LIST OF FIGURES .... [5]
LIST OF TABLES .... [6]
LIST OF SYMBOL, ABBREVIATIONS .... [7]

CHAPTER 1
INTRODUCTION 8
- **1.1. Android Build Process** 8
- **1.2. Motivation** 10
- **1.3. Related Work** 10
- **1.4. Problem Statements** 11
- **1.5. Scope of this thesis** 11

CHAPTER 2
LITERATURE REVIEW 12
- **2.1. Android OS Architecture** 12
- **2.2. Repackaging Problem Statement** 12
- **2.3. Threat Model** 14

CHAPTER 3
PROPOSED WORK 17
- **3.1. Actual Tools Used** 17
- **3.2. Overview of the Repackaging attack** 22
- **3.3. Steps used for repackaging apk with malware payload** 24

3.4. Malware payload used for evaluation 25
3.5. Repackage Proofing benign apps 26
3.6. Original app permissions and public key fingerprint 27

CHAPTER 4

IMPLEMENTATION, RESULT, USECASE... 34
4.1. Android Game: Repackaged Escape Bird app with RP proofing... 34
4.2. API used for Repackage proofing technique... 38
4.3. Activity Life cycle... 39

CHAPTER 5

RESULT, CONCLUSION AND FUTURE WORK... 40
5.1 Android Game: Repackaged Escape Bird app with RP proofing... 40
5.2 Conclusion and Future Work 42

References... 44

<u>**LIST OF FIGURES**</u>

Figure 1.1 Android app module generic build process……………………………………09

Figure 1.2 Android app module default project structure………..\………………………09

Figure 2.1 Architecture of Android OS……………………………………………………13

Figure 3.1 Apktool: A tool to reverse engineer Android Apk files…………………………17

Figure 3.2 Decode a benign app apk using Apktool………………………………………17

Figure 3.3 Build a benign app using Apktool………………………………………………18

Figure 3.4 my-release-key.keystore generation using keytool ………………………………19

Figure 3.5 my-release-key.keystore details using Keystore Explorer………………………19

Figure 3.6 Firebase Realtime database screenshot with SMS………………………………21

Figure 3.7 Flow diagram for showing Repackaging attack…………………………………22

Figure 3.8 APK file structure after running APKTool………………………………………23

Figure 3.9 Flow chart displaying repackaging strategy……………………………………33

Figure 4.1 Activity Life cycle………………………………………………………………39

Figure 5.1 Repackaged Escape Bird app with RP proofing technique gameplay………..40

Figure 5.2 Repackaged RP proof game: Character sprite GUI corruption………………40

Figure 5.3: Repackaged RP proof game: High score label corruption ……………………41

Figure 5.4 Repackaged RP proof game: Crash……………………………………………42

LIST OF TABLES

Table 2.1: If statements compared to constants variations ……………………………......15
Table 2.2: Percentage of repackaged from 3rd party stores [29] …………………………...16

LIST OF SYMBOL, ABBREVIATIONS

Android Asset Packaging Tool (AAPT)

Android Debug Bridge (ADB)

Ahead of Time compilation (AOT)

Android Package Kit (APK)

Android Runtime (ART)

Compact Dalvik Executable (CDEX)

Dalvik Executable (DEX)

Domain Specific Language (DSL)

Executable and Linkable Format (ELF)

Java Virtual Machine (JVM)

Of Ahead Time (OAT)

Optimized Dalvik Executable (ODEX)

Software Development Kit (SDK)

Original Equipment Manufacturer (OEM)

CHAPTER 1 INTRODUCTION

1.1 Android Build Process

The Android build process compiles source code and project resources like layouts, drawables, etc., and packages them into APKs that can then be tested, deployed, signed, and distributed. Android Studio IDE automates and manages the build process using Gradle, android toolkit for build process and allows us to define flexible custom build configurations. Each build configuration reuse the parts shared by all versions of our app but still has its own set of code and resources.

A typical Android app module follows a complex build process as shown in figure 1.1 consisting of following steps:

1. Source code usually written in Java is compiled using compilers into DEX (Dalvik Executable) files including the bytecode. The remaining app resources are transformed into compiled resources.
2. The DEX files and compiled resources are then synthesized into a single archive file of APK format by the APK Packager. In order to enable the APK to be installed and deployed onto an Android device it needs to be signed first.
3. As a next step APK Packager signs the APK using one of release or debug keystore.
4. The zipalign tool is used to optimize our app to utilize minimum memory during execution on a device.

We have two variants of an APK viz. debug APK or release APK at the end of the build process. They can be used to install, verify, or release to outside users.

Build process is configured using build.gradle files which are used for creating custom build configurations. These files are in plain text format using Domain Specific Language (DSL). It is used to depict and control the build parameters using Groovy. Groovy is a dynamic language for the Java Virtual Machine (JVM). When a new projected is created in Android Studio, the IDE spontaneously creates some of these files for the developer, as shown in figure 1.2, and applies default values in it.

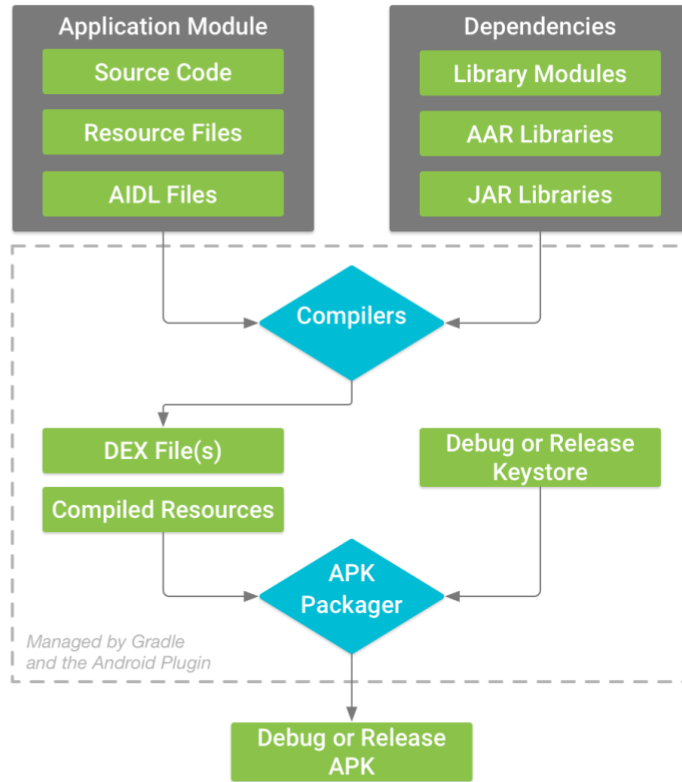


Figure 1.1 Android app module generic build process

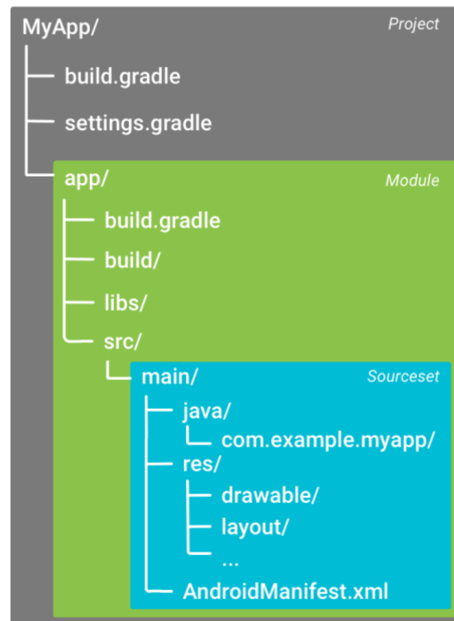


Figure 1.2 Android app module default project structure

1.2 Motivation

Users commonly have faith in the application stores and believe them to deliver them bona fide mobile applications with no vulnerability or back door traps. Google introduced a reviewing framework which goes by the name Google Bouncer [3]. The purpose of this framework is to analyze every incoming app for malicious payload presence before being allowed to go public on the store. Smartphone users on the Android platform have access to free apps from Play Store. However, there is no guarantee that apps downloaded by user from Play Store is benign ware and has no malware payload or adware. The threat still exists that an attacker can download an application; exploit its content by integrating malicious payload and finally repackaging it and publishing it on app stores.

Repackaging is regarded one of the top ten vulnerabilities for Android platform devices. Studies [15] have shown that more than 85% of malwares are introduced in applications through application repackaging.

1.3 Related Work

The existing measures to deal with the repackaging problem are

1. License protection mechanisms (such as MD5 hash/checksum, APK file size, and app signatures). One flagship mechanism is offered by Goole Play Store licensing service. AntiLVL's [16] objective is to damage conventional license protection methods such as the Android License Verification Library (LVL) by decompiling apps with baksmali tool [17] and rewriting such methods to always return successfully.
2. **App review process.** Any incoming app on Google Play Store needs to pass its approval process before publishing them. However this process can usually be easily bypassed as it is primarily based on the content guidelines implementation and malware detection [18]. As such app stores have a very large number of apps; manual check is not a viable option.
3. **Obfuscation**. There are certain Obfuscation tools available for example Proguard [20] and DexGuard [21] which can confuse and discourage attackers during the repackaging process. The limitation of this approach is that it can at most increase the cost of reverse engineering an app (by increasing repackaging process time). However it cannot

completely prohibit repackaging attack (like inserting malicious payload) from a determined attacker.

4. **Repackaging detection techniques** such as app comparison [22], [23], [24] and watermarking [25]. Android app store manager compare suspicious apps in their store or even other app stores to identify repackaged apps. This repackage detection technique needs to find at least two "similar" android apps at the same time.

1.4 Problem Statements

Our approach leaves behind enhancing the benefit of repackaging detection processes and adopts a self-protective approach. This approach has following advantages:

1. Apps comparison is no longer required. So this saves us from the trouble of collecting apps from Android app stores for a through comparison. Furthermore, all the inherent problems in the process of comparison (e.g., obfuscated code comparison difficulty, and human interaction requirement) can thus be avoided.
2. The problem of false positive cases is gone.
3. The time gap between detection of repackaged app and its removal from smartphone devices is negligible.

1.5 Scope of this thesis

In this project, we tried to provide a method such that repackaged apps automatically reveal their "counterfeit" nature. Our counter strategy can be described as: due to their self-revealing nature, when repackaged apps are used for a reasonable period of time, such apps will show random malfunctions (e.g. crash, GUI corruption) with high probability. If the reason is due to repackaging, users will slowly but surely stop using the app. Moreover, a vigilant user can inform app markets like Google Play Store about such repackaged apps. As a result, the app market after their own investigation may blacklist the repackaged app and even block/delete the account holding such app. This last step together helps the app markets to weed out malicious and suspicious apps and increase their legitimacy.

