Investigating the Effectiveness of Obfuscation Against Android Application Reverse Engineering
Rowena Harrison

Technical Report
4 March 2015

Information Security Group
Royal Holloway University of London
Egham, Surrey, TW20 0EX
United Kingdom
Investigating the Effectiveness of Obfuscation Against Android Application Reverse Engineering

Supervisor: Professor Keith Mayes

Submitted as part of the requirements for the award of the MSc in Information Security at Royal Holloway, University of London.

I declare that this assignment is all my own work and that I have acknowledged all quotations from published or unpublished work of other people. I also declare that I have read the statements on plagiarism in Section 1 of the Regulations Governing Examination and Assessment Offences, and in accordance with these regulations I submit this project report as my own work.

Signature:

Date:
Executive Summary

The Android market share of the smart phone industry has grown considerably since its release, and the threat from malware-infested applications has risen along with it. One particular threat is from repackaged applications; legitimate applications that have been reverse engineered, modified to include malicious code, repackaged, and then distributed on app stores for download.

In this work, I focus on the reverse engineering of an Android application I created, RowenApp, using the tools ApkTool and dex2jar. The aim of this project is to use heuristics to investigate the reverse engineering process applied to an obfuscated and unobfuscated version of the app.

I created the app, RowenApp, in the Android Software Development Kit, which stores personal banking information of the app user. I created an obfuscated version of the app, RowenAppobsf using the obfuscation tool ProGuard. I reverse engineered both RowenApp and RowenAppobsf using ApkTool and dex2jar.

Both tools successfully reverse engineered both apps. The dex2jar tool provides an easier way to view the reverse engineered source files than ApkTool since these files are in Java. However only ApkTool can provide the application resources to furnish the reverse engineer with the application’s user interface. Therefore, using the two tools together can provide the necessary means to provide a reasonable understanding of the app.

The reverse engineered output of RowenAppobsf provided more of a challenge than that of RowenApp; it required a more extensive analysis and took several more hours to understand the nature of the app, even with the author’s knowledge of the app source code. ProGuard was effective in hindering the reverse engineering process; however since the purpose of the app was still able to be found, this tool is not infallible. One of the drawbacks of the tool is that it is applied when the app is released; an app writer without adequate knowledge of how ProGuard works may inadvertently leave contextual information available which is not obscured by this tool, aiding reverse engineering.

The availability of powerful tools and online support to app plagiarists and malware writers aids the reverse engineering and repackaging processes, creating a problem when trying to protect apps, one which obfuscation cannot fully address.

To extend this work, further techniques available with commercial tools such as DexGuard and Allatori, which extend the obfuscation offered by ProGuard, could be applied to the app to examine their effectiveness.

More test apps, containing all types of computing logic and app functionality, could be produced in order to test these obfuscation tools’ effectiveness against reverse engineering on all different types of coding elements. This would give insight into how these tools obfuscate different pieces of code, and provide input into how to improve these tools so that they are used to the best of their ability.
Acknowledgements

I would like to thank Professor Keith Mayes for all his help and support in this project. I would like to thank him for his encouragement, his guidance when I got stuck, his helpful suggestions on this dissertation and answering any questions I had. I would also like the Smart Card Centre of Royal Holloway for proposing this project.

I would also like to thank the staff of the Information Security Group of Royal Holloway for making this year so enjoyable. I would like to thank them for making my change from physics to information security an easy one, and expanding my horizons with the topics covered this year.

I would like to thank my fellow Information Security masters students for making this an especially good year, in particular Jamie Ellis for introducing me to Android app writing. Finally, thanks also goes to my mother for her continued support, and my father Nicholas Harrison for proof-reading this dissertation.
## Contents

1 Introduction ......................................................... 7  
  1.1 The Smart Phone Industry and its Risks .......................... 7  
  1.2 Motivation for Project ............................................. 7  
  1.3 Aims of the Project ................................................ 8  

2 Literature Review .................................................. 9  
  2.1 Android Operating System .......................................... 9  
    2.1.1 Overview ....................................................... 9  
    2.1.2 Android Applications ......................................... 11  
    2.1.3 Android Security .............................................. 12  
  2.2 Reverse Engineering ............................................... 13  
    2.2.1 Repackaging .................................................... 13  
    2.2.2 Malware and Repackaged Apps ................................. 14  
    2.2.3 Reverse Engineering Tools .................................. 15  
  2.3 Code Obfuscation ................................................ 16  
    2.3.1 Techniques .................................................... 16  
    2.3.2 Obfuscation Tools ............................................. 19  

3 Android Applications ................................................. 21  
  3.1 Android Software Development Kit ................................. 21  
  3.2 Application Files ................................................. 21  
    3.2.1 AndroidManifest.xml ......................................... 21  
    3.2.2 App Resources ................................................ 22  
    3.2.3 Source Files .................................................. 25  
  3.3 Exporting the Application ....................................... 25  

4 RowenApp .......................................................... 26  
  4.1 RowenApp Overview ............................................... 26  
    4.1.1 Launching the App ............................................. 26  
    4.1.2 Adding Information ........................................... 26  
    4.1.3 Retrieving Information ...................................... 28  
    4.1.4 Error Messages ............................................... 30  
  4.2 Source Files ..................................................... 30  
    4.2.1 DBDetails.java ............................................... 30  
    4.2.2 MyDBHandler ................................................ 31  
    4.2.3 Crypto ........................................................ 33  
    4.2.4 MainActivity ................................................. 34  
    4.2.5 AddDetails .................................................... 34  

5 RowenAppobsf ....................................................... 37  
  5.1 ProGuard ........................................................ 37  
  5.2 Creating RowenAppobsf .......................................... 37  

6 Reverse Engineering Tools and Process ............................ 38  
  6.1 ApkTool & TextEdit ................................................ 38  
    6.1.1 Installing ApkTool ............................................ 38  
    6.1.2 Using ApkTool ................................................. 38  
  6.2 dex2jar & JD-GUI ................................................. 40
6.2.1 Installing dex2jar and JD-GUI ........................................ 40
6.2.2 Using dex2jar ............................................................. 40
6.3 App Files and Tool Outputs .............................................. 41

7 Analysis of Results ......................................................... 42
  7.1 RowenApp ApkTool Output Comparison .............................. 42
    7.1.1 Context ................................................................. 45
    7.1.2 Syntax ................................................................. 45
    7.1.3 Opcodes ............................................................... 45
    7.1.4 Methods ............................................................... 46
    7.1.5 Classes ............................................................... 46
    7.1.6 Line Numbers ........................................................ 46
    7.1.7 Conditional Statements ............................................ 47
    7.1.8 Inner Classes ....................................................... 48
  7.2 RowenApp dex2jar Output Comparison ............................... 50
    7.2.1 Java ................................................................. 50
    7.2.2 Context ............................................................... 50
    7.2.3 Resource Referencing .............................................. 51
    7.2.4 Variable Names .................................................... 52
  7.3 RowenAppobsf Comparisons ........................................... 53
    7.3.1 Class Names ....................................................... 53
    7.3.2 Method Names ..................................................... 53
    7.3.3 Variable Names .................................................... 54
    7.3.4 Remaining Context ................................................ 55
    7.3.5 Code Shrinking .................................................... 56
    7.3.6 Activities ........................................................... 56
  7.4 Summary ...................................................................... 56

8 Conclusions ................................................................. 58
  8.1 Future Work .............................................................. 59
1 Introduction

1.1 The Smart Phone Industry and its Risks

Smart phones are now becoming the norm; emarketer.com [1] estimates that in 2014, smart phone users will total 1.75 billion, accounting for 38.5% of all mobile phone users. The shift from non-smart to smart phones is rising because of the growing range of phone features, such as Internet capability and Global Positioning System (GPS), and the ability to create and download applications.

One significant player in the smart phone arena is Android [2], a Linux-based operating system, released in 2007 by Google. It has grown rapidly and now dominates - from a market share of 2.8% in 2009 [3], to 81% at the end of 2013 [4]. The popularity of the Android operating system, and subsequently its applications, has made it a major target for malicious behaviour, mimicking the trend between Microsoft Windows’ majority market share and the number of malware attacks on personal computers [5]. In fact, 99% of all phone malware is targeted at Android devices [6].

Google Play (formerly Android Market) developed by Google, is the primary dispensing platform for Android apps. It allows users to browse and download paid or free apps. This app store is available via the Google Play app, which is automatically installed on each Android smart phone [7]. There are also other third party app stores such as GetJar [8] and SlideME [9].

With access to millions of applications via these dispensing platforms, apps can be easily plagiarised and have malware incorporated in them through reverse engineering by malicious entities. Google have introduced Bouncer [10], a malware detection tool, which scans all new and existing apps on Google Play in order to reduce the number of malicious apps downloaded by Android users. However, since other third party apps stores are available, harmless apps can be reverse-engineered to include malware and then re-distributed into these markets.

1.2 Motivation for Project

In this project, I am interested in investigating the effectiveness of obfuscation to counter Android application reverse engineering.

The prevalence of plagiarised and repackaged apps (see Section 2.2.1), is a worrying facet of the Android market, along with the fact that repackaged apps make up a high percentage of Android malware. If an app is carrying out a trusted function there is a risk that it could be reverse engineered, and a modified (malicious) version passed off as the original. This project explores the reverse engineering process in practice by creating an Android application and using two tools to reverse engineer the app.

Software developers wary of the reverse engineering process will obfuscate their code, trying to make it harder to read the code and understand its function. Tools are available for Android developers to do this, and yet not all apps incorporate this. This project examines how ProGuard, the tool freely available for Android developers, alters the output of the reverse engineering process, after being run on an application created in this work.
1.3 Aims of the Project

It is known that Android apps are often reverse engineered and plagiarised. Any app on an app store can be downloaded, reverse engineered and have malicious code injected into it. The initial goal of the project is to examine how easy it is to reverse engineer an Android application and discover its purpose and a subsequent goal of the project is to see how the obfuscation process modifies the reverse engineering process.

The aims of the project are as follows:

- to create an Android application, with functionality to encrypt and store personal information.
- to reverse engineer this app using the tools ApkTool and dex2jar.
- to compare the output of the reverse engineering tools to the application source code through heuristic analysis.
- to create an obfuscated version of the Android application.
- to reverse engineer this obfuscated app with ApkTool and dex2jar.
- to compare the output of the tools on the obfuscated app to the output of the tools on the unobfuscated app through heuristic analysis, in order to determine the effectiveness of obfuscation to combat reverse engineering.

Chapter Overview

In Chapter 2, I provide a literature review of existing research relevant to this project, including the Android operating system and applications, the reverse engineering process and code obfuscation. I go on to examine Android applications further in Chapter 3; I describe the files, software and exportation process involved in creating an Android application.

The practical work of the project is introduced in Chapters 4 and 5. In Chapter 4, I describe the app that was created, RowenApp, and the contents of its source files, and include an example emulation of the app. Chapter 5 describes the obfuscated version of RowenApp called RowenAppobsf. I provide a synopsis of the obfuscation tool ProGuard and its implementation in the Android project.

Chapter 6 sets out the methodologies used in this project. I explain the two reverse engineering methods used and demonstrate their installation and use.

In Chapter 7, I report my results; the heuristic comparisons first between the outputs from ApkTool and dex2jar applied to RowenApp and the app source code, and secondly between the output from both tools when applied to RowenApp and RowenAppobsf. I discuss the results and conclusions obtained in this project, as well as future directions in Chapter 8.
2 Literature Review

This chapter describes the literature relevant to the topic of Android application reverse engineering and obfuscation. It provides an overview of existing work, theories and methodologies, and establishes further motivation for the project. This literature review looks at the Android operating system (OS), the reverse engineering process and code obfuscation.

