Verifying the Integrity of Open Source Android Applications
Michael Macnair

Technical Report
RHUL–MA–2015–4
4 March 2015

Information Security Group
Royal Holloway University of London
Egham, Surrey, TW20 0EX
United Kingdom

www.ma.rhul.ac.uk/tech
Michael Macnair
Student number: 100761605

Verifying the Integrity of Open Source Android Applications

28th August 2014

Supervisor: 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I declare that this assignment is all my own work and that I have acknowledged all quotations from published or unpublished work of other people.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:
## Contents

**Executive Summary**  
1

**1 Introduction**  
1.1 Android and Android Applications .................. 3  
1.2 Reverse Engineering ................................. 4  
1.3 Obfuscation ........................................ 4  
1.4 Verifying Integrity of Binary Distributions .......... 5  
1.5 Objectives ........................................ 5  
1.6 Report Structure .................................... 6  

**2 Use Cases for Reversing and Obfuscation**  
2.1 Use Cases ........................................ 7  
2.2 Conclusion ........................................ 10  

**3 Reverse Engineering Android Applications**  
3.1 The APK File Format ................................. 11  
3.2 Techniques .......................................... 12  
3.3 Tools .............................................. 14  

**4 Verifying Android Application Integrity**  
4.1 APKDiff ........................................... 23  
4.2 AppIntegrity ....................................... 26  

**5 Analysis of Applications on the Play Store**  
5.1 RedPhone ........................................... 35  
5.2 ChatSecure ......................................... 37  
5.3 Telegram ........................................... 38  
5.4 TextSecure ......................................... 40  
5.5 Lil' Debi: Debian Installer ............................ 41  
5.6 Summary ............................................ 42  

**6 Conclusion**  
6.1 Reverse Engineering for Android ...................... 45  
6.2 Reproducible Builds and Binary Integrity ............. 45  
6.3 Objectives ........................................ 46  
6.4 Future Work ........................................ 46  

**Bibliography**  
49  

**A aapt Dump Format**  
55
Executive Summary

Android is the most widely used operating system for mobile devices such as smartphones. Users can install additional applications onto Android devices, typically from the Google Play Store. Some of these applications are released under an open source licence, enabling their source code to be freely audited for their security or privacy properties, for example.

Open source Android applications are usually compiled into binary form and published on the Play Store. This means that whilst the source code for an application may be available for review, the user has no guarantees that the application they download from the Play Store was actually built from the published source code.

This project surveys the available reverse engineering tools and techniques that can be used to audit Android application binaries. No tools are currently available that adequately solve the binary integrity verification problem.

To address this we present AppIntegrity, a service that automatically downloads application binaries and sources, builds the sources and compares the resulting applications. The results are presented in the form of a website, allowing reviewers to see the extent of the differences between the published application and the version built from source by the independent AppIntegrity service. This serves two main purposes: encouraging application developers to produce applications with a so-called reproducible or deterministic build that produces the same output no matter who builds it; and providing a check against subverted binary versions of open source applications appearing in the Play Store.

Additionally, vendors of proprietary Android applications can host a private instance of AppIntegrity. This could be used to monitor their own applications for reproducibility, or for compromised binaries being uploaded to the Play Store.

AppIntegrity was used to check the integrity of four open source secure communications applications published on the Play Store: RedPhone, ChatSecure, TextSecure, and Telegram. None of the applications had a reproducible build, but the differences found between the published sources and binaries were consistently benign. The importance of good release practices were highlighted – determining the version of the source code that corresponds to the published binaries proved to be non-trivial in some cases, with the source for some versions of Telegram found to not be published at all.

In December 2014 the AppIntegrity website will go live. The source code for both the AppIntegrity website and the APKDiff tool that performs the comparisons between applications will be published under a free open source licence.
1. Introduction

Mobile applications are entrusted with access to our private data, from messages to photos, location data to banking credentials. Are they worthy of this trust? One way of determining this is through auditing applications, though this can be a challenging task for applications to which the auditor does not have access to the source code.

In such a scenario, an auditor can attempt to reverse engineer an application to understand its properties, for example what it is doing or whether it is secure.

Some application developers help foster trust in their applications by providing access to the source code under an open source licence. However even in the case of an open source application which has had its source code audited, a user of the application cannot easily tell whether the application they have downloaded – typically from the mobile platform’s store – was actually built from the code that was audited. The user is still at risk of subverted versions of the application being uploaded to the store due to either malicious developers, viruses or trojans on a developer’s computer, or the developer’s credentials for the store being compromised.

This study considers reverse engineering tools and techniques that can help in performing an audit of an Android application, then presents a new service for checking the integrity of published open source Android applications against their source code.

1.1 Android and Android Applications

Android is an operating system designed for mobile devices such as smartphones, developed primarily by Google as part of the Open Handset Alliance.

Since their introduction in the mid-2000s, smart-phone usage has proliferated, to the point where most adults in the USA own a smart phone [83]. Phones running the Android platform account for the largest share of the market (over 80% as of late 2013 [50]).

One of the defining features of a smartphone is the ability to install applications that add to the base functionality of the phone. Some applications ship with the device as part of Android (for example a browser, contacts and telephone are a few of the applications that form part of the core Android OS). Users can install additional applications, typically from Google’s Play Store™, but it is also possible to ‘side-load’ applications from other sources.

An Android application is contained within an Android Application Package (APK) file. The APK file is a collection of components [48], which includes everything necessary for the application to run [37]. APK files are ZIP con-
tainers, which can be unpacked using a standard unzip tool to examine the components within the APK. See section 3.1 for greater detail on the format of APK files.

Android applications are run using the Dalvik Virtual Machine (VM) and may also call native code. The Dalvik VM interprets bytecode; the Android Software Development Kit (SDK) produces Dalvik bytecode by compiling Java source code into Java bytecode, then converting the bytecode into the Dex format. Note that the new Android RunTime (ART) VM that will replace Dalvik in versions of Android beyond 4.4 does not alter the distribution format of applications, so all of the work in this project applies whether an application is targeting devices that use Dalvik or ART.

The twin properties of a ZIP container format and use of a bytecode language makes reverse engineering an Android application relatively accessible. This, in turn, leads some developers to try and make this task more difficult through the use of obfuscation. These two aspects are introduced in the next two sections.

1.2 Reverse Engineering

Reverse Engineering, or reversing, is the practice of taking an existing computer program and gaining an understanding of some aspect of its behaviour. Reversing usually involves static analysis – analysing the stored files of a program – but it can also involve dynamic analysis – observing the run time behaviour of a program.

Reverse engineering exists as an engineering discipline because software in the form that it is distributed and executed on user’s machines is often very different to the form it is written in in its original programming language. Software can be written in a higher level language (such as C or Java) and then compiled into a lower level language that is more readily executed by a computer or interpreter (such as x86 machine code or Java bytecode). The lower level interpretations are usually harder to understand as a human compared to the higher level language form of the same program.

Obfuscation presents further difficulties for reverse engineering and hence drives greater sophistication in reverse engineering tools and techniques.

1.3 Obfuscation

On Android, developers are encouraged by the official documentation to use the ProGuard obfuscation tool that is bundled with the Android SDK, so ‘it is difficult for an attacker to reverse engineer security protocols and other application components’ [80].

In general, developers of closed-source applications may wish to prevent others from reverse engineering their applications (see Chapter 2 for a number of use cases). This can be attempted through a number of techniques, for example not including debugging information such as comments and symbol names in the binaries, encrypting portions of the code or including unused misleading code. All of these techniques and more can be used to frustrate attempts to reverse engineer a program.
1.4 Verifying Integrity of Binary Distributions

One of the benefits of open source applications is the ability for anyone to audit the application for, say, its security or privacy properties. If a user subsequently goes on to build this audited code from source, then that user knows that whatever assurance the audit provided applies to the application they are running. For the vast majority of non-technical users though, this is not a practical option. For Android applications, users are much more likely to download the version of the application that is on the Play Store than build their own.

To realise this benefit of open source applications, users need some assurance that the application published to the Play Store is built from the published source code. To verify this, a reviewer could build their own version of the application from the source and see if it is the same. Unfortunately for the reviewer, the two files will not be identical, as the reviewer will not be able to sign the APK with the publisher's private key. More significantly, unless the developer has taken specific steps to ensure that the build is binary-reproducible from the source code, the unzipped contents of the APK will likely be different too. To determine whether a published application was built from the published source, the reviewer must reverse engineer the published application.

The profile of binary reproducibility (also known as deterministic builds) is rising in the security community. The Tor Project are perhaps the highest profile team concerned about reproducible builds, making the Tor Browser Bundle reproducible in the second half of 2013 [69]. The maintainers of the Fedora and Debian Linux distributions are also seeking to have fully reproducible builds [58; 73]. In the words of the Red Hat security team, 'Fedora shouldn’t be forced to say Trust Us when asked about proving the binary RPMs came from the source RPMs.' [58]. In the ongoing audit of the open source TrueCrypt disk encryption software, the auditors state 'Many of our concerns with TrueCrypt could go away if we knew the binaries were compiled from source.' [53]. There is very little work in this area for Android applications, with the Guardian Project being the only high profile developers targeting reproducible builds for their applications (see Section 5.5 for more details).

1.5 Objectives

This project will aim to satisfy the following objectives:

Objective 1: review  Enumerate and describe the reversing tools and techniques available to the Android community.

Objective 2: engineering  Provide tooling and/or services that enable the security community to verify the integrity of open source applications on the Play Store. This is the primary goal of the project.

Objective 3: case study  Use the capability developed under objective 2 to check the integrity of one or more open source Android applications published on the Play Store.
The rationale for these objectives are as follows:

**Objective 1: review** A review of the available tools will guide the engineering aspect of the project to make best use of existing work. Also tooling is an intrinsic part of reverse engineering – this review will provide a valuable resource to those wishing to reverse engineer Android applications.

**Objective 2: engineering** The problem of verifying the integrity of applications published on the Play Store against their source is unresolved. It would be of benefit to progress the state of the art in this area, particularly if the work is made available in a form that is readily usable by the security community (i.e. a free service and/or free open source software).

**Objective 3: case study** This objective delivers value in two ways. Firstly by analysing the work delivered under objective 2: using the tool on real world examples will provide a good opportunity to critically assess the effectiveness of the tool. Secondly by showing results, positive or negative, for applications with a significant user base. These results may fall on a spectrum from demonstrating the integrity of the application(s) with respect to their source, to providing useful feedback to developers on build and release process control, to uncovering evidence of negligence, compromise or malicious behaviour.

### 1.6 Report Structure

This chapter introduced Android, reverse engineering and obfuscation. This chapter also defined the initial objectives of the project.

Chapter 2 provides an in depth treatment of the different use cases for obfuscation and reverse engineering. It is easy to list motivating reasons for work on either obfuscation or reverse engineering alone, but doing so does not properly acknowledge the positive and negative use cases for both techniques, which are in direct conflict with each other. A consideration of both sides of the topic sets the scene for the project.

