S²MAV

Static Slicing for Mobile Application Verification

Christoph Woergoetter

S²MAV

Static Slicing for Mobile Application Verification

Master's Thesis

at

Graz University of Technology

submitted by

Christoph Woergoetter

Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology
A-8010 Graz, Austria

05 Nov 2012

Advisors: Univ.-Prof. Dr. Roderick Bloem
Dipl.-Ing. Daniel Hein
Dr. Peter Teufl

S²MAV

Statisches Slicen zur Mobilen Applikationsverifikation

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

Christoph Woergoetter

Institut für Angewandte Informationsverarbeitung und Kommunikation (IAIK),
Technische Universität Graz
A-8010 Graz

5. November 2012

Diese Arbeit ist in englischer Sprache verfasst.

Begutachter: Univ.-Prof. Dr. Roderick Bloem
Dipl.-Ing. Daniel Hein
Dr. Peter Teufl

Abstract

Today's mobile phones are powerful and complex devices. These devices store sensitive data like business data, contact data, account information, and banking data. All of these data are worthy of protection. Unfortunately, modern phone operating systems have different security issues. These issues are composed of operating system issues, application issues, and implementation issues. Therefore, Android applications with sensitive information have to deal with these issues to minimize the number of security vulnerabilities.

This master's thesis provides a framework to efficiently analyse if an application properly implements security checks to consider Android's different security issues. The idea behind our approach is to use static slicing, which is a well-known analysis technique, to analyse Android's Dalvik instruction set. This technique is used to classify the quality of current Android application in terms of security relevant code. To evaluate the performance of our framework, we analysed 4969 arbitrary applications.

Kurzfassung

Heutige Handys sind leistungsstarke und komplexe Geräte. Diese Geräte speichern viele sensible Daten, wie Unternehmensdaten, Kontaktdaten, Benutzerinformation, und Bankinformationen. Alle diese Daten sind schützenswert. Weiters haben moderne Handybetriebssysteme viele verschiedene Sicherheitsprobleme. Diese Probleme setzen sich aus Problemen des Betriebssystems, der Anwendungsschicht, und der Implementierungen von Anwendungen zusammen. Eine Anwendung, mit sensiblen Daten, muss folglich all diese verschiedenen Probleme behandeln, um die Menge der Sicherheitsschwachstellen zu minimieren.

Diese Masterarbeit stellt ein Framwork zur Verfügung, um zu überprüfen ob Android Anwendungen Sicherheitsüberprüfungen verwenden und diese auch korrekt implementiert haben. Für dieses Ziel verwendet das Framework statisches Slicen, um die Dalvik Instruktionen von Android zu analysieren. Diese Methode wird verwendet, um die Qualität aktueller Android Anwendungen in Bezug auf Sicherheit zu überprüfen. Für die Evaluierung der Qualität unserer Lösung haben wir 4969 zufällige Anwendungen analysiert.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Place	Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Ort	Datum	Unterschrift

Contents

List of Figures

List of Tables

Listings

Acknowledgements

I would like to thank my colleagues at the institute of applied information processing and communication (IAIK) who have provided invaluable help and feedback during the course of my work. I especially wish to thank my advisors Peter Teufl and Daniel Hein for their immediate attention to my questions and endless hours of toil in correcting draft versions of this thesis, giving feedback and showing new ways and impressions during the development of the framework.

Christoph Woergoetter
Graz, Austria, October 2013

Credits

I would like to thank the following individuals and organisations for permission to use their material:

- The thesis was written using Keith Andrews' skeleton thesis [Andrews, 2012].
- Figure 2.1 was extracted from *Android System Architecture* [2013] under the terms of the Cc-by-sa-3.0-migrated, GFDL. Copyright © by the GNU Free Documentation License.

Creative Commons Notice

[1] http://creativecommons.org/licenses/by-sa/3.0/legalcode

Chapter 1

Introduction

" Begin at the beginning and go on till you come to the end; then stop. "

[Lewis Carroll, Alice in Wonderland]

Currently, the most popular smartphone operating system is Android [Spreitzenbarth, 2013]. However, the increased exposure of Android leads to an increased risk of threats to Android phones through poorly, or maliciously designed applications. In this context, Zhou and Jiang [2012] describe Android as the top mobile malware platform.

While Android defines a base set of permissions to protect phone resources and core applications, it does not define what a secure phone is, relying on the applications to act together securely [Enck, Ongtang, and McDaniel, 2008]. Nevertheless, for the protection of sensitive data, Android provides different security mechanisms [Shabtai, Fledel, et al., 2010].

- Operating system mechanisms

  POSIX users to encapsulate application data on file-system level.

  Deactivated root user, to avoid unauthorized system modifications.

- Application's communication-specific mechanisms

  Application permissions to limit application abilities to perform malicious behaviour.

  Component encapsulation, to prevent that one application disturbs another application.

  Own Dalvik virtual machine for each application.

Beside these mechanisms, the applications are also able to implement their own protection mechanisms for their data. For example, a common method to protect data is to encrypt the data of an application with a common encryption algorithm.

Thus, the protection possibilities of Android, and its applications can be separated into three categories. These three categories can be defined as low-level, mid-level, and high-level security mechanisms, where the levels define the place of the security mechanisms. In this context, we defined the following security levels.

Operating system security mechanisms are defined as low-level protection mechanisms based on operating system mechanisms, like file-system permission schema.

Application-specific security mechanisms are mid-level protection mechanisms, which protect the operating system, and the applications against malicious applications. This mechanism is based on the previously defined application's communication-specific mechanisms, like component encapsulation, application permissions, and own Dalvik virtual machines.

Implemented security mechanisms are defined as high-level security mechanisms, which an application implements by itself and is not based on any Android-specific security mechanism, like data encryption.

However, each of these security categories has its weaknesses. Based on the defined security levels, we similarly separate the issues of each security level. These levels, and an exemplary issue for each level can be connected to the previously defined security mechanisms as follows.

Low-level issues are based on weaknesses of operating system security mechanisms. For example the deactivated root user can be activated, which destroys the application encapsulation mechanism on file system level.

Mid-level issues are based on weaknesses of application-specific security mechanisms. For example, application's components are protected by several security mechanisms to avoid unauthorized access, but all of them share the same communication channels for making phone calls, sending and receiving an SMS, and to use the Internet. Thus, these communication channels can be used to infiltrate the security mechanisms of Android applications.

High-level issues are based on weaknesses of implemented security mechanisms. For example, the idea to protect application by additional security mechanisms, like encryption, is good, but if an application uses an already broken, or weak encryption algorithm, then the data protection is useless.

In the following section we discuss the issues of the different levels in more detail, and why it is important to analyse, and to verify applications if certain security mechanisms are correctly implemented. Additionally, we discuss why applications should implement specific measurement mechanisms, to ensure that all operating system security mechanisms are active.

1.1 Android Security Issues

Modern mobile phone operating systems have many different security issues, which are composed of low-level, mid-level, and high-level issues. Low-level issues are defined as issues based on operating system security mechanisms. Mid-level issues are defined as issues based on application-specific security mechanisms, and high-level issues are issues of own implemented security mechanisms.

Thus, an application with sensitive data has to deal with these issues to minimize the number of security vulnerabilities. To gain a better insight, this section describes the different types of issues in more detail.

1.1.1 Low-Level Issues

Low-level issues, which are based on operating system security mechanisms, including, among others, the version of the operating system and the administration user. Currently, there are at least five different Android versions outstanding [Andreas, 2013]. However, security vulnerabilities of older Android versions still exist and application developers must deal with them. One possible solution for this problem is to only support the latest version, but this would decrease the potential amount of customers. Another issue of this category is the administration user. The so-called root user is able to modify all files on the system. Therefore, a security relevant application should check, whether a device is rooted or not. Unfortunately, we do not have any analysis results whether applications make root-checks or not, what we want to change with this work.

Accordingly, one important focus of this master's thesis is the detection of root-checks in applications. This helps us to make assumptions about the security of an application and whether the developers of an application also considered operating system issues or not. However, an analysis of the available Android versions and their vulnerabilities is not part of this work, because this is basically an operating system and market share analysis. The next section describes rooted phones with its advantages, disadvantages, and issues in more detail.

Rooted Phones

In general a rooted device is a device with an active user account that has administration privileges. The term rooted is derived from Unix operating systems, which always have an administration user called root. Nevertheless, this user can be activated, or deactivated. Current Unix operating systems deactivate the root user and allow users with the help of a program, which is executed with administrative rights, to run other programs with administration privileges. The advantage of this solution is that the root user remains inactive and only one application acts as a gatekeeper. This gatekeeper application is able to allow users to execute applications with administrative rights or not.

Android does not have such a program. Furthermore, such a program can not even be installed, because the system prevents the installation and the root user is deactivated by default. Thus, the user does not have the required permissions to grant an application administrative rights.

To circumvent this protection mechanism, a vulnerability must be exploited, but this results in an active root user. Thus, the user, and the applications are able to modify files from other applications, or system files. This is done for various reasons. On the one hand, an owner of the phone can reactivate root to modify the system, or to use low-level functions of the operating system that can not directly be used without administration privileges. On the other hand, malicious applications use administration privileges to embed malicious code deeper in the system and to hide it from antivirus applications.

In contrast to the advanced capabilities of rooted phones, security relevant applications tend to prevent the execution on rooted phones, because the data of the applications could be compromised. So, we defined security relevant applications as applications with sensitive data. That means that, in our case sensitive data are business data, authentication data, and banking information. Security relevant applications could be applications with business data, applications for storing electronic money, banking applications, or other applications with authentication data. If such applications store the sensitive data in plain text or with a weak encryption in the private folder of the application, then another application with administration privileges can easily steal this sensitive data.

In this context, we see a high security vulnerability in rooted phones. Thus, it must be detected, whether applications check if a phone is rooted or not. Furthermore, if an application uses administrative permissions for certain operations, then this application must be found, because it could be that such an application is malicious.

1.1.2 Mid-Level Issues

Mid-level issues are based on the different application-specific security mechanisms. These security mechanisms protect the inter-process communication with other applications, and the usage of Android's communication channels.

In Android the inter-process communication can be done through services, and intents. For example, an application is able to bind to a service from another application to communicate with certain components of the other application. Additionally, it is also possible to use intents to forward some content to another application and to retrieve some response. Chin et al. [2011] discuss issues with inter-process communication in Android. Furthermore, privilege escalation of an application by exploiting components of other applications is also possible with inter-process communication [Davi et al., 2011]. These

are only two of many research papers about IPC in Android. Thus, the analysis of the IPC architecture of Android is not one of the primary goals of this work, but it is discussed how good the detection of such mechanisms would work with our framework.

Furthermore, a smartphone has typically three different communication channels. Firstly, it is able to make phone calls. Secondly, it is possible to send short messages (SMS). Thirdly, a smartphone has access to the Internet. Each of these communication channels can be used for benign, and malicious applications, for different purposes.

Phone calls can be used for several purposes. The typical usage scenario is to call another person, but phone calls can also be used for payment services, emergency services, and scam calls. Payment services, and scam calls are more interesting for security inspection. Calls to premium rate numbers are protected by an additional permission of the system. Thus, a user sees the possibility that an application is able to call premium rate numbers during install-time. Furthermore, the owner of a phone can block incoming and outgoing premium rate numbers from the service provider. Thus, we do not inspect the possibilities of phone calls in this work, because the user has always the possibility to block premium rate numbers through the service provider.

The connectivity to the Internet is the communication channel with the most possibilities. Through the use of this channel, applications are able to do almost all from the other two channels. In that way, it is possible to make calls, send short messages, send mails, download additional content, etc.. To prevent that benign, and malicious applications do not send sensible data to a server, many researchers presented approaches to secure the communication with the Internet. Enck, Gilbert, et al. [2010] discuss the possibilities of an information-flow tracking system to detect data, which are sent to a server. Furthermore, Barrera et al. [2010] formulated a fined-grained permission model to increase the security by reducing the amount of allowed operations on a communication channel. These are only two of many research papers, which primarily discuss issues with the Internet on mobile phones. Thus, we do not focus on this communication channel, because there is already a lot of research in this field.