2.1 Android Operating System

This part reviews the Android OS components, Android applications, and how the Android operating system protects and secures itself.

2.1.1 Overview

The Android OS is made up of several layers; a Linux kernel, open source libraries, a runtime environment, an application framework with a Java interface and an application layer. This basic structure is shown in Figure 1, as well as a more detailed insight into the key components in each layer. At the core of the Android OS is a Linux 2.6 kernel, responsible for the management of memory, security, processes and drivers. It also acts as an abstraction layer between the hardware and software stack [11].

![Figure 1: The Android Operating System. Figure from [12].](image)

Above the kernel, lie the libraries and Android runtime environment. The libraries, written in C/C++ are used to maintain several key components within the system. For instance [11]:
- Media libraries support the playing and recording of audio and video formats such as JPG and MP3.

- The Surface Manager controls access to 2D and 3D graphics layers for different processes and creates drawing surfaces on the mobile screen.

- For maintaining data storage, SQLite is used. It is a software library that supervises a relational database system, linked into the application processes.

The Android runtime environment’s primary constituent is the Dalvik Virtual Machine (VM), which allows applications to be run on Android devices. The Dalvik VM is designed to be suitable and efficient for systems with limited Central Processing Unit (CPU) speed, Random Access Memory (RAM) and battery power, ideal for smart phones.

Android applications are written in the Java language, which is not executable by the Dalvik VM. To overcome this, the Dalvik VM uses the tool dx to convert the Java byte code of the application, the set of instructions designed for execution by a specific software interpreter, to .dex format, an alternative instruction set used by the Dalvik VM. This process can be seen in Figure 2. It is possible to run multiple virtual machines efficiently with the Dalvik VM [11].

![Figure 2: The Conversion of Java Files to Dex Format. Figure from [11].](image)

The next layer up is the Application Framework that is written in Java. It contains a set of application programming interfaces (APIs), used to write applications. It is essentially a toolkit for apps that both come with the mobile device (contacts, media players) and ones built by Google or other developers. The framework has several parts such as [11]:

- The Activity Manager supplies a navigation history or backstack and organises the lifecycle of applications.

- The Package Manager maintains records of applications installed on the device.

- The Window Manager is a Java abstraction layer on top of services distributed by the Surface Manager.

- The Content Providers facilitate the sharing of information between applications.

- The Resource Manager stores and grants access to other resources used in the external components of apps such as layout file descriptions, localised strings and graphics.

- Other parts of the framework include the Telephony Manager, View System and Notification Manager. These contain APIs for calling applications, generating buttons and other user interfaces (UIs), and adapting display and audio alerts respectively.

The top layer of the Android OS is the application layer, which contains all applications available to the user. Android smart phones usually have default applications already installed such as calling applications, Short Message Service (SMS), Internet browsers, maps and calculators.
2.1.2 Android Applications

Android apps are written in the Java programming language and are stored as an Android package, or .apk file. This file has the relevant information to install and run the app on Android devices.

There are four types of key constituents of each app, which have specific objectives and determine how each constituent is used. These four types - activities, services, content provider and broadcast receiver - are described below [13].

Activities

An activity provides a visual interface for the user to interact with the app. Examples of activities are taking a photo, sending an SMS and playing a game. An app usually comprises one main activity, which is displayed when the app is launched and opens as the root activity on the stack. Along with this main activity, there are several different supporting activities that perform different functions.

The current activity on the stack can initiate a new activity in order for the app to proceed to another task. When this is done, the new activity is pushed onto the stack, whilst the current activity is stopped, but saved still on the stack. When the back button is pressed, it is popped from the stack, destroying the new activity, and the previous activity continues. Figure 3 demonstrates this.

![Figure 3: A diagram showing how activities are put on and taken off the backstack. Figure from [14]](image)

Services

A service carries out long-running processes in the background of an application with no UI. An activity can launch a service in order to run in the background or attach itself to it for further interaction.

Content Provider

The content provider is a type of SQLite database, which manages information related to
apps. Other apps may be allowed to query or modify data with the use of the content provider. It can also be used for storing private application data.

**Broadcast Receiver**

This component responds to broadcast announcements, originating from the device itself, for instance factory resets, or from apps e.g. to notify other apps that data has been downloaded or updated. They provide a way to call upon other app components, for example a service or activity may be triggered centred on some event.

These four types of components are all activated by an intent message, which links individual components together at runtime. For activities and services, an intent specifies an action to execute or data to work on, e.g. a request for an activity to display an image. An intent for a broadcast receiver is simply the message to be sent out. The content provider is not activated by intents, but by requests from the content resolver. The content resolver acts as a layer of abstraction by handling all interactions with the content provider and components [13].

2.1.3 **Android Security**

At the heart of the Android OS is the Linux kernel, which employs Linux security fundamentals. The main features of this are user-based permissions and process space protection. This isolates and protects user resources from one another; one user cannot read another user’s files, use CPU resources or exhaust memory [15].

In the Android system, each application is treated as a separate user and given a unique user identity (ID), sandboxing the application into its own process and file space. Each app then runs in an individual Dalvik VM with the assigned user ID, isolated from other applications and their files [15]. This is shown in Figure 4.

![Figure 4: Android Application Sandboxing. Figure from [16]](image-url)
Along with the user ID, applications have permissions associated with them. The Android system obliges applications to declare what permissions they require before installation i.e. what device resources they need, for example access to SMS, GPS location or reading a user’s contacts. This permission architecture allows the operating system to protect against attacks. For instance, if an application tries to access another application’s file or send SMS messages (another separate application), the operating system will not allow these actions since the application does not have permission [15].

This permission feature is regarded as “an Android developer’s best friend and worst enemy” [17]. According to Android developer Prateek Srivastava, quoted in [17] “developers are also lazy”, and may give apps more permissions than they need. An attacker can exploit this by accessing device resources and user data. This problem is exacerbated by user laziness; when installing an app, a user may simply click install without regard for the permissions required, allowing a malicious app to be successfully installed. Fortunately, due to the prevalence of Android malware [6], users are now more aware of the importance of legitimate permissions and trustworthy sources through online articles such as [17], [18] and [19].

Android app developers must digitally sign their apps when uploading them onto Google Play. The app is signed with a certificate with a private key owned by the app’s creator. This certificate associates the app with the app developer. Certificates can also be self-signed, meaning no certificate authority has been used. Unsigned apps will not be installed or executed correctly by the Android OS [20].

### 2.2 Reverse Engineering

In general, reverse engineering is the process of analyzing an object or system’s operation to determine its original design. In a software engineering context, it is the method of obtaining the source code of a program from examining the machine code. It is used at both ends of the spectrum. Malicious developers reverse engineer software and operating systems to find vulnerabilities that can be exploited to access sensitive information or to facilitate taking control of a system. Anti-virus developers scrutinize examples of malware in order to calculate the extent of damage that could be caused and find out ways to remove it from a system [21].

#### 2.2.1 Repackaging

Repackaging of apps is when an app, created and signed by one developer, is reverse engineered and then re-released onto app stores, signed by another developer. The re-released app is usually modified from the original one. This whole process is shown in Figure 5 [22]. [23] explains that apps can be plagiarised, altered and repackaged for monetary reasons; a paid app could be re-released for free, “cracking” the application, and a plagiarist could divert revenue from advertising towards himself by altering the ad libraries. They can also be repackaged with malware in order to gain personal user information or send SMS to premium rate numbers as in the case of HippoSMS [24], as mentioned in [25].

The amount of repackaging is significant. Researchers have estimated the number of repackaged and plagiarised apps in the past few years through tools designed to compare applications. [23] found 4,295 of 265,395 apps were repackaged; [26] discovered 5% to 13% of apps across six third-party Android markets were repackaged; and [25] found 141 repackaged apps of a sample of 75,000 across 13 markets. A study by Gibler et al. [27] in 2013 inspected 265,359 apps over 17 Android markets using the DNADroid tool. It found that the official Android app
store, Google Play, had the highest amount of cloned apps, with approximately 30,000 cloned apps. Tools have also been created to test the likelihood that an app will be plagiarised; [28] indicates that 29.4% of the 158,000 sample of apps they observed are more likely to be plagiarised.

The academic community has also been aware of the prevalence of Android malware in repackaged apps. Zhou et al. [26] found that 86% of its Android malware sample was repackaged apps. [23] used this estimate to find suspicious apps within its clusters of similar apps, and found 243 occurrences of malware, 169 that had not been seen before. However, with over 1 million apps available [29], it is difficult to get an accurate estimate of the total number of malicious repackaged apps.

![Figure 5: The Repackaging Process](image)

2.2.2 Malware and Repackaged Apps

There are several types of malware known to computers such as worms, trojans, viruses and spyware. However, these malwares have morphed to suit their new environment of smart phones in recent years. Khan and Tahir [30], cited in [31] classify malware for smart phones:

- **Ad-Ware** - Malware advertisements on smart phones.
- **Destructive** - Examples include deletion of phone contacts with no user knowledge.
- **Direct Payoff** - Malware that sends an SMS without user consent.
• Premeditated Spyware - Remote listening and tracking.

• Proof of Concept - Malware that proves a particular attack can work (e.g. leaving Bluetooth running to drain the battery without user knowledge).

• Information Scattering - Checking address books, cookies and passwords without user consent.

But what is the big picture? As stated previously, attackers may have financial motives to change advertising libraries, but what other kind of things can repackaged malicious apps do with these different types of malware?

[32] showed a serious attack on a banking app was “possible without having to illegally obtain any of the sender’s personal information, such as the sender’s public key certificate, the password to their bank account, or their security card”. The attack involves an innocent user installing a malicious repackaged banking app developed by an attacker. When the user tries to send an amount of money to the intended recipient, the money is transferred to the attacker instead. The attack is laid out in Figure 6.

![Figure 6: Repackaging Attack on Banking App. Figure from [32].](image)

The simulated attack was successful. The antivirus routine was skipped, integrity checks were forged and messages were replayed in order to trick the banking server into believing that it has sent the correct amount to the intended recipient, whereas it has sent the amount to the attacker. This scenario demonstrates the serious dangers of vulnerable code being repackaged and let loose on Android markets. With a total of 48 billion app downloads from Google Play alone [33], the potential impact of real repackaged banking apps similar to this one could be catastrophic.

2.2.3 Reverse Engineering Tools

There is an abundance of Android tools available to reverse engineer apps. The first tool designed for this process was undx [34] created by Marc Schonefeld. This tool converts the Dalvik bytecode back into Java bytecode, a .jar file, which then can be fed into tools such as a Java
Decompiler (JAD) or Java Decompiler Graphical User Interface (JD-GUI), to obtain the source code. However, problems were reported with undx when more complex Dalvik bytecode was introduced.

To overcome issues with complex Dalvik bytecode, dex2jar was created. This tool, whilst performing the same act as undx, “does a much better job at converting Dalvik to Java” [35]. [23], [25] and [27] all use dex2jar to study the similarity of different applications.

Another way to alter apps is to use smali/baksmali [36], an assembler/disassembler. The baksmali disassembler translates the Dalvik bytecode in to a readable arrangement, the smali code. This code can then be changed and recompiled to use on an Android device. [37] uses baksmali to study app repackaging. ApkTool [38] uses the baksmali disassembler to obtain smali code of an app, but also has the ability to repackage it.

Vibha Manjunath’s work [31], which successfully reverse engineered Android malware, used both dex2jar and ApkTool to perform static analysis. The routines to send premium rate SMS messages were clearly found in the Java and smali code for the malicious application “iCalendar”. This work also highlighted the ease of reverse engineering and repackaging of malicious apps. The app “seismic” was easily altered to include the permissions to read, write and send SMS messages and to read the owner’s data. The altered app was then repackaged and emulated on an Android device with the new permissions.