Chapter 3 describes the APK file format in some detail. It covers reverse engineering techniques that are applicable to Android and in particular the integrity verification problem. It then describes the tools that implement these techniques.

Chapter 4 describes the design and implementation of a tool for comparing two versions of an Android application and a service building on this tool for systematically and semi-automatically verifying the integrity of open source applications published on the Play Store.

Chapter 5 analyses the results of using this service to check the integrity of some popular open source secure communications applications published on the Play Store.

Chapter 6 provides a conclusion, summarising the problem, the available tools, the tools developed to help solve the problem and the results of applying those tools. It also describes areas of future work in the domain.
2. Use Cases for Reversing and Obfuscation

Reverse engineering and obfuscation are two sides of an arms race. This applies in general and on the Android platform. On the one side are developers wishing to prevent reverse engineering of their applications, and on the other are those intent on doing just that. This is comparable to other arms races in information security such as virus detection and anti-virus evasion, but in the case of reversing and obfuscating there are beneficial uses of both techniques. This chapter describes the motivations of the various actors that use obfuscation and carry out reverse engineering on Android.

2.1 Use Cases

2.1.1 Revenue Protection

Developers producing applications for the Play Store can make revenue from three main sources [2]:

1. the initial purchase of the application;
2. advertising served within the application;
3. in-app billing to unlock extra features or obtain content.

Each of these revenue sources is threatened by cloned versions of their applications made available either within or outside of the Play Store. With respect to each of the revenue sources, cloned applications may:

1. be downloaded with no payment to the developer;
2. have advertising removed, or re-targeted such that revenue from the adverts does not go to the original developer;
3. come with all in-app features unlocked.

To clone an application and modify it like this, the application must first be reverse engineered. This provides an incentive for all developers of revenue generating applications to obfuscate their applications to attempt to prevent such cloning.
2.1.2 Malware

2.1.3 Obfuscation of Malware

Developers of Android applications that perform malicious activity have a strong incentive to obfuscate their application to hinder others from fully understanding its behaviour. This could include:

- Hiding the payload, which may not be apparent from dynamic behaviour (e.g. payloads that trigger at a certain time, or only when certain other applications are present, etc);

- Increasing the time it takes for command and control behaviour to be identified and potentially blocked by IP blocking or domain name registration. For example if a domain generation algorithm is used to determine possible command and control servers, future domains can not be easily predicted without knowing the algorithm;

- Hiding anti-virus evasion techniques (for example exploiting a vulnerability in the underlying OS to gain persistence even after the malicious application is uninstalled).

Conversely, security researchers (in particular those working for anti-virus firms) would like to understand the complete behaviour of any malware they are analysing - to do this requires an ability to reverse engineer the sample.

Whilst obfuscation may do no more than delay the full behaviour of the application being understood by security researchers, this may have a direct positive financial impact for the malware authors - the longer the malware is operating without interference the greater the opportunity there is to make a profit.

2.1.4 Reverse Engineering to Produce Malware

As described in Section 2.1.1, applications available on the Play Store can be cloned and made available via alternate sources. This is a vector for malware to be distributed: the legitimate application is reverse engineered, malicious functionality is inserted into it and then the modified application is made available from the alternate source. Clearly the better the reverse engineering tools and techniques the malware authors have access to, the easier this process will be for them – the ideal situation from a malware author’s perspective is a fully automated app-cloning and payload insertion process.

This provides a motivation for all application developers to have access to strong obfuscation techniques and apply them to their applications, to increase the costs of this kind of activity for malware authors.

2.1.5 Privacy

Applications that are not designed as malware may still exhibit privacy violating or otherwise intrusive behaviour. Security researchers, privacy proponents or expert users may want to validate that an application complies with its stated privacy policy, or doesn’t otherwise violate the expectations of its users.
The Android permissions model requires applications to request permission prior to accessing sensitive information such as the device location, the user’s contacts, etc. Whilst the technological enforcement of this may be sound, the user behaviour aspects can be problematic: a large proportion of popular apps request potentially dangerous combinations of permissions such as internet access and precise location [40] and users are not good at recognising or reacting to dangerous combinations of permissions [41].

With many applications installed with permissions that technically enable them to violate privacy policies or expectations, the only way to verify that they do not do this is to reverse engineer the application.

As an example, the Facebook application has over half a billion installs on Android and requests permissions including access to: the internet; microphone and camera; contacts; and precise location [44]. To validate that the Facebook app never uses the microphone or camera without user consent requires an ability to reverse engineer the application to review when these features are used.

2.1.6 Security Review

Benign, non-privacy invading applications can still pose a threat to a user’s privacy and security.

Firstly, an application may be installed that request a dangerous permission but then ‘leak’ this capability to other applications, such that the other applications can use the capability without having requested the permission [49]. As an example, an application may be installed with the “Send SMS” permission, but due to faulty design another (malicious) application with no special permissions is able to send SMS messages to a premium number by using the leaked capabilities of the first application.

Secondly, an application that purports to offer some security feature may not have implemented it correctly. If the application is open source, then reviewers are able to easily review the application’s security; if the source is not available then being able to reverse engineer the application will enable a review to be performed.

2.1.7 Digital Rights Management

Application developers may implement a Digital Rights Management (DRM) system to attempt to control how digital content provided by the application is used. Users wishing to circumvent the system and obtain an unprotected copy of the content will be aided by being able to reverse engineer an application to at least understand how it operates and potentially to modify it to circumvent the DRM protection. This provides a motive for application developers to obfuscate their applications and frustrate attempts to bypass the protection schemes.

2.1.8 Intellectual Property Rights protection

Article 6, 2. (c) of the European directive on the legal protection of computer programs provides explicit protection against decompilation being used
for 'development, production or marketing of a computer program substantially similar in its expression' [34], where expression can be interpreted to mean implementation.

An application developer may have implemented some original functionality in their application and wishes to augment the legal protection of their application code with technical protection, by using obfuscation techniques.

2.1.9 License compliance checking

A developer of open source software may wish to verify that other applications are not using open source components in violation of their licences – copyleft licences like the GNU Public Licence (GPL) place a burden on an application that uses GPL licensed software to publish the application's own source under the GPL licence. Similarly a commercial developer may wish to check that published applications aren't using unlicensed copies of their software components.

2.1.10 Maintenance

A user may wish to fix a bug in an application that has gone out of support, or port it to a platform that is not supported (in the context of Android applications, this could be to port an application to an earlier version of the Android OS). Both of these use cases will involve reverse engineering to some degree, as the user must ascertain which part of the application needs to be changed.

2.1.11 Checking binary integrity

The problem of verifying that distributed binaries were built from the published source code was introduced in Section 1.4.

2.1.12 Software Watermarking

A software developer may wish to indelibly mark a version of the software such that it is uniquely tied to an individual customer, to deter software piracy. Once a watermark is present in an application, obfuscating the application helps prevent the removal of that watermark [93].

2.2 Conclusion

The use cases described above show that different actors have incentives to be able to carry out reverse engineering of applications and to prevent reverse engineering of their own applications. Sometimes the same actors have an incentive to do both! Both obfuscation and reverse engineering are just techniques that can be used for both good and ill. There are legitimate grounds for targeted research into both.

This project presents existing tools that help in carrying out security reviews, then focuses on solving the integrity verification problem by building on these tools.
3. Reverse Engineering Android Applications

In their definitive article on the topic, Chikovsky and Cross define reverse engineering as 'the process of analysing a subject system to: identify the system's components and their interrelationships; and create representations of the system in another form or at a higher level of abstraction.' [21], noting that reverse engineering does not always entail modification of the subject.

This chapter first introduces the APK file format, then considers the reverse engineering techniques and tools that can be used to audit a closed-source Android application or to verify the integrity of an open source Android application.

3.1 The APK File Format

The format of APK files is not covered in the Android developer documentation, as it is not necessary information for developing Android applications: the build tools take care of packaging. The principal components of an APK file are documented in this section.

An APK file is a ZIP archive containing all of the components needed for an Android application to run. Within the archive an APK file contains the following files and directories. Additional files may be present but have no semantic meaning for the Android OS.

- **AndroidManifest.xml**: Describes the name, required permissions, etc.
- **resources.arsc**: Resource mapping and compiled values.
- **classes.dex**: The Dalvik byte-code of the application.
- **META-INF/**: Directory containing files used for signature verification.
- **assets/**: Directory containing arbitrary files used within the application.
- **lib/**: Directory containing native libraries.
- **res/**: Directory containing resource files.

3.1.1 AndroidManifest.xml

This file defines all of the metadata the Android OS needs to know about an application, including name, version, minimum SDK version, any required permissions and the application components including any services and intents they handle.
When creating an Android application this is a normal XML file. When packaged into an APK, this and other Android XML files are compiled into a proprietary binary format. The Android SDK tool aapt is used to compile the XML resources into the binary format, and also provides functionality to inspect the binary XML resources.

3.1.2 resources.arsc and the res directory

Many static resources of an Android application such as strings, layout definitions and images are stored within the res directory [36]. Simple values (such as strings, styles, dimensions, etc) are stored in XML files within the res/values sub-directory. Arbitrary files can be stored within the res/raw sub-directory. During packaging, aapt compiles all of the definitions within the values directory into the resources.arsc file [63]. All other XML files within the res directory are compiled into a binary format and stored in the APK. All other resources (e.g. images, raw files) are stored as-is in the APK. In addition to the compiled values, resources.arsc contains a mapping of resource IDs to the files in the res directory of the APK.

3.1.3 classes.dex

The Java source files of an Android application are compiled into Java bytecode (.class files) by a Java compiler. The Android OS does not have a Java Runtime Environment (JRE) however, instead it uses the Dalvik VM. The dx tool is used as part of the build process to compile the Java class files into Dalvik bytecode in a single file, classes.dex.

3.1.4 The META-INF directory

This directory is based on the JAR file format and is used for signing and verification of the integrity of APKs. It includes a file manifest with hashes of all of the other files within the APK, a signature over this manifest and a public key certificate.

3.1.5 The assets directory

Contains arbitrary assets used by the application, for example images or text files. Similar to res/raw, but accessed via a different Android API.

3.2 Techniques

There are many aspects to reverse engineering, and precisely which techniques are applicable depends on the target application being reverse engineered and the goal of the activity. This section describes some of the techniques that are particularly applicable to auditing Android applications.

3.2.1 Disassembly

Disassembling a binary is the process of converting the machine code (in the case of Android the machine is usually the Dalvik VM, but it could be an ARM
3.2. TECHNIQUES

or x86 processor) to a human readable representation of that code called an assembly language. There is a one-to-one correspondence between the binary and the disassembled output (every instruction has both a machine code and an assembly representation).

3.2.2 Retargeting

In the case of Dalvik executables, retargeting entails conversion to or from Dalvik bytecode to another machine language, typically Java bytecode. In reverse engineering Android applications, retargeting from Dalvik bytecode (.dex) to Java bytecode (.class) enables use of existing Java reverse engineering tools to analyse the code of an Android application.