SMS have an important role as communication channel, because SMS are typically used as second authentication method in two-factor authentication scenarios, and as confirmation messages in electronic depository transfers. Unfortunately, it is very easy to intercept the SMS communication on Android. Malicious applications are able to sniff, and abort SMS without the knowledge of the user. Furthermore, it is also possible to hide such an SMS listener from Android's API. Thus, third-party applications can not easily detect whether an application intercepts the SMS communication or not. Because of the high relevance of the SMS channel, we added the SMS channel to our goals. We want to detect issues with SMS regarding the possibilities what an application can do with an SMS, and which kind of applications sniff or abort SMS. The next section discusses issues with the SMS channel in more detail.

SMS Issues

Our second defined security threat handles vulnerabilities of the SMS architecture on Android, which come from the flexible use of broadcast receivers. SMS is a communication method, but the range of functions for SMS is much higher than only sending text messages. It is used to control the behaviour of the phone, to track the position of the phone, or to do a factory reset in case of a theft. Furthermore, SMS are often used for multi-factor authentication mechanisms, and for electronic banking. Multi-factor authentication can be used to reduce the probability of a hijacked account, by using two communication channels for authenticating a user. For electronic banking, short messages are used to validate a cashless money transfer. This enormous potential of the SMS architecture and the vulnerabilities of the underlying component led us to define this as our second point of interest.

Android's system architecture allows applications to register themselves to receive SMS in two different ways, either dynamically, or statically. The differences of these two registration schemes are not obviously visible. It is only possible to retrieve information of statically registered receivers from other

applications, whereas dynamically registered receivers are not visible by other applications with the help of Android's API. Consequently, it is not easily possible to detect SMS applications with dynamically registered receivers. As a result of this, applications have the possibility to hide their SMS functionality from other applications. The only sign of an SMS receiver is the additional permission for the application, but many applications are over-privileged [Davi et al., 2011]. Therefore, applications that analyse and classify the installed applications do not classify and list such applications as SMS receiving applications.

We believe that also current antivirus applications do not detect this type of applications. In our perspective, these applications are potentially security threats, because they are able to sniff SMS without the knowledge of the user. Our motivation is to detect such applications to notify the user about the existence of such applications on the phone.

1.1.3 High-Level Issues

The last defined security issues are high-level issues, which are based on implemented security mechanisms. Android applications have the possibility to easily replace every default application. Because of this fact, applications must have wide-ranging permissions in the system, to be able to replace core features, like the lock screen, the home screen, the dialer application, or the messaging application. Nevertheless, such wide-ranging permissions can lead to security vulnerabilities, because if an application does not care about the security of a component, then the application will have a security vulnerability. Security relevant implementation issues including among others, input-output operations (IO), and encryption algorithms.

IO operations can be an implementation issue, but this issue depends on the location where a file is stored, read, and which data the file contains. If an application uses the public directory to store configuration properties, or cached data, then this application will be exploitable for another application. The problem with cached data is that they can contain authentication information, or other sensitive information. Because of the relevance of the correct use of the public directory, we also inspect IO operations in this work.

Another implementation issue can be the usage of encryption. Data encryption is basically an efficient method to protect data, but if the used encryption algorithm is already broken, or by default weak, then the encryption is useless. Nevertheless, in certain cases a weak encryption is not an issue by default, because if the encryption must only withstand a short period of time, then a weak encryption algorithm is also suitable. On the one hand, such an usage scenario could be the encrypted transaction of a one-time pad, where the one-time pad is only valid for a couple of minutes. On the other hand, secure applications – like password safe applications – must use a strong encryption algorithm, which is exhaustively analysed and where an attack is computational infeasible. Thus, it is interesting to know if an application uses encryption. Because of these aspects we also added the detection of cryptographic code to the goals of this work.

1.2 The Goals of this Master's Thesis

This section describes the goals of this master's thesis, based on the previously described problems. We do have the following goals:

- Provide a framework for direct execution on Android's mobile platform, to give users the capability to analyse their applications.
- Analyse applications to find the code for root-detection.

- Analyse applications to detect the code for processing SMS, and classify the applications, based on this SMS processing code.
- Evaluate the framework for the applicability to detect cryptographic code snippets in applications.

1.2.1 Android as Mobile Platform

We decided to use Android as the underlying operating system on which our framework should work. This decision is made for several reasons. Android is currently the most common operating system for mobile phones [Zhou and Jiang, 2012]. Thus, our framework covers the most mobile phones, and the amount of malware for Android is probably also higher than for other mobile platforms. Furthermore, it has a more open approach in contrast to the other three common operating systems – iOS, Windows Mobile, and Blackberry OS. Therefore, we can get easier an in-depth knowledge of the operating system. The last property is regarding the execution environment. It is easier to statically analyse intermediate code as native code.

1.2.2 On-device Execution

Our second basic design decision was to build a framework that is capable of running directly on a mobile phone. This is due to the entry barriers of desktop analysis frameworks for mobile applications. We did not choose a cloud-based solution, because we would have more implementation effort for implementing a server, and client application, and to design a communication protocol. Additionally, not each user has a mobile data contract with unlimited data transfer volume. Therefore, we assume that a standalone client is more suitable.

Our experience with desktop analysis frameworks is that they are mostly difficult to use for end users. Nevertheless, the end users should also be able to do basic security checks on installed application and not only trust the application's developers.

We know that this decision has certain disadvantages in calculation power and execution privileges. Thus, our evaluation includes a performance evaluation of our framework to discuss the usability of our framework on mobile phones.

1.2.3 Root Classification

One of our focused goals is the detection of code that is used to classify whether a mobile phone is rooted or not. Thus, we must find applications that check if the phone is rooted or not. With that basis we are able to find common root-detection patterns and compare them with our detection methods. Subsequently, our framework must be evaluated to analyse the reliability of it. We defined the goals for detecting code to detect rooted phones as follows:

1. Search, decompile, and analyse current available applications with root-detection code.
2. Define detection patterns for the framework on the basis of the analysis, and our already known detection methods.
3. Evaluate the defined detection patterns.

1.2.4 SMS Classification

The second goal of this work is the SMS architecture of Android. We want to have a deeper understanding of the architecture itself and its potential threats. In consequence of that, we have to manually analyse many applications with the permission to receive SMS and to classify these applications. Thus,

classification categories must be defined. Based on the newly learned behaviour of the SMS architecture, we must define detection patterns to efficiently classify applications into our defined categories. Finally, we must also evaluate the performance of our framework with our defined detection patterns. Therefore, our goals for classifying the SMS architecture of Android are the following ones.

1. Analyse applications with SMS permissions, define categories, and classify these applications.
2. Define detection patterns to efficiently classify applications in our defined categories.
3. Evaluate the defined detection patterns.

1.2.5 Applicability for Cryptographic Analysis

Beside our previously defined goals, our framework should also be able to detect and to classify other types of code. Thus, we also defined an additional detection pattern to analyse the use of cryptographic code in applications.

Furthermore, it is interesting to know which encryption is used and whether the cryptographic code is used or not. With this information we can conclude the strength of the used cryptography. Nevertheless, we are aware of the fact that static slicing is not the best mechanism to find cryptographic code. Thus, this should only be an experimental setup to examine the applicability for the analysis of cryptographic code in applications.

Therefore, this detection class should be able to find the code for encryption and decryption and to detect the used algorithms for the cryptographic operations. This detection class is also used to analyse whether the framework is portable for other usage scenarios or not.

1.3 Structure of this Thesis

This thesis describes a new approach of analysing Android applications for gathering information about their security. The first part of the thesis (Section 2 and 3) embeds this work into the context of other related work. Furthermore, it describes background knowledge about the Android system, its security mechanisms, and technical background about static slicing. Additionally, this part also describes related frameworks, which are also used to analyse Android applications.

The second part of the thesis handles the technical implementation of the work and the evaluation of the performance of the work. Section 4 describes the technical implementation of the work and Section 5 describes the results of the evaluation of the framework on different evaluation sets.

Finally, the last part outlines some concluding remarks about the work of this thesis in Section 6 and consists of a discussion about future work in Section 7.

Chapter 2

Preliminaries

" Getting information off the Internet is like taking a drink from a fire hydrant. "

[Mitchell Kapor]

This work discusses issues with applications, which leak information on mobile phones powered by Android and proposes a solution for those issues. Here we give a brief introduction about the used technologies. The first section defines the technical vocabulary, which we use in this work. Section 2.2 discusses the Android operating system and its security mechanisms. This platform is used by our proposed solution. Next, an in-depth discussion about the Dalvik file format and the use for static slicing is given. The following sections discuss different analysis methods and existing frameworks, which are based on these analysis methods. Finally, the last section gives a brief introduction of static slicing, with the help of a simple language.

In contrast to the related work chapter, this chapter does not consist of academical research, but rather technical background about the Android system, analysis techniques, and available analysis tools.

2.1 Technical Vocabulary

We must differentiate between mobile phones and smartphones. A smartphone is a mobile phone with more advanced capabilities than a typical mobile phone has. Such advanced capabilities are email functionalities, portable media player, digital camera, GPS navigation, and feature-rich Internet browsers. Nevertheless, in this work both terms are equally used and refer to smartphones. It is possible to root current smartphones. The term root refers to the possibility to gain administrative privileges on the mobile phone. Rooted phones have additional functions, which result from the additional privileges of an application. Such applications can be used to read private data from other applications. Therefore, the data of the applications can be compromised. It is essential to differentiate between data security, and data confidentiality.

Data confidentiality refers to preventing the disclosure of information to unauthorized individuals or systems.

Data security means protecting data against malicious applications and keeping the data confidential.

We also introduce the terms technical expert, non-technical expert, security expert, and user. We use the terms technical expert, and security expert for persons, whose knowledge about the technical details of a specific piece of hardware, or software is higher than for typical end users. For the term end user,

we also use the terms non-technical expert, and user. All of these terms refer to persons, who have basic knowledge about the system, but have not detailed understanding of the underlying technology.

Finally, there are different definitions for malicious applications and its variant. In this work we do not differ between the terms malicious applications, malware, and viruses. In all cases we mean a harmful application.

2.2 Android OS

This section gives a brief overview of the Android system architecture, which is currently the world's most popular smartphone operating system. Android reached a market share of 75% in the third quarter of 2012 [Nagamine, 2012]. Furthermore, Android's official application market reached an amount of 700.000 applications [Islam, 2012]. The success story began when Android was founded in 2003. It took five years until the first Android devices for mass-market were available in 2008. Android has become the market leader in only five years.

One success factor of Android's mobile device operating system is, that it is based on open source licenses [Butler, 2011]. Thus, it allows every company, and especially every developer, to modify and improve the code of the operating system. In general, for most parts of the operating system the Apache License[1] is used, and for the Linux kernel the GNU General Public License[2] is used [Paul, 2007]. These licenses allow developers to modify the software for any purpose. We believe that another success factor of the system is that the system is based on an already existing operating system and Google's marketing strategy. The core of Android comes from Linux, an open source operating system. The main advantage to use an existing operating system is that the basic functionality of an operating system does not need to be written by oneself which can lead to a faster development of the operating system itself. A disadvantage that possibly many vulnerabilities are also ported to the new platform. Nevertheless, if everything is newly developed, it will also have some vulnerabilities. Beside the Android's open source property, Google developed a good marketing strategy, and ecosystem for Android.

We believe that these properties have led many companies to use Android on their different devices. Nowadays, it is used for smart watches, smart TVs, cameras, laptops, etc. Nevertheless, a primary threat of such a wide spread system is that every vulnerability in the software is in every device. Thus, if every device from a user is based on the same operating system, then it will be possible that an attacker can easily infiltrate the whole technical infrastructure of such a user.

Android seems to waste many system resources for protecting application processes from each other, but in more detail the system uses this to efficiently avoid program-flow manipulation by malicious applications. Furthermore, Android has specific security mechanisms, which are designed to protect application components. It is important to use these security mechanisms, because a weak design of applications can lead to security vulnerabilities, which would make it possible for malware to hide malicious code, or to easily gain additional permissions by exploiting benign applications [Felt, Chin, et al., 2011]. The following sections describe the general design of Android, its security vulnerabilities, and the security mechanisms of Android applications.