A tool also exists within the Android Software Development Kit (SDK) called dexdump [39]. It performs a linear sweep algorithm to perform Dalvik bytecode disassembly. This algorithm navigates through the bytecode and anticipates that “each next valid instruction follows the currently analysed one” [40]. This output can be examined to determine the behaviour of the code, as in [28], but can fail if code is obfuscated.

IDA Pro [41] is a commercial tool used for reverse engineering, offering program graph visualisation and plug-in support to expand its functionality.

With ever expanding online resources, it is easy to see why so many Android apps are repackaged. Numerous tools are readily available online with tutorials which walk through the steps to reverse engineer and repackage apps [42, 43], making this process straightforward, even for those who know little about app development.

2.3 Code Obfuscation

To hinder the reverse engineering process, developers obfuscate their code. Code obfuscation purposefully tries to make code as hard as possible for humans to read, in order to conceal its inner workings. Obfuscation, like reverse engineering, is done by both legitimate software developers, and creators of malware; the first, to make it harder to plagiarise their work, and the latter, to elude anti-virus detectors. Code obfuscators are often steered towards combating particular reverse engineering stages, to reduce the effectiveness of tools and heuristic methods [44].

2.3.1 Techniques

One of the necessary fundameitals of code obfuscation is that it does not impact the function or behaviour of a program, but alters the outer representation [44]. There are several techniques
Identifier Mangling

This technique changes the names of classes, methods and fields, called identifiers, from meaningful words to meaningless characters. Identifiers, in personal programs, can be helpful and provide information about the following code and the intention of the program, but in a public sphere such as Android apps, identifiers can leak this information, making the reverse engineering process less time-consuming as human inspection and analysis will be easier [44].

Identifier mangling reduces the amount of meta-information about the program’s function by using random strings, which contain no information about program behaviour. An example of this technique is shown in Figures 7 and 8. It becomes clear in Figure 7 that this part of the program deals with network connections and encryption, important to a malicious developer. Figure 8 shows the resulting code after obfuscation. The original identifiers have been replaced with random letters, furnishing no information of the intended behaviour of the original code [44].

```
1 public class NetworkManager {
2  private String encrypt( String input )
3  { .... }
4  public void send( String Input )
5  { .... }
6 }
```

Figure 7: Unobfuscated Java source code. Identifiers are highlighted in blue. Figure from [44].

```
1 public class a {
2  private String a( String ab )
3  { .... }
4  public void b( String ac )
5  { .... }
6 }
```

Figure 8: Obfuscated Java code. Figure from [44].

String Obfuscation

String obfuscation, like identifier mangling, is a method of removing any meta-data about the program that could be extracted in the reverse engineering process. Strings are arrays of characters used within the program for user interaction. However, strings affect the semantics of the program and the original strings must be available at runtime for the user, so they cannot be mangled in the same way as the identifiers [40].

Many applications come into contact with personal data, a true gold mine for the reverse engineer if stored in plaintext. To combat this, the strings are encrypted to avoid static analysis, but need to be available in plaintext when the app is running so the user can interact with it. String obfuscation is done by passing the string $s$, into an invertible transformation function $F$, and storing the result $F(s)$, within the code. By running the inverse of the transformation
function on $F(s)$, $F^{-1}(F(s)) = s$, the original string is generated when needed at runtime [40]. Schulz [44] remarks that this encryption can be the Advanced Encryption Standard (AES), whilst [40] warns of the poor security practices of using custom XOR ciphers.

Schulz [44] also makes a point of saying that this process “does not make it harder to understand the program code due to that fact that it does not change it”. It reduces the amount of information available to be extracted by static analysis, but during dynamic analysis where the program is executed, the original string can be extracted.

**Dynamic Code Loading**

In order to take out all meaning of the code so that no information can be extracted, the whole program can be encrypted. Like string obfuscation, the program can be decrypted at runtime before it is executed. This process involves creating a packing stub, executed when the app is launched, which decrypts the app, loads it into the process context and executes it [44].

The whole process of the decryption stub is shown in Figure 9. The encrypted application is first fetched into memory (Step 1), by downloading the dex file from an external server or internal structure. The file is then decrypted to generate the original file (Step 2), which can then be loaded (Step 3) and executed (Step 4) in the Dalvik VM [44].

![Figure 9: The Process of Dynamic Code Loading. Figure from [44].](image)

The bytecode of the application is encrypted under dynamic code loading, rendering it hard to be analysed by reverse engineering techniques; the packing stub must be examined which is a slow process. This process also has another advantage: it can be modified to include more advanced obfuscation techniques [44].

**Self-Modifying Code**

Self-modifying code transforms itself whilst being executed; the instructions observed in the code initially may not be the code running when the application is being executed. This process, commonly used by malware alongside buffer overflow attacks, specifically hinders dynamic analysis [44].

Self-modifying code is not possible within the Dalvik VM itself, since the limited instruction set of the Dalvik VM does not allow access to the bytecode using an instruction. It is possible
however, with the use of a Java Native Interface (JNI), enabling the execution of native code within the current process.

Schulz [44] gives an example of how this could be done:

“The Java source code and therefore also the Dalvik bytecode consists of three parts:

- A JNI call in order to execute our native code,
- A magic byte constant, which is used to locate our Dalvik bytecode,
- The Dalvik bytecode, which should be modified.

The native code will be called by our Dalvik bytecode. It has a search routine that starts to look for the magic byte constant within the memory. This process with the intention to gather position information is well known from shellcode and called ‘egg-hunting’. After the magic byte constant has been identified, we can modify the target Dalvik bytecode, which is located at a previously measured offset to the magic byte constant. By this we mean modify our bytecode. After returning from the native code, the modified bytecode will be executed.”

This technique means that the Android application cannot be reverse engineered from the static analysis of Dalvik bytecode since the bytecode can modify itself when being run. The reverse engineering process is made even harder; the native code will have to be examined, as the Dalvik bytecode is no longer an isolated system component [44].

Junkbytes

This technique alters the flow of the program by adding in lines of code, which is never executed. This confuses disassemblers by introducing errors and diverts attention away from the true meaning of the code. Junkbytes are injected into the bytecode where the disassembler anticipates an instruction [40].

The location of the junkbyte has to be precise to maximise the obfuscation, but must be placed where it is not going to be executed since this would culminate in unwanted actions in the app. The junkbyte is put after an unconditional or conditional branch so that the code jumps to where the branch points [44].

The act of junkbyte injection forces linear-sweep algorithms, like the one used in dexdump, to misconstrue the bytecode and produce inaccurate output. This allows junkbyte instructions to be hidden during the disassembling process, making it very difficult to analyse and prolongs the process. Many reverse engineering tools have problems with junkbyte insertion and can be tricked by it [44].

2.3.2 Obfuscation Tools

As seen from the obfuscation techniques above, obfuscation can take place on the source code level or bytecode level. There are several tools which obfuscate Android code:

- ProGuard: A Java source code obfuscator included in the Android SDK, which according to its website [45], “detects and removes unused classes, fields, methods, and attributes. It optimises bytecode and removes unused instructions.” It also uses identifier mangling, using lexical-sorted strings [45] not random characters to minimise memory usage.
• DexGuard: A commercially available Android obfuscator [46]. It works on both source code and bytecode to obfuscate strings and identifiers, but also encrypts strings and libraries and offers tamper detection.

• Allatori: Whilst offering similar features to ProGuard, Allatori also includes string encryption and flow obfuscation (junkbyte injection), which obscures the program flow but does not change runtime code [47].

• Dalvik-obfuscator: This tool obfuscates bytecode utilising junkbyte insertion [48]. When given a standard Android application package, .apk, file as input, it produces an obfuscated .apk file output.
3 Android Applications

This chapter describes the software used to create the Android app in this project and also the file content and architecture of Android apps.

3.1 Android Software Development Kit

The Android SDK provides the relevant libraries and API's used to develop Android applications, along with an emulator. The implementation of the Android SDK used in this work is with the Integrated Development Environment (IDE) Eclipse with the Android Development Tool (ADT) plug-in. The Android SDK includes a tutorial and sample code, and is available for download at [49].

3.2 Application Files

Android applications comprise several files which provide core app fundamentals, app resources and the source material for the app. These source files for the application are typically written in Java, and the resource files are written in the Extensible Markup Language (XML) format. The XML language is designed to store and transport information [50].

3.2.1 AndroidManifest.xml

The AndroidManifest.xml file provides the fundamental structure of the application to the Android system. It declares the activities, intents, broadcast receivers and content providers that the app is made up of. This file also outlines the permissions the application must have, the minimum SDK version and the libraries that must be linked to the application [51].

A sample AndroidManifest.xml file is set out in Figure 10 showing the file's typical construction. All elements the app contains, for example, activities, services and permissions, are listed in the file, along with their attributes. Specific attributes must be set out so that the element can perform its desired task. In this manifest file, most attributes are listed with a prefix of `android:`; e.g. `android:name`. Attributes can also display things to the user, for instance graphics, usually set from a resource or theme file. An icon can be set in order to launch the application on the Android device in this way. This is done by specifying the icon attribute as `android:icon = "@drawable/ic_launcher"` using the ic_launcher file from the drawable resource.

As stated in Section 2.1.2, the fundamental constituents of an application are launched by intents. Intents carry a lot of information including the type of app constituent (activity, service, content provider or broadcast receiver) that should perform the intent, which the Android system then finds to respond to the intent. The Android system then opens up a new instance of that constituent if necessary and passes the Intent object to it. Intent filters are defined in the manifest file so that the constituents of the app can list the types of intents they can respond to. These are listed as `<intent-filter>` elements within the manifest file, of which each constituent can have many, each specifying the capability of the constituent.

Permissions are another important aspect of the AndroidManifest.xml file. The Android Developers guide [51] states that “A permission is a restriction limiting access to a part of the code or to data on the device.” In the manifest file, labels are used to distinguish each permission, typically expressing the restricted action. Any permissions that the application needs in order to
access other features, must be proclaimed in the manifest file, using the `<uses-permission>` element. The user must agree to the permissions the app requires during installation. Each constituent of the app can define its own permissions or new ones using the `<permission>` element. This can be done to protect the application’s own constituents [51].

```
<manifest>
  <uses-permission />  
  <permission />  
  <permission-tree />  
  <permission-group />  
  <instrumentation />  
  <uses-sdk />  
  <uses-feature />  
  <supports-screens />  
  <compatible-screens />  
  <supports-gl-texture />  
  <application>
    <activity>
      <intent-filter>
        <action />
        <category />
      </intent-filter>
    </activity>
    <activity-alias>
      <intent-filter> ... </intent-filter>
    </activity-alias>
    <service>
      <intent-filter> ... </intent-filter>
    </service>
    <receiver>
      <intent-filter> ... </intent-filter>
    </receiver>
    <provider>
      <grant-uri-permission />
    </provider>
  </application>
</manifest>
```

Figure 10: Sample AndroidManifest.xml file. Figure from [51].

3.2.2 App Resources

Each Android project contains its resources within the `/res` directory. These resources are split into subdirectories according to their type. Typical subdirectories include [51]:

- **drawable/** - This subdirectory encapsulates the drawable resources which the application can draw to the screen. Examples of drawable resources are bitmap graphics files, shapes
and animation drawables.