3.2.3 Decompilation

Decompilation is the process of taking a compiled form of a program and producing source code which, if compiled, would produce the original compiled form (or at least functionally equivalent code). For Android applications the target source code is usually Java, and the compiled form is Dalvik bytecode, unless it has already been retargetted to Java bytecode. There is an equivalent decompilation process for native code, where the target source code is more likely to be C or C++.

3.2.4 Dynamic Analysis

The techniques described so far are static – they inspect and process the target application as collections of structured data, some of which happen to be executable instructions. Dynamic reverse engineering techniques actually execute these instructions, and derive information about the applications' dynamic behaviour. The application may execute on the intended target (e.g. a mobile device running Android), or on an emulator. To support the analysis, sometimes the application is modified (instrumented) and often the execution environment contains instrumentation. A common trait of dynamic analysis techniques is that the analysis only reveals information about what was executed. Code paths that were not invoked might be identified as not having been invoked, but the conditions under which they would be invoked and the consequent behaviour of the application remain unknown. For this reason, dynamic analysis is not an ideal tool for an engineer wishing to verify the integrity of an application for which the engineer posses the purported source. Having carried out dynamic analysis of the application, little is known about the range of possible behaviours that were not seen during the combined runs. These behaviours could include backdoors, information leakage, etc. We do not consider dynamic analysis further in this report.

3.2.5 APK Specific Techniques

Some resources in an Android application require specific reversing techniques. In addition to the compiled form of the AndroidManifest.xml file and the resource XML files, so-called Nine-patches – images with nine different segments for use at different scales – are lightly transformed during packaging.
3.3 Tools

This section describes reverse engineering tools that implement one or more of the above techniques. Unless specifically included due to their popularity in the Android reversing community, certain categories of tools were omitted as they were considered to be of little value for static reverse engineering of Android applications: pure modification tools (for example ‘themers’ and other tools that focus on changing resource values); thin wrappers of other tools; tools which are no longer available; and tools that identify potentially malicious or insecure code but don’t provide output for the rest of the application.

There have been a number of previous efforts to collect and describe these tools [91; 38, Appendix A; 88, p. 73; 60, p. 9; 79], plus several presenters at security conferences list their preferred tools of the trade, for example [13; 33]. Despite the exclusions mentioned above there are still many more relevant tools than described in these existing collections. Whilst it is hard to claim to be comprehensive in this relatively new and fragmented field, presented below is: an extensive collection of Android specific static reverse engineering tools; Linux distributions that aim to provide a ‘one-stop-shop’ for Android reversing tools; and an overview of Java decompilers which can be used following retargeting of Dalvik bytecode. The tools are listed in alphabetical order within the categories.

See the following chapter for experiences with some of these tools, but note that not all of them were tested as part of this project, so inclusion should not be considered endorsement.

Tables 3.1, 3.2, 3.3, 3.4 and 3.5 list the tools, the licences under which they can be obtained and their last update as of August 2014. The licensing information is included to assist with tool selection, as some projects may have particular licence constraints (such as an inability to incorporate non-GPL compatible licences). The year of last update is included to provide an indication of whether the tools are actively maintained.

3.3.1 Android Static Reverse Engineering Tools

Android Asset Packaging Tool (aapt)

aapt is a tool that forms part of the Android SDK. It has two primary modes of operation: packaging up APK files, and dumping information about them. The dump feature can be used to output [1]:

- The APK’s label and icon (badging);
- The permissions requested in the manifest file;
- The resource table;
- The different configurations defined in the APK (configurations are device-dependent resource sets);
- The parse tree for a compiled XML asset;
- The strings defined in a specified compiled XML asset.

Note that aapt outputs the parse tree for XML files; it does not output decompiled XML files.
3.3. TOOLS

Androguard [4]
Contains an extensive set of features for static analysis of Android applications. The functionality most relevant to integrity verification of applications is detailed below.

Aprvillé describes a method hiding technique that works against Androguard ([14], see the description under baksmali for details).

DAD A decompiler from Dalvik bytecode (classes.dex) to Java source. Susceptible to obfuscation as per most Java decompilers [60]. Androguard also supports two external decompilation options: DED and dex2jar + Jad.

Androdiff / elsim Comparison of new, removed or similar methods between two classes.dex files. Does not compare resources, assets, the manifest, or other files in the APK.

Androlyze Interactive reverse engineering tool, extends the iPython shell with functionality for inspecting APK files.

Androaxml Converts Android’s binary XML format back into standard XML.

APKtool [11]
Performs common APK reversing activities including disassembly/reassembly using baksmali/smali, XML and ARSC resource conversion to and from normal XML files, and conversion of 9-patch image files. APKtool automates the process of unpacking and repackaging an application using aapt.

apktools [12]
Also called the APK Resource Toolkit, a Ruby library for extracting XML resources from APKs. Can be used to return the plain XML version of a specified XML file, and provides an interface for parsing the keys and values of the resources.arsc file, with support for configuration-dependent values.

Axml Printer 2 [16]
Converts Android’s binary XML format into human readable XML. Development ceased in 2008. Tools such as Androguard’s androaxml were based on this implementation [27].

baksmali [82]
A disassembler for the dex format. Its output format is ‘smali’, which is based on the Jasmin assembly language for Java. The most popular Android disassembler, in part because of its companion tool smali, which allows for assembly of the smali representation back into a classes.dex file.

It uses a recursive traversal of the bytecode (following the addresses in jump opcodes) and is capable of disassembling most .dex files. Some obfuscation techniques prevent successful disassembly with baksmali [86; 15; 92].
Axelle Apvrille demonstrated [14] that it is possible to hide methods in classes such that they don’t appear in the output from `baksmai`, but can still be invoked from within the application. This is of much greater significance for integrity verification of applications that an inability to disassemble obfuscated code. The technique still works on the current version of `baksmai` (2.0.3).

Dare [23]

Retargets Dalvik bytecode (`.dex`) to Java bytecode. The successor tool to DED (see below), by the same authors. More reliable than DED – in their study the authors retargetted the top 1100 free applications on the Play Store with all of the classes in approximately 99% of applications being successfully retargeted (approximately 99.99% of all of the encountered classes) [67, p. 9]. The authors compare the retargeting success rate of Dare to `dex2jar`, with `dex2jar` retargeting approximately 60% of applications successfully [67, p. 10]. Their report does not include information on the decompilation success rate, though the tool has an option to decompile the retargetted code using soot.

DED [25]

Retargets Dalvik bytecode (.dex) to Java bytecode. Popular and simple to use, though no longer maintained as it was succeeded by the Dare project. DED initially retargets the classes.dex file to Java bytecode (using a linear sweep algorithm), then Soot (see below) can be used to optimise and decompile the class files to Java source. The authors used DED to perform a study of the security of the most popular 1100 free Android applications at the time [39]; DED with Soot generated Java source for 95% of the classes encountered [66]. In the subsequent Dare report [67], the authors acknowledged that ‘about half’ of the applications studied contained a class that could not be decompiled.

dedexer [26]

A disassembler that produces a Jasmin-like output (not smali).

As per `dexdump`, it uses a linear sweep algorithm and is susceptible to the same simple obfuscation techniques.

DEX Studio [29]

A GUI tool that supports resource analysis and bytecode disassembly, and a comparison feature. The comparison functionality compares resources (identical, present in only one, differing) and method signatures (name, parameters, etc).

At the time of writing, neither binaries nor source are available for download or purchase, and no information is available as to the licensing of the tool. Included here because of the reported comparison feature, which is directly applicable to this project.

dex2jar [30]

Retargets Dalvik instructions into Java bytecode, presented to the user as converting an APK file into a Java Archive (JAR). The conversion is only of
the classes.dex file within the APK – the manifest, other resources and assets are not retained in the JAR. Aprville describes a method hiding technique that works against dex2jar ([14], see the description under baksmali for details). A very popular tool in the community.

dexdump
A disassembler that is part of the Android platform SDK. Uses a linear sweep algorithm that assumes opcodes will follow sequentially, which is not necessarily the case, especially for obfuscated code.

Doesn’t support all of the features of the Dalvik bytecode such as labels and debugging information.

Dexpler [31]
Dexpler can convert Dalvik bytecode into Jimple [17] (Soot’s intermediate representation, see below), using ddedexer to initially disassemble the classes.dex file. Once in Jimple form, Soot with Dava can be used to analyse and decompile the application (note that the Dexpler developers don’t describe using dava with Dexpler in their report).

Dexter [32]
A free online service for analysing user uploaded APK files, providing high level information such as permissions, activities and intents; APK contents browsing; class disassembly and dependency analysis, and string enumeration.

The developers identified diff functionality as future work for the project [64, slide 24], but this is yet to be implemented.

iceditor [51]
A tool with no documentation, binaries or build instructions, but mentioned here as the source includes an APK comparison algorithm. The APK files are unzipped and baksmali is used to disassemble the source, then all of the files are checked and non-identical text files are compared using a Java diff library.

IDA [52]
IDA is a commercial disassembler and debugger for a wide range of architectures, including the Dalvik VM. Aprville describes a method hiding technique that works against IDA ([14], see the description under baksmali for details).

jadx [54]
A new .dex to Java decompiler, that unlike most other decompilers does not first retarget to Java bytecode.

JEB [57]
A commercial tool that supports interactive resource extraction, disassembly to Smali or ‘simplified Dalvik assembly’ [56] and decompilation to Java source using a proprietary decompiler.
The literature has limited coverage of JEB's performance. Apvrille reports her method hiding technique works against JEB ([14], see the description under baksmani for details), and Bremer indicates in [19] that JEB is more capable than dex2jar and dexdump. The JEB website [57] compares JEB's decompilation results with JAD and JD-GUI, but does not specify what bytecode retargeter was used – these two tools are Java decompilers, not Dalvik (see below for details).

Kivlad [59]

A Dalvik decompiler that goes directly from a .dex file to Java source (presented in HTML). Released by Matasano as an alpha release in 2011 and not updated since – reported as not functional for real world applications [76].

radare2 [71]

A general purpose reverse engineering framework, providing an interactive editing and inspecting mode plus a scriptable API. Radare2 includes a plugin that adds support for the Dalvik instruction format.

The radare2 tool performs (interactive) disassembly of Dalvik bytecode. The radiff2 tool performs binary difference analysis, but does not offer difference analysis of disassembled code.

undx [87]

Translates Dalvik bytecode to Java bytecode. Uses dexdump to do the majority of the classes.dex parsing [78]. No longer maintained and no downloads or source are present on the site. Reported to be less capable than dex2jar [18].

3.3.2 Integrated Tools

These tools primarily integrate the functionality of one or more of the aforementioned tools, though some provide additional functional as well.

AndroChef [3]

A commercial IDE for Windows which, whilst not documented as such, appears to wrap dex2jar and the FernFlower Java decompiler.

APK Multi-Tool [8]

Formerly known as APK Manager, a Windows batch script and Linux shell script that automates many of the higher-level tasks achieved using smali/baksmani, APKtool and the Android SDK tools.

