Code obfuscating a Kotlin-based App with Proguard

Yash Shah
Computer Science, SBMP
Mumbai, India
yashshah518@gmail.com

Jimil Shah
Computer Science, SBMP
Mumbai, India
shahjimil35@gmail.com

Krishna Kansara
Computer Science, SBMP
Mumbai, India
kansarakrishna@rediffmail.com

Abstract— **The convenience of using Android-based smartphones has made Android immensely popular. Until now, Java dominated other Android App Development languages such as C# and Corona, while Kotlin seems to be the future of Android App Development. Though Kotlin provides additional perks to Android app developers, some loopholes still exist that provide a way for security threats to penetrate. One of the most common security threats is App Repackaging. It is a dangerous and malicious attack which exploits the open-source platform. App Repackaging can be minimized by Proguard, a multi-purpose code obfuscator. In this paper, we have implemented prevention against App Repackaging using Proguard with Kotlin-developed Android app. It aims to assist android app developers in securing their apps against App Repackaging attacks.**

Keywords— Android; Kotlin; APK; App Repackaging; Code obfuscation; Proguard

I. INTRODUCTION

Smartphones have come a long way since their original release in 1992. Not only have they evolved in design and architecture, but also in their primary purpose. From being designed to be used for making phone calls, smartphones today can be used for taking pictures, browsing the web, playing music and even as a remote control for IR-controlled devices. Mobile operating systems have had to be at par with these new purposes of smartphones. Mobile operating systems have evolved a lot in the last decade, or so. A variety of mobile operating systems are available right now in the market. With more than 2 billion monthly active devices, as stated in Google I/O 2017, Android is one of the most preferred mobile operating system [1]. Android is a mobile operating systems for smartphones and tablets, built on a Linux foundation. Google purchased the initial developer of the software, Android Inc., in 2005. On November 5, 2007, the unveiling of Android distribution was announced with the founding of the Open Handset Alliance [2]. Providing just a mobile operating system isn't enough. There's a need for providing applications that may be run on the operating system for providing various functionalities. Android provides a rich application framework for developing apps. App developers are provided with a range of object-oriented languages for development purposes, including Java, C#, Python, Corona and many more. Java is the primary language used by most Android App Developers. However, the Google I/O 2017, held from May 17 – May 19, witnessed granting of first class citizenship to Kotlin for app development [3]. Kotlin is a language from JetBrains, and is statically typed. Kotlin identifies the data type using type inference, unlike Java, where type has to be specified by programmer. Nevertheless, Kotlin is fully 'inter-operable' with Java. Kotlin has gained some popularity and was already being used by various Android app developers using a plugin. Several popular apps (eg: Basecamp) were developed solely using Kotlin [4]. Adrian Ludwig, director of Android security states, "Our goal is to make Android the safest computing platform in the world." [5] But, being open source makes Android a fertile ground for a large variety of Android vulnerabilities, which may also appear in benign apps through the accidental inclusion of coding mistakes or design flaws. There are a variety of security issues on Android phones, such as unauthorized access from one app to the others, permission escalation, repackaging apps to inject malicious code, colluding, and Denial of Service (DoS) attacks [6]. This paper pivots around app repackaging attack on Android apps. We build an Android app using Kotlin, and implement protection against reverse engineering practice using Proguard.

The outline of the paper is organized as follows: Part II is the literature review, Part III is the System Overview, Part IV is the proposed system, and Part V is the result and analysis section.

II. LITERATURE REVIEW

In [6] Bahman Rashidi and Carol Fung enlist security threats of android. The different types of attacks on android apps are explained. This paper also explains the apk structure which is critical when dealing with android security. It also explains mechanisms that can be employed to maintain the security of an android app.

In [7], Aniket Kulkarni explains the elementary concepts related to code obfuscation. Code obfuscation is a technique that is used to avert reverse-engineering attacks on software. The paper contains a thorough analysis of the various code obfuscation techniques and an example of code obfuscation on an application has also been demonstrated.

We learnt about various security threats and code obfuscation from the above mentioned literature. We implement an advanced method to develop an android app and obfuscate the code.