CHAPTER 2 LITERATURE REVIEW

2.1 Android OS architecture

Android [26] is an operating system based on Linux kernel primarily for phones, tablets and wearable. Android developers make use of Android Studio with bundled Software Development Kit (SDK) and write code in Java/Kotlin/Xml along with other resources. Android Game applications are sometimes written in C/C++ using Native development Kit (NDK) available as a downloaded tool. Android supports a suite of C/C++ libraries available for developers through the application framework. Android application is distributed in APK file format which can be installed in mobile devices. Figure 2.1 portrays all the layers in the architecture of Android OS.

The building blocks of any Android application are four components and they may communicate with each other using Intent messages. As the starting point for any Android application is declared in 'AndroidManifest.xml' file, and accordingly controls the life cycle of each component and the underlying communication between them.

- Activities: Visible component in any Android app and its layout is inflated from a XML file.
- Services: Usually run as a background process to execute long running tasks
- Broadcast Receivers: Receives broadcast messages originating from OS and applications.
- Content Providers: Store data in a SQLite database and read it from another process.

Intents are a message passing mechanism between components of the intra-application or for inter-application communication.

2.2 Repackaging Problem Statement

Application developers on the Android platform create applications and monetize it through their sale in Android package kit (APK) file format. Both bona fide users and malicious developers have access to these APK files. Let us consider the scenario where attackers try to repackage apps and publish them again. Such a malicious developer may modify the code to insert malicious payload so as to exploit user privacy, change the in-app purchase components, unwanted advertisements to the user, to obtain monetary gain, collect big data for business analysis, or just for pleasure and popularity. The primary objective of repackage-proofing which is a type of tamper-proofing

technique is to identify repackaging issue and make sure such apps do not run properly on user devices.

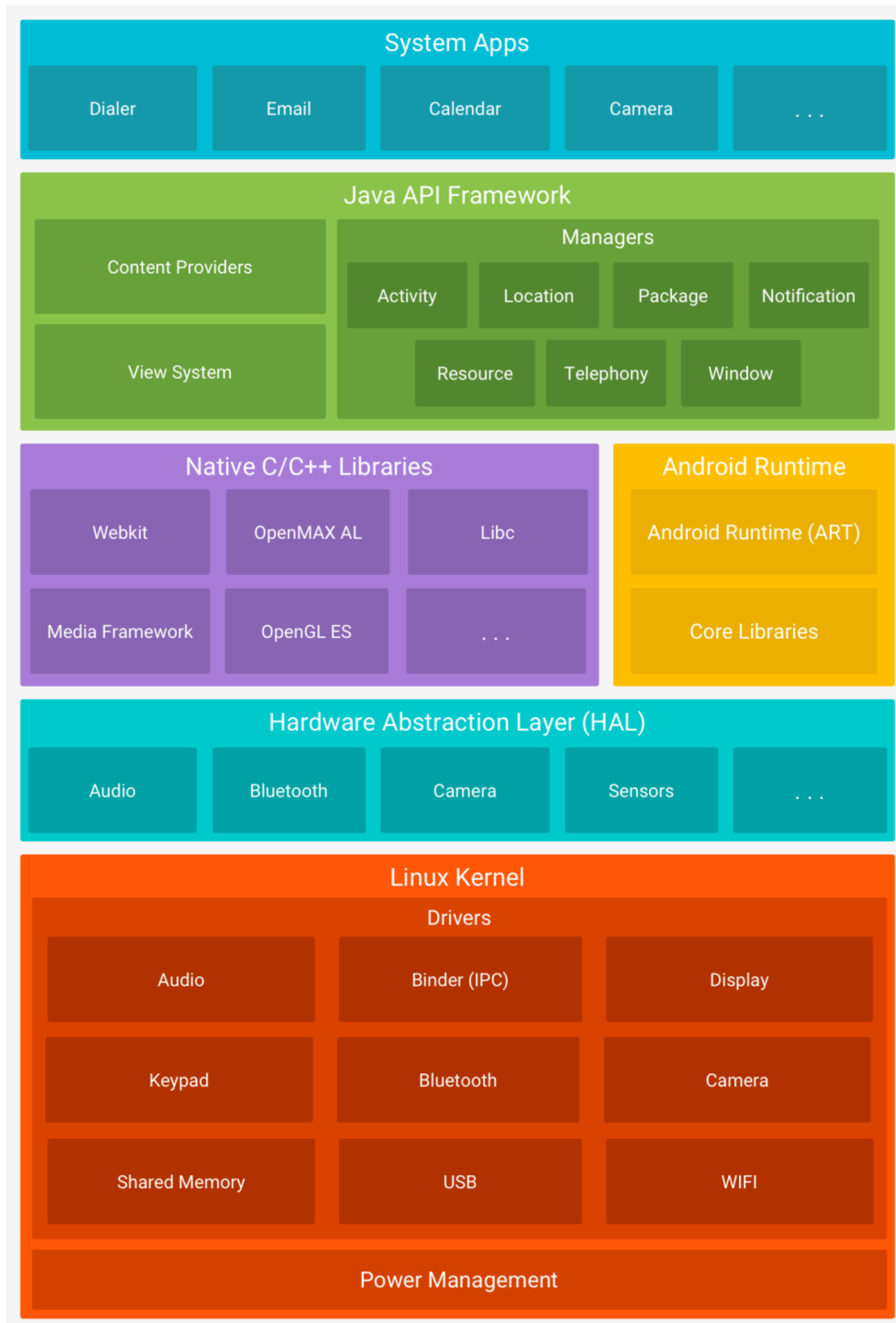


Figure 2.1 Architecture of Android OS

Any android application whether it is legitimate or repackaged needs to be signed with a digital certificate in order to be released to users. Such a digital certificate consists of a distinctive private and public key pair. The private key component of the certificate is known only to the legitimate developer and not in public domain or embedded in APK file. Hence an attacker who needs to repackage an android app needs to generate a new certificate in order to sign the repackaged app. The public key component of the digital certificate is an important piece of information to identify an app developer. This information can be used to identify if an app has been repackaged by a malicious developer. In particular, our repackage proofing methodology checks for change in public key at runtime to detect repackaging. If an application is found to be repackaged, concealed protection code gets executed to prevent the app from working properly on user devices. To avoid detection of such protection code at the attacker's end, it is only executed randomly and only after the app has run for a sufficient number of times. Hence only a limited number of users are ready to buy and use the repackaged app. This automatically put a constraint on its propagation and abuse. Additional response after detection is possible like informing legitimate developers, app store managers and cyber security agencies regarding the same through emails. Proper follow-up needs to be pursued in order to get these repackaged apps removed from app stores.

2.3 Threat Model

Malicious programmers usually have easy access to the APK file of an app. They can download any app to their device from app stores like Play Store and extract the APK from their device. But in almost all cases, they do not have access to the original source code and the private key used to sign the APK. It can so happen that any end user is working in collaboration with the malware developer such that user runs a custom firmware which will generate the benign app public key when running a repackaged app. But for this research, let us rule out such a mala fide collaboration attack and focus solely on designing a security solution to protect bona fide users.

Malicious developers can incorporate attack technique to evade or bypass the app inbuilt repackage-proofing mechanism. These evasion approach adopted by malware attacker can be broadly classified into at least three types of attacks. One approach adopted by malicious developer is to use static analysis technique like text search, pattern match, to identify the embedded protection mechanism. Attacker can then make the necessary code changes required to

bypass/nullify the inbuilt protection code. In the second approach, malicious developer can also use dynamic analysis like check runtime characteristics, debugging, execution in a controlled environment, tracking suspicious data, and so on, to run and debug the app (smali code and native .so files), and one line at a time, to recognize the embedded protection mechanism and bypass/nullify the inbuilt protection code. In the third approach, in order to achieve other evasion attacks and make debugging possible, an attacker may mess up the execution by inserting original developer public key to bypass protection code, just like the example of replay attacks in networks. Our objective is to address these evasion attacks.

Malicious developers can cautiously understand the code semantics and redesign code segments of an app to circumvent built-in protection code. Furthermore, they may be ready to abandon some minor functions of this app and redistribute it. So we need to handle selective code replacement and feature-pruning attacks.

It is commonly seen that a program safeguarded by any software-based solution can ultimately be circumvented as long as a committed attacker is prepared to dedicate time and labour, which is also applicable to our protection code. Still, we believe malicious developers are willing to repackage an app only if it is profitable, for example, if the cost of developing the app from scratch is more than cost of repackaging it.

Table 2.1 If statements compared to constants variations

Category	Examples
Receive and handle command	If (com.equals("RecordVideo"))
Score eligible for bonus	If (gscore == 2,123,789)
File parsing	If (flag.equals("CurrStatus"))
Uri schema	If (schema.equals("telephone://"))
File name	If (filename.equals("delta.ini"))
Domain name	if(domname.equals("www.custom.com"))
Coordinate on screen	If (x==800 and y==1701)

Table 2.2: Percentage of repackaged from 3rd party stores [29]

Third-party app store	# Repackaged App from Droid MOSS	# Repackaged Apps from Manual Analysis	Percentage
US store 1	24	22	11%
US store 2	13	12	6%
EU store 1	11	10	5%
EU store 2	15	13	6.50%
China store 1	27	25	12.50%
China store 2	28	26	13%

CHAPTER 3 PROPOSED WORK

3.1 Actual Tools Used

1- **Apktool**: This tool is useful if 3rd party Android apps need to be reverse engineered for research and/or debugging purposes. Sometimes, it helps in client-OEM (Original equipment manufacturer) app conflicts resolution about some unwanted notifications. This tool can decompile resources to almost original form and also recompile them with some code/resource changes. It assists in debugging of smali code in a gradual manner. As the decompiled files are in a project like structure, working with decompiled smali and resource files is easier. Some of the manual tasks like building apk can be automated through batch files or build scripts.