APKtool

Described in Section 3.3.1, repeated here as it integrates the (dis)assembly and packaging functionality of (baks)mani and aapt.
### 3.3. TOOLS

<table>
<thead>
<tr>
<th>Tool</th>
<th>Licence</th>
<th>Updated</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Androguard</td>
<td>Apache 2.0</td>
<td>2014</td>
<td></td>
</tr>
<tr>
<td>ApkAnalyzer</td>
<td>Apache 2.0</td>
<td>2012</td>
<td></td>
</tr>
<tr>
<td>apkinspector</td>
<td>GPLv2</td>
<td>2012</td>
<td></td>
</tr>
<tr>
<td>Baksmali</td>
<td>BSD 3-Clause</td>
<td>2014</td>
<td></td>
</tr>
<tr>
<td>dedexer</td>
<td>Public domain</td>
<td>2013</td>
<td></td>
</tr>
<tr>
<td>DEX Studio</td>
<td>Unknown(^1)</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>dexdump</td>
<td>Apache 2.0</td>
<td>2014</td>
<td></td>
</tr>
<tr>
<td>dexter</td>
<td>Commercial(^2)</td>
<td>Unknown</td>
<td></td>
</tr>
<tr>
<td>IDA</td>
<td>Commercial</td>
<td>2014</td>
<td></td>
</tr>
<tr>
<td>JEB</td>
<td>Commercial</td>
<td>2014</td>
<td></td>
</tr>
<tr>
<td>radare2</td>
<td>GPLv3, LGPLv3</td>
<td>2013</td>
<td></td>
</tr>
</tbody>
</table>

\(^1\) Not currently available  
\(^2\) Free web service

Table 3.1: Dalvik Disassemblers

<table>
<thead>
<tr>
<th>Tool</th>
<th>Licence</th>
<th>Updated</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dare</td>
<td>GPLv2</td>
<td>2012</td>
<td>Java bytecode</td>
</tr>
<tr>
<td>ded</td>
<td>Closed</td>
<td>2011(^1)</td>
<td>Java bytecode</td>
</tr>
<tr>
<td>dex2jar</td>
<td>Apache 2.0</td>
<td>2012</td>
<td>Java bytecode</td>
</tr>
<tr>
<td>Dexpler(^2)</td>
<td>LGPLv2.1</td>
<td>2013</td>
<td>Jimple</td>
</tr>
<tr>
<td>undx</td>
<td>GPLv3</td>
<td>2013</td>
<td>Java bytecode</td>
</tr>
</tbody>
</table>

\(^1\) Estimated  
\(^2\) As part of the soot framework

Table 3.2: Dalvik Bytecode Retargeting Tools

Apk2Java [9]

A simple Windows script that wraps up `dex2jar`, `jad`, `apktool` and `aapt`.

ApkAnalyzer [7]

A GUI from the Sony Xperia team integrating smali/baksmali, `APKtool` and `AXML Printer 2`, adding additional functionality aimed at developers to warn of potential resource issues.

APKinsector [10]

A GUI integrating the output of a number of other tools. Incorporates disassembly using `baksmali`, Java decompilation using `APKtool` and `jad`, and a call flow graph analysis.

Virtuous Ten Studio [90]

A commercial IDE that simplifies the work-flow of a number of the tools described here, including `APKtool`, `baksmali` and `dex2jar`. 
CHAPTER 3. REVERSE ENGINEERING ANDROID APPLICATIONS

<table>
<thead>
<tr>
<th>Tool</th>
<th>Licence</th>
<th>Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Androguard (DAD)</td>
<td>Apache 2.0</td>
<td>2014</td>
</tr>
<tr>
<td>Dexter</td>
<td>Commercial¹</td>
<td>Unknown</td>
</tr>
<tr>
<td>jadx</td>
<td>Apache 2.0</td>
<td>2014</td>
</tr>
<tr>
<td>JEB</td>
<td>Commercial</td>
<td>2014</td>
</tr>
<tr>
<td>kivlad</td>
<td>Not specified</td>
<td>2011</td>
</tr>
</tbody>
</table>

¹ Free web service

Table 3.3: Dalvik Decompilers

<table>
<thead>
<tr>
<th>Tool</th>
<th>Licence</th>
<th>Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>aapt</td>
<td>Apache 2.0</td>
<td>2012</td>
</tr>
<tr>
<td>Androguard</td>
<td>Apache 2.0</td>
<td>2014</td>
</tr>
<tr>
<td>APKtool</td>
<td>Apache 2.0</td>
<td>2014</td>
</tr>
<tr>
<td>apktools</td>
<td>MIT</td>
<td>2013</td>
</tr>
<tr>
<td>Axml Printer 2</td>
<td>Apache 2.0</td>
<td>2008</td>
</tr>
<tr>
<td>dedexer</td>
<td>Public domain</td>
<td>2013</td>
</tr>
<tr>
<td>DEX Studio</td>
<td>Unknown¹</td>
<td>2014</td>
</tr>
<tr>
<td>Dexter</td>
<td>Commercial²</td>
<td>Unknown</td>
</tr>
<tr>
<td>JEB</td>
<td>Commercial</td>
<td>2014</td>
</tr>
</tbody>
</table>

¹ Not currently available
² Free web service

Table 3.4: APK Resource Extraction Tools

3.3.3 Linux Distributions and VMs

Due to the large number of tools that can be useful when performing security testing or analysis of Android applications, a number of Linux distributions have been produced that come with many of these tools pre-installed.

Android Malware Analysis Toolkit (AMAT) [5]

A single version of this VM was released in 2013, there is minimal published information on it and it is not maintained.

Android Tamer [6]

As the name suggests, an Android-specific distribution, including several reversing tools: Androguard, APK Analyzer, AXML Printer 2, jad, jdk-gui, smal-i/baksmali and dex2jar. Last updated October 2013.

ARE VM: Android Reverse Engineering Virtual Machine [89]

A VM that contains several Free Open Source Software (FOSS) Android reversing tools. Last updated in February 2012.
3.3. TOOLS

MobiSec [65]

The MobiSec Live Environment Mobile Testing is a security testing Linux distribution in the tradition of Kali and Backtrack but with a focus on mobile devices. Can be installed, live booted or run as a VM. Includes JD as well as some Android specific tools: the Android SDK, APKtool and Dex2Jar.

REMnux [72]

A general purpose reverse engineering Linux distro based on Ubuntu. Contains a wide range of reversing tools, including two that have specific utility in reversing Dalvik applications: ndare2 and JD. Currently maintained.

Santoku Linux [77]

A Linux distribution by via Forensics focused on mobile security. Includes several Android reversing tools. Can be run as a VM or installed as a normal Linux OS. Last updated in May 2013.

3.3.4 Native Code Reversing Tools

Where Android applications use native code via the Java Native Interface (JNI), traditional reverse engineering tools for the platform can be used (typically ARM, but there are some x86 Android devices).

This section will not detail these, other than to note that two of the previously mentioned tools, IDA Pro and ndare2, were originally designed for and are quite capable of such reverse engineering tasks.

3.3.5 Java Decompilers

In this section we consider Java tools that can be used once the Dalvik bytecode has been re-targetted to Java bytecode.

Unmaintained Decompilers

There are a large number of unmaintained tools from the early 2000s or 1990s, that don’t support newer versions of Java. Most notably of these is Jad, last updated in 2001, the most popular of the older decompilers, and still frequently referenced in Android reverse engineering guides. See [94] for a comprehensive list of old decompilers.

Modern Decompilers

CFR [20] The most actively developed of the tools listed here. Described by the author of Prowyn as ‘well on its way to becoming the premier Java Decompiler’ [70].

JD and JD-GUI [55] A freeware but not open source decompiler. JD is one of the two most frequently referenced decompilers in the Android reversing community alongside JAD.
### Table 3.5: Java Decompilers

<table>
<thead>
<tr>
<th>Tool</th>
<th>Licence</th>
<th>Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFR</td>
<td>MIT(^1)</td>
<td>2014</td>
</tr>
<tr>
<td>Jad</td>
<td>Closed</td>
<td>2001</td>
</tr>
<tr>
<td>JD / JD-GUI</td>
<td>Closed</td>
<td>2013</td>
</tr>
<tr>
<td>Krakatau</td>
<td>GPLv3</td>
<td>2014</td>
</tr>
<tr>
<td>Procyon</td>
<td>Apache 2.0</td>
<td>2014</td>
</tr>
<tr>
<td>Soot, Dava</td>
<td>LGPL v2.1</td>
<td>2014</td>
</tr>
</tbody>
</table>

\(^1\) The source has not yet been published

Krackatau [61]  Does not attempt to reconstruct source code that is identical to the original source, in exchange for being more robust at decompiling obfuscated bytecode. Also includes a disassembler, producing Jasmin-like output. A Python library with a command line interface.

Procyon [70]  Written partly due to JD-GUI being unable to decompile certain constructs; the author provides some example test cases where the output from Procyon compares favourably with JD-GUI. Luyten [62] is a GUI frontend project for Procyon.

Soot [84] and Dava [24]  Soot is a Java optimisation framework that uses an intermediate representation for Java bytecode called Jimple. Soot can analyse Jimple code, for example to produce a call flow graph. Dava, a project built on Soot, can be used to decompile the Jimple representation into Java source.
4. Verifying Android Application Integrity

The introduction described that for source code audit of open source Android applications to be meaningful, there needs to be a means to verify the integrity of an APK with respect to its source. None of the tools identified in the review automate this process, inhibiting routine application of this technique.

This chapter presents the design of AppIntegrity, an open source web application deployed as a public website for verifying the integrity of open source Android applications on the Play Store with respect to their source repositories. (Note that in accordance with RHUL project rules, the website will go live and the source for the applications will be published following marking, in December 2014.)

There are two different approaches that could be taken to check that an APK is really built from a given set of sources. The code in the APK could be decompiled, and the resources restored to their original form, in an attempt to directly compare the APK with the unmodified sources. This approach is limited by the fact that some information is lost in the compilation process, so decompilers are unable to reproduce the exact source: this will lead to there always being large amounts of minor differences that require analysis to confirm they are logically equivalent.

A key observation of this project is that there is a better approach: compile the sources using the same build tools as were used in publication and then compare the resultant APKs. The transformation from source to APK should result in a very similar APK to what was published. Some differences are inevitable: APKs are signed using a private key which is unlikely to be present in a public source code repository. So, the two APK files can be reverse engineered using a variety of tools and techniques, and the output of these tools compared.

To that end we first present APKDiff, a tool for analysing differences in similar APK files.

4.1 APKDiff

4.1.1 High Level Design

The build-reverse-compare pattern is the central design philosophy of APKDiff: compare the two files at a binary level and if they are not equal, apply one or more reverse engineering techniques to the object and compare the reversed forms. The process starts with the APKs (which are the same only in the trivial case), unzips and then compares each of the files within (manifest, classes.dex,
resources, etc). If the files differ APKDiff attempts to reverse them and output a comparison of the reversed forms.