- **values/** - This contains xml files which represent resources such as colours, strings and styles. The values/ directory has many resources, each with its own file. Examples of the resources in this subdirectory are:
  
  - **colors** which defines colour resources in hexadecimal values. They can be referenced in order to combine with other simple resources. Figure 11 is an example of the res/values/colors.xml file. It shows the definitions of two colour resources, opaque_red and translucent_red, along with the hexadecimal colour values. This hexadecimal value begins with a # character.

  ```xml
  <?xml version="1.0" encoding="utf-8"?>
  <resources>
    <color name="opaque_red">#f00</color>
    <color name="translucent_red">#80ff0000</color>
  </resources>
  
  Figure 11: Sample color.xml file. Figure from [51].
  ```

  - **strings** defines the strings that can be referenced all over the application. The string resources are defined using the `<string>` element, and are identified by the string resources name attribute. An example of a strings.xml file is shown in Figure 12. It shows the string resource identified as “hello” and its value of “Hello!”.

  ```xml
  <?xml version="1.0" encoding="utf-8"?>
  <resources>
    <string name="hello">Hello!</string>
  </resources>
  
  Figure 12: Sample strings.xml file. Figure from [51].
  ```

  - **styles** contains the style resources which define the appearance of a UI. Style resources must have a name to identify them, a parent reference to the style properties that the resource should inherit, and also define the properties of the style resource. A sample styles file can be seen in Figure 13. It defines the style “CustomText”, which has a Text parent style resource. The properties of this style resource, the text size and text colour, are set out too.

- **layout/** - The layout of the UI for the activities in the application are contained in xml files within this subdirectory. This file contains a root element that encapsulates all the other elements in the UI and can be one of three things [52]:
  
  - A `<ViewGroup>` element which acts as a container for View elements. Different ViewGroup objects specify the architecture of the elements within itself, differently. For instance, LinearLayout, a ViewGroup object, aligns the child elements in one direction.
– A `<View>` element which represents an individual component in the UI. Examples of View objects include Button, EditText and TextView.

– A `<merge>` element, which is useful when this particular layout is applied to a layout containing the relevant parent View to encase the `<merge>` element’s child elements.

Each `<ViewGroup>` and `<View>` element must specify height and width parameters using the `android:layout.width` and `android:layout.height` prefixes. Identifiers can be declared for View elements as well using the `android:id` prefix. String resources can be used in the layout file by referencing the entries in the strings.xml file, e.g. `android:text = "@string/resource"`.

An example of a layout file for an activity is shown in Figure 14. The example uses the LinearLayout object as its root element, with two child elements of a TextView and a Button. The heights and widths of all the elements in the file have been declared, along with the vertical orientation of the activity. Text has also been included in the child elements. The XML namespace has been declared in the top line of Figure 11, which bypasses element name conflicts [53].

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">
    <TextView
        android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a TextView" />
    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a Button" />
</LinearLayout>
```

Figure 14: Sample Layout file. Figure from [51].
All application resources, whether they be strings, layouts or drawable objects, have identifiers called resource IDs. These resource IDs are defined in an Android project’s automatically generated R class. Each resource ID in the R class is made up of an R subclass of the resource type and the resource name, represented as a static integer. For instance, the string resource from Figure 12 has the resource ID of R.string.hello. This string resource can be accessed using its static integer of the R class (R.string.hello) or using xml syntax (@string/hello) [51].

3.2.3 Source Files

The application’s main source files are within the src/ directory. This directory houses the Java files for the application to run. It includes a file for each activity in the app, along with other classes needed in the app.

Most activities allow the user to interact with the application. The Activity class is used for this UI. The first method to be called when the activity is first created is the `onCreate()` method. This method sets up the UI on the Android device’s screen, binds any information needed to the activity and maintains the activity’s previously frozen state. This method utilises the `setContentView(View)` method to set up the UI using the `layout/` files. This is shown in an example piece of code in Figure 15. When the activity launches, the layout resource `main_activity.xml` file is accessed from the R class and used to set up the UI, and the activity maintains the Bundle with the previously frozen state, `savedInstanceState` [51].