III. SYSTEM OVERVIEW

Section A describes Kotlin and its advantages over Java. Section B illustrates the APK structure. Section C explains App Repackaging.

A. Kotlin and its advantages over Java

Kotlin is a JVM based language developed by JetBrains. Kotlin was created with Java developers in mind, and with IntelliJ as its main development IDE. The primary goal of Kotlin is to provide a more null-safe, more productive and more concise alternative to all the platforms that currently use Java [8] [9].

Kotlin is apt for developing Android applications, inculcating all the benefits of a modern language for the developers without imposing any new restrictions:

- Compatibility: Kotlin is fully supported in Android Studio and compatible with the Android build system. It is fully compatible with JDK 6, assuring that it runs without any issues on older Android devices.
- Performance: Kotlin brought in the support for inline functions, making lambda codes run even quicker than the same one written in Java. It has a very similar bytecode structure, and hence, a Kotlin application runs as swiftly as an equivalent Java app.
- Interoperability: Being totally interoperable with Java, Kotlin allows app developers to use all existing Android libraries within their Kotlin applications.
- Size: The runtime library of Kotlin is quite compact, because Kotlin runtime adds only a few methods and less than 100K to the size of the .apk file. The size can be further lessened by the use of Proguard.
- Compilation Time: While there are some additional overheads for clean builds, Kotlin supports efficient incremental compilation, which is as fast or sometimes even faster than Java, thus allowing working software to be available at an early stage in the software development life cycle.
- Easy to Learn: Getting started with Kotlin is quite manageable for a Java developer. They can start off with the guidance of the automated Java to Kotlin converter, built-in with the Kotlin plugin. Early developers may also try their hands on the language using Kotlin Koans or online Kotlin development environment i.e. https://try.kotlinlang.org
- Safer: Possible null situations can be checked for in compile time, to prevent execution time exceptions since Kotlin is null safe. However, if we need an object to be null, we need to explicitly specify that it can be null, and then check its nullity before using it [10].

B. *APK Structure*

APK stands for android application package. It is the default application package format for devices that run on android. An APK file is a zip archive file by nature. It includes all the necessary files that make up your app. The files that comprise an APK file are: Java class files, Resource files, and file consisting compiled resources. Pre-installed Android APKs are stored in system/app folder and the user installed ones are stored in data/app folder. If you want to view the contents of an APK file you must change the extension of the file to.zip and open it.

An APK contains the following directories:

- Assets: this directory contains documents in HTML format that provides information about the apps, license agreement and various and various FAQs.
- META-INF: This directory is responsible for maintaining the integrity and security of the APK package and the system. There are numerous files in this directory viz. CERT.RSA, CERT.DSA, CERT.SF, MANIFEST.MF, etc.
- Res: This file includes all the resource files like sounds, graphics, settings etc.
- AndroidManifest.xml: this file covers the name, version, access rights and also references to library files. This file employs Android's binary XML format.
- Resources.arsc: This file contains compiled resources in binary format. It consists of XML content from all configurations of res/values/folder. The contents are stored in archived form.
- Classes.dex: It is a Dalvik Virtual Machine executable file. It is not possible for java Runtime environments to execute a DEX file. Such files can only be executed in Dalvik virtual machine. It consists of compiled java source codes.
- Lib: It contains a compiled code corresponding to the software layer of each category of processor. This directory includes a subdirectory for each category of processor like armeabi, armeabi-v7a, arm64-v8a, x86, x86_64 and mips.

C. *App Repackaging*

A plagiarist may make changes to the app during repackaging of apps. The modifications performed include: replacing of an API library with a plagiarist-owned library, modifying the ad revenue of the app, adding some ads, and introducing unfavourable code into existing method(s), or adding specific method(s) or a class specifically for introducing the malware code. Once the necessary modifications are made, the plagiarist prepares the APK file again. The plagiarist signs the app with his/her own private key so that the public key in the META-INF directory corresponds to that private key. This repackaged harmful app is now released on some unofficial market where users fall victim to it [11].