We have seen that there is a plethora of existing FOSS tools for reverse engineering Android applications. APKDiff makes use of these existing tools, as well as FOSS Python libraries to quickly build an effective and robust means of comparing APK files.

There are three main components to APKDiff: the main module that provides a command line interface to the library; the apk module that defines the APK class which provides an abstraction of a single APK file, as well as Resource, Manifest, XML and ARSC classes to represent the various standard components of an APK which require reversing; and the diff module that defines the diff class which compares two APK objects. APKDiff can be used as either a command line application or a Python 2 or 3 library (through the apk and diff modules).

4.1.2 main module

The main module is a simple argparse based script, accepting user arguments for: the APK files to compare; where to unpack them to and whether to remove those directories after the comparison is complete; how verbose to be; and the location of some reversing tools. Having parsed the arguments, the tool simply outputs the textual representation of the diff.

4.1.3 apk module

The APK class provides properties and methods over the APK file that abstract the process of reversing the APK. This includes a path to the unzipped APK and its disassembled or decompiled classes.dex, instances of the ARSC and Manifest classes, and a dictionary containing all of the XML resources. An APK class is initialised with the path to an APK file. It unzips it using Python's zipfile standard library. The baksmai disassembler was chosen due to its ubiquitous usage within the community: an Android reverse engineer is more likely to be familiar with smali syntax than any other.

For decompilation, bytecode retargeting followed by use of a Java decompiler was favoured over direct decompilation from Dalvik bytecode due to the maturity of the available tools. As tools such as jadx and DAD improve, direct decompilation may offer a better alternative in future. For bytecode retargeting, Dare was selected due to its favourable comparison with dex2jar (see Section 3.3.1) and dexpler in informal testing as part of this project. The other retargeters, undx and ded, were not considered due to their lack of availability. Of the modern decompilers identified in Chapter 3, those under an open source licence were considered. The Krakatau decompiler was trialled as it presents a Python API, but it was found to not complete the decompilation process on the applications under test (others report success so this is likely due to user error). The CFR decompiler was trialled and worked well for all of the applications under test; APKDiff uses CFR in the current release, but is designed to easily support additional decompilers.

The Resource abstract class is implemented by the Manifest and ARSC classes. It provides an interface to the raw data of the AndroidManifest.xml and resources.arsc files, as well as their human-readable rendering provided by
4.1. APKDIFF

The tool was chosen for stability: as a part of the Android SDK, it is likely to always be current with any changes to the format of these files.

The XML class is instantiated for each XML file within the res directory of the APK. As per the Resource class, it provides an interface to the raw data (for simple comparison of files). It also provides access to the XML form of the files, using Androguard's AXMLPrinter to convert the binary format into XML and the defusedxml Python library to parse the XML. Androguard was chosen for its convenience of integration (it is a Python library) and during testing no problems were encountered in parsing XML files.

APKDiff is a Python 2.7 and Python 3.4+ compatible application. If run under Python 3 it gracefully degrades and Python 2-only libraries such as Androguard are not used (but the dependent functionality is also not available).

4.1.4 diff modules

The primary external interface for performing comparisons of APKs is the Diff class in the diff module. The class is instantiated with two APK objects (a left and a right) and optionally a function for performing textual comparisons, to allow the caller to control the output format. The Diff class has a __str__ method to provide an overall textual representation of the differences between the two APKs (when an object is converted to a string in Python, the __str__ method is used to provide the representation). The default textual comparison function is difflib's unified_diff from the Python standard library, chosen as the unified diff is a common format for expressing diffs and the functionality is part of the standard library.

The Diff class doesn't contain any logic for performing comparisons, rather it exposes the comparison functionality of other classes and gathers the results from them. In addition to the diff module holding the Diff class, there are modules for each of the components of an APK: diff_classes, diff_manifest, diff_resources, and diff_files for all other files within the APK. These modules all contain a class or set of classes for performing the comparison.

The FilesDiff class implements a recursive directory comparison, returning a dictionary of files that either only appear in the left or right directory, or appear in both but differ. Where files differ, a simple heuristic is used to determine whether the file is a binary or text file\(^1\) and text files are compared using the specified text comparison function. This class is used to provide a comparison of all files that aren't covered in the other classes, but is also reused in the other comparisons.

The classes.dex file containing the Dalvik bytecode for the application is initially compared at a binary level. If they differ, the APK object's baks-mali method is invoked to disassemble the bytecode. This creates an assembly listing with one file per class; an instance of the FilesDiff class is created to compare these files. In a similar manner, the APKs are also decompiled and compared. The comparisons are exposed through the properties assembly_diff and java_diff. The ClassesDiff class exposes the diff results as a FilesDiff object.

\(^1\)Using the Python library binaryornot v0.3.0.
The ManifestDiff class performs a binary comparison of the compiled manifests, and if they differ uses the APK object's aapt dump of the manifest for a textual comparison.

Resources are in two forms: the values and ID mapping in resources.arsc, and the resources in the res directory, some of which will be binary forms of the original XML resources. The arsc files are compared for equality, if they differ then the APK object uses aapt to dump the file in text form, and the output is compared. The files in the res directory are compared using the FilesDiff class, having first converted all of the binary XML files into standard XML as described in Section 4.1.3.

4.1.5 Cached property design pattern

Many of the operations performed by APKDiff are computationally expensive: disassembly, decompilation and performing a recursive diff over large numbers of files are particularly costly. As a diff is expected to be performed between two unchanging APK files, once an operation has been performed the results can be cached. Also, to avoid performing unnecessary operations whilst still providing a clean API, these operations are performed only when required – typically the class can be instantiated very quickly and the expensive operations are only carried out if a corresponding property that depends on them is accessed.

This is implemented within APKDiff using the @cached_property decorator pattern: a decorator is defined which extends the built in python @property decorator to check for the existence of a cached result, and if it doesn't find it performs the operation and caches the result. This example is from the FilesDiff class; instead of comparing the file tree each time the result is requested, or computing it when the class is instantiated, it is calculated and cached like so:

```python
@cached_property
def diff(self):
    result = # ... expensive directory traversal ...
    return result
```

4.2 AppIntegrity

The AppIntegrity website presents to the visitor a comparison of open source Android applications that are published on the Play Store with versions of the applications that have been built from the published source.

4.2.1 High Level Design

AppIntegrity uses APKDiff to compare the published source of an application against the APK published to the Google Play Store. For each application, AppIntegrity obtains the APK from the Play Store, obtains the source that should correspond to that version of the APK, builds that source and then compares the published and built APKs. The results are stored in a database and presented to the user through a website built on the Django framework\(^2\).

\(^2\)Django v1.7, https://www.djangoproject.com
4.2. APPINTEGRITY

Django was chosen as a framework due to its maturity and the ease with which APKDiff could be integrated – Django is written in Python and Django web applications can easily use other Python libraries.

4.2.2 Django Website Architecture

The Django framework divides applications into models, views and templates. AppIntegrity’s model does most of the heavy lifting, with the Version class implementing methods to download the Play Store APK, do a build, compare the APKs and store the results.

The model is illustrated by the Python class diagram in Figure 4.1, which Django transparently maps onto a relational database. Attributes with brackets such as `dir()` are implemented as properties in the model, so are calculated not stored directly in the database. For further information on custom Managers or the relationship between the classes and the database, refer to the Django documentation[35]. For brevity, types are omitted from the model diagram.

The view simply extracts the appropriate information from the model for presentation by the templates. See Section 4.2.5 for more information about the presentation of the data by the templates.

4.2.3 Play Store Automation

The Play Store does not have an official API, however a number of unofficial open source APIs have been developed. This project uses Emilien Girault’s `googleplay-api` as it provides a Python API that enables downloading of APKs from the store (the only required feature for this application) and has a liberal licence.

To access the Play Store a Google account is required which is registered with the Play Store. Registration usually occurs when a user signs in with their Google account to an Android phone. The Play Store will only present versions of applications to a user that their device can support, and some applications are restricted to particular carrier/country combinations. The `Android Checkin` tool was used to register an account with a phone type of Galaxy Nexus/Android 4.3. The country and carrier are specified in the API calls to the store and are currently fixed.

Users identify versions of applications by their name and user-visible version string, for example RedPhone 0.9.6. However this name is not guaranteed to uniquely identify the application, and the version string is not guaranteed to uniquely identify the version of the application. The Play Store API uses package names (a Java-style package name) and the versionCode attribute (an integer that the Play Store enforces is larger than the versionCode of any previous version of the application). For the RedPhone example we have org.thoughtcrime.redphone.versionCode 28. AppIntegrity makes use of the package name and versionCode for identifiers, but also presents to the user the familiar application name and version string.

---

3 `googleplay-api` commit c463cb e589, http://github.com/egirault/googleplay-api
4 `Android Checkin` commit 6f8b968922, http://github.com/nviennot/android-checkin
Figure 4.1: AppIntegrity model diagram
4.2. APPINTEGRITY

```python
def get_commit(version_name, version_code):
    return 'tags/v' + version_name
```

Figure 4.2: Example get_commit function for the TextSecure repository

4.2.4 Builds

Different applications are built in different ways and with different dependencies. For AppIntegrity to scale and be sustainable, an important design goal was to minimise the manual effort involved in the build process. Two industry standard tools are used to assist with this: Vagrant\(^5\), a virtual development environment manager and Puppet\(^6\), a configuration manager.

Each application version is built in its own virtual machine. The virtual machines are managed using Vagrant, which uses the configuration defined in a ‘Vagrantfile’ to bring up a base ‘box’ and configures it with any dependencies of the build using Puppet. A minimal Android build base box was produced, based on Ubuntu 14.04 with some additional packages: Puppet and various Puppet modules; a Java Development Kit (JDK); the Android SDK manager; and git. Whilst the base box isn’t necessary, it does speed up builds by avoiding the need to install this common subset of packages each time a build is performed.

The configuration elements of the build system are summarised in Figure 4.3 using a UML-like diagram indicating the purpose and specificity of the different configuration files. A single common Vagrantfile is used across all applications and versions which specifies the base box, the virtual environment provider (see Deployment in Section 4.2.7 for details of providers), the provisioning method (i.e. Puppet and the location of the provisioning script) and the virtual machine name. A Python configuration file is defined for each application that specifies the source code repository location and provides a function that returns the matching commit within the repository for a provided versionName and versionCode; see Figure 4.2 for a simple example where the repository consistently uses tags to label releases. The per-version configuration consists of a build script and a Puppet manifest. The manifest defines any additional dependencies for the build, which would include the required Android SDK version at a minimum. The build script downloads the source, runs the build and outputs the APK into a common location.

As the service identifies and downloads new application versions, the most recent configuration is copied to create a new build environment for the new version. Of course this will not always work: from time to time build dependencies and methods will change. For example during development of AppIntegrity the RedPhone project switched from ant to gradle for its build system. When a non-backwards-compatible build change such as this occurs, the build and/or provisioning script for the version requires manual modification. The output from the build script is logged to the database, allowing diagnosis of any build issues.