```java
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_activity);
}
```

Figure 15: Sample Activity File. Figure from [51].

After the activity’s layout has become visible to the user, the `onStart()` method is called, which gives instructions to the application after the creation of the activity and after resuming the activity, if it had gone into the background [51]. The activity contains other methods dependent on the purpose of the app. For instance, a calculator app will use the press of buttons to perform calculations and the methods used when buttons are pressed will be in the calculator activity.

3.3 Exporting the Application

After the activities, resources, classes and AndroidManifest files have been completed to create the final version of the application, the application must be packaged and exported to generate an .apk file in order for it to be usable on an Android device. The application can be exported either unsigned or signed with a certificate [51]. In Eclipse, the application is exported by right-clicking on the project in the Project Explorer, clicking on Android Tools then Export Signed Application or Export Unsigned Application, then saved [54].

The next chapter shows how the information in this chapter is used in practice with the creation of RowenApp.
4 RowenApp

To investigate the reverse engineering process, an app was created. This app, named RowenApp, is designed to store personal banking information of the user or users in a database and then find and retrieve this information when requested by the user. The version of the ADT used to build the application created by the author was 22.3.0-887826, and was simulated on an Android Virtual Device with platform number 4.4 and API Level 19. RowenApp was exported unsigned.

4.1 RowenApp Overview

This section gives an overview of how the app works in practice with the UI. It provides an emulation of RowenApp to demonstrate the app’s function with example user data.

4.1.1 Launching the App

When RowenApp is launched, its main activity appears on the screen to welcome the user to the application. This can be seen in Figure 16. It includes a button to launch another activity in order for the user to add details about themselves.

![Figure 16: Main Activity screen.](image)

4.1.2 Adding Information

After pressing the “Add details” button, the AddDetails activity is launched. This is the interface for the user to input their personal information into the app which can be seen in Figure 17. The personal information includes the user’s first name, surname, account number, sort code, a passcode or password and a user ID. The user ID is used to identify the user and
the password is used to encrypt the personal information of the user in an SQLite database.

Figure 17: The AddDetails activity screen.

The AddDetails activity screen includes two buttons; one for adding information to the database, and one for finding information from the database. To input information into the database, the user simply inputs their details into the text boxes provided and then clicks the “Enter details” button. This has been illustrated with an example in Figure 18.

In this example, the information provided by the user is:

- First name: Mike
- Surname: Smith
- Account no: 123456789
- Sort code: 123456
- Passcode: password1234
- User ID: MSmith1234
4.1.3 Retrieving Information

To find a user’s account number and sort code from the database, the user ID, passcode, first name and surname fields must be completed, then the “Find Record” button must be clicked. Figures 19 and 20 demonstrate this process by inputting Mike Smith’s name, user ID and passcode to find his account number and sort code, which are then shown in the relevant text boxes.
Figure 19: The AddDetails activity screen with the correct fields completed to find Mike Smith’s account number and sort code. The Account Number and Sort Code fields are blank.

Figure 20: The AddDetails activity screen with all of Mike Smith’s details.
4.1.4 Error Messages

RowenApp has error messages displayed when user input is incorrect. When the user ID is incorrect, the app is unable to find the correct entry in the database and sends an error message. This process can be seen in Figure 21, where (a) shows the incorrect user ID for Mike Smith and (b) demonstrates the error message. An error message also appears when the first name or surname is entered incorrectly.

Figure 21: The process of displaying an error message when the user ID is entered incorrectly.

4.2 Source Files

RowenApp has five source files within the src/ directory. They are the two activity classes, MainActivity and AddDetails, one class dealing with the cryptography, Crypto, and two classes dealing with inputting data into a SQLite database to store the personal banking information, DBDetails and MyDBHandler.

4.2.1 DBDetails.java

This file creates a new class called DBDetails, which is used to group the information to be put into the database. It has eight private variables:

id An integer used to order entries in the database.

userID A string representing the identification of the user.

firstname A string representing the first name of the user.
surname A string representing the surname of the user.

accountno A string representing the account number of the user.

sortcode A string representing the sort code of the user.

salt A string used as a per-user diversification input to the cryptographic algorithms.

iv A string used as a per-user initialisation vector input to the cryptographic algorithms.

The class then defines three public constructors: one with null arguments, one with all eight variables, and one with seven variables (without the variable id).

DBDetails has sixteen methods which describe the dynamic behaviour of the class. Each variable has two methods. It has one method to set the value of the variable, using that value as an argument to the method, e.g. `setfirstname(user’s first name)`, returning void. It also has one to return the value of the variable which requires no arguments e.g. `getfirstname()`.

4.2.2 MyDBHandler

MyDBHandler extends the SQLiteOpenHelper class, meaning it is a subclass of SQLiteOpenHelper. It inherits all the fields and methods of SQLiteOpenHelper, meaning that MyDBHandler can reuse these fields and methods. This is one of the perks of the Android SDK; developers need not write and debug common classes needed in Android app development [55]. This class also imports other SQLite database classes.

MyDBHandler has 11 variables; one integer for the database version, two string variables naming the database name and table name, “detailsDB.db” and “details” respectively, and eight strings used to name the columns of the database. The significant methods of MyDBHandler are outlined below:

`onCreate(SQLiteDatabase db)`

This method is called when a new instance of MyDBHandler is created. `onCreate` uses the method `execSQL` from the SQLiteDatabase class which executes a SQL string query of the form “CREATE TABLE” + “TABLE NAME” + “(" + “COLUMN 1” + “TYPE” + . . . + “)”, in order to create a table. The actual query uses the column names for COLUMN 1, COLUMN 2 etc., and the expected type of data to be put into the column (text, integers etc.) is used for the TYPE sections. The structure of the “details” table created in this method is shown in Table 1 below, along with example user data.

Table 1: The representation of the table “details” created when `onCreate` is used in MyDBHandler. The table also includes the example user data of Mike Smith first introduced in Section 4.1.

<table>
<thead>
<tr>
<th>ID</th>
<th>First name</th>
<th>Surname</th>
<th>Account no</th>
<th>Sort code</th>
<th>Salt</th>
<th>User ID</th>
<th>IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mike</td>
<td>Smith</td>
<td>123456789</td>
<td>123456</td>
<td>salt</td>
<td>MSmith1234</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
addDetails(DBDetails detail)

This method adds a row of user information to the database table. The user information is contained within the DBDetails object which is an argument of addDetails.

To insert data into the database, the insert method of the SQLiteDatabase class is called, requiring the name of the table to insert the data into and the actual data to insert. The actual data must be contained within an instance of the ContentValues class, a common class which stores a set of values [51].

The user information is retrieved from the DBDetails object using the class’s get methods e.g. getfirstname. This, along with the returned user information’s corresponding column headers, is stored in the ContentValues object, using the ContentValues class’s put method [51]. Once all the information has been stored within the ContentValues object, the database is opened and the insert method is used to input the user information stored in the ContentValues object. The database is then closed.

selectrowbyID(String userID)

This method selects all information in a row of the database which corresponds to a particular entry of the User ID column. The particular entry of the User ID column is the string argument of the method. A DBDetails object is returned by the method containing the user information corresponding to the userID.

Firstly, the string query is defined to select everything from the “details” table, where the User ID column is equal to the string argument. The database is then opened and the query is executed using the SQLiteDatabase method, rawQuery. rawQuery takes the string query as input and returns a Cursor over the result set. A Cursor object is a collection of the queried data [51].

A new DBDetails object is created, and if the Cursor object is not null, i.e user information was found corresponding to the userID input, the user information in the Cursor object is put into the DBDetails object using the set methods of the class, for instance setsurname. The Cursor is then closed and the DBDetails object is returned. If the Cursor is null, this means the query returned no information, the DBDetails object is set to null and returned.

deletedetails(String firstname)

This method deletes a row of user data from the database which coincides with an individual record in the first name column, the string argument to the method. The method returns a boolean. This method is not actually used when the app is running on a device.

This method follows closely the structure of selectrowbyID; the query is declared and executed using the rawQuery method to return a Cursor object with the queried data. If no data was found and the Cursor object is null, the method returns false. However if the data is found, the delete method is then used to delete the row of user data corresponding to the string input and the method returns true.
4.2.3 Crypto

The Android SDK incorporates its own crypto package, javax.crypto which contains the corresponding classes to exercise encryption/decryption and key agreement algorithms for applications wishing to use cryptographic processes [51]. RowenApp uses these crypto classes to implement the encryption and decryption of the database entries.

RowenApp uses symmetric encryption to encrypt and decrypt the user information as symmetric encryption is very fast. The encryption algorithm used is AES, used in cipher-block chaining (CBC) mode. An encryption key and initialisation vector (IV) is needed to perform this encryption.

As the user is inputting their own data into the app and also wants the ability to retrieve their own data, it was deemed preferable to have a user-supplied password to generate the symmetric key. This is because a typical user is not going to remember a long binary or hexadecimal key comfortably, and a password is easily remembered and can be passed through a method to generate the symmetric key. A salt was used in the key generation process as a per-user diversification of the key generation process. This means that even if two users have the same password, it will not generate the same key.

The methods to implement the cryptographic processes are listed below:

`generatesalt()`

This method generates a random number stored in a 32-bit byte array. A new instance of the SecureRandom class is created. SecureRandom generates pseudo-random numbers used for cryptographic processes. The SecureRandom method `nextBytes` with an argument of a 32-bit byte array named “salt” is called. This method stores random bytes for each array element in the “salt” array [51]. `generatesalt` then returns the “salt” array.

`generatekey(char[ ] password, byte[ ] salt)`

This method generates a 256-bit AES key using the user password and salt as input. This method uses the SecretKeyFactory class to generate the symmetric key. It first creates a new instance of SecretKeyFactory for the algorithm “PBKDF2WithHmacSHA1”, one of the key generation algorithms available for this class [51].

Next, a KeySpec object is created. This is a specification of the material being used to generate the key. This KeySpec object is a password-based encryption (PBE) KeySpec, using the password, the salt, an iteration counter and the desired key length to formulate the PBEKeySpec. The key is then produced by using the SecretKeyFactory method `generateSecret` using the PBEKeySpec as input [51].

`generateiv()`

This method is similar to that of `generatesalt`; it uses a SecureRandom object in order to generate a pseudo-random initialisation vector, only in this case a Cipher object [51] is created with the appropriate encryption algorithm in order to get the correct block size for the IV.
encrypt(String message, char[] password, byte[] salt, byte[] iv)

This method is used to encrypt a string input using the user’s password, salt and IV, with the AES algorithm.

The method creates a Cipher object which provides access to cryptographic cipher implementations. The algorithm implementation used is “AES/CBC/PKCS5Padding” which is of the form “algorithm/mode/padding”. The key is then generated within encrypt using generatekey to hide the encryption key generation within the encryption function so it is not passed directly into it. The specification of the IV parameters are set up using an IVParameterSpec object, which, along with the key, are passed into the Cipher object’s init method to set up the encryption mode of the Cipher object [51].

The message to be encrypted is then put into bytes using UTF-8 (‘U’ from Universal Character Set + Transformation Format8-bit) encoding and this is passed into the doFinal method of the Cipher object which then encrypts the message. encrypt then returns the hexadecimal representation of this encryption.

decrypt(String encrypted, char[] password, byte[] salt, byte[] iv)

This method is responsible for the decryption of a string of a hexadecimal number, which is passed into the method. It follows the same steps as encrypt, it sets up the Cipher object, the key (using the password and salt inputs) and the IV parameters. However, the init method is established in decryption mode. The encrypted message is changed from its hexadecimal form to bytes and then decrypted using the doFinal method and returned in string format.

4.2.4 MainActivity

MainActivity extends the Activity class and so inherits the methods and fields of the Activity class. It begins with the onCreate method and sets up the UI for this activity using code similar to that of Figure 15. Figure 16 shows the UI of MainActivity, and shows the “Add details” button widget. The purpose of the button is to launch the AddDetails activity and the steps below outline this procedure within the onCreate method [51]:

- A Button object btn is created to represent the push-button widget from the UI of Main-Activity. The button is identified as “btn” in the activity_main.xml layout file and assigned to this newly created Button object by using the method findViewById.
- An OnClickListener object listnr is then created. OnClickListener provides a way for a callback function, onClick, to be invoked when a view is clicked.
- The method btn.setOnClickListener(listnr) is called. It registers the listnr callback function to be invoked when the button is pressed in the application.
- onClick(View v) is the callback function called upon when the button is pressed. This function creates an explicit intent for the AddDetails class and executes the method startActivityForResult on the intent.

4.2.5 AddDetails

The AddDetails activity firstly defines six EditText variables, firstnameBox, surnameBox . . . etc. to represent the EditText boxes set up in the layout file of this activity. The methods in Ad-
onCreate

This method sets up the UI for this activity by using the method `setContentView` to display the contents of the layout file for this activity. It is similar to the method of the same name in Figure 15.

onStart()

Each of the EditText widgets created in the layout file is distinguished by an identifier, the `android:id` attribute. In this method, each of the EditText variables defined at the beginning of the AddDetails class is assigned its corresponding EditText widget, e.g. the EditText widget with `android:id = "@+id/firstname"` is assigned to the firstnameBox variable. The function `findViewById` is used to assign the identified EditText boxes to the corresponding variables.

newDetail(View v)

This method adds the information in the EditText widgets to the database in a series of steps:

- Firstly, the text within the EditText widgets is captured and stored in string variables. The string representing the password is put into a character array, ready to use for the encryption process.
- Secondly, a salt and IV are generated for the user using the `generatesalt` and `generateiv` Crypto methods respectively. The first name, surname, account number and sort code of the user are encrypted using the `encrypt` method using the password, salt and IV.
- Thirdly, the salt and IV variables, originally of the type `byte[]`, are now turned into string variables.
- Fourthly, a new DBDetails object is created, storing the user ID, encrypted user information, salt and IV. This information is inserted into the database using the MyDBHandler method `addDetails`.
- Fifthly, the EditText widgets on the activity screen are cleared, ready for the next user or information retrieval.

lookDetail(View view)

This method looks up user account numbers and sort codes using the first name and surname, password and user ID. The procedure goes as follows:

- A MyDBHandler object is created so interaction with the database can be achieved. The relevant values of the EditText widgets (first name, surname, user ID and password) are stored as strings.
- The method `selectrowbyuID` is called to find the user information corresponding to the user ID gathered from the relevant EditText widget, and a DBDetails object is returned with the user information.

...
• If the DBDetails object is null, no user information was found corresponding to the inputted user ID. An AlertDialog.Builder object [51] is set up to inform the user that the user ID is incorrect. The EditText widgets are then blanked out.

• If the DBDetails object is not null, the salt and IV for the user are found using the getsalt and getiv methods from the DBDetails class. The password is put into a character array for the cryptographic processes.

• The DBDetails object returned from selectrowbyuID contains the encrypted first name, surname, account number and sort code. In order to release the user’s account number and sort code, the encrypted first name and surname from the DBDetails object is decrypted using the decrypt method, using the user’s password, salt and iv, and then compared to the first name and surname written by the user in the EditText widgets.

• If the decrypted first name and surname match the first name and surname from the EditText widgets, the account number and sort code are decrypted from the DBDetails object and displayed in their corresponding EditText widgets.

• If they do not match, another AlertDialog.Builder is constructed, telling the user that the first name or surname is incorrect and blanks out the EditText widgets.

Chapter 5 goes on to explain how RowenApp was obfuscated in order to test the effectiveness of obfuscation against reverse engineering.
5 RowenAppobsf

As mentioned in Section 2.3, obfuscation is designed to make reading code as hard as possible using a variety of techniques. This is done to remove contextual information or meta-data from the code to hide the code’s true purpose and inner workings.

RowenApp was obfuscated using the tool ProGuard, available in the Android SDK. This chapter deals with the ProGuard tool, and its use in the obfuscation of RowenApp, to make RowenAppobsf. RowenAppobsf was created with the same version of the ADT as RowenApp, and exported unsigned. No changes were made by the author to any of the source or resource files belonging to RowenApp before ProGuard was run.

5.1 ProGuard

ProGuard, a free tool which is available as a stand alone product and within the Android SDK, does four things [45]:

Shrinking This includes removing classes, methods and fields which are unused, but leaves the function of the program unchanged. This is done through bytecode analysis.

Obfuscation This process replaces names of classes, methods and variables by meaningless characters. This decreases the amount of contextual debugging information in the compiled bytecode, making it harder to decompile the bytecode and reverse engineer the program. This process also compacts the code further.

Preverification This step performs verification of the byte code when the class files are loaded, to confirm that the program code cannot break out of the virtual machine’s sandbox. This process attaches preverification information to the class files, so the class loader’s verification step is simpler, allowing class files to be loaded quickly and more efficiently.

Optimisation ProGuard carries out different analysis techniques on the bytecode to remove unnecessary field accesses, method calls, branches and comparisons. It can also remove unused code, reduce the allocation of variables and inline methods, constants and return values. The optimisation step’s goal is to reduce the time overhead for functions within the program, and make the code more efficient.

5.2 Creating RowenAppobsf

To obfuscate RowenApp, ProGuard was enabled in Eclipse. This was done simply by uncommenting a piece of code within the project.properties file, to set the proguard.config property [56].

ProGuard only operates on the application when it is built in release mode, i.e. when the application is exported. This means that the writer of the application can edit their own code in the debug mode and not interact with the obfuscated code. It processes the bytecode of the application before packaging it into the apk file and creates four files containing obfuscation information after it runs [56].
6 Reverse Engineering Tools and Process

This chapter focuses on the reverse engineering methodologies used in this work. The first method concentrates on the tool ApkTool and the text editor TextEdit. The second examines the dex2jar tool and the use of the graphical utility JD-GUI. Both of these methods were applied to RowenApp.apk and RowenAppobsf.apk. The examples discussed in the remainder of this chapter refer to RowenApp.apk, but exactly the same processes were applied to RowenAppobsf.apk.

6.1 ApkTool & TextEdit

As previously mentioned in Section 2.2.3, ApkTool [38] has been used in [31] and [37] to investigate app repackaging and reverse engineering. ApkTool is able to disassemble Android applications into smali/baksmali, allow the user of the tool to debug and modify the code, and then repackage the application. The smali files which are the output of ApkTool can be read by various editors, for instance [31] used the editor Notepad++ [57]. This work was completed on a computer running OS X, which has the editor TextEdit [58] already installed, so this editor was used.

6.1.1 Installing ApkTool

[38] provides the downloadable content for ApkTool, along with an installation guide. There are also dependencies for this software for Windows, Mac OS X and Linux. The version of ApkTool downloaded and installed by the author was 1.5.2 from the file apktool1.5.2.tar.bz2, and the dependencies installed were from the file apktool-install-macosx-r05-ibot.tar.bz2. These two files were downloaded from the Downloads tab of [38].

Using ApkTool’s user guide [38] and [59], the downloaded files were “unzipped” and stored in a single folder “apktool1.5.2” in the Documents folder. The folder “apktool1.5.2” contained three files after this process: aapt and apktool, both Unix Executable Files, and apktool.jar, a Java JAR file. These three files were unpacked to the /usr/local/bin directory by copying them to this directory using the command cp. This whole process can be seen in Figure 22. The three files can be seen in the apktool1.5.2 folder and the copying of the directory to the /usr/local/bin directory is there too.

Figure 22: The unpacking of the ApkTool files to the /usr/local/bin directory.

6.1.2 Using ApkTool

Once ApkTool has been installed, the application to be decompiled must be stored within the same folder as the ApkTool. The command to decompile the application’s .apk file is typically 

```
apktool d filename.apk
```

[59]. However since relative paths are used to execute ApkTool and the present working directory is apktool1.5.2, the command must be prefixed with ./ to make the command./apktool d filename.apk [60].

Figure 23 demonstrates the process the author used to decompile RowenApp.apk. The command ./apktool d RowenApp.apk can be seen clearly, along with the output of this command,
ApkTool stores the output of this decompilation in a new folder named “RowenApp” within the “apktool1.5.2” directory.

The output of this tool, the “RowenApp” folder, contains further files and folders [31]:

- res folder: This contains the Android resources used in the app which includes the xml files used for the layout, string and drawable resources.
- smali folder: This contains the smali code of the source files in the application, and also the Android support files.
- AndroidManifest.xml: The AndroidManifest file used to define the architecture of the app.
- apktool.yml: A YAML (YAML Ain’t Markup Language) document containing information on the ApkTool version, SDK version and the application package.

The contents of these files can then be opened and examined using the TextEdit application, as seen in Figure 24.