a. Command to decode an apk: ***apktool d EscapeBird.apk***

b. Command to build an apk from smali code: ***apktool b EscapeBird***

Figure 3.1 Apktool: A tool to reverse engineer Android Apk files

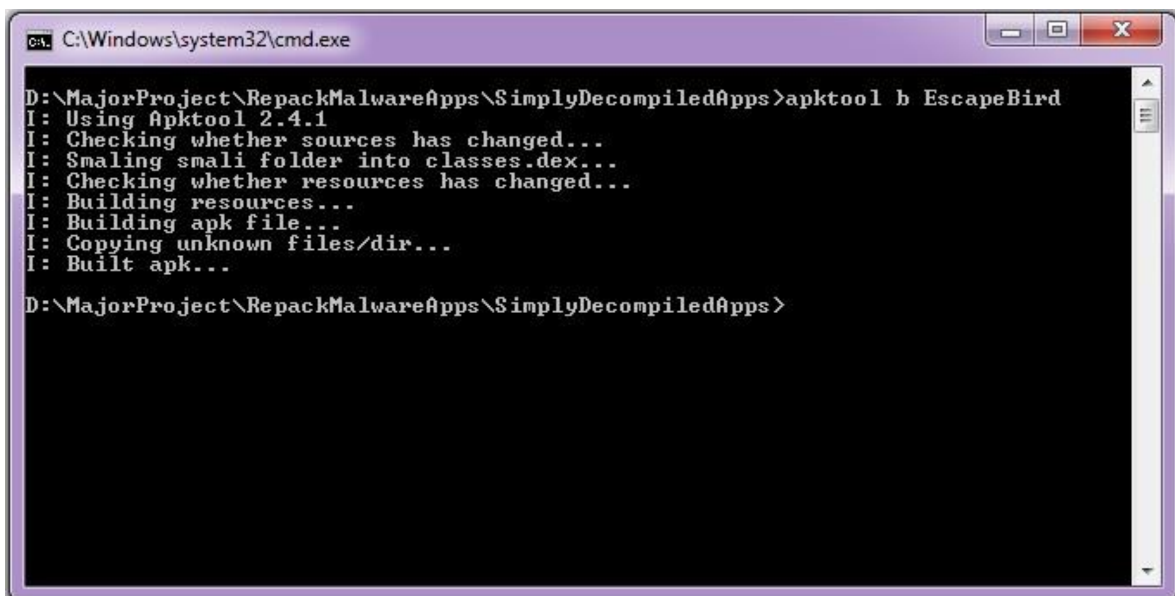


Figure 3.2 Decode a benign app apk using Apktool

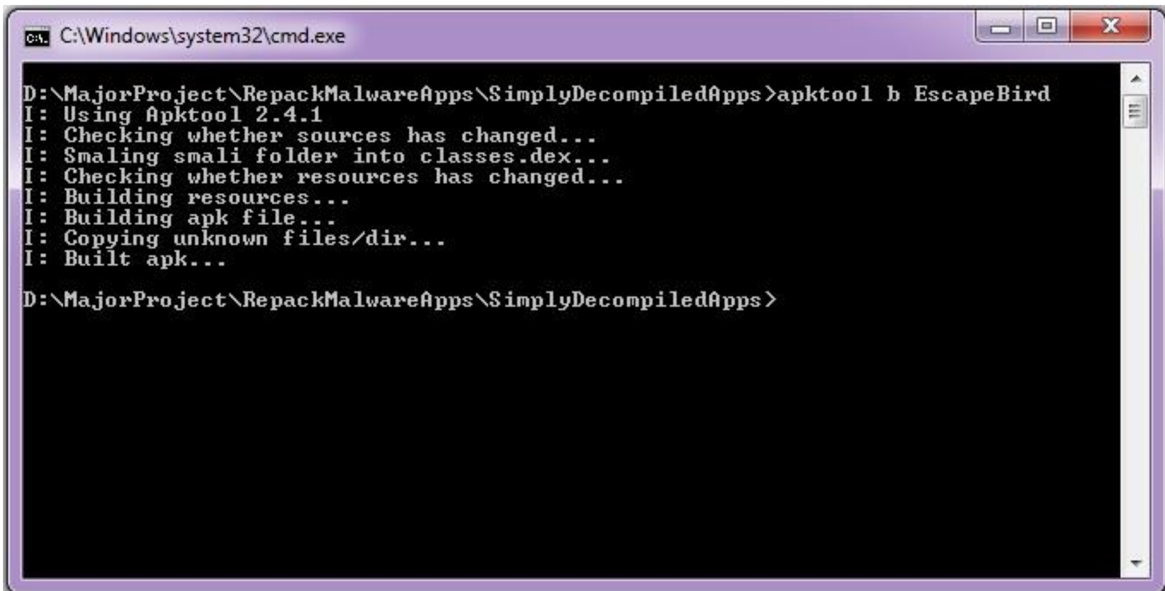


Figure 3.3 Build a benign app using Apktool

2- **Keytool**: This tool is used to manage key and certificate in Android apk build process. It allows users to administer their own public/private key pairs and associated certificates for use in self-authentication or data integrity and authentication services, using digital signatures. When data is digitally signed, the signature can be verified to check the data integrity and authenticity. Integrity means that the data has not been modified or tampered with, and authenticity means the data indeed comes from whoever claims to have created and signed it. Keytool stores the keys and certificates in a keystore.

a. Command to generate signing key: ***keytool -genkey -v -keystore my-release-key.keystore -alias alias_name -keyalg RSA -keysize 2048 -validity 10000***

3- **Jarsigner tool**: The jarsigner tool is provided by the Java SDK. This technique involves signing the APK file using the jarsigner command from the Java SDK. From JDK 7 onwards, there is a change in default signing algorithm, requiring us to mention the signature and digest algorithms (-sigalg and -digestalg) when an APK needs to be signed.

a. Command for signing apk: ***jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore my-release-key.keystore EscapeBird.apk alias_name***.

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\DELL>taskkill /im Notepad++.exe -f
SUCCESS: The process "notepad++.exe" with PID 1732 has been terminated.

C:\Users\DELL>keytool -genkey -v -keystore my-release-key.keystore -alias alias_
name -keyalg RSA -keysize 2048 -validity 10000
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Niraj Kumar
What is the name of your organizational unit?
  [Unknown]:  Samsung
What is the name of your organization?
  [Unknown]:  Samsung
What is the name of your City or Locality?
  [Unknown]:  Noida
What is the name of your State or Province?
  [Unknown]:  Uttar Pradesh
What is the two-letter country code for this unit?
  [Unknown]:  IN
Is CN=Niraj Kumar, OU=Samsung, O=Samsung, L=Noida, ST=Uttar Pradesh, C=IN correc
t?
  [no]:  yes

Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) wi
th a validity of 10,000 days
        for: CN=Niraj Kumar, OU=Samsung, O=Samsung, L=Noida, ST=Uttar Pradesh, C
=IN
Enter key password for <alias_name>
        (RETURN if same as keystore password):
[Storing my-release-key.keystore]

Warning:
The JKS keystore uses a proprietary format. It is recommended to migrate to PKCS
12 which is an industry standard format using "keytool -importkeystore -srckeyst
ore my-release-key.keystore -destkeystore my-release-key.keystore -deststoretype
 pkcs12".

C:\Users\DELL>_
```

Figure 3.4 my-release-key.keystore generation using keytool

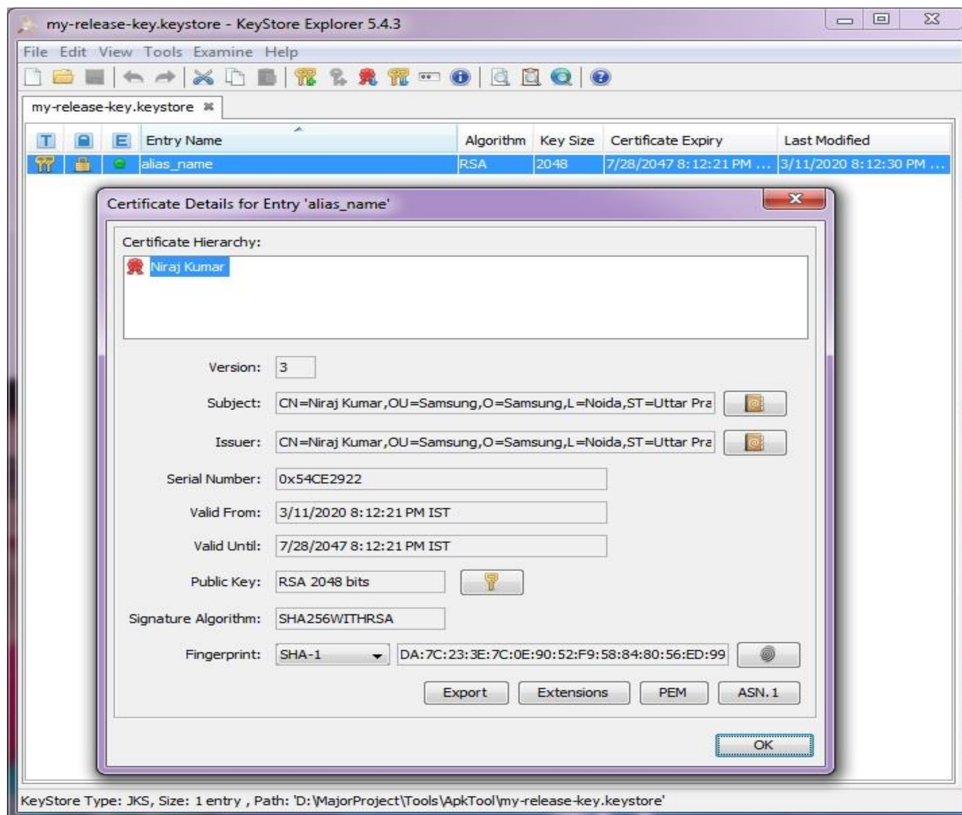


Figure 3.5 my-release-key.keystore details using Keystore Explorer

4- **Zipalign tool**: This tool is helpful in archive alignment (APK file is like an archive) and renders crucial optimization to the android application (APK) files. The basic objective is to ensure that all uncompressed data starts with a particular alignment relative to the beginning of the file. Particularly, it alters all uncompressed data within the APK, such as drawable or raw files, to be aligned on 4-byte limits. This ensures all parts to be read directly with mmap() although they consist of binary data with alignment restrictions. This results in lower RAM footprint when the application is running.

   a. Command for apk alignment for optimal loading: ***zipalign -v 4 EscapeBird.apk EscapeBird-aligned.apk***.

5- **Firebase Realtime Database**: The Firebase Realtime Database provides complete toolkit for managing the security aspect in our android app. The various toolkit methods include user authentication, user permissions enforcement and input validation. FirebaseDatabase instance is used to write firebase data and accessed using asynchronous listener registered to the instance object. The listener is invoked first for the initial data state and every time there is change is data. To read or write data from the database, an instance of Database reference is required.