2.2.1 Android Architecture

Android is built on top of a modified Linux kernel. The kernel is adapted to fulfill the requirements of mobile devices. One example is Android's better power management to increase the uptime of mobile devices [*Android Kernel Development* 2013]. Android's applications are written in Java, but they do not use the Java virtual machine (JVM) for execution. Android has its own virtual machine to execute the

[1] http://www.apache.org/licenses/LICENSE-2.0.html
[2] http://www.gnu.org/licenses/old-licenses/gpl-2.0.txt

code of applications. Android's runtime environment consists of core libraries, which are written in Java, and the Dalvik virtual machine (DVM). The DVM interprets the byte code of the applications.

Furthermore, it does not matter if the application is pre-installed, or a third party application, because by design Android handles every application in the same way. As a result of this, it is possible to replace the functionality of every pre-installed application by a third party application. Third party applications can be used to replace for example the pre-installed messenger application. Furthermore, Android's developers have added many libraries for third-party developers to reduce the effort of developing new applications for the system.

Figure 2.1 shows the Android architecture with its components divided into five layers. The root of the operating system is the Linux kernel, on which Android is built on. The second layers consist of Android's specific libraries. Android's runtime layer with the DVM and the core libraries is embedded between the library layer and the application framework layer. The last two layers at the top of the hierarchy are for applications and the application's framework that is used by the applications [Developers, 2011].

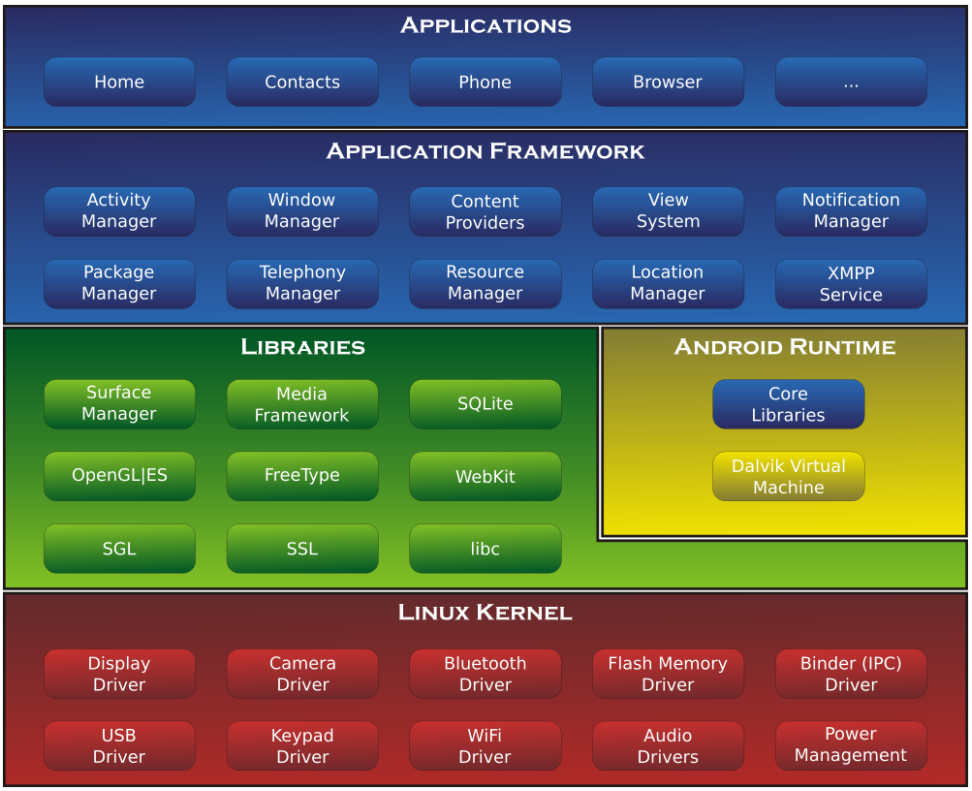

Figure 2.1: Android's system architecture divided into five layers. [Image extracted from *Android System Architecture* [2013] under the terms of the Cc-by-sa-3.0-migrated, GFDL. Copyright © by the GNU Free Documentation License.]

2.2.2 Android Application Structure

Every Android application consists of the manifest file, the resource files, and the source code files. The manifest file declares the different components of an application. In addition, the entry point of the application, and the requested permissions are also declared in the manifest file. Beside the manifest file, every application also contains some resource files. These files are used for additional resources as well as for user interface declarations.

The different components of an Android application, their properties, and a brief discussion about the relevance of the component in terms of the security are described next. Android and its different appli-

cation components suffer from specific security vulnerabilities [Enck, Ongtang, and McDaniel, 2009b]. The security vulnerabilities of Android, and the security vulnerabilities of the different components of an application are described in later sections.

2.2.3 Android Application Components

Activities

Activities are the primary components of Android applications. Activities are used as the presentation layer of an application. Hence, activities are used to handle an application's user input and output. Every Activity must be declared in the manifest file. The manifest file is used to declare all components and permissions, which are requested. Furthermore, for every application one defined Activity must hold an Android specific action[3], which defines the entry point of an application. This is similar to the main function for applications written in C/C++. One important property of an Activity is that it is not possible to run more than one Activity at the same time. Thus, if an Activity calls another Activity, the first is paused and the second is started [*Android Developer Guide - Activity* 2013]. Another property of an Activity is that it is executed on the main thread. This is the only thread that is allowed to draw elements on the screen [*Android Developer Guide - Activity* 2013; Kroop, 2008].

Activities are less important in the context of a security analysis, because malicious applications have not many advantages if they start an Activity of another application. Except starting an exploitable Activity to circumvent Android's permission model by taking advantage of granted permissions of another application [Orthacker et al., 2012]. Basically, the idea of this is that an benign application has an exploitable Activity with a relevant system permission like the Internet permission. Then a malicious application can exploit this service to gain access to the Internet without the permission for the Internet by itself.

Broadcast Receiver

One of the most interesting components of the Android system is the broadcast receiver. Broadcast receivers play an important role in the whole design of Android. They are used for delivering system events, simple inter-process communication, and for broadcasting short messages [*Android Developer Guide - Receiver* 2013; Kroop, 2008].

From the security point of view, broadcasting short messages, or in general inter-process communication, is an interesting aspect, because it is possible that such messages include private information which should not be read or forwarded to certain applications [Chin et al., 2011].

Background Tasks

Services: If an application has a long running operation, then this operation must not be done on the main thread, because this prevents every input and output. If the main thread is blocked, then Android terminates the blocking application, and throws an application not responding (ANR) exception. Android has services, which are components, designed for long running operations. Services run in the background to execute an operation even if the user opens another application. Additionally, a property of a service is that it allows other components to use it.

This could be a potential security threat if the service is defined wrongly in the manifest file [*Android Developer Guide - Service* 2013; Kroop, 2008]. Wrongly defined services can be misused by malicious applications for example to simulate the behaviour of another application. A service can be exploited and misused, if the service does not check the sent data of the calling application.

[3] android.intent.action.MAIN

Asynchronous Tasks: Beside services, Android provides another concept for handling long running tasks in Android. So-called asynchronous tasks are used to execute an operation in the background as well. Such tasks are basically extensions of the Java thread mechanism and thus, they are not a component of an application. Furthermore, asynchronous tasks are designed for one-time operations, like loading some data from the web, to embed it into an application.

In contrast to services, asynchronous tasks have two advantages. Firstly, they have a simple architecture and secondly, the pre-defined call-back architecture to the UI-thread makes it easy to inform the user about progress updates. Our framework also uses an asynchronous task to handle long running operations and to easily inform the user about the current status of the analyser.

Storage

Content Provider: For data storage, content providers are the most common components, because with the help of content providers every application is able to retrieve data and can easily store data in a database. This component is the recommended way to connect to a database in an application. The advantage is that this component encapsulates the data and provides more security mechanisms than other storage solutions would do [*Android Developer Guide - Provider* 2013; Kroop, 2008].

Files: Another common approach is to store data in a file. This approach is not restricted to the Android environment. It is a common solution on Unix systems to store configuration properties of applications in files. Nevertheless, a disadvantage of this simple storage solution is that the files are only protected by the permission model of the system. Thus, the security depends where the file is stored [Becker and Pant, 2010].

2.2.4 Application Start-Up Methods

Android applications consist of different components and three of these components can be used to start an application. These three are activities, sending broadcasts, or starting services. The regular way to start an application is to click on the icon of the application. This tells Android to start the main Activity of the application. The other two start-up methods are hidden and not visible for the user. The system fires broadcasts for pre-defined events, like boot completed or message received. Thus, every application is able to register a receiver for these events. Android starts all registered receivers, when the event occurs. The callback method of the broadcast receiver can be used to do event based operations or to execute malicious code. It is possible that a malicious application starts after the boot of the operating system by registering a receiver for the boot completed event.

Receivers and activities can not be used for long-running operations. Therefore, services can be used to keep the application running in the background without the knowledge of the user. Applications use the combination of the broadcast receiver, and the service component to start at boot and to keep themselves alive to execute code in the background.

2.2.5 Android's Security Mechanisms

This section describes the security properties of the Android system. One important security mechanism is the idea of application isolation through sandboxing [Enck, Octeau, et al., 2011]. A sandbox in the field of computer security is a well known technique to encapsulate an application in a way that this application can not harm the hosting system or another application. Android and other systems use this mechanism to avoid that an application is able to alter data from other applications. Nevertheless, it must be possible to communicate with other applications. So, Android weakens the sandbox principle. One case is the usage of different components where each component can be reused and integrated in other applications in different ways [Felt, H. J. Wang, et al., 2011].

Three security mechanisms implement sandboxing on Android. Firstly, Android's application-specific user management on file-system level uses well known mechanisms to manage the access to the files of applications on the disk. This mechanism comes from Android's kernel, which is adapted from the Linux kernel. Secondly, the execution of an application is encapsulated in its own Dalvik virtual machine [Enck, Ongtang, and McDaniel, 2009b]. Thirdly, to protect the application interfaces, Android provides a permission schema to grant access to specific application components. This mechanism is also used by Android to protect its components from not permitted access [Felt, H. J. Wang, et al., 2011]. All of these mechanisms are described next.

System Permission Model

Android uses the multiuser concept, which comes from the Linux kernel, to encapsulate applications. Basically, this prevents process A to access process B's data, because Android assigns to every application its own identity. Every application, which is not signed by the same developer gets a different Android ID (AID), which consists of a Linux User ID (UID) and Group ID (GID). Consequently, each process from a different developer is executed with its own identity. Therefore, the owner of the files of two applications from different developers are not the same. The principle of shared identities is described in the next section.

If a process tries to access a file from another process then the file system checks the permissions of the file and prevents the access because of the different owners of the accessing process and the file itself.

This mechanism restricts access to files of applications on the file system. Thus, if the device is not rooted, then it is not possible to access private application folders outside an application. Furthermore, it is not possible to access all system folders. Some of the system folders are only accessible by the system. Nevertheless, if a device is rooted the permission structure is useless, because root has per design access to every file, even if the file does not allow the access through the permission system [Enck, Ongtang, and McDaniel, 2009b]. As a consequence of this, it is important to check if the device is rooted or not, because smartphones with root access do not have this security function. The absence of this security feature allows malicious applications to access private folders of applications. For benign applications this is not important, but security critical applications, like banking or business applications, have information leaks [Scheid, 2012].

Listing 2.1 shows the file and permission structure of Android's application cache directory. If it is possible to access this folder, then it will be possible for applications to easily extract cached files from an application, which can include authentication credentials, or other sensitive data.

```
1
2     android:/# ls -l /data/data/
3
4     ...
5     drwxr-x--x app_0    app_0     com.cyanogenmod.theme.Achromatic
6     drwxr-x--x app_2    app_2     com.boombuler.system.appwidgetpicker
7     drwxr-x--x app_1    app_1     com.anddoes.launcher
8     drwxr-x--x app_3    app_3     com.android.providers.applications
9     drwxr-x--x app_3    app_3     com.android.providers.contacts
10    drwxr-x--x app_3    app_3     com.android.providers.userdictionary
11    drwxr-x--x app_3    app_3     com.android.contacts
12    drwxr-x--x app_4    app_4     com.android.backupconfirm
13    ...
14    drwxr-x--x app_100  app_100   com.valvesoftware.android.steam.
          community
15    drwxr-x--x app_101  app_101   com.google.android.street
```

```
16    drwxr-x--x app_102  app_102   jp.takke.cpustats
17    drwxr-x--x app_103  app_103   jp.susatthi.ManifestViewer
18    drwxr-x--x app_104  app_104   woergi.tools
19    ...
20
21    android:/# ls -l data/data/woergi.tools/
22    drwxrwx--x app_104  app_104   cache
23    drwxr-xr-x system   system    lib
```

Listing 2.1: A typical permission structure of the /data/data directory, which is the installed application cache directory.