```
Figure 23: The decompiling of RowenApp.apk.

The application TextEdit displaying one of the smali files.
```

```
Figure 24: The application TextEdit displaying one of the smali files.
```
6.2  dex2jar & JD-GUI

dex2jar and JD-GUI provides an easier way to view the source code of an application. Instead of viewing the smali code from disassembling the apk file (a completely different format to the language the application was written in), dex2jar converts the Dalvik bytecode back into Java bytecode. This means the source code can be viewed in the Java format it was written in, using the JD-GUI, as done in [31].

6.2.1  Installing dex2jar and JD-GUI

dex2jar, similarly to ApkTool, has the downloadable files and user guide, all on its website [43]. Following the user guide’s instructions, the file dex2jar-0.0.9.15.zip was downloaded and unzipped. The resulting dex2jar-0.0.9.15 folder containing the dex2jar version 0.0.9.15 was moved into the Documents directory. The JD-GUI software was downloaded from [61]. The file jd-gui-0.3.5osx.i686.dmg was downloaded; it contained the 0.3.5 version of JD-GUI along with a readme.txt file.

6.2.2  Using dex2jar

After the installation of dex2jar and JD-GUI, dex2jar was used on RowenApp.apk to recover the Java source code files. The command to initiate this process on a Linux or Unix system is of the form

```
sh /user/.../dex2jar -version/d2j -dex2jar.sh /user/.../workingfolder/application.apk
```

This process can be seen in Figure 25. The whole sh command can be seen, along with the location of the dex2jar-0.0.9.15 folder in the Documents Directory. The working folder in this case is the apktool1.5.2 folder as this folder already contains the RowenApp.apk file. The generated file from the dex2jar process, RowenApp-dex2jar.jar, can also be seen in the apktool1.5.2 folder with the use of the ls command.

```
ls RowenApp RowenApp.jar apktool RowenApp-dex2jar.jar
```

Figure 25: The process of recovering the .jar file from RowenApp.apk

The JD-GUI is then able to open the generated .jar file and the source files can be viewed, as seen in Figure 26.
6.3 App Files and Tool Outputs

A copy of the outputs of ApkTool and dex2jar applied on RowenApp and RowenAppobsf have been deposited with the Smart Card Centre of Royal Holloway, University of London, along with the application package files of RowenApp and RowenAppobsf.
7 Analysis of Results

Both RowenApp and RowenAppobsf were reverse engineered using the two methods in Chapter 6. This chapter analyses the outputs from the two tools, ApkTool and dex2jar, for both apps. Firstly, the outputs from RowenApp are examined and compared to the original application source files viewed in Eclipse. Secondly, the outputs from RowenAppobsf are compared to the outputs from RowenApp, to analyse how the obfuscation process has changed the reverse engineering outputs. The results are summarised in tables at the end of the chapter.

7.1 RowenApp ApkTool Output Comparison

The RowenApp.apk file was decompiled using ApkTool using the method seen in Section 6.1.2 and the smali source code files were examined using TextEdit. The “RowenApp” folder generated in this process also contained the resources folder, android manifest file and the apktool file for RowenApp.

This section firstly gives initial observations the author made when first examining the ApkTool output. It gives a brief overview of how the smali files are structured, their syntax and how methods are constructed, and also an insight into how the author became familiar with smali, with the help of [64] and [65]. This section then goes on to compare different features of the smali code with the original source files, starting at 7.1.1.

The beginning of the smali file for DBDetails is shown in Figure 27 below. The name of the class, the superclass and the original source file are shown in the first three lines of the smali file. Private variables and their types are then listed in the figure, using the form variable name: type, for instance the entry _accountno:Ljava/lang/String defines the string variable _accountno.

Figure 27: The opening section of the DBDetails.smali file
The smali file in Figure 27 shows the direct methods of the class. Methods begin with \texttt{.method} and end with \texttt{.end method}. The method shown in this figure is the default constructor, which has no arguments as seen in the empty brackets when the method is defined. The \texttt{.locals} provides the number of local parameters in the method, which in this case is 0.

The constructor uses \texttt{invoke-direct \{} p0, Ljava/lang/Object; \} \texttt{<init> ()V}. This is a Dalvik opcode of the form \texttt{invoke-direct \{} \texttt{parameters}, \texttt{method to call}, which invokes a method with the parameters \cite{65}. The opcode from Figure 27 invokes the \texttt{init} method of the object with the parameter p0, which is the “this” reference in Java. This line has the effect of initialising a DBDetails object. The method then ends and returns \texttt{void}.

The corresponding part of the DBDetails Java class from Eclipse is shown in Figure 28 below. Note the similarity between the listing of the private variable of the class in Figures 27 and 28. The default constructor of DBDetails is shown below the variables in Figure 28; it provides the code to initialise a DBDetails object.

![Figure 28: The opening section of the DBDetails.java file.](image)

The line numbers corresponding to parts of the DBDetails.java file in Figure 28 can be seen in the smali file in Figure 27. The line showing initialisation of the object using the p0 parameter is under a reference for line 14 and the line showing the ending of the method and the void return is under a reference for line 16. This coincides with the line number in Figure 28, line 14 is the beginning of the constructor and line 16 is the end of the constructor.

The next constructor in the smali file, is the constructor which includes all eight variables. This is shown in Figure 29. The constructor now has 8 arguments; their type is declared in the brackets and their name is listed below as parameters \cite{64}. The constructor goes on to use these parameters and references them as the letter p and a number which corresponds to the parameter’s position in the list. For instance p1 is “id”, p2 is “userID” etc.

The object is initialised using the invoke-direct line. The next opcode used is the input-opcode. This stores p1, the parameter “id”, in the field \_id of the DBDetails object, and is referenced by p0, the “this” reference. The next opcode to be used is the input-object opcode, which essentially does the same as input, but the first parameter is an object reference. Following this logic, it can be deduced that this constructor stores the parameters passed into the

43
constructor in their corresponding fields, and the DBDetails object can be referenced by “this” [64, 65].

Figure 29: The eight variable DBDetails constructor.

Now compare this to Figure 30, the DBDetails Java class, both constructors are seen achieving the same effect. Figure 30 stores the arguments passed into the constructor in the same way as Figure 29. The actions completed on each line in Figure 30 agree with those in Figure 29, proven by the line numbers.

From Figures 27 to 30, it is easy to conclude what the opening sections of the DBDetails class does.
Following on from these initial observations, the remainder of Section 7.1 compares and contrasts specific aspects of the smali code with the application’s source files.

### 7.1.1 Context

The context of the application remains in the smali files; all classes, methods and variables have the same names as in the Java files. This means it is easy to deduce the purpose of classes and their methods, for instance the Crypto class is for cryptographic purposes, and its method `encrypt` provides encryption.

### 7.1.2 Syntax

The syntax of smali is very unlike Java, but in some ways is similar to that of assembler. Parameters and variables are stored in “registers”, v1, v2, p0, p1 etc. and opcodes operate on these registers using an assembler-like syntax. For instance the line `new-instance v1, Landroid/content/ContentValues;` creates a new instance of the ContentValues class and stores it in v1.

### 7.1.3 Opcodes

Methods require more lines of code due to the assembler-like syntax of smali. Since a new line is needed for each Dalvik opcode, one line of Java code can amount to many lines of smali. This is demonstrated in Figure 31 showing the beginning of the `addDetails` method of the MyDBHandler.smali file. A new instance of the ContentValues class is created and stored in v1 (red box), the ContentValues object in v1 is initialised (green box), and the ContentValues object in v1 is named “values” (blue box). In other words, a ContentValues object is created called “values”. This process takes 3 opcodes and 3 lines of code.

```smali
.method public addDetails(Lcom/example/rowenapp/DBDetails;)V
 .locals 4
 .parameter "detail"
 .prologue
 .line 55
 new-instance v1, Landroid/content/ContentValues;
 invoke-direct {v1}, Landroid/content/ContentValues;-><init>()V
 .line 57
 local v1, values:Landroid/content/ContentValues;
```

Figure 31: The opening line from the `addDetails` method from MyDBHandler.smali.

However, the same process takes one line in Java, shown in Figure 32, an excerpt from the
addDetails method from the Java file. Line 55 shows the creation of the ContentValues object “values”.

```
52 public void addDetails(DBDetails detail) {
53     ContentValues values = new ContentValues();
```

Figure 32: The opening line from the addDetails method from MyDBHandler.java.

### 7.1.4 Methods

The process of calling a class’s method in the smali files is different to that of Java. The smali files do not use the format, class.method() as in Java; methods are called using arrows, seen in Figure 33. This excerpt in Figure 33 is the following lines of code after the excerpt in Figure 31. v1 is the ContentValues object, “values”, and p1 is the DBDetails object passed into the method. The blue box represents the code calling the getfirstname method of the DBDetails class on p1. The result of this is stored in v3. The purple box represents the values object v1 calling the method put using v2 and v3 as arguments. Both these methods are called using the − > symbol.

```
const-string v2, "firstname"
invoke-virtual (p1), Lcom/example/rowenapp/DBDetails;->getfirstname()Ljava/lang/String;
move-result-object v3
invoke-virtual (v1, v2, v3), Landroid/content/ContentValues;->put(Ljava/lang/String;Ljava/lang/String;I)V
```

Figure 33: Excerpt from addDetails.smali.

The equivalent code in Java is shown in Figure 34. The blue and purple boxes represent the corresponding calling of methods from Figure 33. The methods here are called using a dot. The variable “COLUMN_FIRSTNAME” is a string variable representing the string “firstname”, a column header for the database, and equivalent to v2 in Figure 33.

```
values.put(COLUMN_FIRSTNAME, detail.getfirstname());
```

Figure 34: Excerpt from addDetails.java.

### 7.1.5 Classes

When instances of classes are used in the application, in the smali code they are represented in terms of the package they are from, giving more information away to the reader about where the classes come from. Going back to Figure 31, the ContentValues class is always represented as Landroid/content/ContentValues, and from Figure 33, the DBDetails class is represented as Lcom/example/rowenapp/DBDetails.

### 7.1.6 Line Numbers

Collections of opcodes in the smali files are grouped together under the line number corresponding to their actions in the Java file. This means it is a bit easier for the reader of the smali file to discern how the code is put together, even though not all methods within the smali file are in the same order as the ones in the Java file.
7.1.7 Conditional Statements

The conditional statements in the `lookDetail` method of the AddDetails activity, used to determine whether to send error messages if incorrect information is provided, are represented by if opcodes in the smali files. The conditional statements are presented by the specific conditional if opcode, the parameter which is subject to the condition and a target position the code jumps to, if the condition is fulfilled [65].

Take the example shown in Figure 35, an excerpt from AddDetails.smali, where the opcode `if − nez v14, :cond_0` is used. This line checks the DBDetails object `lookupdetail` stored in v14 and jumps to : cond_0 is v14 is non-zero [65]. If the v14 is equal to zero, the program will follow the next line of code to create a new instance of an AlertDialogBuilder and not jump.