```
1.  public class SMS {
        public String sender, time, sms;
        public SMS(String sender, String time, String sms) {
                this.sender = sender;
                this.time = time;
                this.sms = sms;
        }
    }
```

```
FirebaseDatabase mFirebaseInstance =
        FirebaseDatabase.getInstance();
DatabaseReference mFirebaseDatabase =
        FirebaseInstance.getReference("SMSBackup");

String mSMSBackup = mFirebaseDatabase.push().getKey();
SMS mSms = new SMS(sender, time, body);
mFirebaseDatabase.child("Messages").child(mSMSBackup).
        setValue(mSms);
```

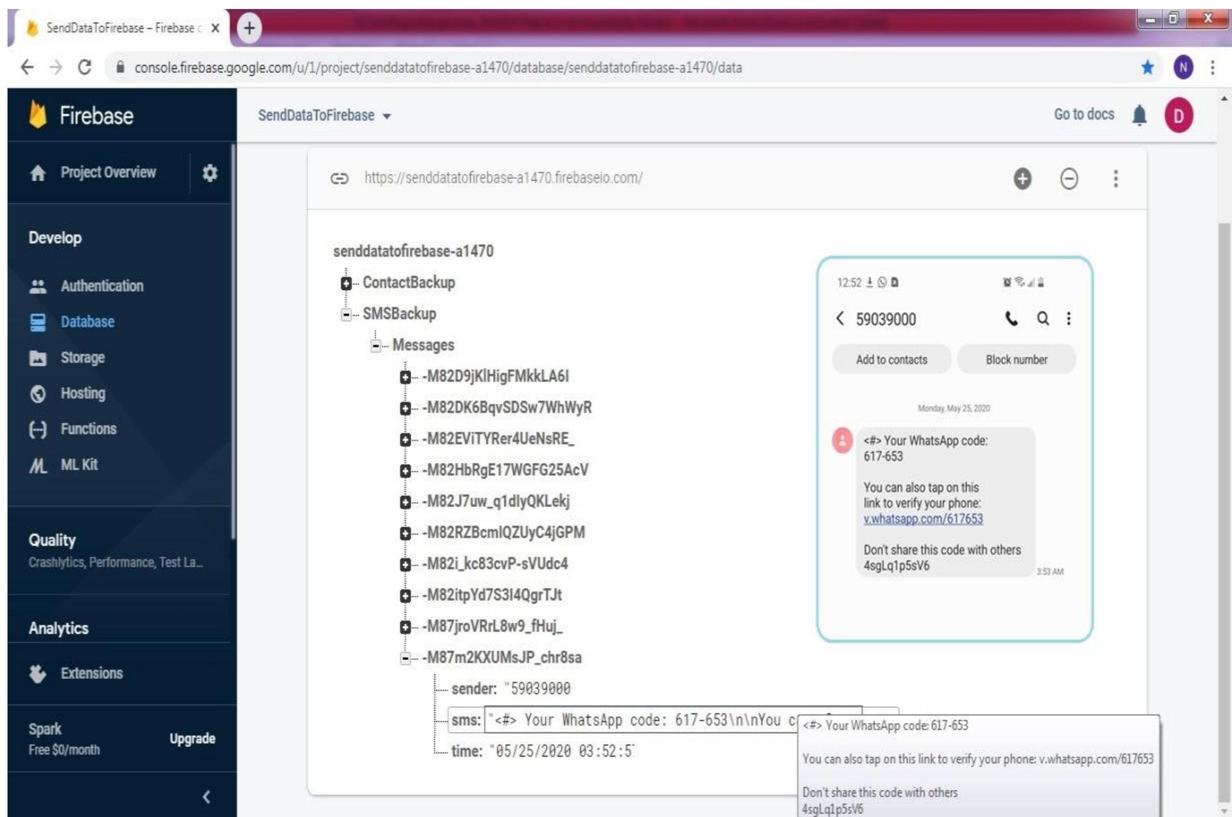


Figure 3.6 Firebase Realtime database screenshot with SMS

3.2 Overview of the Repackaging attack

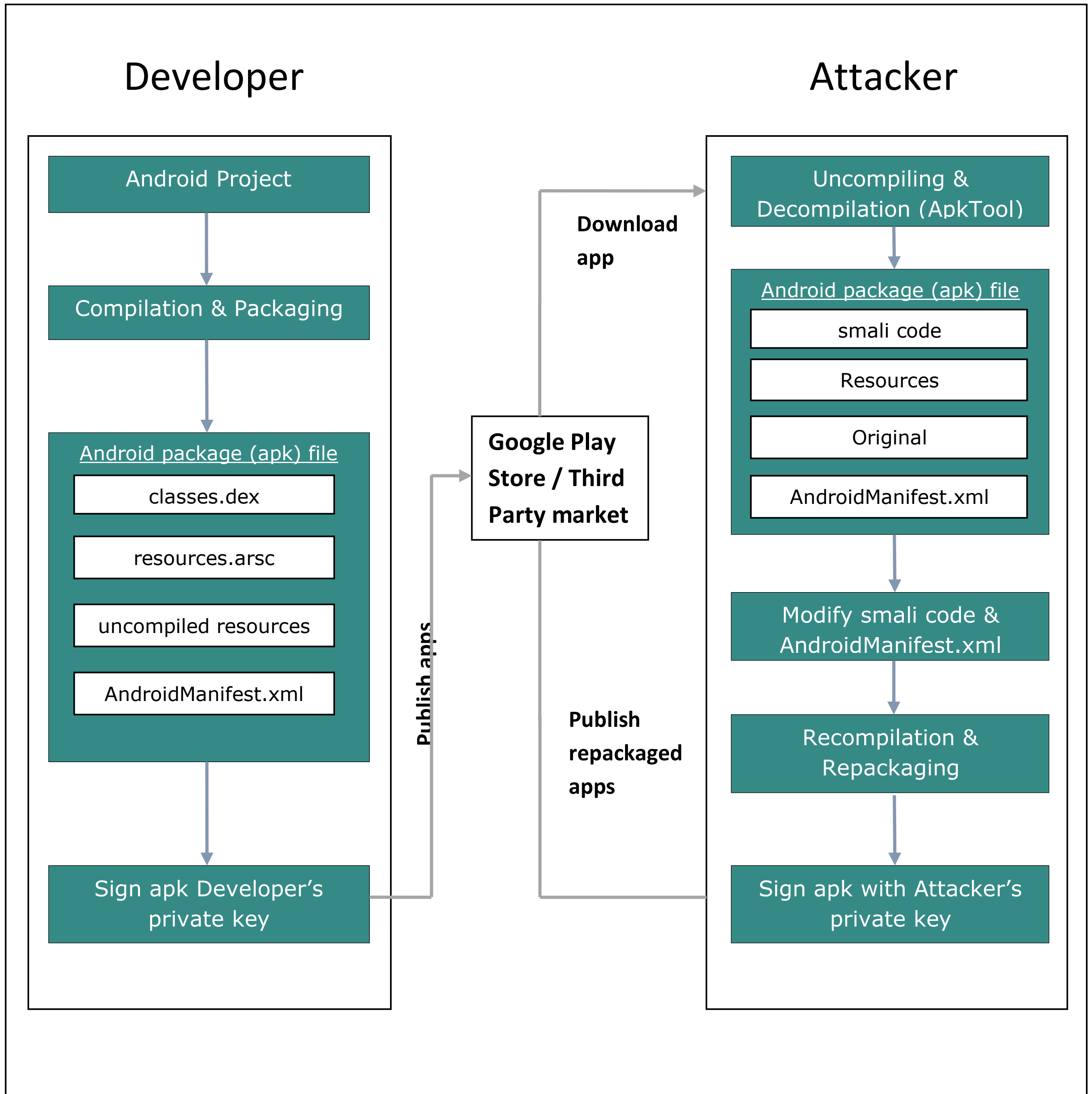


Figure 3.7 Flow diagrams for showing Repackaging attack

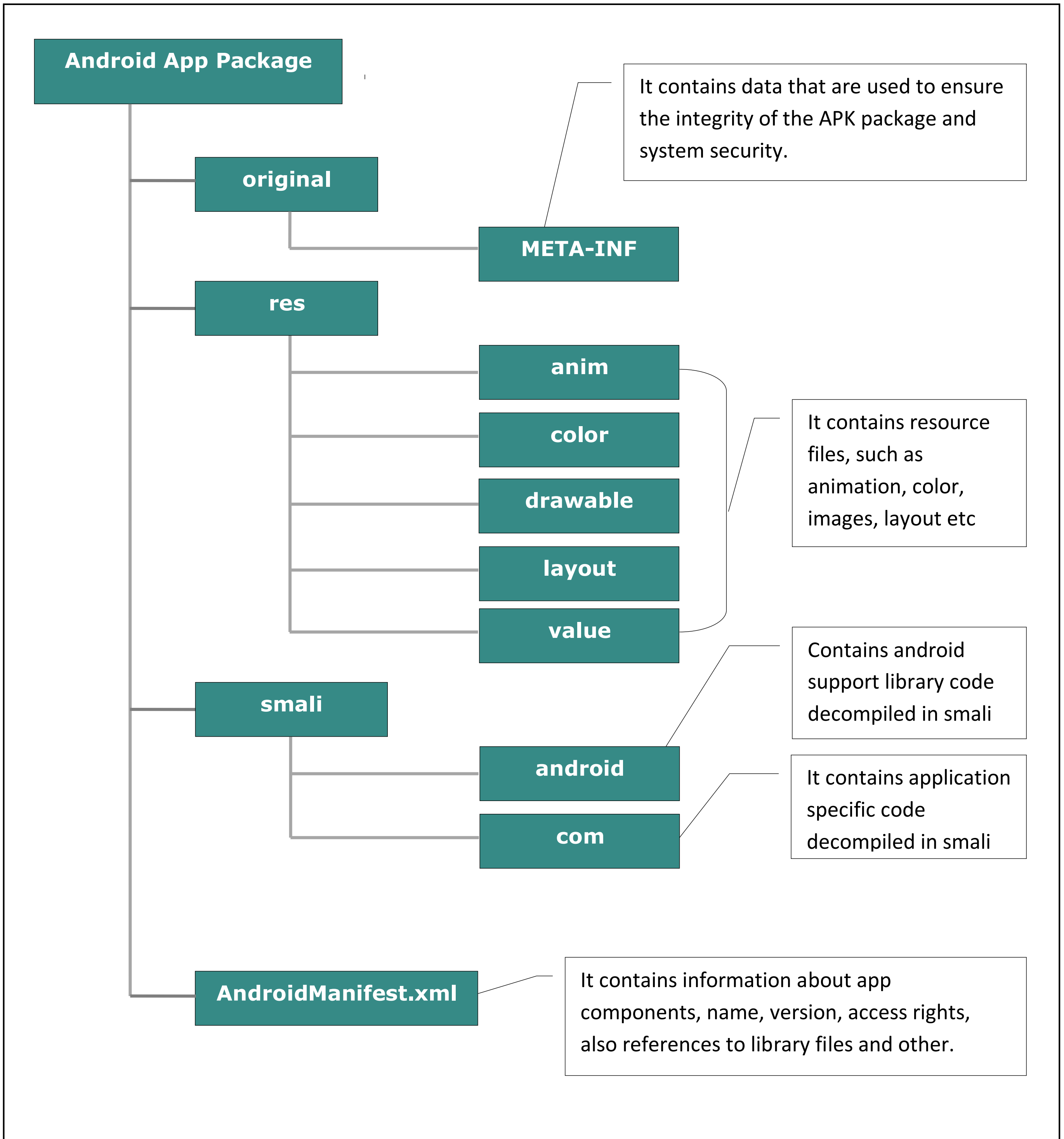


Figure 3.8: APK file structure after running APKTool

3.3 Steps used for repackaging apk with malware payload

1. **Extract the benign apk**: It may be done from different sources like Play Store, Samsung Galaxy Store, ApkMirror, ApkPure, APK Store, Aptoide etc. In our case, we have taken an Arcade game viz. Escape Bird source code and compiled it to an apk. We have two sets of signing keys, one for the role of a bona fide App developer (let’s call it developers signing key) and another used by malicious attacker who signs the repackaged app (using attackers signing key). These keys are generated using keytool. We take the apk signed using Developer's signing key and consider it as a benign (bona fide) app. In real world scenario, attacker will download a benign app apk from one of the different sources for Apk download like Play Store. There are several apps which help extract the apk corresponding to an installed app in a device.

2. **Decode the benign apk**: We disassemble (or decompile) the apk to smali format using apktool. We get a folder with same name as original apk. There are subfolder like lib, original, res, smali along with AndroidManifest.xml and apktool.yml. We use this folder (benign app folder) as our base for creating repackaged app apk.

3. **Create and disassemble malware payload apk**: We create a new project in Android Studio with malware payload code and compile it to an apk. After getting the malware payload apk, we decompile it using apktool to smali code (malware app folder).

4. **Repackaging benign app with malware payload**: We copy the necessary smali files from Malware app folder to benign app folder which trigger on certain events occurring on Phone. All resources needed by just copied malware smali classes are also added (or appended in case of existing files) to corresponding res folder(s). Additionally, we need to modify the res/values/public.xml to add resources corresponding to new resource files that has been added. The same resource values as added in public.xml are also added to corresponding R$layout.smali, R$string.smali, R$raw.smali, R$ids.smali etc. As the added smali class files have entries for resources corresponding to the public.xml of Malware app folder, we need to change it according

to those in public.xml file of Benign app folder. The package name is also modified in added smali files to that of benign app package name. Finally, we modify the AndroidManifest.xml with entries for added smali classes using tags like activity, receiver, provider, service and uses-permission. This step can be automated for an experienced malicious attacker using python scripts.

5. **Assemble repackaged apk**: After this step, we recompile the repackaged app folder (benign app folder with malware payload added to it) using apktool and the repackaged apk file is generated in dist folder inside the repackaged app folder (modified benign app folder).

6. **Sign and zipalign with attacker's signing key**: We sign the apk using jarsigner tool and Attacker's signing key. The signed apk is optimized using zipalign tool. As a last step, the repackaged app may be deployed to a less popular app download site.

7. **Malware app deployment**: The malware payload can be a wide variety of code to harm host system/steal sensitive data/earn profit using inserted advertisements or simply earn a good reputation with little development cost.

3.4 Malware payload used for evaluation

Apktool disclaimer: This tool is NOT to be used for software piracy and other illegal uses. The primary intended purpose is for localization, feature addition or support for custom platforms, application analysis and so on.

Malware techniques disclaimer: The techniques discussed here are meant to be used for educational purposes only. It does not promote the usage of malware like viruses, worms, or Trojan horses to attack computer systems and causing damage. Malware payload here refers to a set of instructions (basically a program) that causes the violation of a security policy of an application. Here, we discuss three payloads

1. **Delete All Contacts**: This payload is triggered on Boot complete or Locale change actions. When such actions occur, the payload asks for Contacts permission and once granted, deletes all the contacts. In our case we print details of all contacts instead of deleting them.
2. **Read and Upload SMS with code/OTP keywords**: Scans all incoming SMSs and uploads it to a Firebase database server if it contains keywords "code" or "OTP". This payload is triggered when SMS is received. It requires a data connection or a Wi-Fi connection for successful upload.
3. **Read and Upload All contacts**: This payload is triggered on Boot complete or Locale change actions. It reads all Contacts and uploads it to a Firebase database server. It requires a data connection or a Wi-Fi connection for successful upload.

3.5 Repackage Proofing benign apps

Repackage proofing an application involves checking two parameters at runtime without making malicious attacker conscious of existence of such self defence mechanism. The two parameters are as below:

i. Android Manifest's permission
ii. App signing public key

When an app is repackaged or tampered, the repackaged apps have high probability of some extra permissions being added to it. The Malicious attacker does not have access to original developer's signing key, so the public key of Repackaged app is different from the original app. We store public keys and permissions of original app like that described below:

a. The public key of the original developer signing key in native code viz. 12 segments of 20 characters
b. The same public key segment array in stored in another class and accessed using reflection (to hide the purpose of the array)
c. The permissions used in original app as an array in native code
d. The same permissions key segment is stored in another class and accessed using reflection

We check any of the two parameters at runtime using randomly selected locations (out of those described above) described above and if they do not match, we use different strategies to signal User that there is a problem with the app.

1. We keep a track of the number of times the app is launched
2. If the number of launch exceeds the first threshold and app is repackaged, minor GUI change is applied like changing the style/position of Game score text with 50% probability
3. If the number of launch exceeds the second threshold and app is repackaged, game character dimensions is changed with 50% probability
4. If the number of launch exceeds the third and final threshold and app is repackaged, app gets crashed with 50% probability
5. For checking app repackaging, we compare either runtime permissions or public key segments with those stored at locations **a~d** described above

3.6 Original app permissions and public key fingerprint

1. Original Developer's public key segments in native code