The original app is first unpacked, to obtain the source code files. The obtained source code is then decompiled. The decompiled source code is then altered by the plagiarist to introduce malware or other threats. This altered source code is recompiled and released again as known app. Fig. 1. shows the flowchart for app repackaging.

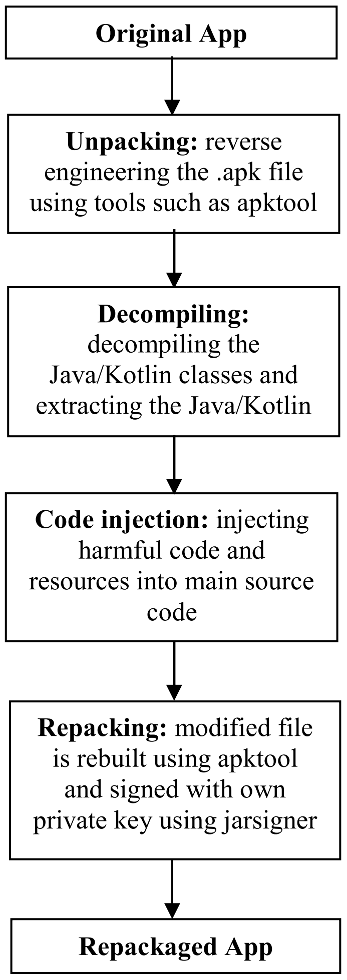

Original App
Unpacking: reverse engineering the .apk file using tools such as apktool
Decompiling: decompiling the Java/Kotlin classes and extracting the Java/Kotlin
Code injection: injecting harmful code and resources into main source code
Repacking: modified file is rebuilt using apktool and signed with own private key using jarsigner
Repackaged App

Fig. 1. Flowchart for App Repackaging

The most fundamental approach to prevent app repackaging is code obfuscation. Code obfuscation is the act of deliberately obscuring the code, complicating it and making it less readable and hard to reverse-engineer. Code obfuscation preserves the semantics and functionality of the original programs and at the same time, prevent the code from reverse engineering attacks.

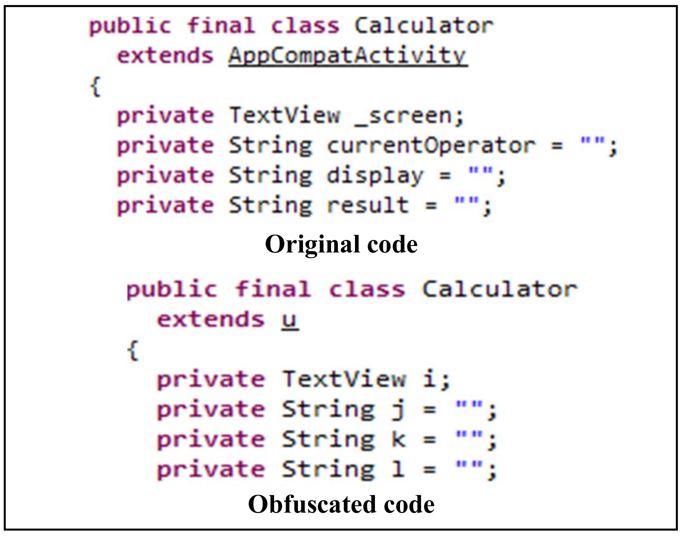
If P → P' is an obfuscating transformation of original program P into obfuscated program P', the following conditions must hold:

- P' may or may not terminate when P fails to terminate or terminates with an error condition.
- If there are no errors, P' must terminate successfully and produce the same output as P [7].

Fig. 2. illustrates an example of code obfuscation.