```
.line 126
.local v14, lookupdetail:Lcom/example/rowenapp/DBDetails;
if−nez v14, :cond_0

[line 127]
new-instance v10, Landroid/app/AlertDialog$Builder;
```

Figure 35: Excerpt from AddDetails.smali showing an example of conditional statements.

Figure 36 shows another excerpt from the AddDetails.smali file. It shows the placement of : cond_0, the target line of code where the program jumps to if v14 is not equal to zero.

```
[line 150]
.end local v10  #dolgAlert:Landroid/app/AlertDialog$Builder;
:cond_0
invoke-virtual {v14}, Lcom/example/rowenapp/DBDetails;−>getsalt()Ljava/lang/String;
```

Figure 36: Excerpt from AddDetails.smali showing : cond_0.

These two AddDetails.smali excerpts represent the if condition from AddDetails.java shown in Figure 37. The line numbers for the if condition statement and the target to jump to both agree in Figures 35, 36 and 37.
7.1.8 Inner Classes

Inner classes within classes are defined in separate files. Within the smali files for AddDetails and MainActivity, there exists references to other smali files, AddDetails$1, AddDetails$2 and MainActivity$1. These files, also within the smali folder of the RowenApp folder generated by ApkTool, are the inner classes of AddDetails and MainActivity [64]. They are used to define listeners which register callback functions when buttons are clicked.

Figure 38 shows an excerpt of MainActivity.smali. It can be seen that a new instance of Lcom/example/rowenapp/MainActivity$1 is created and stored in v1, initialised then named listnr.

Figure 38: Excerpt from MainActivity.smali showing the creation of the MainActivity$1 inner class.

Figures 39 and 40 show excerpts from MainActivity$1.smali. Figure 39 shows the name and source of the class and also that it implements the interface OnClickListener. Figure 40 shows the onClick callback method which is used when a button is clicked. It can easily be seen that an intent is created, and with further examination, the purpose of the intent is to start the AddDetails activity. Figure 41 shows an excerpt of the MainActivity.java source file, with the OnClickListener inner class within the source file as well as the callback function.
.class Lcom/example/rowenapp/MainActivity$1;
.super Ljava/lang/Object;
.source "MainActivity.java"

# interfaces
.implments Landroid/view/View$0OnClickListener;

# virtual methods
.method public onClick(Landroid/view/View;)V
    .locals 3
    .parameter "v"
    .prologue
    .line 25
    new-instance v0, Landroid/content/Intent;
    .line 26
    iget-object v1, p0, Lcom/example/rowenapp/MainActivity$1;->this$0:Lcom/example/rowenapp/
    MainActivity;
    invoke-virtual {v1}, Lcom/example/rowenapp/MainActivity;->getBaseContext()Ljava/content/Context;
    move-result-object v1
    .line 27
    const-class v2, Lcom/example/rowenapp/AddDetails;
    invoke-direct {v0, v1, v2}, Landroid/content/Intent;-><init>(Ljava/content/Context;Ljava/lang/
    Class;)V
    .line 28
    .local v0, i:Landroid/content/Intent;
    iget-object v1, p0, Lcom/example/rowenapp/MainActivity$1;->this$0:Lcom/example/rowenapp/
    MainActivity;
    invoke-virtual {v1, v0}, Lcom/example/rowenapp/MainActivity;->startActivity(Landroid/content/
    Intent;)V
    .line 29
    return-void
.end method

Figure 39: Excerpt from the top of MainActivity$1.smali.

Figure 40: Excerpt from MainActivity$1.smali showing the onClick method.
7.2 RowenApp dex2jar Output Comparison

RowenApp was operated on by the tool dex2jar as described in Section 6.2.2 and the RowenApp-dex2jar.jar file was created with the output of this tool. This file was opened and viewed with JD-GUI. A comparison of the contents of this .jar file to the original source code was carried out and the main points are set out below. The output of dex2jar, the .jar file, only contained the source files, it did not contain the app resources or manifest file.

7.2.1 Java

The dex2jar tool and JD-GUI converted RowenApp.apk back into Java language, so the resulting source files use the same language that it was written in and are almost identical to those written by the author. The Java language is high-level so it is more natural to read and understand.

7.2.2 Context

The context of the application is still evident with all classes, methods and global variables retaining their names so the purpose of the application is easily discovered. However, even though global variable names are remain the same, local variables do not retain their names.

This is shown in Figure 42, an excerpt from the reverse engineered AddDetails class, viewed in JD-GUI. The class retains its name of AddDetails and the names of the global EditText variables e.g. accountnoBox, sortcodeBox etc., but in the lookDetail method, the local variable names are different to those in the actual source code, shown in Figure 43.

We can see that in Figure 43, the text in each of the EditText widgets is defined as a string variable and given a name that signifies what the text represents, e.g. the text from the firstnameBox is given the name firstname. However in Figure 42, each of the strings here are...
given the name of str1, str2, str3 and so on, removing some of the context and some of the ease of reading the code.

```java
package com.example.rowenapp;
import android.app.Activity;

public class AddDetails extends Activity {
    EditText accountBox;
    EditText firstnameBox;
    TextView idView;
    EditText passBox;
    EditText sortBox;
    EditText surnameBox;
    EditText uIDBox;

    public void lookDetail(View paramView)
    throws Exception
    {
        MyDBHandler localHandler = new MyDBHandler(this, null, null, 1);
        String str1 = this.passBox.getText().toString();
        String str2 = this.firstnameBox.getText().toString();
        String str3 = this.surnameBox.getText().toString();
    }

    /** Function to lookup account information using first name and surname and password **/  
    public void lookDetail (View view) throws Exception {
        MyDBHandler dbHandler = new MyDBHandler(this, null, null, 1);

        //set up database handler
        String passno = passBox.getText().toString();
        String firstname = firstnameBox.getText().toString();
        String surname = surnameBox.getText().toString();
        String uID = uIDBox.getText().toString();
    }
}
```

Figure 42: Excerpt from AddDetails class from dex2jar output.

Figure 43: Excerpt from AddDetails.java showing the lookDetail method.

7.2.3 Resource Referencing

The reverse engineered files differ from the original source code files, in such that the references to resources is different. As explained in Section 3.2.2, resources can be referenced using resource IDs of the form R.type of resource.resource name. For instance, the MainActivity file sets up the user interface using the setContentView method with the resource ID R.layout.activity_main, as seen in Figure 44.

```java
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
```

Figure 44: Excerpt from MainActivity.java from the original source files.
However, the dex2jar output files do not reference the resource ID in this way, they reference the static integer within the R.class file itself. Figure 45 below, shows the same part of the MainActivity file as Figure 44, but is from the reverse engineering process. The same setContentView function and findViewById methods are used, but they use the integers corresponding to the resource IDs in the R.class file in Figure 46.

### Figure 45: Excerpt from MainActivity.java from the reverse engineered files.

```java
public class MainActivity extends Activity {
    protected void onCreate(Bundle paramBundle) {
        super.onCreate(paramBundle);
        setContentView(2130903841);
    }
}
```

### Figure 46: Excerpt from R.class from the reverse engineered files.

```java
public static final class layout {
    public static final int activity_details = 2130903840;
    public static final int activity_main = 2130903041;
}
```

This is a legitimate way to reference the resources, but using the static integers removes more context from the code at first glance. The reader of the file need only to examine the R.class file to determine which resources have been referenced, however this makes the understanding of the application more time-consuming.

#### 7.2.4 Variable Names

The output of dex2jar is slightly different to that written by the author. The allocation of variables has been reduced from that of the original source code. Local variables that are only used once in a method are referenced when needed and not stored. An example of this is from the Cipher class. In this excerpt of the decrypt method, the variables cipher, key, ivParams, have been defined which are all highlighted in Figure 47 below. This figure is from the original source code.

### Figure 47: Excerpt from Crypto.java from the source files.

```java
public static String decrypt(String encrypted, char[] password, byte[] salt, byte[] iv) throws In
NoSuchAlgorithmException, NoSuchPaddingException, IllegalBlockSizeException, BadPaddingException, InvalidKeySpecException, InvalidAlgorithmParameterEx
ction {
    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");

    try{
        SecretKeySpec key = generateKey(password, salt);
        IvParameterSpec ivParams = new IvParameterSpec(iv);
        cipher.init(Cipher.DECRYPT_MODE, key, ivParams);
    }
    // Get a cipher object.
    public static String decrypt(String encrypted, char[] password, byte[] salt, byte[] iv) throws In
    NoSuchAlgorithmException, NoSuchPaddingException, IllegalBlockSizeException, BadPaddingException, InvalidKeySpecException, InvalidAlgorithmParameterEx
```

### Figure 48: Excerpt from Crypto.java from the source files.

Figure 48 is the corresponding excerpt of decrypt in the dex2jar output of RowenApp. In this figure only the variable localCipher is defined. The arguments in the init method of the
Cipher object, cipher, correspond to key and ivParams of Figure 47, but they are not referenced under variable names, they are referenced via their methods. The highlighted portions of Figure 48 correspond to the highlighted variables of Figure 47.

Figure 48: Excerpt from Crypto.class from the reverse engineered files.

7.3 RowenAppobsf Comparisons

The obfuscated version of RowenApp, RowenAppobsf, was decompiled and the smali files were examined also using ApkTool and TextEdit. RowenAppobsf and was also operated on by dex2jar and viewed with JD-GUI. The folder RowenAppobsf was generated by ApkTool and the RowenAppobsf-dex2jar.jar file was generated by dex2jar. Again, the ApkTool output contained the resource files and manifest file, but the dex2jar output did not.

This section aims to analyse how the reverse engineering process of RowenApp is altered by the obfuscation of its source code. It attempts to determine how the output of ApkTool and dex2jar has changed after app obfuscation and what impact these changes have. The main points are expanded on in the subsections below.

7.3.1 Class Names

The names of the classes used in RowenAppobsf have been changed from the original RowenApp. The collection of files resulting from both reverse engineering process include the following files: MainActivity, AddDetails, a, b, c, d, e, and f.

The dex2jar output now contains eight files, same as both ApkTool outputs. This differs from the five source files generated from the RowenApp dex2jar process. From examining these files, files a, b and e are the inner classes of the AddDetails and MainActivity activities, from their use of the OnClickListener, evident both in the ApkTool output and dex2jar output.

The naming of non-activity classes as single letters removes the context gathered from the names of the classes. The classes are no longer named Crypto, DBDetails and MyDBHandler, so the information as to the purpose of these classes can no longer be retained from their names. The reader of the files cannot infer that, for instance, class f handles cryptography, the way the reader could of the Crypto class.

7.3.2 Method Names

The names of the methods within these classes with single-letter names are now labelled with single letters such as a, b, c, d etc. This mirrors the effect from changing the class names; it removes information and context about the purpose of these methods. These method names are also often used multiple times within each class. Each class tries to have as many methods called a as possible until it uses the method name b. The method names can be duplicated

```java
public static String decrypt(String paramString, char[] paramArrayOfChar, byte[] paramArrayOfByte1, byte[] paramArrayOfByte2)
    throws InvalidKeyException, NoSuchAlgorithmException, NoSuchPaddingException, IllegalBlockSizeException, BadPaddingException
{
    Cipher localCipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    try{
        localCipher.init(?, new javax.crypto.spec.X509EncodedKeySpec(paramArrayOfByte2));
    }
```
because each class has methods with different numbers of argument, different types of argument and different types of return value and can be distinguished by these factors, and not the name. Multiple methods named the same letter makes it hard for the reader of the code to differentiate between the methods, making the understanding of the output of these tools more time-consuming.

The reader of the code gets no information of the purpose or of result when these methods are called in the activity class, for instance the lines of code in Figure 49. This is a sample of the AddDetails.class file from the dex2jar output. No contextual information about the strings str4 and str5 can be ascertained, only that they are output of class c’s b method with the arguments listed. The functions within the c class must be examined independently to discover their purpose to find out more information on str4 and str5.