Weakening Android's System Permission Model

Applications with limited communication channels are more secure, than applications with many communication channels [Chin et al., 2011]. This is obvious, because for example an application with no communication possibilities can hardly be modified by a malware and has no data, except the binary of the program itself, which can be exploited by a malware. Nevertheless, applications with no communication channels, except the UI channel, have limited usability and they are only used for real offline applications, like a calculator, or a heart-rate monitor. Therefore, a compromise between data protection and communication capabilities must be achieved, by the underlying operating system.

For that reason, Android added a feature that easily allows developers to communicate with their own applications, but avoids unprivileged access by other applications. In detail, this feature weakens the user and file permission system of Android. If a developer defines a *sharedUserId* in an application's manifest file, any application with the same ID is executed in the same process and has the same user ID in the system. Therefore, they share the same sandbox [Bugiel et al., 2011]. Listing 2.2 and Listing 2.3 show this mechanism in detail. The first listing shows the needed tag embedded in the manifest file and the second listing shows the directory of the shared applications. Both applications are able to read each others private data. Furthermore, they share the same process, so they also share system preferences and other process resources.

```
1
2     <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3         package="iaik.thesis.example1"
4         android:versionCode="1"
5         android:versionName="1.0"
6
7         android:sharedUserId="iaik.thesis"
8     >
9      ...
```

Listing 2.2: This listing shows an example usage of the sharedUserId-flag, which puts two applications into the same sandbox (Adapted from the Source printed in Becker and Pant [2010, page 31]).

```
1
2   android:/# ls -l /data/data/
3     ...
4     drwxr-x--x app_114  app_114    iaik.thesis.example1
5     drwxr-x--x app_114  app_114    iaik.thesis.example2
6     ...
```

Listing 2.3: This listing shows two applications, which share the same AID. (This listing uses the example code from Listing 2.2 for the two applications.).

File System Security

In early Android versions the private application folder was a common place to store passwords and other configuration files. One advantage of a file-based persistence model is that it is possible to store and retrieve data without any deeper knowledge of Android [Hyojun Kim, Agrawal, and Ungureanu, 2012]. It is also possible to quickly port existing application and its configuration files to the Android platform without deeper knowledge of Android's architecture.

Listing 2.4 shows a code snippet that accesses the private folder of an application. If the data should be accessible for every user, the mode must be replaced by one of the public-modes. These modes can be used to share information with other applications. For that, two modes are available:

- MODE_WORLD_WRITEABLE
- MODE_WORLD_READABLE

Nevertheless, public files are dangerous and they are a potential security leak for the application. As a result of this, these two modes are marked as deprecated and should not be used anymore [*Android Developer Guide - Context* 2013]. Thus, files for applications should be only stored with the private mode.

```
1
2   void storeData(String fname, String data) {
3       FileOutputStream f = openFileOutput(fname, Context.MODE_PRIVATE);
4       f.write(data.getBytes());
5       f.close();
6   }
```

Listing 2.4: This source code example shows how data can be stored in the internal private directory.

Application Security

Beside the system permission model, Android provides mechanisms to secure every application component by itself. The communication between application components can easily become an attack vector. If an application has an unprotected component, an attacker is able to force the program to interpret his message which leads to unpredicted behaviour and possibly to information leaks. Furthermore, if a developer accidentally sends data to the wrong recipient, it will probably leak information too. Chin et al. [2011] describe security threats of Android applications and its components. Furthermore, Enck, Ongtang, and McDaniel [2009b] and Kuhn, Ritter, and Mitschang [2012] also discuss security threats in Android's component mechanism. This section describes security threats and Android's protection mechanisms for applications and their components.

Application's Component Security:

The components of an application are defined in its manifest file. In terms of security, the manifest holds certain important flags, which must be inspected to make an assumption about the accessibility of application components and therefore, for the overall security of the application. For example, each component can be assigned as external accessible or not in the manifest. In addition, different component types have different communication capabilities. Three of four component types are able to receive Intents (broadcast receiver, activities, and services). Intents are the message passing system of Android, which can be used to easily communicate with the components of applications. Whether a component of an application is able to receive Intents from outside the application or not depends on the *exported* flag. This flag can be declared for each component in the manifest. If this flag is declared for a component in the manifest, it is easily possible to derive the accessibility of the component. If it is not declared for a component, then it depends on the implicit assignment by the system. The implicit assignment of the exported flag depends on the component type, and is defined as follows [*Android Developer Guide - Manifest* 2013].

Services, *Receiver*, *Activities* are by default exported if they hold at least one intent filter. If they do not hold an intent filter, then Android assumes that these services are only for internal use and sets the exported flag to false [*Android Developer Guide - Manifest* 2013].

Providers are by default exported, but for applications with a minimum-sdk-version number or a target-sdk-version number above 16 are by default not exported [*Android Developer Guide - Manifest* 2013].

Application's Permission Model:

Android also provides a permission model on application level. This model is used for two different things. Firstly, it protects application programming interface (API) calls from unauthorized access and secondly it protects exported broadcast receiver components from unauthorized access. Nevertheless, the permissions are granted at install-time. Thus, the protection of API calls is very limited, because the user decides during installation, if an application gets access to certain functionality or not [Orthacker et al., 2012].

As a consequence of this, the permission model of Android can protect applications, and stored data on the phone, as long as the phone is not rooted. Compared to other mobile operating systems, Android grants the permissions at install-time and not at run-time [*App Attack-Surviving the Explosive Growth of Mobile Apps*]. Thus, users install over-privileged applications on the phone, because they are not aware of the the problems with over-privileged applications [Felt, Chin, et al., 2011]. This can lead to security leaks.

For Android's permission model, the permissions can be separated into four different protection levels [*Android Developer Guide - Permission's Protection Level* 2013]:

Normal: The system automatically grants this type of permission to a requesting application at installation, without asking for the user.

Dangerous: The user can grant these permissions during installation.

Signature: These permissions are only granted if the requesting application is signed by the same developer that defined the permission.

SignatureOrSystem: This category is the highest one. The system grants such permissions only to applications that are in the Android system image or that are signed with the same certificate as the application that declared the permission.

Reverse Engineering

Another issue is that it is possible to reverse engineer the code of an application. It must be differentiated between byte code and native code. Byte code is compiled to a higher-level portable code that operates on classes and primitive types. Native code is compiled to a low-level instruction set that operates on raw binary data and is specific to the target hardware, such as x86. In general, it is easier to reverse engineer the byte code of an application, than the native code of an application. This results from the fact that byte code has still more information than native code.

The reasons for security analysts to reverse engineer the code of applications are to understand the behaviour of the code faster, and to easier analyse it. Unfortunately, malware developers also use this technique to find vulnerabilities in applications.

To increase the complexity of reverse engineering, application developers can obfuscate their code. Obfuscation methods restructure the code and potentially add many useless operations to the code. This is done to increase the effort of following the program-flow in the reversed program. In general, obfuscation methods can be divided into three categories [Nolan, 2012, page 122ff]:

Layout obfuscation: This technique renames the names of the variables, methods and classes which makes the code harder to read.

Control obfuscation: If many useless commands are inserted into the code, the reversed code is harder to read, because these useless code breaks the control-flow.

Data obfuscation: Data obfuscation restructures the data by dividing the data into smaller pieces, encoding it or by reordering the data.

Another simple technique to protect code against reverse engineering is to hide it in native libraries. Native libraries are more complex to reverse engineer as byte code.

2.3 Dalvik Architecture

The Dalvik VM is the basis of every Android application. Android has its own virtual machine which is similar to Java's virtual machine. It interprets Dalvik bytecode. Nevertheless, the Dalvik bytecode format differs in many aspects to Java's bytecode architecture [Bender, 2010]. Figure 2.2 shows the differences between Dalvik and Java binaries. Java compiles each file by its own and Dalvik compiles all files and removes duplicate entries for type declarations, identical strings and other identical data by putting everything in a global section. Another difference is that Java byte code is based on stack-based execution model, whereas Dalvik is based on a register-based execution model. That means that the Java byte code mainly uses a stack for handling data, which leads to many load and store operations. On the other hand Dalvik uses a registers-based architecture which is faster and reduces the amount of instructions [Nolan, 2012]. Both, the register-based architecture and the Dalvik file architecture with global and not class-based sections, lead to a smaller code-size as for Java archive (JAR) files. Ehringer [2010] includes a sample measurement between uncompressed Dalvik Executable (DEX) files, and uncompressed and compressed JAR files. An uncompressed DEX file is approximately 45 percent smaller than an uncompressed JAR file and approximately 5 percent smaller than a compressed JAR file [Ehringer, 2010].

For analysing an application, the most important section is the one, where the instructions are stored. Dalvik adds the instructions into the data section. The data section can be accessed through the class section, which holds a reference to the data of a class in the data section. This class-data element holds information about the methods, variables, etc. of a class. Each method of a class has a reference to a code-item. Finally, such code-items hold the instructions of a method in the data section [Nolan, 2012].

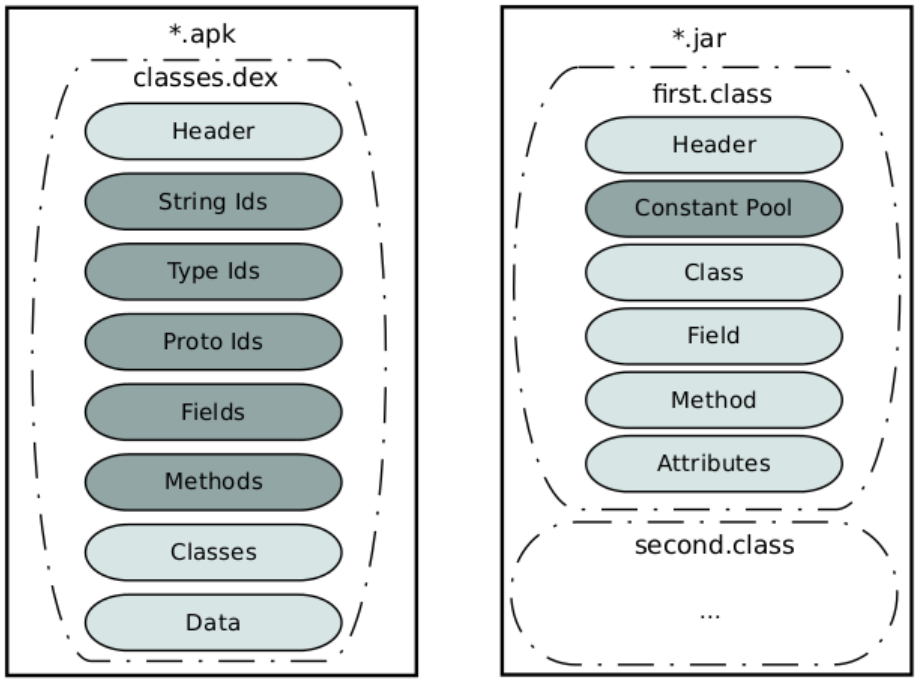

Figure 2.2: This picture compares the Java architecture to the Dalvik architecture. On the one hand Java compiles every class individually and adds every file into an archive. On the other hand Dalvik produces only one file containing compiled classes (The picture is adapted from Nolan [2012, page 58]).

2.3.1 Dalvik Instructions

Dalvik assigns to each instruction a specific instruction format. This instruction format defines the size of the instruction, the number of parameters, and some additional properties of the opcode, like install-time static linking for faster code execution [Paller, 2012].