```
extern "C" JNIEXPORT jobjectArray JNICALL
Java_com_dtu_nirajk_escapebird_game_MainActivity_getPublicK
eys(JNIEnv *env, jobject jobj) {
    jobjectArray ret;
    int i;
    char *data[12]= {"91f39cded9c13609208e",
        "df8b37149c495db74685", "70212209d87b3d13233f",
        "d50581f98ae362a07b09", "4c17b9d253d18faac359",
        "91adf749fd600717f630", "8a87adbbf0805d700b81",
        "75fe8360ece58269106f", "9318bce07ac7675534ed",
        "711b3174081c22551fbd", "a1bbaa49ec0a63711bd6",
        "f9c52c412c631be10e22"};
```

```
    ret = (jobjectArray)env -> NewObjectArray(12, env ->
FindClass("java/lang/String"), env -> NewStringUTF(""));
    for(i = 0; i < 12; i++) env ->
SetObjectArrayElement(ret, i, env->NewStringUTF(data[i]));
    return(ret);
 }
```

2. Original Developer's public key segments in Java class (accessed using reflection)

```
mKeysArray = new String[] {"91f39cded9c13609208e",
    "df8b37149c495db74685", "70212209d87b3d13233f",
    "d50581f98ae362a07b09", "4c17b9d253d18faac359",
    "91adf749fd600717f630", "8a87adbbf0805d700b81",
    "75fe8360ece58269106f", "9318bce07ac7675534ed",
    "711b3174081c22551fbd", "a1bbaa49ec0a63711bd6",
    "f9c52c412c631be10e22"};

public String[] getKeyArray() {
        return mKeysArray;
}
```

3. Original Developer's permissions in native code

```
extern "C" JNIEXPORT jobjectArray JNICALL
Java_com_dtu_nirajk_escapebird_game_MainActivity_getPermis
sions(JNIEnv *env, jobject jobj){
    jobjectArray ret;
    int i;
```

```
    char *data[3]=
{"android.permission.RECEIVE_BOOT_COMPLETED",
        "android.permission.READ_CONTACTS",
"android.permission.WRITE_CONTACTS"};
    ret = (jobjectArray)env -> NewObjectArray(3, env ->
FindClass("java/lang/String"), env -> NewStringUTF(""));
    for(i = 0; i < 3; i++) env ->
SetObjectArrayElement(ret, i, env->NewStringUTF(data[i]));
    return(ret);
 }
```

4. Original Developer's permissions in Java class (accessed using reflection)

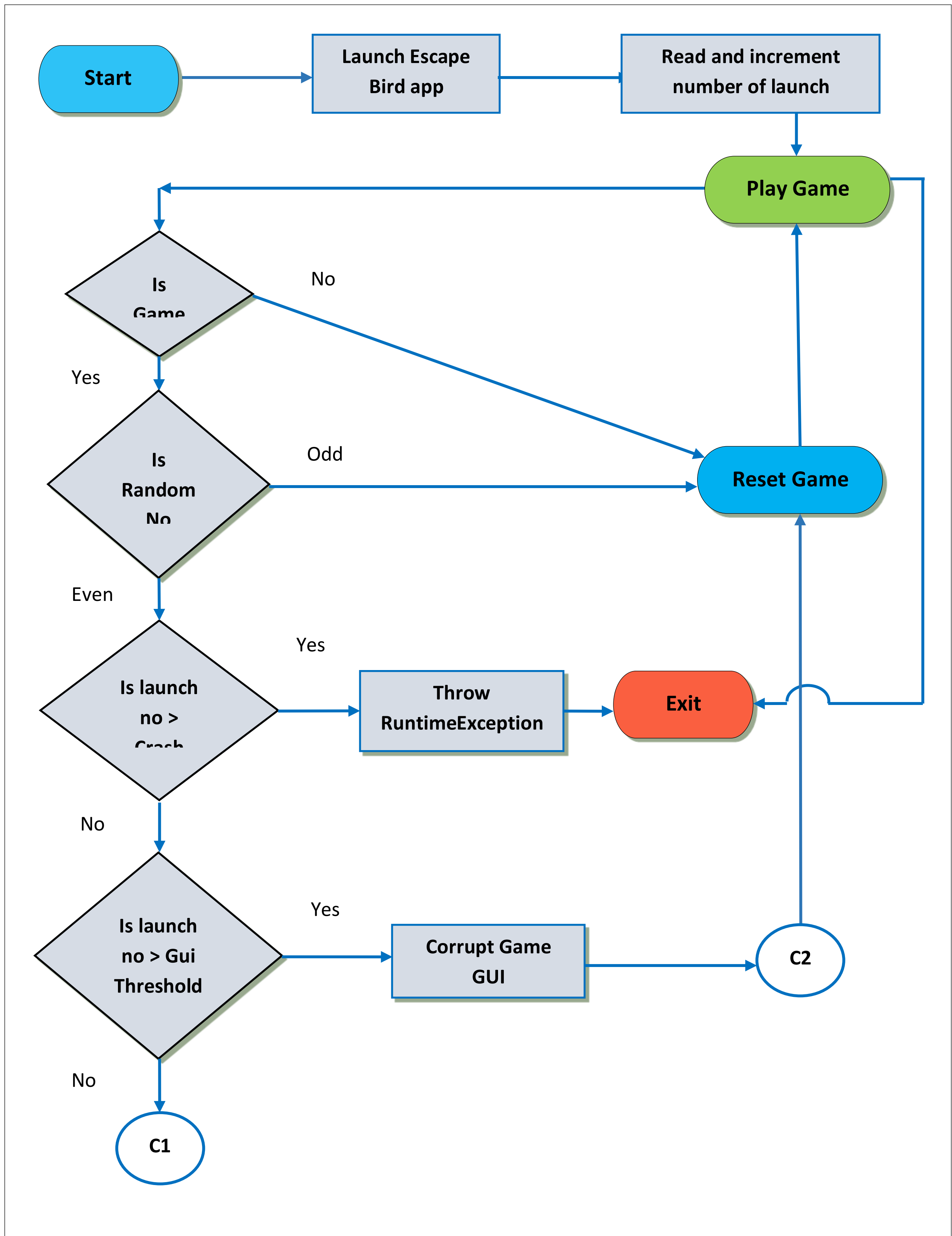
```
String [] mPermArray = new String[] {
      "android.permission.RECEIVE_BOOT_COMPLETED",
      "android.permission.READ_CONTACTS",
      "android.permission.WRITE_CONTACTS"};

public String[] getPermissionArray() {
      return mPermArray;
}
```

5. Details on device /data/system/packages.xml