\(^5\)Vagrant v1.6, https://www.vagrantup.com
\(^6\)Puppet v3.6, https://www.puppetlabs.com
CHAPTER 4. VERIFYING ANDROID APPLICATION INTEGRITY

Figure 4.3: Build environment configuration files

### 4.2.5 Appearance

The website uses the Bootstrap framework\(^7\) for styling and responsive behaviour across devices. The application listing is presented in Figure 4.4, showing the layout for a narrow screen. Application icons are hosted on the Play Store, the URL is automatically retrieved when an application is first added to AppIntegrity.

The report page for a particular application version identifies which APK was downloaded from the store (giving both the user friendly versionName and the precise versionCode) as well as the source control reference that the reference APK was built from (typically a git commit hash). It then shows five collapsible elements, one for each of: Manifest; Classes; resources.arsc; the res directory; and all other files. Where these elements are binary identical, there is nothing to expand, the element is green, and the title states the element is identical. If they differ, the title includes a summary of the magnitude of the differences and the element expands when clicked on. The differences are presented in the typical manner: present only in the source or store versions; different binary files; or a unified diff for text files. There is a final collapsible element containing the per-version build configuration and the output from the build process.

To create the unified diff in a form suitable for the website, the diff function provided to APKDiff is from the \(^8\)library. This directly outputs unified diffs in HTML format for use with the ghdiff CSS rules. ghdiff is based on the GitHub diff style, an easy to use presentation of diffs.

As the classes are both disassembled and decompiled, the Classes element contains sub-elements, one for the assembly comparison and one for the Java comparison.

A sample report page is presented in Figure 4.5, showing elements of the report for ChatSecure. The analysis for this report is given in Section 5.2.

### 4.2.6 Scheduling

Interactions with the Play Store and building and comparing of applications are both long tasks which can’t be run on the main thread of the Django

---

\(^7\)Bootstrap v3.2, http://getbootstrap.com/
\(^8\)ghdiff v0.4, https://github.com/kilink/ghdiff
application without rendering the site unresponsive. A Celery\textsuperscript{9} asynchronous task queue is used to handle these jobs, with Celery Beat scheduling periodic checks of the Play Store.

As the Play Store API is unofficial, there is no usage guidance from Google. To reduce the risk of being considered an abusive user of the API due to frequency of requests, a rate limiting layer was implemented on top of googleplay-api. Another Python decorator called \texttt{RateLimited} is defined, and applied to the function through which all API calls are passed. This rate is defined in the Django project settings file, and for AppIntegrity has been set to at most one query every two seconds. Whilst no rate-limiting issues have been encountered, occasionally requests do temporarily fail for an unknown reason. To work around this sporadic failure, the single API interface is also wrapped in a \texttt{Retry} decorator, causing the request to be repeated if it raises an exception on the first or second attempts.

\subsection*{4.2.7 Deployment}

AppIntegrity has been designed to be easily deployed into different environments. Vagrant is again used to manage the website environment. In a development setting, the default provider of the website environment is VirtualBox; in a production setting this could easily be changed for a Virtual Private Server (VPS) provider such as Amazon EC2 or DigitalOcean (Vagrant has explicit support for interacting with these providers amongst others). Build machines are managed using Vagrant as described in Section 4.2.4, in a production environment the provider could be any VPS provider and in a development setting the default VM provider is \texttt{lxc} – Linux containers. Linux Containers

\footnote{\texttt{Celery} v3.1, \url{http://www.celeryproject.org}. Backed by RabbitMQ v3.3 \url{http://www.rabbitmq.com}}
Chapter 4. Verifying Android Application Integrity

Figure 4.5: Report page for ChatSecure

Built using commit tags: 13.1.3 from https://github.com/guardianproject/ChatSecureAndroid.git

Manifest: identical

Classes
  Smali (2 files differ)
    /data/app/com.actionbashfork/RSStyleable.smali
    /data/app/com.actionbashfork/RSStyleable.smali

  Java (2 files differ)

ARSC Resource Mapping and Values: identical

Resources: identical

Other Files (2 files differ)
  META-INF: only in store
  assets/giberbot.properties: only in store

Build
4.2. **APPINTEGRITY**

are a light-weight virtualisation solution that re-uses the host machine's kernel. So even though the website environment is already within a VirtualBox guest, that guest can itself host and provision Linux containers to perform the builds.

This setup allows a developer to install Vagrant, clone the AppIntegrity sources, then after setting their play store credentials issue a single `vagrant up` command and have a fully functional local copy of the web application, including the build system.

In development, Django's built in web-server and database back-end are sufficient although slow in rendering some large report pages. In production, these will be switched for the common Django deployment configuration of Apache with mod_wsgi and PostgreSQL.

4.2.8 Admin

AppIntegrity has a web admin interface, utilising the Django framework's out-of-the-box admin feature. This contains user and group management, as well as providing the ability to add Applications and trigger rebuilds of existing versions.

4.2.9 Community and Contribution

Whilst much effort has been put into minimising the manual overhead of maintaining AppIntegrity by automating the download, build and comparison processes, it still takes some effort to add applications and support changes to the build process. To allow AppIntegrity to scale, it must be possible for the wider security and Android developer communities to contribute to the site. Ideally, developers of some open source applications will find it a useful quality and security check and some will be willing to maintain their own build configs. To allow for this, all of the application and version configurations are stored in a separate git repository. This repository will be shared on GitHub, with users encouraged to fork, deploy locally using the development environment described in Section 4.2.7, and submit pull requests with new app configurations in. The process is as simple as running `vagrant up` to launch a local copy of the website, creating the three config files for an app (config.py, build.sh and provision.pp), and submitting a pull request for the new configuration to be integrated into the central website.

In addition to the website which is the focus of this project, APKDiff and AppIntegrity are both open source projects released under the Apache 2.0 licence, open to improvement and adaptation with or without involvement of the original author.

As AppIntegrity is freely available, the web application could also be used by vendors of proprietary Android applications. In this scenario the vendor host their own instance of AppIntegrity, configuring it with access to the source code for their published applications. The service then provides a check for the vendor that the integrity of their published applications has not been compromised, through exposure of store credentials for example.
5. Analysis of Applications on the Play Store

This chapter analyses the results from the AppIntegrity service for four popular Android secure communication applications, plus a further application by the Guardian Project that is advertised as having a reproducible build. The analysis of each application has the following structure:

- The background to the application is given;
- The versions analysed are listed and the project’s release practices are reviewed;
- The experience of setting up the build is described;
- The differences between the Play Store version of the app and the version built from source are analysed;
- A summary of the analysis is given.

5.1 RedPhone

RedPhone by Open Whisper Systems is a secure call app that encrypts phone conversations end-to-end. It is released under the GNU GPLv3 licence and has between 100k and 500k installs from the Play Store. In the description on the store, they advertise that RedPhone is ‘Free and Open Source, enabling anyone to verify its security by auditing the code.’ [45].

5.1.1 Versions

The latest version of RedPhone was released in October 2013, has a version string of 0.9.6, and is version number 28 on the Play Store. This version has been analysed with AppIntegrity.

The RedPhone source is published on GitHub and uses GitHub’s releases feature and git tags to identify released versions of the code. The latest version has not been tagged though, so the appropriate commit (16e82f1...) was determined by manual inspection, corresponding to the commit where the version in the manifest was changed to match the version in the released APK and Play Store metadata.

5.1.2 Build

The build process is documented in the de-facto standard BUILDING file. Two undocumented changes to the build process were required. The ActionBarSherlock library is referenced by a commit to the ActionBarSherlock repository,
however the version of the Android Support Library that is present in this commit had to be replaced with the version that is present in the RedPhone repository. Additionally the default.properties file in this commit appeared to be broken, and required the following two lines to be appended to it:

```
split.density = false
target=android-16
```

### 5.1.3 Comparison

The manifest is identical. The classes.dex file differs, however the disassembled and decompiled forms are identical: both the assembly output from baksmbali and the Java sources output by CFR and dare. The cause of the difference in the classes.dex files is not known, but one possible explanation is slightly different versions of the Java compiler being used.

The values defined in resources.arsc are identical, with one tiny exception: a floating point dialogue dimension differs in the least significant bit of the mantissa, as illustrated in Figure 5.1. The difference is so small, that `aapt`’s rendering of the value to 6 decimal places is identical. It is not clear why this difference might be present. The definition of the value is 80% in the source code for ActionBarSherlock – both these floating point representations can be considered to be 80%. Perhaps `aapt` is dependent on the underlying processor architecture’s floating point behaviour and this differed between the AppIntegrity system and the system used to package the application for the Play Store.

The other resource files are identical. The META-INF directory only appears in the published version; this is expected as no private key for signing was present during the from-source build.

### 5.1.4 Summary

Aside from what appears to be a one-off omission of a release tag and a couple of build configuration items that were not checked in, RedPhone has a highly reproducible build. The project should be commended for generally good configuration management practices and for releasing source code that very closely corresponds to the application published on the Play Store. Users have reasonable assurance that the results of any audit performed on this stable version of RedPhone’s source equally apply to the binary form of the application they have downloaded from the Play Store.
5.2 ChatSecure

ChatSecure by The Guardian Project is a secure messaging app that runs over existing chat services. It is released under the Apache 2.0 licence and has between 100k and 500k install from the Play Store [43]. The iOS version of ChatSecure was audited by OpenITP’s Peer Review Board [68], though the results of the audit have yet to be published.

5.2.1 Versions

The latest version of ChatSecure was released in January 2014, has a version string of 13.1.2 and is version number 28 on the Play Store. This version has been analysed with AppIntegrity.

ChatSecure source is published on GitHub and uses GitHub’s releases feature and git tags to identify the source code corresponding to released versions.

5.2.2 Build

The build process is documented and straightforward, with only the location of the Android SDK needing to be specified in the local.properties file. A git submodule URL needed modifying, but this fix was submitted to the maintainers and has subsequently been merged into the repository, so will not be required for the next release.

5.2.3 Comparison

The manifest is identical.

The classes.dex file differs, with the difference appearing in both the assembly listing and the decompiled Java source code. The difference is a constant used in styling the ActionBarSherlock spinner control. It’s not clear why this difference arises.

The resources.arsc files are identical, as are the other resources in the res folder.

The META-INF directory only appears in the store version, which is expected as no private key was available during the build from source. An asset is present in the store version of the application that isn’t in the built from source version, a file called gibberbot.properties (ChatSecure was previously called gibberbot). This file is explicitly excluded from the source repository by being listed in the "gitignore" file and was removed from the source tree shortly before this ignore was added. This file is only used within the code to specify a default locale for the application. It seems likely that this is an accidental inclusion, a byproduct of some other build activity or a leftover on the developer’s machine from before it was removed from the repository.

5.2.4 Summary

ChatSecure requires little effort to create a reproducible build. The only difference between the published APK and a version built from source are one benign extraneous file appearing in the store version and a small difference in the styling of one control. The project should be commended for generally good configuration management practices and for releasing source code
that very closely corresponds to the application published on the Play Store. Users have reasonable assurance that the results of any audit performed on this stable version of ChatSecure's source equally apply to the binary form of the application they have downloaded from the Play Store.