Dalvik has 30 different instruction formats. Listing 2.5 shows the format of a Dalvik method-invocation instruction. The first half defines the byte code format, with the opcode, the instruction format, the name of the instruction, the parameters. Thus, the instruction *invoke-virtual* has five argument register and the reference to the method object. The second half defines the instruction format for *35c*, which is used by the *invoke-virtual* instruction. Each word is separated by a space and consists of 16 bits. Each character in a word represents four bits read from high bits to low. Uppercase letters in sequence from *A* are used to indicate fields, and the term *op* is used to indicate the position of an eight-bit opcode.

```
1  Byte code format
2  6e35c invoke-virtual {vC, vD, vE, vF, vG}, meth@BBBB
3  ^ ^   ^               ^                     ^
4  | |   |               |                     Method reference
5  | |   |               Argument registers
6  | |   Instruction name
7  | Instruction format
8  Opcode
9
10 Dalvik VM instruction format 35c
11 A|G|op BBBB F|E|D|C
12 ^ ^ ^  ^    ^ ^ ^ ^
```

```
 | | |  |    4 argument registers (each 4 bit)
 | | |  Method reference (16 bit)
 | | Opcode (8 bit)
 | Argument register (4 bit)
 Argument word count (4 bit)

 [A=5] op {vC, vD, vD, vF, vG}, meth@BBBB
 ...
 [A=0] op {}, meth@BBBB
```

Listing 2.5: Dalvik Instruction Format

Table C.1 on page 93 holds all opcodes, whereas this section describes the most important instructions, their instruction formates and their use in more detail. Basically, instructions can be separated in different categories, based on their use.

- Instructions for data retrieval.
- Instructions for arithmetic operations.
- Instructions for branches.
- Instructions for method calls.

Data Retrieval

This section describes four different types of data retrieval.

1. Loading data into a register.
2. Reading an instance variable.
3. Reading a static variable.
4. Extracting an element from an array.

Loading data into a register can be basically done in two different ways. Either it loads a constant into a register, or it moves a value into a register. Firstly, loading a constant into a register can be done with one of the 12 *const* instructions. For example to load a string into the register the instruction in Listing 2.6 can be used. Secondly, one of the 13 *move* instructions can be used to move a value. Listing 2.7 shows a simple move from one register to another register.

```
Byte code format
1a21c const-string vAA, BBBB
^ ^   ^            ^    ^
| |   |            |    String index
| |   |            Destination register
| |   Instruction name
| Instruction format
Opcode

Dalvik VM instruction format 21c
AA|op BBBB
```

Listing 2.6: Dalvik Instruction: const-string

```
Byte code format
0112x move vA, vB
^ ^   ^    ^   ^
| |   |    |   Source register
| |   |    Destination register
| |   Instruction name
| Instruction format
Opcode

Dalvik VM instruction format 12x
B|A|op
```

Listing 2.7: Dalvik Instruction: move

Instance, and static variables have an additional set of instructions. The *move* instructions are used to move the values of such variables, and the *const* instructions are used to initialise such variables. Nevertheless, both variable types have an additional instruction set for storing and loading data from a variable into a register. Without these instructions, they are identically to local variables. Listing 2.8 shows a get and put operation for an instance variable. Static variables have a similar instructions as the put and get instructions for instance variables and the identical instruction format as constant strings.

```
Byte code format
5b22c iput-object vA, vB, field@CCCC
5422c iget-object vA, vB, field@CCCC
^ ^   ^           ^   ^   ^
| |   |           |   |   Instance field reference index
| |   |           |   Object register
| |   |           Source or destination register
| |   Instruction name
| Instruction format
Opcode

Dalvik VM instruction format 22c
B|A|op CCCC
```

Listing 2.8: Dalvik instructions for instance get and put

Another way to store data is to store it in arrays. An array is a set of values from the same type. Dalvik provides specific instructions to get, and put a value in the set. The structure of these instructions is also similar to the structure of the instance instructions, except they refer to an array register and not to an object register.

Arithmetic operations

Arithmetic operations have different instruction formats, which depends on the different instruction sizes. The arithmetic operations with the small instruction size can only be used from the first 16 registers. This results from the fact that for each register only four bits in the instruction format are reserved. The other instruction format reserves eight bit for each register.

Conditions and Jumps

Dalvik has several possibilities for program-flow manipulation. Common terms in a high-level language, like Java, for program-flow manipulations are conditions, and loops. Dalvik has two similar constructs. Firstly, Dalvik has different instructions for jumping to a program location, based on a specific condition. Listing 2.9 shows such an instruction, and its format. The instruction compares the values of two registers, and if the condition is fulfilled, then Dalvik continues with the execution at the position stored in the third register.

```
1  Byte code format
2  3222t if-eq vA, vB +CCCC
3  ^ ^   ^     ^   ^  ^
4  | |   |     |   |  Signed branch offset
5  | |   |     |   Second register
6  | |   |     First register
7  | |   Instruction name
8  | Instruction format
9  Opcode
10
11 Dalvik VM instruction format 22t
12 B|A|op CCCC
```

Listing 2.9: Dalvik instructions: if-eq

Secondly, to achieve loops in Dalvik, a condition instruction must be combined with a jump instruction. Jump instructions are similar to *goto* statements in *C*. Such instructions only hold an offset address to a specific position in the code.

Method calls

For method calls, Dalvik has two different formats. The first one is able to handle five parameter registers. For that instruction format, the first 16 registers can be used as parameter register. This limitation makes the size of the instruction smaller [Paller, 2012]. For methods with more than five parameters or for passing a register, which is not one of the first 16, Dalvik has another instructions – *invoke-kind/range*. For the parameters of that instructions, the registers can be in the range of 0-65535. Nevertheless all parameters must be consecutive, because only the number of the first register and the amount of the passed registers are passed to the instruction. Additionally, both instructions have a method reference, which refers to the invoked method. The first parameter of a non-static invocation is the *this* reference. Therefore, the first instruction type can be used for an instance method with a maximum of four parameters and for static methods with a maximum of five parameters.

If the method has a return-value, then the result of the method is stored in a hidden result register. To use the return-value of a method, the result must be moved from the hidden register to a register, which is accessible by other instructions. This can be done with one of the *move-result* instructions [Paller, 2012]. Listing 2.10 shows the two different method-invocations, and Listing 2.7 shows the basic move instruction, which is similar to the *move-result* instruction.

```
1  Byte code format
2  7035c invoke-direct {vC, vD, vE, vF, vG}, meth@BBBB
3  ^ ^   ^              ^                    ^
4  | |   |              |                    Method reference index
5  | |   |              Argument registers
```

```
6  | |   Instruction name
7  | Instruction format
8  Opcode
9
10 Dalvik VM instruction format 35c
11 A|G|op BBBB F|E|D|C
12 [A=5] op {vC, vD, vE, vF, vG}, meth@BBBB
13 [A=0] op {}, meth@BBBB
14 ^
15 |
16 Argument count
17
18 Byte code format
19 743rc invoke-direct/range {vCCCC .. vNNNN}, meth@BBBB
20 ^ ^   ^                    ^        ^       ^
21 | |   |                    |        |       Method reference index
22 | |   |                    |        Last argument register: NNNN = CCCC +
       AA - 1
23 | |   |                    First argument register
24 | |   Instruction name
25 | Instruction format
26 Opcode
27
28 Dalvik VM instruction format 3rc
29 AA|op BBBB CCCC
30 ^
31 |
32 Determines the count 0..255
```

Listing 2.10: Dalvik instructions: invoke-direct and invoke-direct/range

2.4 Code Analysis

Information-flow analysis methods and malware detection methods are mainly based on code analysis techniques. In general, there are two different code analysis techniques – either static code analysis or dynamic code analysis. The main difference between these two categories is that for dynamic analysis the code under inspection must be executed, whereas for static analysis only the byte-code is inspected. This section describes the properties of both analysis techniques and their advantages and disadvantages. Additionally, it discusses the usability for direct use on a mobile device.

Mobile devices are limited in resources. In addition, Android applications are limited in their granted permissions by the protection mechanisms of Android, which leads to further issues for static and dynamic analysis techniques. Therefore, not every code analysis technique is suitable for mobile platforms.

2.4.1 Static Analysis

In contrast to dynamic analysis, which needs a complete runtime environment, static analysis only needs the application binary, and a parser for the code. Thus, static analysis frameworks requires less privileges on a system, which stems from the fact that it only needs access to the application. Additionally, it is easier to execute and to automate frameworks based on static analysis techniques on restricted environments. This results from the fact that static analysis frameworks requires less privileges on a system. This property is very important for restricted environments like the Android system, because on a non-rooted

Android system, it is not as easily possible to intercept the execution of an application or to sniff system calls as on desktop operating systems.

Nevertheless, static code analysis also has some negative properties or disadvantages. The usage of resources can grow exponentially, because each possible execution path produces a new possible result, which must be tracked for a complete analysis of an application. Therefore, a complete analysis of an application could result in many possible execution paths, where each of them consume memory. Many of those cold never practically be realised.

2.4.2 Dynamic Analysis

In contrast to static code analysis, dynamic code analysis needs more than only the application. It also needs a modified execution environment to track the execution path of an application. To track such an execution path of an application a dynamic analysis framework must also be able to interact with the application. For example, this can be done manually, or by tools like Monkey[4], which is a tool for Android applications that simulates user interactions.

A dynamic code analysis technique can not guarantee that every branch in the code is executed and inspected, because dynamic analysis is performed on specific execution paths with specific variable assignments. These variable assignments potentially do not match all possible variable assignments of the program, and thus not all execution paths are analysed. For example, if a malicious code is triggered by an external event, which is hard to guess, then it will not be detected during a dynamic analysis. This results from the fact that the specific execution path is not executed.

Dynamic code analysis inspects the executing program, which can be done if the framework is able to intercept the communication of the application and the underlying operating system [Blasing et al., 2010]. Furthermore, dynamic code analysis can also be used to inject additional code in an application, to analyse the internal communication of this applications.

Unfortunately, a stock Android system does not allow to inspect the communication between an application and the operating system, so for this dynamic analysis technique the device has to be rooted first. The so-called application instrumentation method can be used to analyse the application itself, or to circumvent the restrictions of a stock Android by intercepting the communication between the operating system and the application [Enck, Gilbert, et al., 2010].

2.5 Static Slicing

Static slicing is an established method for analysing applications, especially for debugging and testing. Weiser [1981] introduced the concept of a program slice. He defined a slice as follows:

> A static program slice S is a reduced executable program derived from program P, and consists of all statements in P that may affect the value of variable v at some program point p, such that S replicates parts of the behaviour of P.
>
> The slice S is defined for a slicing criterion $C = (p, V)$, where p is in P, and V is a subset of variables in P.

K. J. Ottenstein and L. M. Ottenstein [1984] showed that a program can be represented by its program dependency graph (PDG) and the computation of a slice can be simplified to a reachability problem in the PDG.

[4] http://developer.android.com/tools/help/monkey.html

A number of different static and dynamic slicing algorithms, with different definitions, are published. This section describes the basics of static forward and backward slicing in association with a simple language.

2.5.1 Language Definition

This simple language we use to illustrate is shown in Listing 2.11. In this language, a program P is a block B, in which a number of statements S are defined. The language consists of arithmetic operations, binary operations, assignments and two different types of branches. One branch is to construct conditions, and the other one is to construct loops in a program. It lacks classes, methods, concurrency, IPC, etc., because for simplicity reasons.

```
1   P  := B
2   B  := begin S* end;
3   S  := identifier = E;
4      |  if E then B [else B] fi;
5      |  while E do B od;
6   E  := num
7      |  identifier
8      |  ( E OP E)
9      |  ( UOP E)
10  OP := + | - | * | / | && | || | > | < | == | ...
11  UOP := ! | ~
```

Listing 2.11: A very compact and abstract language which is used in the static slice examples. The language is adopted from Wotawa [2002].

2.5.2 Backward Slicing with a PDG

For backward slicing with a PDG the following definitions are made. Firstly, a node j *postdominates* another node i, if every path from i to the exit of the graph goes through j [Lam et al., 2006]. A path is a sequence of edges, which connects a sequence of vertices. Secondly, $DEF(i)$ is a definition set that denotes the set of variables, which are defined at node i. Thirdly, $REF(j)$ is a reference set that denotes the set of variables, which are referenced at node j [Krinke, 2003].

Based on the language in Listing 2.11, a PDG is a directed graph of a program P. The nodes are the statements S of the program P and the edges are control- and data-dependencies. A node i has a control-dependency to another node j if and only if

- there exists a path R from i to j such that any node $u \neq i, j$ in R is post-dominated by j and
- i is not post-dominated by j.