```
byte[] arrayOfByte1 = c.a(locald.f());
byte[] arrayOfByte2 = c.a(locald.g());
char[] arrayOfChar = a(str1);
String str4 = c.b(locald.b(), arrayOfChar, arrayOfByte1, arrayOfByte2);
String str5 = c.b(locald.c(), arrayOfChar, arrayOfByte1, arrayOfByte2);
```

Figure 49: Excerpt from AddDetails.class from the reverse engineered RowenAppobsf.

### 7.3.3 Variable Names

The names of global variables in each class have been altered to remove context from the application. They, like the classes and methods, have been given single letter names. Figure 50 (a) and (b) show this. (a) is from d.smali and (b) is from AddDetails.smali, and these excerpts show the global variables of these two classes have been renamed a, b, c, d etc. Again, this removes the context of these variables, and is especially difficult for the d class, where its variables and methods are both labelled a, b, c, d etc.

```
# instance fields
.field private a:
.field private b:Ljava/lang/String;
.field private c:Ljava/lang/String;
.field private d:Ljava/lang/String;
.field private f:Ljava/lang/String;
.field private h:Ljava/lang/String;

# instance fields
.field a:android/widget/EditText;
.field b:android/widget/EditText;
.field c:android/widget/EditText;
.field d:android/widget/EditText;
.field f:android/widget/EditText;
.field h:android/widget/EditText;
```

(a) Excerpt from d.smali. (b) Excerpt from AddDetails.smali.

Figure 50: Excerpts of the smali output of RowenAppobsf, showing the naming of variables.
7.3.4 Remaining Context

The context has been removed from the classes and methods written by the author. However, context is still retained within the imported Android classes. This app and its classes use Android packages and classes in order to make them work, and their class and method names are left unchanged.

This remaining context is shown:

- when Android packages, classes and methods are used, e.g. the new instance of the ContentValues class in Figure 51 below.

```java
ContentValues localContentValues = new ContentValues();
```

Figure 51: Excerpt from f.class from the reverse engineered RowenAppobsf.

- when packages are imported, e.g. in the f.class below in Figure 52, six Android packages are imported, three of which are on the subject of SQLite databases.

```java
import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteDatabase.CursorFactory;
```

Figure 52: Excerpt from f.class from the reverse engineered RowenAppobsf.

- when classes are extended, e.g. when class f is extends SQLiteOpenHelper in Figure 53 and its superclass is shown the f.smali file in Figure 54.

```java
public class f extends SQLiteOpenHelper
```

Figure 53: Excerpt from f.class from the reverse engineered RowenAppobsf.

```java
.class public lcom/example/rowenapp/f;
.super Landroid/database/sqlite/SQLiteOpenHelper;
```

Figure 54: Excerpt from f.smali from the reverse engineered RowenAppobsf.

The context of the application is very apparent in the f class, in both smali and Java class files. The class imports SQLite database packages, evidence that the class interacts with a SQLite database and is the equivalent of the MyDBHandler class in the unobfuscated version. Examining f further, the column headers have remained unobfuscated, and retain their names, meaning that the methods creating and interacting with database are clear, giving away the structure of the database.

The column names also give context to the d class and its methods. The function a in the f class, creates a ContentValues object and inputs a column name with a value returned from a method operating on a d object. An example of this is shown in Figure 55, the column name
"firstname" is grouped with `paramd.b()`, implying that this returned value is the first name of someone. The `b` method of the `d` class, returns the second variable of the `d` constructor, giving more context to the `d` class and its use within the application.

```java
public void a(d paramd)
{
    ContentValues localContentValues = new ContentValues();
    localContentValues.put("firstname", paramd.b());
}
```

Figure 55: Excerpt from f.class from the reverse engineered RowenAppobsf.

From examining the files from both reverse engineering methods, the obfuscated files have been identified:

- `a` : AddDetails$1 (not present in dex2jar output of RowenApp)
- `b` : AddDetails$2 (not present in dex2jar output of RowenApp)
- `c` : Crypto
- `d` : DBDetails
- `e` : MainActivity$1 (not present in dex2jar output of RowenApp)
- `f` : MyDBHandler

### 7.3.5 Code Shrinking

The obfuscation process has also shrunk the amount of code in the output files by removing unused code. Unused methods have been taken out of some of the classes which is evident in the shorter file lengths resulting from both reverse engineering tools.

### 7.3.6 Activities

Most of the method names in both the activity classes remain. Important methods for instance `onCreate`, `onStart` have not been obfuscated as these are important methods which control the activity lifecycle for the application. Two other methods in AddDetails retain their names, `lookDetail` and `newDetails`, possibly because the buttons set up in the unobfuscated layout resource file have an attribute named `onClick` and this is equal to the method defined in AddDetails.

### 7.4 Summary

The tables below summarise the results of the heuristic examination by the author of the tools and processes used to reverse engineer RowenApp and RowenAppobsf.

One column indicates the “Aids to Understanding”, the aspects of the tool outputs which made the heuristic analysis easier.

The “Difficulties to Understanding” column represents the aspects of the reverse engineering output which hindered the heuristic analysis and discovery of the app’s function.
Table 2: RowenApp. Use of tools, ApkTool and dex2jar, applied to original application.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Difficulties to Understanding</th>
<th>Aids to Understanding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApkTool</td>
<td>• Output in low-level smali code</td>
<td>• Represents all files accurately</td>
</tr>
<tr>
<td></td>
<td>• Increased number of lines of code</td>
<td>• All resource files and manifest file included</td>
</tr>
<tr>
<td></td>
<td>• Context of app remains</td>
<td>• Line numbers from original source code files remain</td>
</tr>
<tr>
<td></td>
<td>• Methods and their classes can be easily seen with their package information</td>
<td></td>
</tr>
<tr>
<td>dex2jar</td>
<td>• Resources referenced by integers</td>
<td>• Files are in Java</td>
</tr>
<tr>
<td></td>
<td>• Variables renamed with non-contextual names</td>
<td>• Represents all source files almost exactly like original files</td>
</tr>
<tr>
<td></td>
<td>• No resource files or manifest file</td>
<td>• Context of app remains</td>
</tr>
</tbody>
</table>

Table 3: RowenAppobsf. Use of tools, ApkTool and dex2jar, applied to the obfuscated application. The bullet points in *italics* indicate the changes observed on the obfuscated app.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Difficulties to Understanding</th>
<th>Aids to Understanding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApkTool</td>
<td>• Output in low-level smali code</td>
<td>• Still represents all files accurately</td>
</tr>
<tr>
<td></td>
<td>• Increased number of lines of code</td>
<td>• <em>Some context can be deduced within source files</em></td>
</tr>
<tr>
<td></td>
<td>• Context of <em>classes, methods and variables has been removed</em></td>
<td>• Xml resource files and manifest included</td>
</tr>
<tr>
<td></td>
<td>• <em>R files no longer included in output</em></td>
<td>• Line numbers from original source code remain</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <em>Unused code has been removed so output represents app function fully</em></td>
</tr>
<tr>
<td>dex2jar</td>
<td>• Resources referenced by integers</td>
<td>• Files are in Java</td>
</tr>
<tr>
<td></td>
<td>• <em>Context from classes, methods and variables has been removed</em></td>
<td>• Represents all files accurately</td>
</tr>
<tr>
<td></td>
<td>• <em>R files no longer included in output</em></td>
<td>• <em>Some context can be deduced within source files</em></td>
</tr>
<tr>
<td></td>
<td>• No resource files or manifest file</td>
<td>• <em>Unused code has been removed so output represents app function fully</em></td>
</tr>
</tbody>
</table>
8 Conclusions

The growing smart phone industry has encouraged advances in the software development of mobile operating systems and applications. This development has lead to the creation and distribution of malicious software throughout the smart phone market. The most popular operating system, Android, is the target for nearly all phone malware, with one of the mechanisms to introduce malware being the reverse engineering of legitimate applications, installation of malicious code and its re-uploading to a third party app store.

This project aimed to investigate the reverse engineering process on an Android application, motivated by the myriad of plagiarised and repackaged applications found in studies. This project compared how well two tools, ApkTool and dex2jar, were able to reverse engineer an app written by the author. The impact of obfuscation of the app on the reverse engineering process was investigated by comparing the outputs of the two tools before and after obfuscation was applied.

The Android application, RowenApp was created by the author. Its purpose is to encrypt and store and retrieve personal banking information of the application’s user. The app was then obfuscated using the ProGuard tool within the Android SDK. Two versions of the app, an unobfuscated and obfuscated one, were exported and stored as RowenApp.apk and RowenAppobsf.apk respectively. Both of these apps were reverse engineered using the tools ApkTool and dex2jar and their output was examined.

Both ApkTool and dex2jar successfully reverse engineered RowenApp, and the nature and purpose of the application can be discovered by both tools. The dex2jar output of the source files was easier and quicker to understand than the ApkTool output as it was written in the high level Java language which the app was originally written in, and not the low level smali format of the ApkTool.

The ApkTool output, but not the dex2jar output, contained all the application resources needed in the app in their xml format. This created a considerable disadvantage to the author; it was not possible to be certain how some of the methods are used in the user interface with dex2jar. To get a full picture of the application, both tools could be used together, dex2jar for the source code in Java which is easier to understand, and ApkTool for finding the resources which are not present in the dex2jar output.

From the comparison between the tool outputs of RowenApp and RowenAppobsf, the main result is that the obfuscation process makes the reverse engineering process from both tools harder. The output from both tools operating on RowenAppobsf, find the context and purpose removed from classes, methods and variables; thus the app and tool output are more complex to understand.

A more laborious and time-consuming examination of the code was required to understand RowenAppobsf. It took the author over four hours to examine the output from both tools, compared to RowenApp which took only one hour, concluding that code obfuscation by ProGuard has therefore been effective. This is reinforced by the fact that the author (although not a knowledgeable reverse engineer) had the advantage of knowing the app’s function and source code, and could compare the unobfuscated and obfuscated versions.

58
The obfuscation process also removed the R file from the output of both tools, leaving the reader of the output unaware of the resources being used in the application, further complicating the reverse engineering of the app.

Configuring and applying ProGuard to the app was very simple and it adds difficulty into the reverse engineering process, so it is worth doing. However, the context and purpose of the application was still able to be deduced through examination of the f file. This file included column headings that were not obfuscated, leading to the conclusion that the obfuscation process used in this project is far from foolproof.

The obfuscation is applied when the app is built in release mode; the app writer has no control on the final version of their obfuscated code. The inexperienced app writer with only a basic knowledge of ProGuard, may assume all variables and methods are going to be obfuscated, but they may not be. This may leave important clues and information for a reverse engineer to find, as was the case in this project, as this work was the first experience of the author with Android application writing and ProGuard.

ProGuard and other obfuscation tools provide a way to hinder the reverse engineering process. However, the unrestricted access of malicious parties to powerful tools, online tutorials and resources, is a major problem when trying to protect an app, which cannot be solved by applying tools or processes to the app code.

8.1 Future Work

To explore the results and conclusions obtained from this work, the obfuscation process should be investigated further by using obfuscation techniques other than those provided by ProGuard. The application could be obfuscated using commercial tools, such as DexGuard, Allatori and Dalvik-obfuscator which provide encryption and junkbyte insertion, and the reverse engineering process conducted again. The output of other reverse engineering tools, such as IDAPro, could be analysed to give a fuller analysis of the reverse engineering process.

As well as investigating obfuscation tools, the effectiveness of obfuscation on different code and logic could be investigated. More test apps could be produced, covering all types of app coding and functionality. The obfuscation tools listed above could be applied and their effectiveness in countering the reverse engineering process on all the different coding elements could be tested. These results could perhaps be used to influence obfuscation tool design and improvement in the future.
References