5.3 Telegram

Telegram by Telegram Messenger LLP is a secure messaging app. It is released under the GNU GPLv2 licence and has between 10M and 50M installs from the Play Store [46].

5.3.1 Versions

Telegram is undergoing active development, with several versions released during the development of AppIntegrity. Three versions have been analysed with AppIntegrity: version number 284 (version string 1.6.1), version number 288 (version string 1.6.1) and version number 307 (version string 1.7.0), the current version at the time of writing. Here we see the ability of a publisher to choose arbitrary values for the version string: the user facing version string 1.6.1 was used for at least two distinct versions of the application that were uploaded to the Play Store.

The source code for Telegram is released on GitHub, but the commits corresponding to released versions of code are not identified. A normally reliable method of identifying the commit corresponding to a release is to find the version of the manifest file (or of the build properties file that generates the manifest) for which the version string or number is the same as the released version. For Telegram this is not always possible: the commits in the repository frequently jump over released versions. It would appear that the source code is not available for several versions of Telegram published on the Play Store.

One of the issues raised on the GitHub project discusses repository management, where the author describes ‘github contains only major updates that will go to google play’ [81], with most work being done on a private repository. This practice was changed in March 2014 to start using the dev branch for all commits, yet version 284 which was released after March does not appear to have any corresponding source in any branch of the repository.

The remainder of this analysis is for Telegram versions 1.6.1 (288) and 1.7.0 (307). Version 1.6.1 (284) differs significantly from any published source and as we know these differences are due to the source not being available, they are not analysing further.

5.3.2 Build

The build process is not documented, but follows fairly standard Android build conventions. The location of the Android SDK must be specified and a dummy keystore needs to be created. This is in contrast to the other applications reviewed, which have explicit support for building without a signing key configured.
5.3. TELEGRAM

The manifest is identical in both versions.

The classes.dex file differs in both versions. In version 288, the assembly listings of 11 classes differs, but only 5 Java files differ after decompilation. In version 307 a similar difference is apparent: 12 classes differ at the assembly level, but only 6 at the Java level. The differences are the same across the two versions with the exception of the additional class that differs in 307. The differences that only appear in the small assembly and not the decompiled Java are consistently due to how the toString function is invoked; an example is given in Figure 5.2. In the version built from source, the invoke-interface opcode is called on the CharSequence class, in the store version the invoke-virtual opcode is called with the Object class. Both of these instructions decompile to the same Java code.

One of the differing classes is due to differing input sources. The file Build-Vars.java in the public git repository contains different information to the version used to compile the store APK, as this includes details such as identifying IDs for the Telegram server API. Presumably this is intended to allow the Telegram server maintainers to identify whether a request is established with the version of the application they have published or a version built from the source by someone else. It also defines bug reporting information; in the public version the application will not send bug reports to the original Telegram author, rather the person building the application is reasonably expected to fill in their own details.

All of the remaining differences do not appear to be due to differences in the input source. In the GroupCreateActivity classes, very minor differences are seen with how a style is referenced - in the from-source variant the style is imported and used directly, in the store version it is referenced via another object. In the TLRPC class, the store version has a repeated type cast. The FileLoader class contains subtly different implementations of the same iteration logic. In version 307, an unused variable is not declared in the store version of the extra varying class. It is likely that a different version of the JDK was used to compile the code to produce the published version and the reference version, resulting in slightly differing bytecode for the same source code. Specifying the version in the Gradle build.properties file would either fix the issue or eliminate this as a possible cause.

On first inspection, the resources.arsc file appears to be wildly different: the diff runs to roughly 5000 lines. All of these differences though are down to the index into the string table for different strings. The string IDs, names and values are all identical, it is just their internal position within the table that varies. This would be expected if at least one string was present in one version but not the other, as then all the offsets would differ. This is not the

![Figure 5.2: Excerpt from the AppIntegrity output for Telegram, showing different ways of invoking toString().](image-url)
case though, and this analysis presents no other likely alternative explanations. The other resources in the res directory are identical in both versions. The META-INF directory contains different contents in both versions, this is expected due to the other differences described above causing files to have different hashes, and the different private key used to sign the APK.

5.3.4 Summary

Telegram's configuration control practices could be improved in simple ways that would greatly increase confidence that published applications corresponded to published source: providing build instructions; tagging releases; and ensuring all published versions of the application are built from a checked in version of the source. Current practices have lead to unpublished source for some published versions and significant effort and uncertainty is involved in identifying the corresponding source for a published application version. There are some unexplained but benign differences in the exact binary form of the published APK and the version built by AppIntegrity; specifying the precise JDK version is suggested as a first step to resolving these differences.

5.4 TextSecure

TextSecure by Open Whisper Systems is a secure message app that encrypts messages end-to-end. It is released under the GNU GPLv3 licence and has between 100k and 500k installs from the Play Store. As with RedPhone, they advertise that 'TextSecure is Free and Open Source, enabling anyone to verify its security by auditing the code.' [47]. TextSecure was audited by iSEC Partners as a requirement of their funding from the Open Technology Fund [75], though the report is not public.

5.4.1 Versions

TextSecure is under active development, and during this project AppIntegrity picked up version numbers 71, 72, 73 and the current version 78 (version strings 2.0.8 to 2.1.6).

For brevity this analysis will focus on version numbers 71 and 78. The analyses for versions 72 and 73 are very similar to the analysis for version 78.

As for RedPhone, the TextSecure source is published on GitHub and uses GitHub's releases feature and git tags to identify released versions of the code. Not every version is tagged, for example tags for three versions are missing from the last eight, though the historic record seems to be more consistent with these omissions being almost the only ones. The two versions under consideration are tagged in the repository.

5.4.2 Build

The build process is documented and very straightforward: only the location of the Android SDK needs to be specified in the local configuration file.
5.4.3 Comparison

Version number 71 (2.0.8) of TextSecure is very nearly a reproducible build: the manifest is identical; classes.dex differs but the disassembled and decompiled forms are identical (as for RedPhone, the cause of this is not known); resources.arsc is identical; the other resource files are identical; and the only other difference is the expected presence of META-INF in the store version. Notably, the ‘other files’ category for TextSecure includes native shared objects, that are also built from source.

Version number 78 also has identical manifest, resources.arsc, res folder and the expected META-INF difference. However, the initial analysis for classes.dex was very different. In this version, 255 class files had different assembly listings, with 51 of these class files also differing after decompilation. The differences were non-functional differences. The assembly level differences that didn’t manifest at at Java level appeared to be solely due to use of different registers. Classes of decompiled Java source code differences included: variable declaration ordering; type specificity; variable reuse vs declaring new variables; conditional structure (e.g. \texttt{if (x) return; ... vs if (!x) ...; return}); and variable names.

As these changes were strongly suggestive of differences in either the compiler or the bytecode retargeter, the build was repeated with different versions of the Java JDK. The default version used was OpenJDK 7; after trialling Oracle’s JDK version 6, 7 and 8, Oracle JDK 8 was found to produce the least differences.

When compiled with Oracle JDK 8, there is no difference in the decompiled Java source, though 18 classes have differing disassembled forms. The differences in the smali code are the introduction of MemberClasses annotations in the store version of the APK that aren’t present in the reference version. A MemberClasses annotation is a type of system annotation defined in the Dalvik Executable Format documentation: ‘System annotations are used to represent various pieces of reflective information about classes (and methods and fields).’ [28] The annotations are all for anonymous inner classes (all named ‘$1’), but it is not clear why the difference appears – perhaps the precise JDK version used still differs.

5.4.4 Summary

TextSecure demonstrates good practices in build reproducibility and release management. Specifying the exact JDK version used to produce the published application (ideally within the Gradle build configuration) would ensure that builds can be easily reproduced by third parties.

5.5 Lil’ Debi: Debian Installer

This application allows a user of a rooted Android device to install a Debian system in parallel to the Android system on the their phone. The Guardian Project announced [85] in June 2014 that version 0.4.7 of Lil’ Debi was the project’s first deterministic Android build.
5.5.1 Versions

Version 0.4.7 is the latest version of the application and the only version that claims to have a deterministic build.

The source code is published on GitHub and follows the common practice of using GitHub’s releases feature and tagging releases by their version string.

5.5.2 Build

The build script that performs the deterministic build made a number of assumptions about the build system that don’t generally hold, but once the provisioning script produced a system that matched these assumptions the documented build instructions did work.

The ant build process exhibited buggy behaviour, hanging on certain steps of the build. The cause isn’t known, but retrying the process sufficient times did eventually lead to the successful completion of the build.

Whilst the documentation acknowledges that the output of the build is dependent on the JDK used, the project does not specify the exact version of OpenJDK 7 that was used for this release. The AppIntegrity build used version 7u51-2.4.6-1ubuntu4.

5.5.3 Comparison

The reference version of Lil’ Debi built by AppIntegrity is identical at a binary level to the store version, with the exception of the META-INF directory as expected.

5.5.4 Summary

Lil’ Debi is an exemplary Android application for build reproducibility and release management. The build documentation should identify the exact version of the JDK used for the release, as different versions of OpenJDK 7 may produce different bytecode for the same input. Ideally, the JDK version used should be recorded in the repository so the build could notify the user if they use a different version.

5.6 Summary

Overall, the applications analysed had good levels of build reproducibility. All of the differences observed between the published versions of APKs and source are believed to be due to minor build system differences or, in the case of Telegram, deliberate changes to differentiate applications built by the publisher from those built by others. Perhaps the easiest area of improvement across the applications would be to consistently identify the version of source code that corresponds to published versions of the application. In Telegram’s case, this extends to ensuring that the source code is always published for each published version of the application.

The Guardian Project have recently demonstrated with Lil’ Debi that it is possible to have a reproducible build for an Android application.
## 5.6. SUMMARY

Table 5.1 summarises the results of the analysis of the applications against each of the components of an APK. A ‘✓’ symbol indicates the component was identical. A ‘–’ indicates the component was almost identical. A ‘×’ indicates there were significant differences. The notes below the table provide an abbreviated description of any differences.

<table>
<thead>
<tr>
<th>Application</th>
<th>Manifest</th>
<th>Classes</th>
<th>resources.arsc</th>
<th>res/other</th>
</tr>
</thead>
<tbody>
<tr>
<td>RedPhone</td>
<td>✓</td>
<td>✓(^1)</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>ChatSecure</td>
<td>✓</td>
<td>–(^3)</td>
<td>✓</td>
<td>–(^4)</td>
</tr>
<tr>
<td>Telegram</td>
<td>✓</td>
<td>×(^5)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TextSecure (2.0.8)</td>
<td>✓</td>
<td>✓(^1)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TextSecure (2.1.0+)</td>
<td>✓</td>
<td>–(^7)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Lil’ Debi</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

1. classes.dex files differ; reversed forms are identical.
2. Float differs in least significant bit.
3. ActionBarSherlock styling constant difference.
4. gibberbot.properties file present in store APK.
5. Compiler suspected to have minor differences and BuildVars.java differs.
6. String indices differ; values are identical.
7. Annotations of some anonymous inner classes differ.