Without modifications, the control dependence subgraphs have no single root because the top most statements will not be control dependent on anything. On the other hand, it is desirable that there exists a single root, which should be the $ENTRY$ node. This is usually achieved by inserting an irrelevant control flow edge from the $ENTRY$ to the $EXIT$ node. The effect is that no other node than the $EXIT$ node post-dominates the $ENTRY$ node, and the $ENTRY$ node will be the root in the control dependence subgraph. Normally, the $EXIT$ node will be omitted from the program dependence graph, as it has no in-, or outgoing dependence edges [Krinke, 2003].

Additionally, a node has a data-dependency to another node if there exists a variable x such that:

- $x \in DEF(i)$,
- $x \in REF(j)$ and
- there exists a path R from i to j without intervening definitions of x.

Based on the simple language, Listing 2.12 shows a basic program with a loop, and a condition. This program is used to discuss the principle of static backward slicing with a PDG.

```
1   begin
2     n = 4;
3     r = 0;
4     i = 0;
5     while (i < n) do
6       begin
7         if (i == 1) then
8           begin
9             r = r + 1;
10          end
11        fi;
12        i = i + 1;
13        n = n - 1;
14      end;
15    od;
16    n = i;
17  end;
```

Listing 2.12: A very compact program to demonstrate static slicing with a PDG.

To generate the PDG, the control-, and data-dependencies must be computed. As defined before, a control-dependence is defined in terms of post-domination. For the computation of the PDG, each statement in a branch of an *if* or *while* block depends on the specific control statement. In addition, data-dependencies are defined through the definition, and reference set. Thus, we can say that the definition of x at node i is a reaching definition for node j.

The slicing criterion is identified with a node in the PDG, where the node corresponds to the program point p in the slicing criterion $C = (p, V)$. For the defined language, the slice S with respect to p consists of all nodes from which p is reachable, either via a data-, or control-dependency. Thus, to compute S we have to find all nodes in the PDG, which have a path to p.

Figure 2.3 shows the computed PDG from the example program, which is shown in Listing 2.12. The solid lines are the computed control-dependencies and the broken lines are the computed data-dependencies. For an exemplary slicing criterion $C = (16, \{i\})$, the resulting slice $S_{(16,\{i\})}$ is $\{2, 4, 5, 12, 13\}$. The specific nodes for this slicing criterion and the resulting slice are marked in Figure 2.3.

2.5.3 Forward Slicing

Forward slicing uses the basic idea of slicing in execution direction. In contrast to backward slicing, it does not inspect the predecessors $pred(n)$ of a statement, it inspects the successors $succ(n)$ of that statements.

To inspect the successors of a statement, a control flow graph (CFG) is used. If a path made up of one or more successive edges leads from x to y, then y is said to be a successor of x, and x is said to be a predecessor of y. In other words, if y is reachable from x, then x is a predecessor of y, and y is a successor of x.

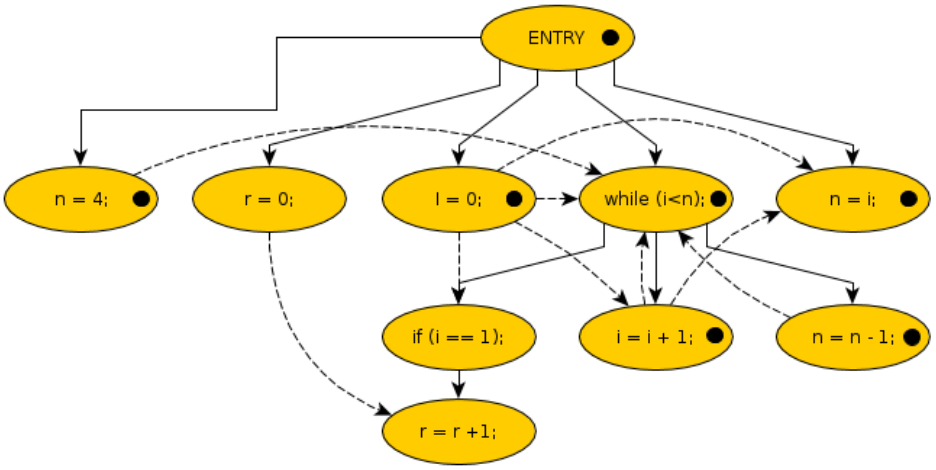

Figure 2.3: PDG for Listing 2.12

A CFG is a directed attributed graph G with a node set N and an edge set E. The nodes are the statements S of the program P, and the directed edges represent the jumps in the control flow [Zeller, 2009].

For forward slicing a flow propagation algorithm must be applied on the CFG, to compute all influenced successors of a variable v. This algorithm is based on the definitions of DEF, REF, an influenced set $Infl(n)$ for every node n, which holds the influenced variables v of n, a logical value $inSlice(n)$, which defines if a node is part of the slice, and the following rules:

- $in(n) = \cup_{p \in predecessor(n)} out(p)$
- $out(n) = GEN(n) \cup (in(n) \setminus KILL(n))$
- $GEN(n) = v | v \in DEF(n) \wedge (REF(n) \cap in(n) \neq \{\} \vee inSlice(n))$
- $KILL(n) = v | v \in DEF(n)$
- $inSlice(n) = True | REF(n) \cap out(n) \neq \{\}$
- $inSlice(m) = True | n \in N(m \in Infl(n) \wedge inSlice(n))$

The logical value $inSlice(n)$ is at the beginning of the computation $false$ for all nodes, except for the node for which the forward slice should be calculated. After calculation of the relevant variables, $inSlice(n)$ is set to $true$ for all nodes n where $REF(n) \cap out(n) \neq \{\}$. Furthermore, $inSlice(m)$ is set to $true$ for all nodes m, which are in the influence of $n (m \in Infl(n))$ if $inSlice(n)$ is $true$. At the end, the slice S_C consists of all nodes n where $inSlice(n) = True$.

The flow propagation algorithm adds a variable v to the tracking set if:

- v is defined in the statement n and the statement n is already in the forward slice or
- v is defined in the statement n and a relevant variable of a predecessor node is referenced $v_2 \in REF(n) \wedge v_2 \in in(n)$

Listing 2.13 shows the pseudo code of the generic flow propagation algorithm. The algorithm is from Lam et al. [2006]; Tonella [2005].

```
1   Input: A set of nodes N, and the slicing criterion C
2   Output: The resulting slice S_C
3
4   ∀n ∈ N
5     in(n) = {}
6     out(n) = GEN(n) ∪ (in(n) \ KILL(n))
7     KILL(n) = v|v ∈ DEF(n)
8     GEN(n) = v|v ∈ DEF(n) ∧ (REF(n) ∩ in(n) ≠ {})
9     inSlice(n) = False
10
11  while any inSlice(n) changes
12    while any in(n) or out(n) changes
13      ∀n ∈ N
14        in(n) = ∪_{p∈predecessor(n)} out(p)
15        out(n) = GEN(n) ∪ (in(n) \ KILL(n))
16        GEN(n) = v|v ∈ DEF(n) ∧ (REF(n) ∩ in(n) ≠ {} ∨ inSlice(n))
17        inSlice(n) = True|REF(n) ∩ out(n) ≠ {}
18    end while
19      ∀m ∈ N
20        inSlice(m) = True|n ∈ N(m ∈ Infl(n) ∧ inSlice(n))
21  end while
22
23  ∀n ∈ N
24    S_C = S_C ∪ n|inSlice(n) = True
```

Listing 2.13: The pseudo code of the flow propagation algorithm for slicing applications.

For every statement, the flow propagation algorithm computes a generation-, and an outgoing-set. All other sets implicitly exist from the current statement and its predecessors. The definition set can only hold at the maximum one variable, because each node is one statement and each statement can only be one assignment operation. The reference set contains all variables referenced from the current statement and the incoming set contains all outgoing variables of all predecessors. Finally, the kill set is a duplication of the definition set.

The generation and outgoing set can be easily computed by applying the previously defined rules. After generating all sets, the next step is to compute the statements of the slice. For that the definition of $inSlice(n)$ and $inSlice(m)$ is used. A statement is in the slice if it is directly influenced. This is described by the intersection of the reference and the outgoing set. Additionally, a node is indirectly influenced if a predecessor is influenced on which the node depends on.

To visualise the behaviour of forward slicing, the slicing criterion $C = (3, \{i\})$ is applied on P, which is shown in Listing 2.14. The CFG for this program is shown in Figure 2.4. Figure 2.5 shows the computed results of the flow propagation algorithm. The computed slice S_C is $\{3, 4, 6, 7\}$. Line 3 is the slicing criterion itself, line 4, and 6 fulfill the condition $REF(n) \wedge out(n) \neq \{\}$, and line 6 is indirectly influenced, and thus in the slice, because at line 4 $inSlice$ is $true$, which influences line 6.

```
1   begin
2     r = 0;
3     i = 0;
4     while (i < x) do
5       begin
6         r = r + y;
7         i = i + 1;
8       end;
```

```
9       od;
10      ...
11    end;
```

Listing 2.14: A very compact program to demonstrate forward slicing.

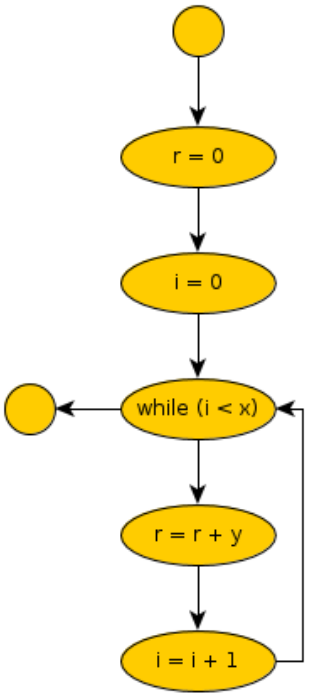

Figure 2.4: CFG for Listing 2.14.

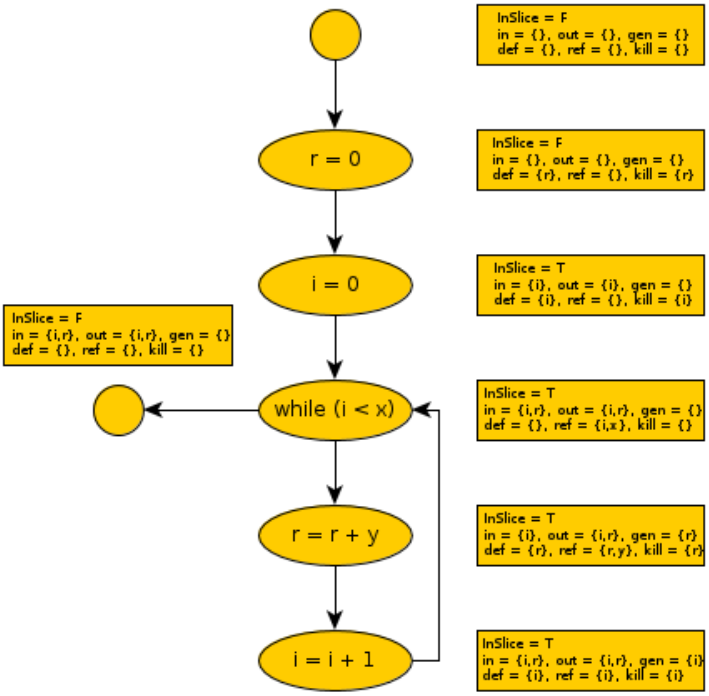

Figure 2.5: CFG for Listing 2.14 with Applied Flow Propagation Algorithm.

2.5.4 Limitations

The static slicing algorithm, as defined in this section, has some limitations. The complexity of slicing increases, if the program is object-oriented, or uses concurrency, and aliasing [Wotawa, 2002].

Object-oriented programs have classes, methods, and instance and static variables. A static slicing algorithm must consider the scope, and lifetime of these objects.

Aliasing describes a situation in which data can be accessed through different names in a program. Thus, modifying the data through one name implicitly modifies the values associated with all aliased names.

Concurrency: A thread is a part of a program, which must be executed on a single processor. Threads may be executed in parallel. Thus, a static slicing algorithm must consider threaded programs, beside object-oriented programs.