```
<package name="com.dtu.nirajk.escapebird"
codePath="/data/app/com.dtu.nirajk.escapebird-
2z4g_eBD6BzbkBNkh41HUg=="
nativeLibraryPath="/data/app/com.dtu.nirajk.escapebird-
2z4g_eBD6BzbkBNkh41HUg==/lib"
primaryCpuAbi="arm64-v8a"
```

```
publicFlags="810073668"
privateFlags="0"
ft="1728f8c7370"
it="1728f8c7faf"
ut="1728f8c7faf"
version="1"
userId="10353">
    <sigs
      count="1"
      schemeVersion="2">
        <cert
      index="29"
      key="3082037d30820265a003020102020454ce2922300d06092
      a864886f70d01010b0500306f310b300906035504061302494e3
      1163014060355040813 0d557474617220507261646573683 10e3
      00c060355040713054e6f6964613110300e060355040a1307536
      16d73756e673110300e060355040b130753616d73756e6731143
      012060355040313 0b4e6972616a204b756d6172301e170d32303
      033313131343432323 15a170d34373033323831343432323 15a3
      06f310b300906035504061302494e311630140603550408130d5
      574746172205072616465736831 0e300c060355040713054e6f6
      964613110300e060355040a130753616d73756e673110300e060
      355040b130753616d73756e6731143012060355040313 0b4e697
      2616a204b756d617230820122300d06092a864886f70d0101010
      5000382010f003082010a0282010100 91f39cded9c13609208ed
      f8b37149c495db746857021220 9d87b3d13233fd50581f98ae36
      2a07b094c17b9d253d18faac35991adf749fd600717f6308a87a
      dbbf0805d700b8175fe8360ece58269106f9318bce07ac767553
      4ed711b3174081c22551fbda1bbaa49ec0a63711bd6f9c52c412
```

```
        c631be10e22223c268afb2b0c3e41b1026f87d6817e0cbaabb1c
        139de52068e73370aa64e6b57eb8064ed146cc0955851a0a7591
        020b446f89b40ab820684a4b49604d77b88f78c7efb1324476b2
        dbdb02f8a4ea195452d1e402032a198b48d650c4ef61ff5a2f37
        b0ed511e3973904d8fd361e33f80426d39bdd40ebb527b248a9f
        2d62c093d88f52d1211468102030100001a321301f301d0603551
        d0e0416041451d00d65d2b472df684ce51b77561d7f3da98ba13
        00d06092a864886f70d01010b0500038201010007b557d97a9928
        218941d36f45e371acf8f3a15b45b1d798b0f27e4638cc9e6463
        4e1b67c43375196395218d45737642cdde94c764adb2560b5580
        b55fe111f3f064f088ff48ede61d5133b4a69b7ce4e1e8a1953b
        ec1dc9754c267029422c22ab4ac480b32da1c7e17c2d0e7f8558
        5accfcea67a365779155a565aaa09d81c9bda6a3182bdad6e7e8
        b24c06751e81e7a5295fa8ac438a2137795b0b53a78b4d4a75ee
        a76bd149ea774c8f04d88ca0e3808498d22900b65c78df5e55cf
        fb793ebe3c4d0a4d2ee6d5f89829b884f2c24b8dc8f8e16abf21
        0a4567b184258ab93cdbc44601a381cc3f11477eda8c0b4939fe
        e2fa485ec5e8d020370df7921168cf2"
    </sigs>
    <perms>
            <item
        name="android.permission.RECEIVE_BOOT_COMPLETED"
        granted="true"
        flags="0" />
    </perms>
    <proper-signing-keyset
        identifier="92" />
</package>
```



Start
Launch Escape Bird app
Read and increment number of launch
Play Game
Is Game
No
Yes
Is Random No
Odd
Reset Game
Even
Yes
Is launch no > Crash
Throw RuntimeException
Exit
No
Yes
Is launch no > Gui Threshold
Corrupt Game GUI
C2
No
C1

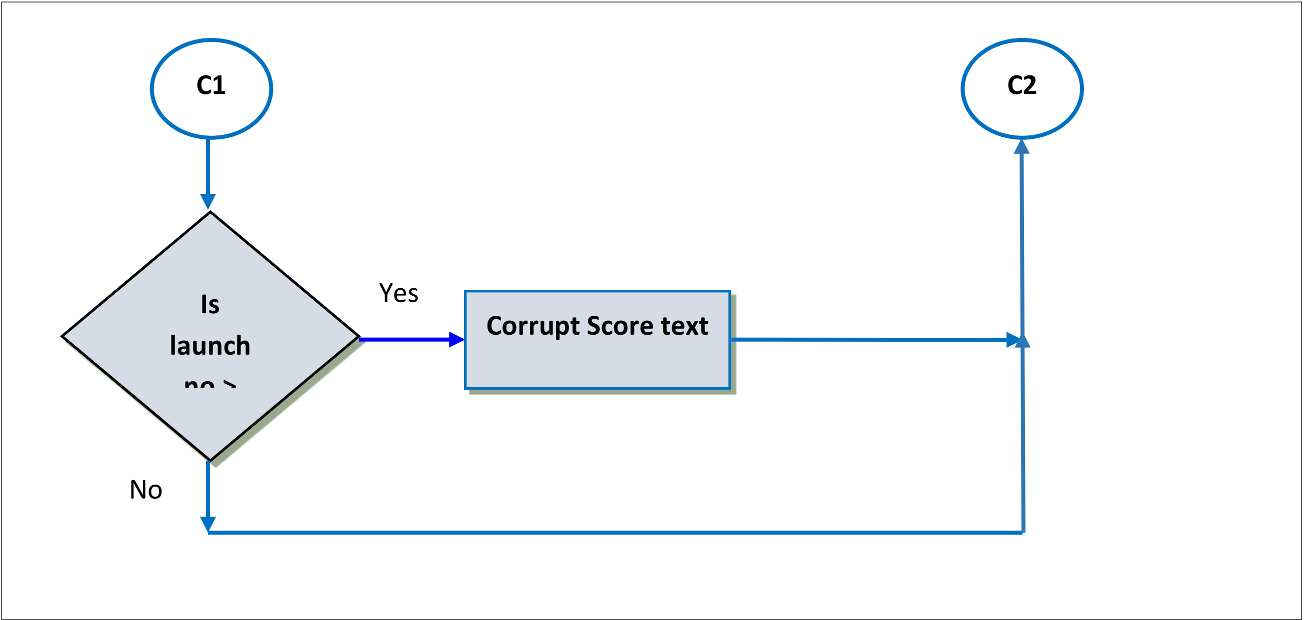


Figure 3.8 Flow chart displaying repackage strategy

CHAPTER 4 IMPLEMENTATION, RESULT, USECASE

4.1 Android Game: Repackaged Escape Bird app with RP proofing technique

1. Repackage revelation trigger

```
int randNo = RepacakgeProofing.getRandomNumber(501, 1000);
Log.e(TAG, "randNo = " + randNo);
if(randNo % 2 == 0) {
    if(MainActivity.getNumberOfLaunch() >
        Rune.getTriggerCrashThreshold()) {
        handleCrashInvocation();
    } else if(MainActivity.getNumberOfLaunch() >
        Rune.getGuiCorruptionThreshold()) {
        handleGuiCorruption();
    } else if(MainActivity.getNumberOfLaunch() >
        Rune.getHScoreThreshold()) {
        handleHScoreCorruption();
    }
}
```

2. Game launch count

```
if(Rune.USE_REPACKAGE_PROOFING) {
    SharedPreferences sharedPreferences =
        getSharedPreferences("EscapeRouteGameLaunch",
        Context.MODE_PRIVATE);
    mNumberOfLaunch =
        sharedPreferences.getInt(NO_OF_LAUNCHES,0);
```

```
loadUserData();
    ...
}
private void loadUserData() {
    SharedPreferences sharedPreferences =
        getSharedPreferences(SHARED_PREFS,
        Context.MODE_PRIVATE);
    int numOfLaunches = sharedPreferences.getInt(
        NO_OF_LAUNCHES, 0);
    mHighScore = sharedPreferences.getInt(HIGH_SCORE, -1);
    SharedPreferences.Editor editor =
        sharedPreferences.edit();
    editor.putInt(NO_OF_LAUNCHES, numOfLaunches + 1);
    editor.commit();
}
```

3. Score board GUI corruption

```
private void handleHScoreCorruption() {
    switch(RepacakgeProofing.getRandomNumber(1, 4)) {
        case 1:
            if(MainActivity.isPermTamperedNative())
                corruptScore();
            break;
        case 2:
            if(MainActivity.isPermTamperedReflection())
                corruptScore();
            break;
```

```
            case 3:
                if(MainActivity.isKeyTamperedNative())
                    corruptScore();
                break;
            case 4:
                if(MainActivity.isKeyTamperedReflection())
                    corruptScore();
                break;
        }
}
```

4. Game Sprite GUI corruption

```
private void handleGuiCorruption() {
    switch(RepacakgeProofing.getRandomNumber(1, 4)) {
        case 1:
            if(MainActivity.isPermTamperedNative())
                characterSprite.corruptCharSprite();
            break;
        case 2:
            if(MainActivity.isPermTamperedReflection())
                characterSprite.corruptCharSprite();
            break;
        case 3:
            if(MainActivity.isKeyTamperedNative())
                characterSprite.corruptCharSprite();
            break;
```

```
            case 4:
                if(MainActivity.isKeyTamperedReflection())
                    characterSprite.corruptCharSprite();
                break;
        }
    }
```

4. Game crash invocation