Code obfuscation techniques are classified into the following:

- Layout Obfuscation - In this technique, layout of the program is changed, by changing original identifiers with random identifiers and removing code comments.



```
public final class Calculator
  extends AppCompatActivity
{
  private TextView _screen;
  private String currentOperator = "";
  private String display = "";
  private String result = "";
```

Original code

```
public final class Calculator
  extends u
{
  private TextView i;
  private String j = "";
  private String k = "";
  private String l = "";
```

Obfuscated code

Fig. 2. An example of code obfuscation

- Control Obfuscation - In this technique, control flow of the program is changed by substituting method calls with method body, creating identical methods with same behaviour, altering loop conditions and body or changing the order of statements in the code. Authors and Affiliations
- Data Obfuscation - In this technique, program data is obfuscated by splitting original variables into multiple variables (preserving the traits of the original variable) and changing variable scope.
- Preventive Obfuscation - The goal of this technique is not to obfuscate the code, but make it tougher to break for deobfuscators. This may be done by exploiting the known weaknesses of the deobfuscators [7] [12].

IV. SYSTEM STRUCTURE

In this paper, we explore the layout obfuscation technique for software protection. To demonstrate about this obfuscation technique, we first develop an app in Kotlin. After developing the app, we apply the obfuscation rules to the app. For obfuscating the Kotlin app in Android Studio, we make use of Proguard. Proguard is specifically a Java file shrinker, optimizer, obfuscator and preverifier. Nonetheless, Proguard works with Kotlin as well. Proguard shrinks the Kotlin app by detecting and removing unused classes, fields, methods and attributes. The next stage is where bytecode of the methods is analyzed and optimized. The remaining classes, fields and methods are then obfuscated with shorter names that make no sense. The app is finally built into an APK file, after Proguard is applied. The generated apk is converted into .zip format, and classes.dex file is extracted from the .zip file. The classes.dex file is given to dex2jar that decompiles the classes.dex into classes.jar. This classes.jar file is opened using jd-gui. The obfuscated source code can now be viewed. The primary purpose of applying Proguard on the developed app is to make reverse engineering of the app and insertion of any malicious source code very difficult. Fig. 3. explains the blocks of our system.

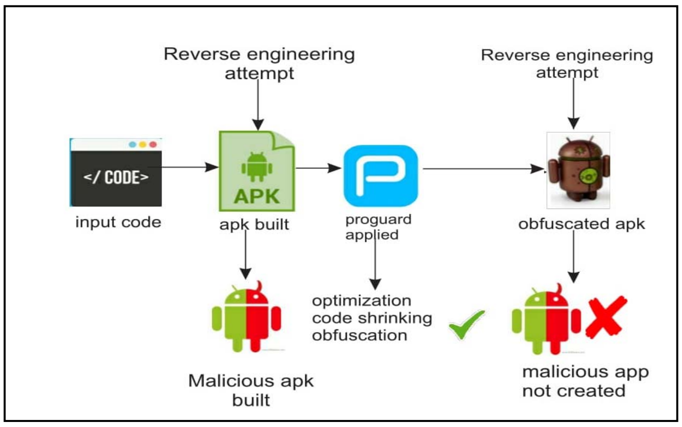
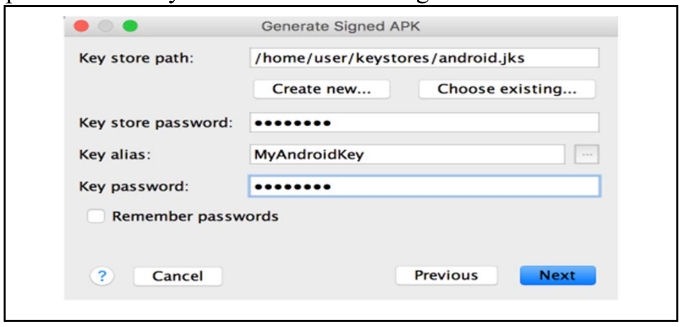


Fig. 3. Blocks of the system

V. RESULT & ANALYSIS

After developing the application, and building an apk file, we decompiled it. We observed that the source code was readable and very easy to alter. Then we built the same application by enabling Proguard. Upon decompiling this apk file, we observed that the source code was obfuscated and was hardly interpretable. Furthermore, the size of this obfuscated file was lesser as compared to the previous apk file. Thus, we inferred that using Proguard provides security and optimization.