Section 4.2 discusses the modifications, and restrictions of the slicing algorithm, for Android applications. This includes the issues with concurrency, aliasing, object-oriented properties, etc.

Chapter 3

Related Work

" Learn from yesterday, live for today, hope for tomorrow. The important thing is not to stop questioning. "

[Albert Einstein]

It is important to know current work regarding malware detection and avoidance, information leaks, application analysis, and Android analysis. Furthermore, this chapter also discusses limitations of the proposed approaches.

The chapter itself is structured into four sections. First, related work regarding malware and its detection is discussed. The next section describes the issues of mobile devices regarding information leaks. The third section deals with related work regarding the chosen analysis method for this master thesis in general, and the last section discusses related work regarding the analysis of Android applications.

3.1 Malware

Mobile phones have evolved from simple communication devices to powerful computers. The growing popularity of smartphones has turned them into attractive targets for malware. The continuing increase of malware for smartphones has forced researchers to think about detection mechanisms for mobile malware and the security of current mobile operating systems.

This chapter describes the historical development and the current situation of mobile malware. Furthermore, it describes the issues of different detection strategies and why it would be better to check applications for their security and not to search for malicious applications. As a consequence, this section is divided into four parts. The first one describes the evolution of malware on desktop computers and finally on mobile devices. Next, a short introduction in common malware detection mechanisms and their problems are described. The last two parts describe further issues, which are especially important for mobile devices, and a new approach which uses the discussed malware detection mechanisms to make assumptions about the security of an application. We also use this new approach in this master thesis.

3.1.1 Malware Evolution

Malware for personal computers really started in the 90s of the 20th century as the Internet and Microsoft Windows came up. Before this era there had been some viruses, which infected the boot sector of floppy disk to spread [Milošević, 2013]. Nowadays, malicious applications are based on the technology developed in the 90s of the 20th century. They use vulnerabilities to infect computers and spread over the

Internet. In 2000, the first trojan horse for a mobile device was reported [Leavitt, 2005]. After four years, in 2004, the first mobile-phone virus was reported [Lawton, 2008]. Probably, the reason why mobile phones were not interesting for attackers over a long period of time is that mobile phones were not more than simple communication devices without any useful information for an attacker. Nevertheless, mobile devices have changed to a fully equipped computer with an enormous amount of personal information [Goasduff and Pettey, 2011].

The problem of current mobile operating systems is that their systems are based on desktop operating systems, which are ported to mobile devices. Basically, at the perspective of functionality and development, the porting of a desktop operating system to a mobile device is a wise decision, but the problem is that current security vulnerabilities are also ported. However, during a complete new development of an operating system, security vulnerabilities will also be created.

Malicious applications for smartphones are increasing rapidly [Schmidt et al., 2009; "Malicious mobile threats report 2010/2011" 2011]. In addition, the amount of personal and sensitive information stored is rapidly increasing on smartphones [Zhou, Zhang, et al., 2011]. Zhou and Jiang [2012] show a growth from 23 malware samples at the beginning of 2011, to 1260 at the end of 2011. One reason for that is the fact that smartphones stores more sensitive information [Rao and Minakakis, 2003]. Schmidt et al. [2009] and F-Secure [2012] show a massive growth of mobile malware, especially of trojan horses for controlling the victim's phone. Consequently, it is getting more and more important to secure sensitive data on mobile devices against those new threats.

The field of research for malware must find more efficient techniques against malicious applications which are continuously increasing in complexity. The following section describes methods how malware can be detected, and how precise such detection mechanisms are. Furthermore, the next sections describe why it is not sufficient to try to detect malware with classic detection techniques like antivirus applications, which are mainly based on static signature detection.

3.1.2 Malware Detection

Applications for detecting malicious applications, and malicious applications have had a similar development. In the early beginning of viruses, the only way to infect computers was through floppy disks. Thus, the first antivirus applications only had to check the boot sector of floppy disk and the executables on the disk, because only an executable file was able to infect the computer. Nevertheless, researcher quickly realised that it is not easily possible to find every malicious code, because protecting the system from some kind of malicious code is as difficult as the halting problem [Cohen, 1984]. Especially, early denial of service attacks are good examples. Listing 3.1 shows such an early denial of service attack. The first routine is used to trigger the malicious code. The malicious code itself is an endless loop. Such simple code was able to freeze computers for a significant period of time by consuming all computing power.

```
1  ...
2  void doDamage( ) {
3    loop: goto loop;
4  }
5
6  void trigger( ) {
7    if (year > 1984)
8      doDamage( ); // Trigger subroutine
9
10   // Otherwise return.
11 }
12 ...
```

Listing 3.1: A typical code used as denial of service attack in the very beginning of the computer era [Cohen, 1984].

Cohen [1988] wrote a paper about defence mechanisms against malicious code. Many of that strategies were picked up by later antivirus applications. Some of these common strategies for detecting malicious code in applications are discussed in the following section.

Detection Methods

Early antivirus applications are based on signature detection. Newer, and more sophisticated antivirus applications are based on heuristic analysis techniques. Nowadays, in combination with the Internet, many antivirus vendors combine an offline with an online approach to achieve a higher detection rate, and to provide an up-to-date database.

Signature-based detection mechanisms have a long history. In general, such applications use a database to compare entire files with malware signatures stored in the database [Landesman, 2012]. In the early beginning of those applications, the signature databases were updated very infrequently. Unfortunately, one of the problems of this technology is that it always takes some time until new virus signatures are inserted in the signature database and distributed to the customers. Therefore, virus authors are always one step ahead of antivirus developers. Another issue of this method is that developers of malicious applications continuously enhance their code to prevent applying signature checks. Such malicious applications consists of oligomorphic, polymorphic or metamorphic code.

Oligomorphic code generates a decryptor for itself by selecting each piece from several predefined alternatives. This pieces are usually too common, and so they can not be detected by signature-based technologies.

Polymorphic code is a piece of code, which changes every time the code is executed. That means, that the underlying algorithm and behaviour stays the same, but the code looks differently after each run.

Metamorphic code outputs a logical equivalent of itself. The differences to polymorphic code, which also outputs a modified version of itself, is that metamorphic code modifies the whole code, which includes the metamorphic engine as well.

Newer and more sophisticated antivirus applications are based on a heuristic approach. Such applications use heuristic analysis to distinguish between malicious and trustworthy applications. Thus, heuristic methods are able to detect new and currently unknown viruses, because they are not based on a signature database or something similar. Heuristic methods can be used in a static and dynamic way [Mishra, 2012]. Nevertheless, heuristic methods are limited in detecting malicious applications. This results from the fact that malicious applications are able to implement identical behaviours in many different ways and it is not possible to definitely categorise every piece of code. If heuristic methods are combined with emulation techniques, then it is possible that the application must run for a long time until the malicious code executes if ever. Furthermore, it still has the same problem that the application must recognise that a malicious piece of code is triggered and executed at the moment.

Further detection methods are cloud-based approaches and online checking solutions. These methods are only improvements of common signature-based or heuristic methods. The advantage of such a hybrid solution is that the signature database can be centralised and need not to be distributed.

Further Issues of Malware Identification

Malware-detection applications must always fight against more and more complex malicious code. Especially, on smartphones it is not easy to detect malicious applications. This results from two primary problems. First, malware on smartphones is younger and antivirus vendors must adapt their antivirus solution for different smartphone operating systems. Secondly, considering the level of resources needed by antivirus techniques and the power and memory constraints of mobile phones, comprehensive analyses are not sufficient on the phone [Burguera, Zurutuza, and Nadjm-Tehrani, 2011].

Section 2.2 describes the security properties of Android, which is currently the most popular smartphone operating system today [Burguera, Zurutuza, and Nadjm-Tehrani, 2011]. This section mainly refers to issues of detecting malicious applications on Android devices.

Beside different code modification techniques (polymorphic, metamorphic, and oligomorphic), development kits for mobile applications support code obfuscation. This technology is used to obfuscate the code of an application to protect intellectual properties, and to make it harder to circumvent copy protections or licence checks. Unfortunately, an obfuscated code has one disadvantage in the context of security analysis. It is more difficult to analyse, and so this protection mechanism increases the effort for security experts to analyse the code of an application.

Another issue of detecting code on mobile phones is the code structure itself. It is possible to pack malicious code into native libraries. Furthermore, it is possible to split the malicious code and to pack it into more than one application to circumvent the permission structure of Android [Orthacker et al., 2012]. For that, two applications, which seems to be benign, request a typical permission to work, but if both applications are installed on the same phone, they will work together and leak information, by using the permissions of each other. Such behaviour is hard to detect, because it would mainly depend on the context in which an application is executed. Furthermore, this is similar for the combination of libraries and applications. It would be possible to act as a regular application and at any time it loads a library from the Internet or as an update from the market and reacts as a malicious application afterwards.

Detection Rates on Mobile Devices

Ching and John [2012] describe an automatic stress test framework for antivirus systems for Android. The result from this paper is that all current antivirus solutions achieve a detection rate of over 90%. However, the detection rate is based on a small set of malicious applications, and slight variants of these malicious applications. On the other hand Zhou and Jiang [2012] describe different types of viruses and the detection rates of four antivirus applications where the worst achieved result of an antivirus program is by 20%. These two papers highly differ in their result. We think, that this large gap occurs, because of the used malware samples. They did not use the same sample set, and thus, that on one set the inspected antivirus applications are more suitable than on the other one. Therefore, on smartphones it is not sufficient to only rely on antivirus applications.

3.1.3 Tolerate Malware

Another approach is to try to secure the system and its applications in a way that malicious applications are not able to harm the system, and are not able to steal any sensitive information. This approach is common on mobile platforms and is enforced by different security barriers.

Those include the use of official markets, where applications can be downloaded, and only installed if they are correctly signed. Here, it is still possible that malicious applications are in the market, and find their way on the device [Zhou, Z. Wang, et al., 2012]. Thus, preventing malware with the help of an official market is not enough. In addition to official markets, Android also allows the installation of applications from other sources.

Consequently, Android must execute applications in sandboxes, where they theoretically can not harm other applications or the system itself. A perfect solution is found if malicious applications and applications with sensitive data are able to coexist on the same device without any information leaks. Operating system developers, security experts, and application developers are continuously improving current systems and applications to achieve a more secure system, where malicious code is not able to harm the system or to steal data of an application.

To protect the data of an application it is possible to use antivirus applications, which try to detect and to delete malware on the phone. Furthermore, there are also information-flow analysis frameworks. The aim of these frameworks is to detect if some applications leak sensitive information. This can be detected, if the pre-defined information ends at specific API calls. Nevertheless, a complete co-existence of malware and applications with sensible information without any negative influences is not currently possible. One reason for that is that application interfaces are always security vulnerabilities and weaken the sandbox principle [Davi et al., 2011].

3.2 Information Leaks

Beside malicious applications, information leaks are a huge threat on mobile devices. Information leaks result not only from malicious applications, because well-known applications also use sensitive data, which they do not necessarily need to run properly. This results from the use of advertising libraries. Gibler et al. [2012] describe that a lot of advertising libraries leak information. Advertising libraries especially tend to retrieve the phone Identifier (ID), which is unacceptable for some user. Another issue which can lead to an information leak is the wide misuse of permissions. Android's permission system is designed to protect access to sensitive data and to protect sensitive API calls. Unfortunately, if an application has more permissions than it really needs and if it has a security issue, then another malicious application will be able to retrieve sensitive information without holding a permission for this information by using components of the benign application [Orthacker et al., 2012]. Such information leaks exist on Android, because of their application architecture.

Barrera et al. [2010] analysed Android's permission model and formulated some possible enhancements. The most important enhancement proposes to make a more fine grained hierarchical permission scheme. Their opinion is that, such a permission scheme improves the user's understanding for the requested permissions of an application and it additionally improve the security of the whole system. A typical example for this idea is the permission to access the Internet[1]. Nearly all applications and especially free applications use this permission and in most cases only to download advertising from a server. With a hierarchical permission scheme an additional permission for advertising[2] is possible. Thus, the security is increased, because such a permission can be used to only grant limited access to the Internet [Barrera et al., 2010].