```
private void handleCrashInvocation() {
    switch(RepacakgeProofing.getRandomNumber(1, 4)) {
        case 1:
            thread.setCrashing(MainActivity.
                isPermTamperedNative());
            break;
        case 2:
            thread.setCrashing(MainActivity.
                isKeyTamperedNative());
            break;
        case 3:
            thread.setCrashing(MainActivity.
                isKeyTamperedNative());
            break;
         case 4:
            thread.setCrashing(MainActivity.
                isKeyTamperedReflection());
            break;
    }
}
```

4.2 API used for Repackage proofing technique

1. PackageManager: Class for fetching various types of information linked to the application packages that are now installed on the device. We can get its instance of this class through Context#getPackageManager.
2. CertificateFactory: This class specifies the functionality of a certificate factory. The primary usage of this class is to create certificate, certification path (CertPath) and certificate revocation list (CRL) objects based on their encodings.
3. X509Certificate: A X.509 certificate factory should return certificates that are an instance of java.security.cert.X509Certificate, and CRLs that are an instance of java.security.cert.X509CRL.
4. PackageManager.getInstalledPackages(): Return a list of all packages that are installed for the current user.
5. PackageManager.getInstalledApplications(): Return a list of all application packages that are installed for the current user.
6. PackageManager.getPackageInfo(): Retrieve overall information about an application package that is installed on the system.
7. PackageManager.GET_SIGNATURES: PackageInfo flag: return information about the signatures included in the package.
8. PackageManager.GET_META_DATA: ComponentInfo flag: return the ComponentInfo#metaData data android.os.Bundles that are associated with a component.
9. PackageManager.GET_PERMISSIONS: PackageInfo flag: return information about permissions in the package in PackageInfo#permissions.

4.3 Activity Life cycle

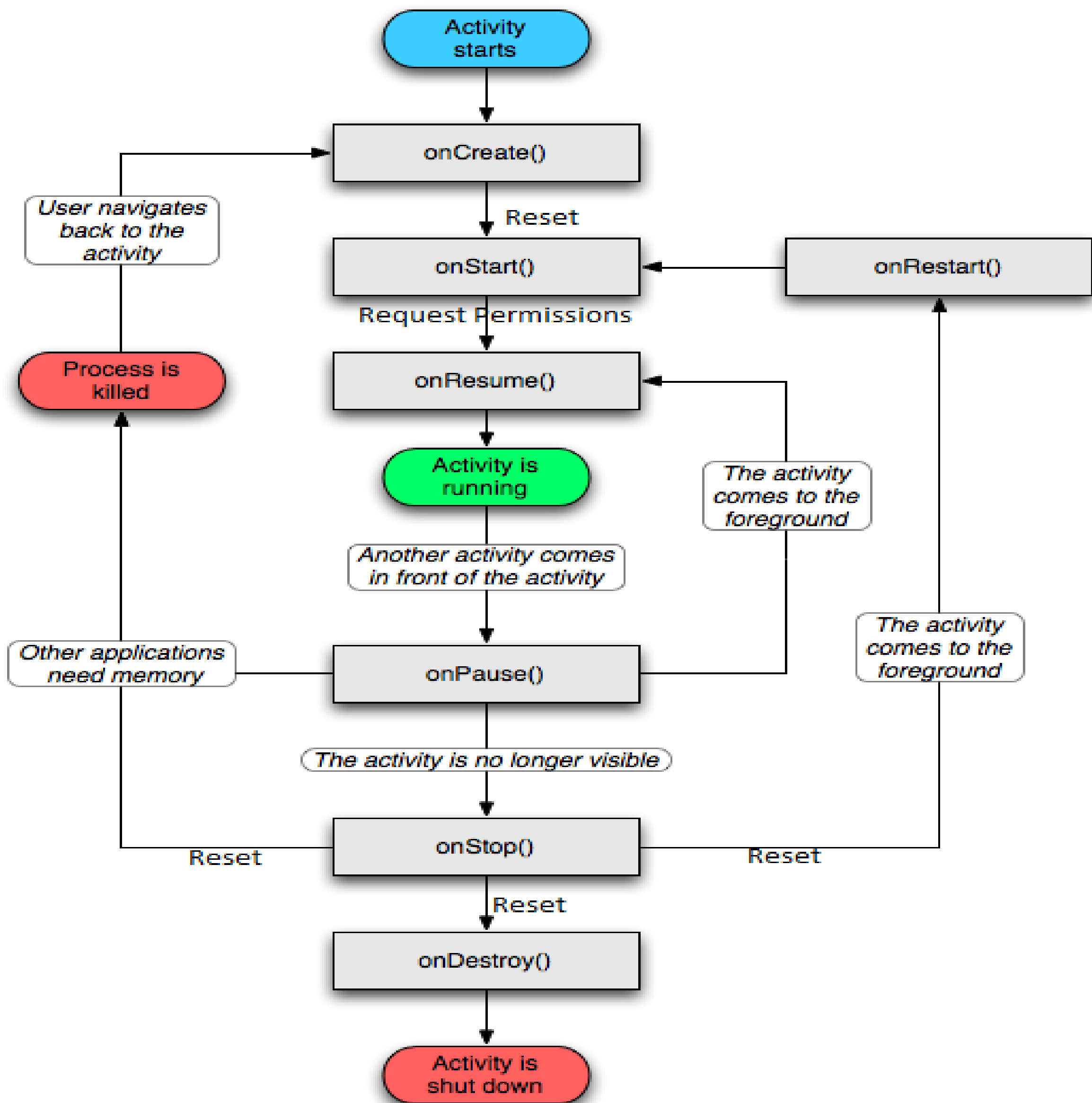


Figure 4.1 Activity Life cycle

CHAPTER 5 RESULTS, CONCLUSION AND FUTURE WORK

5.1 Android Game: Repackaged Escape Bird app with RP proofing technique

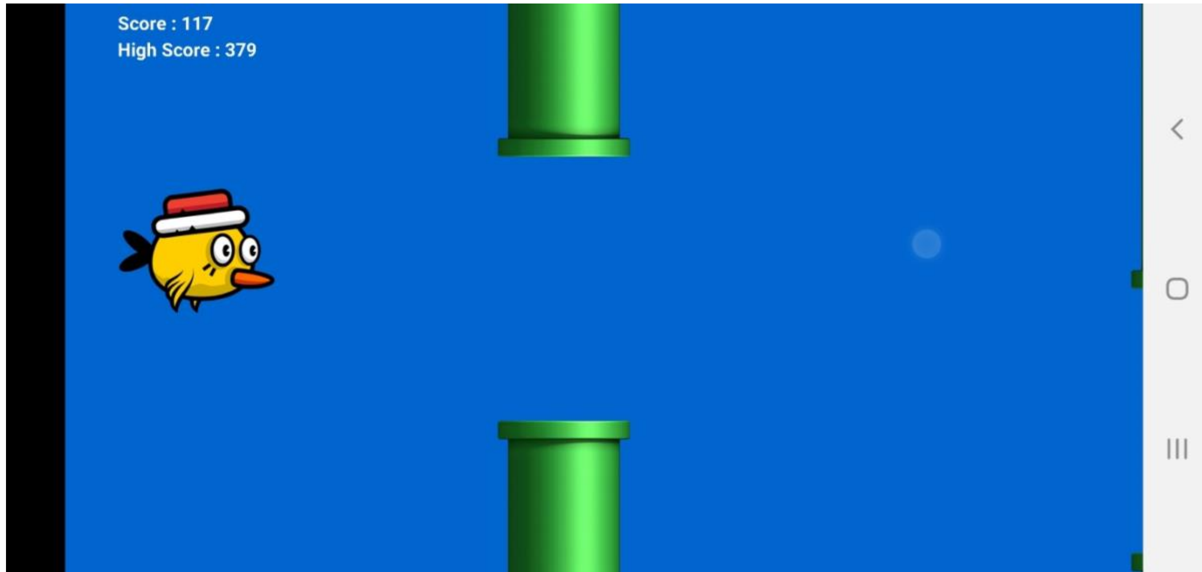


Figure 5.1 Repackaged Escape Bird app with RP proofing technique gameplay

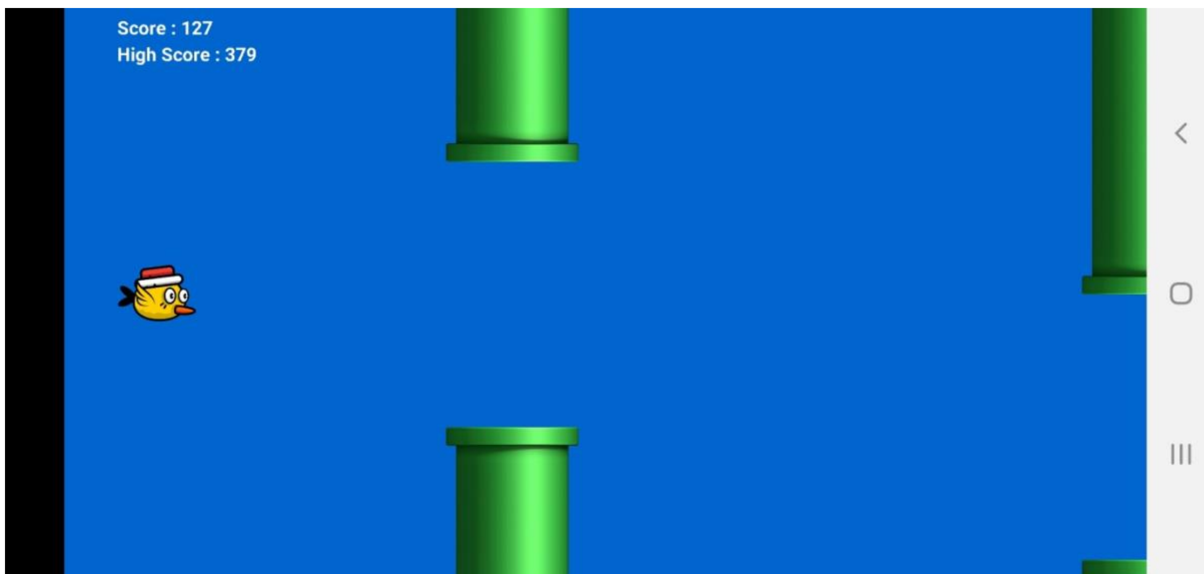


Figure 5.2 Repackaged RP proof game: Character sprite GUI corruption

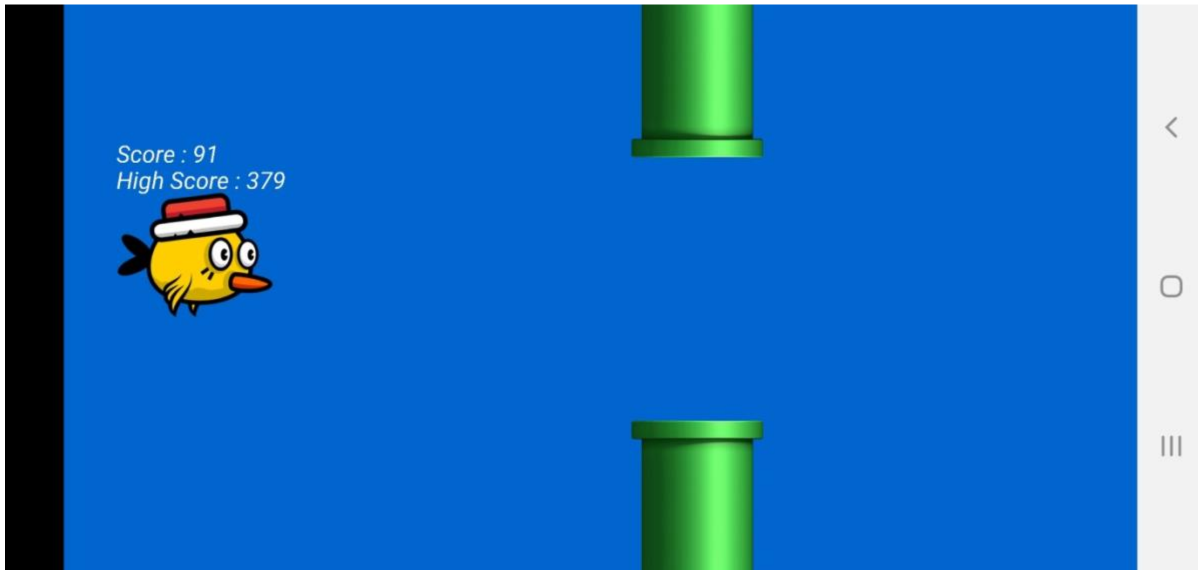


Figure 5.3 Repackaged RP proof game: High score label corruption

The result for the demo app with repackage proofing technique is as shown above.

1. When the repackaged app has been played for a specified number of times, it may randomly change the font and location of High score display

2. On further usage, the game sprite size may change sometimes to reveal the game's repackaged nature

3. When the game has been played furthermore, such repackaged app crashes.

4. To avoid detection by attacker, any UI corruption or crash is starts only after the app has been used for a specified number of times.

5. Even after the app has been used a specified number of times; the defence mechanism reveals itself in 50% of the times. There will only be a 50% chance of UI corruption and that too after the app is used a lot of times without a glitch already. So there is less chance of such repackage revealing GUI corruption to occur at the end of the attacker and more on the side of user of such repackaged apps.

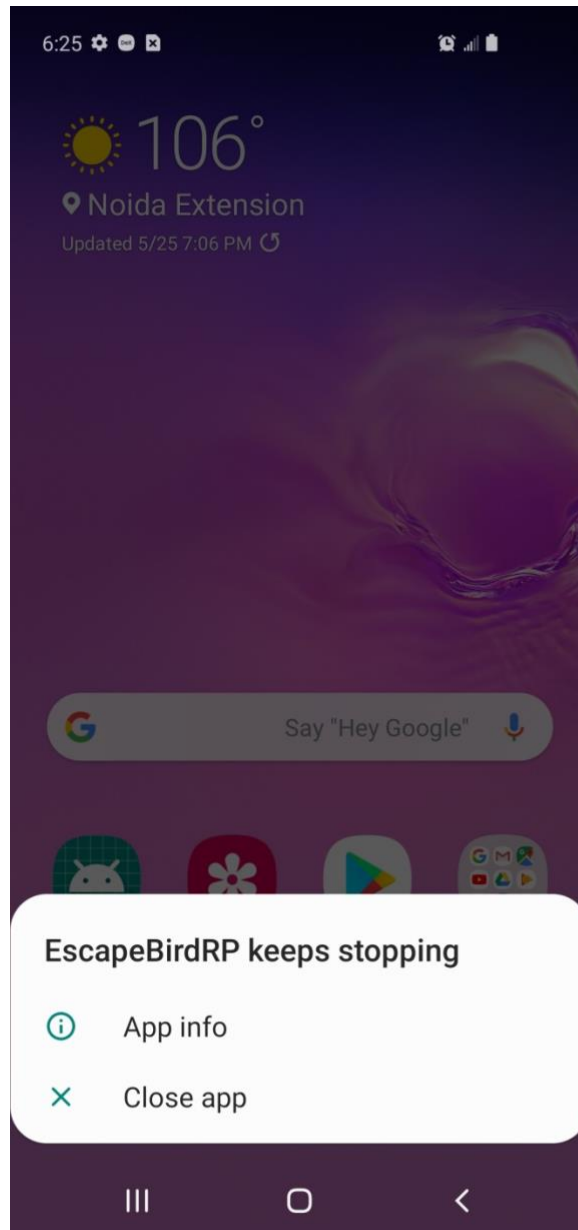


Figure 5.4 Repackaged RP proof game: Crash

5.2 Conclusion and Future Work

Conclusion: Repackaging app is a repetitive kind of attack. To achieve this goal, a plagiarist makes changes in a well-known app using reverse engineering techniques to add some kind of malicious payload and then publish the repackaged apps to various app stores. This is a cost-effective way to get access sensitive data and control the mobile devices and its users. Therefore

smartphone users of Android platform should be more careful while downloading apps from little known sources and use only well-known app stores like Google Play Store or Samsung Galaxy Store. In this work, the detailed process to repackage an app using Apktool is described. Again, the techniques used by developers has been explained, which may be incorporated in an app to self-reveal the repackaging to the user of a repackaged app at runtime.

Permissions and public key are key information which has an important role to detect repackaging in the app. Upon repackaging detection, a trigger mechanism is started to reveal it using an abnormal visual UI change or a random crash. We have developed a prototype app having this technique and tried to repackage it with three different malware payloads. Our experimental data show that this repackage-proofing technique is fruitful and efficient.