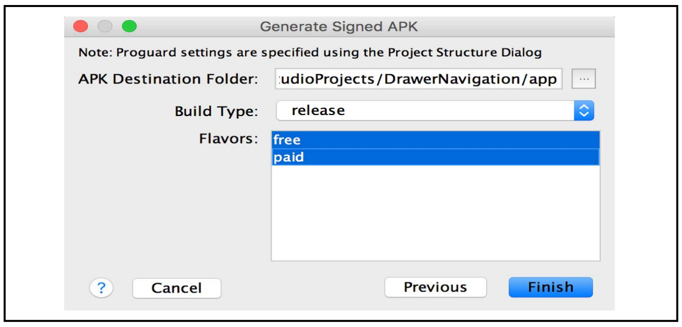
1) The first step to generate the apk of release type is to go to *Build -> Generate Signed APK* and specify the keystore path and the key that will be used to sign the APK.



Key store path: /home/user/keystores/android.jks
Key store password: ••••••••
Key alias: MyAndroidKey
Key password: ••••••••
Remember passwords

Fig. 4. Specify the keystore path and key to use in Android Studio [13]

2) The next step is to specify the details of the APK file that is going to be generated.



Note: Proguard settings are specified using the Project Structure Dialog
APK Destination Folder: udioProjects/DrawerNavigation/app
Build Type: release
Flavors: free, paid

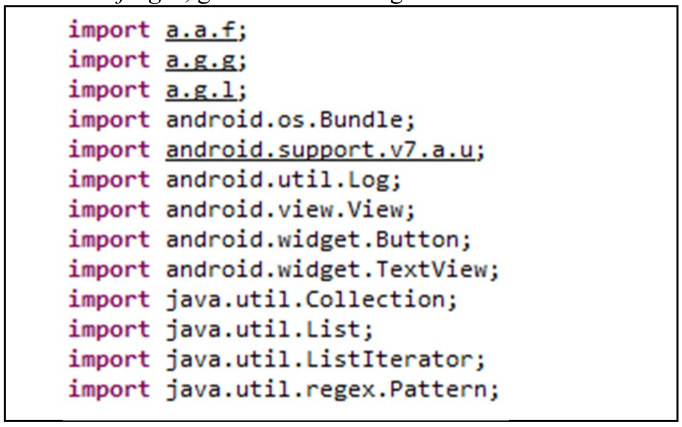
Fig. 5. Generating the signed APK [13]

3) Proguard is enabled in Android Studio by setting the *minifyEnabled* property to true. After the apk is generated, the classes.dex file is extracted by renaming the .apk file with .zip extension.

4) The extracted classes.dex file is then given to dex2jar converter which converts the classes.dex file into a .jar file.

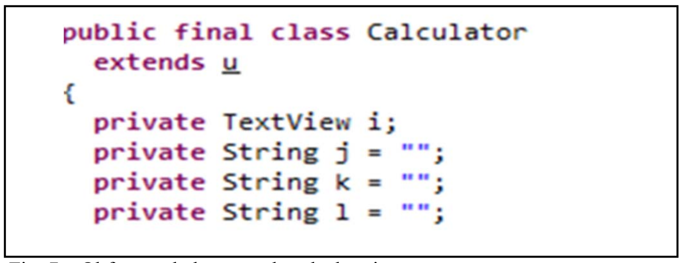
5) The contents of the generated .jar file can be viewed by opening the .jar file in *jd-gui*.

6) The Calculator.class class in our classes.dex, when viewed in *jd-gui*, gives the following obfuscated code:



```
import a.a.f;
import a.g.g;
import a.g.l;
import android.os.Bundle;
import android.support.v7.a.u;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import java.util.Collection;
import java.util.List;
import java.util.ListIterator;
import java.util.regex.Pattern;
```

Fig. 6. Obfuscated import statements



```
public final class Calculator
  extends u
{
  private TextView i;
  private String j = "";
  private String k = "";
  private String l = "";
```

Fig. 7. Obfuscated class member declarations

7) This obfuscated code is hard to make sense of. Deobfuscating this code is tedious and difficult. So, this method of obfuscation lowers the possibilities of a repackaging attack.