Table 5.1: Summary of build integrity
6. Conclusion

This chapter summarises the work done and its findings, revisits the objectives and identifies areas of potential future work.

6.1 Reverse Engineering for Android

There are a wide range of tools, mostly free and mostly open source, that can help with reverse engineering an Android application. The experience on this project shows that all of the common compilation and packaging steps involved in a non-obfuscated Android application can be reversed using a subset of the free open source tools.

6.2 Reproducible Builds and Binary Integrity

The analysis of four popular applications using AppIntegrity confirmed the hypothesis that Android builds are not reproducible by default. Even when two builds of the classes.dex file contain identical bytecode (as determined by disassembly), the files themselves can differ significantly at the binary level—an issue that requires further research. The resources.arsc file seems to be fragile, potentially dependent on processor architectures of the build system, but is possible to be exactly reproduced. All of the other components of an APK are easily reproducible, with the inevitable exception of the signature information. The Guardian Project’s Lil’ Debi application demonstrates that it is possible to have reproducible Android applications.

Verifying the binary integrity of Android applications is not a problem area that has received much attention. AppIntegrity contributes by providing a means of easily identifying what the differences are between an APK on the Play Store and a version built by an independent third party. This can be used both by developers seeking to make reproducible builds and by reviewers wishing to confirm the integrity of (non-reproducible) published applications.

Four high profile open source security applications were analysed and found to have a range of differences between the published applications and the reference versions built from the published source. All of the differences were benign, having little to no functional impact on the program. A prerequisite to determining how closely a published binary application corresponds to its source code is for the correct version of source to be identified. This requires that the project has good configuration management practices that are consistently applied, something that was not the case for all of the applications under review.
6.3 Objectives

This section reviews how the project objectives were met.

Objective 1: review This objective was met in Chapter 3. The techniques involved in reverse engineering an Android application were described in Section 3.2. The tools that implement these techniques for Android applications were then described in Section 3.3.

Objective 2: engineering The primary objective of the project, this was met through the development of the AppIntegrity website, underpinned by the APKDiff tool. The design of the tool and service were described in Chapter 4. Reproducible builds is a topic that is just beginning to gather pace, with high profile instances such as the TrueCrypt audit and the Debian projects aspirations for all binary packages to be reproducible. When deployed, the AppIntegrity website will help raise the profile of reproducible builds on Android, as well as helping developers see what areas of their builds are not currently reproducible.

AppIntegrity provides a continuous service that compares all versions of the configured applications that are published on the Play Store. This provides a mechanism for detecting malicious versions of an application on the store, which could be uploaded due to compromise of a developer's machine, for example.

Objective 3: case study This objective was met through an analysis of four popular open source security applications: TextSecure, ChatSecure, Telegram and RedPhone. In each case, differences between the store version and the reference version were found, but the analysis confirmed the differences were benign. Recommendations were made as to how to minimise the scale of the differences. The Lil' Debi application was also analysed and the claim by the Guardian Project that it has a reproducible build was corroborated.

6.4 Future Work

This work presents many new questions and the tools developed have potential for improvement in various ways. This section concludes the report by presenting areas of improvement for APKDiff and AppIntegrity, and discussing some of the work that would be beneficial beyond the scope of these tools. It is the author's intention that several of the improvements to the tools will be implemented prior to publication.

6.4.1 APKDiff

As we saw in the analysis, a large diff output does not necessarily mean a large semantic difference in the applications. An option to ignore differences in the precise layout of the string table in resources.arsc would help in some comparisons. Specifically where two value entries of type t=0x03 differ in their data value, but the actual referenced string is the same, this could be suppressed or summarised as 'string pool ordering differences'.
The service currently does nothing more than a binary comparison of native code shared objects. We saw with TextSecure and Lil' Debi that it is quite possible to have reproducible shared objects, however this is not always guaranteed to be the case. Disassembly (with a tool such as radare2) and decompilation of shared objects, such that they can be analysed in the same manner as Dalvik code, is a natural improvement.

6.4.2 AppIntegrity

It would be beneficial to extend the set of reverse engineering tools used in AppIntegrity, allowing the user to view comparisons from different sources. For example, the CFR decompiler may fail to decompile a particular class that Procyon succeeds on. More combinations could be added of the retargeters, decompilers (bytecode to Java and binary XML to XML) and disassemblers identified in Chapter 3.

The tool is currently of primary use to Android experts wishing to review published applications against their sources. However, especially for cases where there are non-trivial differences, it can be hard for non-experts to determine the significance of the results. An expert summary of the analysis would be valuable. There is scope for including code-review type functionality to allow manual analysis to be captured and presented inline on the website. Elements from open source code review tools such as Rietveld [74] and Gerrit [42] would be well suited to this task, as they are expressly designed for commenting on diffs.

The service is limited by the device type, country and carrier it uses to interact with the Play Store API. This prevents it from receiving and hence analysing the full range of applications available on the Play Store, in particular older versions of apps where the newer version doesn't support an older API level. Additional configurations should be added covering a wider range of API levels and regions.

The service is currently limited to the Google Play Store. There are other third party app stores such as the Amazon App Store which would benefit from the service.

There are security aspects with the build system that need consideration prior to deployment, as it provides an interface to allow repository owners to obtain arbitrary code execution within the virtual build machine. This problem isn't unique to AppIntegrity of course: anyone who checks out and builds a remote code base is subject to the same risk.

When performing some analyses, it would be helpful if the analyst could review the diff in a larger context, for example by downloading the files or expanding the context lines.

6.4.3 Wider Topics

This project has focused on analysing the discrepancies between published source and published binaries. The reasons for the observed differences in binaries produced using the various build systems from the same source warrants further investigation. With the unavoidable exception of the signing data, it is a reasonable goal for a build system to produce identical APK files when run on the same source. One specific example is aapt's floating point behaviour, which
was called into question in the analysis of RedPhone in Section 5.1.3. Another is the unexplained varying string indexes seen with Telegram in Section 5.3.3.

Following on from this, guidance for developers on how to make reproducible builds would be a valuable contribution to the community. This could cover source control (what to commit for different project types), the release process, and compiler-specific options or settings for ensuring consistent output. As of this writing, the steps the Guardian Project took to make Lil’ Debi’s build reproducible have not been documented.

The tools and techniques used in this project could be equally applied to analysing the differences between subsequent versions of closed source applications. This could be useful for example to reduce the effort required to re-audit a new version of an application that has already been audited.

Aprville’s method hiding technique [14] warrants further research and improvement in the reverse engineering tools which are susceptible to it. Currently, reviewers of source code must be particularly vigilant around reflection code, as it could conceivably be used to hide different behaviour in a binary that would not be revealed by the tools used by APKDiff.

The service only compares published binaries against published sources. Backdoors or other malicious behaviour can still be hidden in the source code or build scripts of the application. In a sense this service is ahead of its time: it doesn’t matter whether the code in the repository matches the app in the store, if no-one is auditing the code! Another viewpoint is that once it is easy to confirm the correspondence between source and published binary, code audits will have more value and will be carried out more frequently. Improving the culture of security audits of open source applications, perhaps supported by open collaborative code review tools, should be a priority for the security community. Whilst there have been some recent improvements in this area such as the creation of the Core Infrastructure Initiative [22], we still have a long way to go.
Bibliography

[1] *Output from Android Asset Packaging Tool (aapt) v0.2 usage help.*


[23] Dare website. URL: http://siis.cse.psu.edu/dare.


[54] jadx website. URL: https://github.com/skylot/jadx.


Kivlad website. URL: http://matasano.com/research/kivlad.


Krakatau website. URL: https://github.com/Storyyeller/Krakatau.

Luyten website. URL: https://github.com/deathmarine/Luyten.


MobiSec website. URL: http://mobisec.professionallyevil.com.


Procyon website. URL: https://bitbucket.org/mstrobel/procyon.

radare website. URL: http://www.radare.org.

REMnux Linux website. URL: http://zeltser.com/remnux.


Rietveld project website. URL: https://code.google.com/p/rietveld/.


[82] Smali website. URL: https://code.google.com/p/smali/.


A. aapt Dump Format

There is no documentation for the output format of the `aapt` tool’s dump command. The following information was determined by review of the source code of the Android frameworks project.

A value is represented as a line containing these elements:

```
resource <ID> <package>:<type>/<name> t=<dataType> d=<data> (s=<size> r=<res0>)
```

**ID** The numeric ID used to reference the resource, defined in R.java.

**package** The package that this resource is defined in and will be the same for all the resources in the package.

**type** The resource’s type, one of a number of strings defined elsewhere in the resource table.

**name** The resource’s name.

**data** The resource value, interpreted according to `dataType` (see below for types; the type-dependent interpretation rules are not described here).

**size** The size of the value struct (not of the value). Seems to always be 0x0008.

**res0** Always 0x00.

**dataType** One of the constants defined in the following excerpt from the source code:

```c
// Contains no data.
TYPE_NULL = 0x00,
// The 'data' holds a ResTable_ref, a reference to another resource
// table entry.
TYPE_REFERENCE = 0x01,
// The 'data' holds an attribute resource identifier.
TYPE_ATTRIBUTE = 0x02,
// The 'data' holds an index into the containing resource table's
// global value string pool.
TYPE_STRING = 0x03,
// The 'data' holds a single-precision floating point number.
TYPE_FLOAT = 0x04,
// The 'data' holds a complex number encoding a dimension value,
// such as "100in".
TYPE_DIMENSION = 0x05,
// The 'data' holds a complex number encoding a fraction of a
// container.
TYPE_FRACTION = 0x06,

// Beginning of integer flavors...
```
APPENDIX A. AAPT DUMP FORMAT

TYPE_FIRST_INT = 0x10,
// The 'data' is a raw integer value of the form n..n.
TYPE_INT_DEC = 0x10,
// The 'data' is a raw integer value of the form 0xn..n.
TYPE_INT_HEX = 0x11,
// The 'data' is either 0 or 1, for input "false" or "true" respectively.
TYPE_INT_BOOLEAN = 0x12,

// Beginning of color integer flavors...
TYPE_FIRST_COLOR_INT = 0x1c,

// The 'data' is a raw integer value of the form #aarrggbb.
TYPE_INT_COLOR_ARGB8 = 0x1c,
// The 'data' is a raw integer value of the form #rrggb.
TYPE_INT_COLOR_RGB8 = 0x1d,
// The 'data' is a raw integer value of the form #argb.
TYPE_INT_COLOR_ARGB4 = 0x1e,
// The 'data' is a raw integer value of the form #rgb.
TYPE_INT_COLOR_RGBA4 = 0x1f,

// ...end of integer flavors.
TYPE_LAST_COLOR_INT = 0x1f,

// ...end of integer flavors.
TYPE_LAST_INT = 0x1f