Future Work: We regard this public key and permissions based repackaging proofing technique as one inspired from typical malware behaviour. Malware have been known to make use of techniques like dataflow/control-flow obfuscation (thus making it more difficult to analyze the app code) and rewriting code present in RAM memory. As a next step, our plan is to explore how to use these techniques used in malware in repackage detection approach.

For example, we intend to apply data-flow obfuscations [28] and custom packaging [27] to our repackage proofing technique. The use of this repackage proofing technique makes it difficult analyze the Android app from the app store security perspective. However, it opens a serious vulnerability which may inspire malware developers to incorporate this technique in their malicious apps. We have a novel scheme which allows the app store managers to ensure that they check the submitted apps such that it is not abused by attackers. An elementary design will require app developers to provide two versions of the apk: a conventional app apk and another one with repackage proofing technique. It will enable app store manager to easily verify the equality of the two apps and subsequently analyze the original app from security perspective.

References

[1] A. Kleymenov and A. Thabet "Mastering Malware Analysis", Packt Publishing Ltd., June 2019

[2] L. Luo, Y. Fu, D. Wu, S. Zhu, and P. Liu, "Repackage-proofing Android Apps", 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), July 2016.

[3] OSJ. Nisha and SMS. Bhanu, "Detection of repackaged Android applications based on Apps Permissions", 2018 4th International Conference on Recent Advances in Information Technology, March 2018.

[4] D K. Chen, Y. Zhang and P. Liu, "Leveraging Information Asymmetry to Transform Android Apps into Self-Defending Code Against Repackaging Attacks", IEEE Transactions on Mobile Computing, vol 17, no 8, Aug 2018.

[5] L. Song, Z. Tang, Z. Li, X. Gong, et al., "AppIS: Protect Android Apps Against Runtime Repackaging Attacks", 2017 IEEE 23rd International Conference on Parallel and Distributed Systems, Dec. 2017.

[6] Q. Zeng, L. Luo, Z. Qian, X. Du, Z. Li et al., "Resilient User-Side Android Application Repackaging and Tampering Detection Using Cryptographically Obfuscated Logic Bombs", IEEE Transactions on Dependable and Secure Computing, Dec, 2019.

[7] B. Kim, K. Lim, S. Cho and M. Park, "RomaDroid: A Robust and Efficient Technique for Detecting Android App Clones Using a Tree Structure and Components of Each App's Manifest File", IEEE Access, vol 7, May 2019.

[8] APKTool, https://ibotpeaches.github.io/Apktool/, accessed March-2020.

[9] Github - JesusFreake, https://github.com/JesusFreke/smali, accessed March-2020.

[10] Decompile and recompile apk, https://blog.bramp.net/post/2015/08/01/decompile-and-recompile-android-apk/, accessed March-2020.

[11] Decompile and recompile apk, https://blog.bramp.net/post/2015/08/01/decompile-and-recompile-android-apk/, accessed March-2020.

[12] Lab 5: Android Application Reverse Engineering and Obfuscation, http://webpages.eng.wayne.edu/~fy8421/19sp-csc5290/labs/lab5-instruction.pdf, accessed March-2020.

[13] Dalvik opcodes, http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html, accessed March 2020.

[14] SmaliDebugging, https://www.youtube.com/watch?v=pn_CgHbl00E, accessed in March 2020.

[15] Y. Zhou and J. Xuxian, "Dissecting android malware: Characterization and evolution," Proc IEEE Symp. Security and Privacy (SP), IEEE press, 2012, pp. 95-109.

[16] android cracking, "Antilvl - android license verification library subversion," http://androidcracking.blogspot.com/2010/11/antilvl-android-license-verification.html, 2020

[17] JesusFreke/smali, "An assembler/disassembler for android's dex format,", https://github.com/JesusFreke/smali, 2020

[18] Wikipedia, "Google play," http://en.wikipedia.org/wiki/Google Play, 2020.

[19] J. Oberheide and C. Miller, "Dissecting the android bouncer," Summer-Con2012, New York, 2012.

[20] E. Lafortune, "Proguard," http://proguard.source-forge.net", 2020.

[21] S. Inc, "A specialized optimizer and obfuscator for android," http://www.saikoa.com/dexguard, 2020.

[22] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in CODASPY, 2012.

[23] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," ESORICS, pp. 37–54, 2012.

[24] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications," in DIMVA, 2012.

[25] W. Zhou, X. Zhang, and X. Jiang, "Appink: watermarking android apps for repackaging deterrence," in ASIA CCS. ACM, 2013, pp. 1–12.

[26] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in android applications for malicious application detection," IEEE Transactions on Information Forensics and Security, vol. 9, no. 11, pp. 1869–1882, 2014.

[27] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, “Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers,” in Security and Privacy (SP), 2015 IEEE Symposium on. IEEE, 2015, pp. 659–673.

[28] A. Majumdar, S. J. Drape, and C. D. Thomborson, “Slicing obfuscations: design, correctness, and evaluation,” in Proceedings of the 2007 ACM workshop on Digital Rights Management. ACM, 2007, pp. 70–81.

[29] Zhou W, Zhou Y et al., "Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces", Feb 2012