8) Another advantage of using Proguard is that it provides resource shrinking and code optimization as well. The *app-debug.apk* file, generated without Proguard enabled, is of size *1.6 MB*. The *app-release.apk* file, with Proguard enabled, is of size *800 KB*.

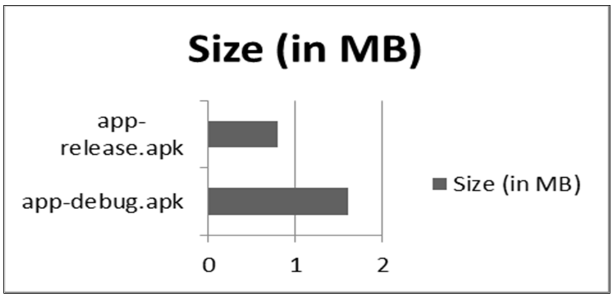


Fig. 8. Size comparison of app-debug.apk file and app-release.apk file

VI. FUTURE SCOPE

Out of the finite code obfuscators available in the market, Proguard is one of the most widely opted for. Proguard's open source and free availability are major factors that need to be rewarded for making it one of the most widely used. Furthermore, Proguard is integrated with several IDEs like Android Studio, Eclipse ME etc. Proguard simply acts as a general optimizer for Java bytecode and fails to provide protection against dynamic analysis attacks. Dexguard hardens the Android-specific object code with multi-layer encryption and also integrates a series of runtime security mechanisms into the app. Apart from the source code, Dexguard also obfuscates, optimizes and encrypts native libraries, manifest files, asset files and resource files. Thus, Dexguard can be used for providing increased security in enterprise-level apps.

VII. CONCLUSION

With several advantages over Java, support from various libraries and first-class language citizenship from Android Studio 3.0, new app developers could surely switch over to Kotlin as their primary language. It minimizes the work for developers by providing null-safe objects and interoperability with the already existing Java code, while on the other hand, it cannot provide protection against malicious attacks on its own. Code obfuscation is a simple approach which can be used to help app developers in securing their apps against repackaging attacks and to efficiently optimize and shrink the source code. This paper has demonstrated how to perform code obfuscation on a Kotlin App in Android Studio, using the Proguard utility. The obfuscated source code obtained is difficult to reverse-engineer and of smaller size than the original app.

REFERENCES

[1] B. Popper, "Google announces over 2 billion monthly active devices on Android", The Verge, 2017. [Online]. Available: https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users/.

[2] "Beginner's guide to Android - Introduction to Android", Beginandroid.com, 2017. [Online]. Available: http://beginandroid.com/intro.shtml.

[3] "Google I/O 2017", Google I/O 2017, 2017. [Online]. Available: https://events.google.com/io/schedule/.

[4] "An introduction to Kotlin for Android development", Android Authority, 2017. [Online]. Available: http://www.androidauthority.com/introduction-to-kotlin-for-android-775678/.

[5] "Security Centre - Overview", Android, 2017. [Online]. Available: https://www.android.com/security-center/.

[6] Bahman Rashidi and Carol Fung, "A Survey of Android Security Threats and Defenses", unpublished

[7] A. Kulkarni, "Software Protection through Code Obfuscation", College of Engineering Pune, 2012.

[8] Antonio Leiva, Kotlin For Android Developers, 1st Ed., North Charleston : CreateSpace Publishing Platform, 2016

[9] Dmitry Jemerov and Svetlana Isakova, Kotlin in Action, Connecticut : Manning Publications, 2017

[10] Kotlin, 2017. [Online]. Available: https://kotlinlang.org/docs/reference/android-overview.html.

[11] S. Rastogi, K. Bhushan and B. Gupta, "Android Applications Repackaging Detection Techniques for Smartphone Devices", ScienceDirect, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050916000089.

[12] "Code Obfuscation", Paladion.net, 2005. [Online]. Available: http://paladion.net/code-obfuscation/.

[13] "Sign Your App | Android Studio", Developer.android.com, 2017. [Online]. Available:https://developer.android.com/studio/publish/app-signing.html.