Felt, Greenwood, and Wagner [2011] discuss over-privileged applications. They analysed the requested permissions of 956 applications. They concluded that all analysed Android applications asked for less than half of the Android permissions and a majority requested less than four. The most common permissions are the permission for accessing the Internet, and the permission to retrieve the GPS position. They also manually analysed the 36 most popular applications, which showed that four of them are over-privileged. Three of them unnecessarily requests the Internet permission, and one application unnecessarily request the permission for accessing the GPS position.

Another common approach to retrieve some sensitive data is not to circumvent some security mechanisms of the system, but to apply a simple phishing attack. Felt and Wagner [2011] analysed such phishing attacks on mobile applications and websites, and found out that they interact in a way which can be exploited by an attacker. They describe four different attack vectors. These four attack vectors

[1] android.permission.INTERNET
[2] android.permission.INTERNET.ADVERTISING

are defined through the role as attacker, or target. In this scenario, a mobile phone, or a web site can be the attacker, or the target for a phishing attack.

Each of these attack vectors can be exploited. Mobile targets can be exploited either by key logging and faked login screens, or website spoofing. On the other side web targets can be exploited either by URL spoofing, or active network attacks.

Many researchers have started to develop detection frameworks which are able to detect information leaks and to track sensitive data through the system, to categorize applications in malicious and trustworthy ones.

3.2.1 Information-Flow Analysis

One issue for detecting malicious applications is, that in many cases the context, in which the application is executed, decides if the application is a malicious one or not. Typical applications for such a dual-behaviour are position tracking applications. In general, these applications are trustworthy, but if another person as the owner of the phone installs the position tracking application, then such applications will typically be malicious. However, information-flow analysis techniques can not solve such categorisation problems, but they improve the security by observing sensitive information, and by detecting if sensitive information leave the phone. Therefore, this technology can also detect unexpected behaviour of benign applications. This section describes related work about information-flow analysis.

One common information-flow analysis framework for Android is TaintDroid [Enck, Gilbert, et al., 2010]. TaintDroid adds a tag to sensitive data and tracks the flow of this data through the system. For that TaintDroid modifies the Dalvik VM to apply the tag propagation algorithm. This tracking system is able to track sensitive information like the device ID. Thus, it can detect information leaks from applications. Unfortunately, by design it is not possible to apply TaintDroid on a regular phone, because it needs to exchange the system image by a modified one. Furthermore, it is currently very easy to circumvent the tainting mechanism, because TaintDroid is only able to trace explicit data flow and not implicit data flow. Therefore, sensitive data can be easily decoupled from the taint that it is not tracked anymore. We analysed this behaviour with different implicit data flow techniques. Listing 3.2 shows an obvious solution of such an indirect data flow.

Another approach to circumvent the taint tracking mechanism of TaintDroid is to decouple the taint and the data by using system processes. We tested this solution by writing sensitive data to an arbitrary file and then reading it in through a Java process, which executes the Unix command `cat` to read the data from the file again. This mechanism opens a new process and by reading its output stream the taint gets lost, because the process for `cat` does not run in the virtual machine, and thus, the tag propagation mechanism of Taintdroid does not retrieve the taint.

```
1  ...
2  String trespasser = "";
3  for(int i = 0; i < imei.length(); i++)
4    switch(imei.charAt(i)){
5      case '0': trespasser += '0'; break;
6      case '1': trespasser += '1'; break;
7      case '2': trespasser += '2'; break;
8      case '3': trespasser += '3'; break;
9      case '4': trespasser += '4'; break;
10     case '5': trespasser += '5'; break;
11     case '6': trespasser += '6'; break;
12     case '7': trespasser += '7'; break;
13     case '8': trespasser += '8'; break;
14     case '9': trespasser += '9'; break;
15     case '-': trespasser += '-'; break;
```

```
16          default : break;
17        }
18      ...
```

Listing 3.2: A simple example how the tainting mechanism from TaintDroid can be circumvented.

ScanDroid has a completely different approach. It statically analyses data flows through Android applications, and can make security-relevant decisions automatically, based on such flows [Fuchs, Chaudhuri, and Foster, 2009]. In particular, it decides if an application is safe to run with certain permissions, based on the permissions enforced by other applications. The framework is based on T. J. Watson Libraries for Analysis (WALA), which is a collection of open-source libraries for Java code analysis. Furthermore, the analysis is based on Java source code, so it is not immediately possible to apply the framework on Android applications. Another property of that system is that it also finds security violations in benign applications. The problem is that many benign applications also have one of the defined security specifications. For example, if a mail reader application has the permission to read and write contacts as well as to send and receive messages, then this application breach a security specification of ScanDroid and the system falsely detect a security violation.

A similar detection framework to ScanDroid is AndroidLeaks [Gibler et al., 2012]. This system extracts the code from the APK and reverse-engineers it to a JAR. This step is done by some external tools – like dex2jar. Afterwards they used WALA[Center, 2011] to build a call-graph, and to statically analyse the resulting Java bytecode for a connection between a pre-defined source and sink. This is done in a two-step procedure. First the application must include the API call for the source and the sink, and then the static analyser searches for a connection between the found sources and sinks. One of the potential sources is the device ID and one of the potential sinks is the network interface. Both can be detected by a specific API call. They analysed 24350 applications with this framework in 30 hours. The evaluation shows, that nearly every advertising library leaks Android's phone data, and location information. Nevertheless, a property of this framework is, that it can not be executed directly on a phone. Thus, an application must always be reverse-engineered until the framework is able to classify it. Furthermore, that means, that a non-technical expert can not easily use this tool to quickly analyse their installed applications to find potential information leaks.

For Apple's iOS Egele et al. [2011] wrote a framework to detect privacy leaks in iOS applications written in Objective-C. This framework is based on IDA Pro, which is a disassembler. The analysis itself is a plug-in for the IDA-python interface. IDA Pro is used for the reverse-engineering part and the plug-in builds the Control Flow Graph (CFG) and does the analysis based on the CFG. To find a potential privacy leak, the framework performs a reachability analysis. More precisely, they check the graph for the presence of paths from sources to sinks. In this environment a source is defined as a function, which retrieves sensitive data from the system and a sink is defined as a function, which is able to transmit data over the network. The evaluation is done with 1407 iOS applications. They found out that the device ID is leaked most frequent. Nevertheless, from 1407 analysed iOS applications only 195 applications leak the device ID. The second most important information leak is the location, which is on iOS not really a dangerous leak because iOS warns users whenever an application tries to access the fine location [Egele et al., 2011].

3.2.2 Information Protection

Beside information-flow analysis techniques it is also important to do some research in the area of information protection. There is the focus to protect sensitive data and to increase the privacy on mobile devices. This section describes related work from this research area.

Beside TaintDroid, MockDroid [Beresford et al., 2011] is a system for privacy enhancing on Android devices, but with a slightly different aim. The aim of MockDroid is to increase the privacy by spoofing API calls for certain applications, where TaintDroid tries to track a specific piece of information, and to informs the user about information leaks. Beresford et al. [2011] developed MockDroid, which is a modified Android operating system with some additional functionality in terms of privacy and avoiding information leaks. MockDroid mocks certain API calls in a way that applications do not recognize that the underlying system rejects every request. For example, if for an application the network interface is mocked, then a socket never connects and always throws a timeout exception. Such an exception is common and an application can not realize that the access has been rejected. Nevertheless, this system has similar issues as TaintDroid. Hence, it needs to replace the installed Android system by the modified one. Additionally, MockDroid adapts Android 2.2.1 which is out of date and could be vulnerable to other security threats regarding the old Android version.

Taming Information-Stealing Smartphone Applications (TISSA) is similar to MockDroid. They also modified an Android version and replaced Android's permission scheme [Zhou, Zhang, et al., 2011]. They extended Android's permission scheme with additional permissions to enforce a better privacy on the system. For each application, the user can choose, if the application retrieves real-world data from the system or faked ones. Thus, it is possible to only grant certain applications access to sensor values or to sensitive information even if the permission for the usage is granted during the installation. For the correct use of this framework, the user needs a deeper knowledge about the permission model and the effects of providing faked or real values for an application. A typical use case for such a framework is the use for applications with advertising. To avoid the leak of the device ID, the device ID can be replaced by a dummy one, which leads to non-personalised advertising. Nevertheless, TISSA is based on Android 2.1, so it is older than MockDroid. Thus, it is not meaningful to use this system anymore, because such an old Android version is vulnerable to other security threats.

Hornyack et al. [2011] used data shadowing to protect sensitive information against unauthorized applications. If an unauthorized application requests the location, or the device ID, then it always gets the same pre-defined position and it always gets a spoofed device ID. If the resource is stored in a database and accessed by a Uniform Resource Identifier (URI) then it will create a shadow database and a new cursor to the new database. Thus, a user can define, which applications operate on real-world data, and which ones operate on spoofed information. In contrast to TISSA, this framework does not replace Android's system image, but it only works on rooted devices, because Android's core libraries and framework must be modified.

3.3 Security Analysis

Static security analysis, especially with the help of static slicing, is a well-known technique to analyse the security of applications. It is primarily used for information-flow analysis, because static slicing can be used to over-approximate information flows in a program in order to ensure information-flow security [Pistoia et al., 2007]. This section presents a survey of related work for analysing applications with the help of static as well as dynamic analysis methods.

Static slicing is to extract code snippets of a program based on certain conditions, to generate a subset of the whole program. This mechanism can be used for various scenarios. It can be used to classify applications based on their code, by using static slicing to build a System Dependence Graph (SDG), and analyse it, based on the so-called tell-tale signs [Bergeron, Debbabi, Erhioui, et al., 1999]. The tell-tale signs are defined as properties of a program that can be used to discriminate between malicious and benign programs [Lo, Levitt, and Olsson, 1995]. In general, tell-tale signs are various security properties of a program. It is also possible to make assumptions about the behaviour of benign applications. Tell-tale signs must be simple enough, so that their identification can be automated and must be fundamental enough, so that certain security relevant actions are impossible without showing tell-tale signs. Lo,

Levitt, and Olsson [1995] categorise tell-tale signs in the following three groups:

Tell-tale signs identified by program slicing are file operations, process creation and execution, network access, change of privileges on the file system, race conditions, and input dependent system calls.

Tell-tale signs based on data-flow information are anomalous data flow. For example, such an anomalous data flow is opening and writing to executable files.

Tell-tale signs based on program-specific information are based on the authentication process of a program. If a program has an authentication process, like a login screen, then the input values must be tracked.

Secure information flow is also important in the context of web applications [Pistoia et al., 2007]. Therefore, a number of information-flow analysis approaches handle network-based traffic. For example, Myers and Liskov [1997] discuss the use of static as well as dynamic analysis to enforce information flow policies for applications. The idea is that each user defines an information flow policy at the level of individual data items, to protect sensitive data. On the other side Newsome and Song [2005] propose a dynamic tainted-variable analysis that catches errors and detects malicious behaviour by monitoring tainted variables at runtime. In detail, they taint all variables, which are derived from untrusted sources, such as the network interface. If a variable is used in a dangerous way, then an attack is detected in this scenario. A dangerous behaviour is defined by:

- Overwrite variables with jump addresses, to redirect control flow.
- Access to unauthorized memory regions.
- Access to unauthorized system calls.

Furthermore, Wagner and Dean [2001] propose an approach for intrusion detection via static analysis. They want to detect if an application is penetrated and then exploited to harm other parts of the system. To achieve this, they defined specifications of expected application behaviours, and monitored the actual behaviour to see whether the behaviour deviates from the specifications or not. They also mentioned that it is still possible for attackers to circumvent the detection, if they do not cause any harm, because then they do not interact with the operating system in a way the framework is able to detect.

Beside the analysis of web applications and network communication, static analysis can also be used to detect security-correct programs [Snelting, Robschink, and Krinke, 2006]. They make the observation that Program Dependence Graph (PDG)s and non-interference are related. Thus, in a security correct program a statement s_1 must not be in the backward slice of another statement s_2, if the predecessor set of s_1 is not a subset of the predecessor set of s_2.

Another common vulnerability for applications is the insufficient protection against buffer overflows. This attack vector is ranked as the top vulnerability in RPC services to UNIX-systems [*The twenty most critical internet security vulnerabilities.*]. Such vulnerabilities are easy to exploit and step-by-step guides are available, to construct such exploits [One, 1996]. Additionally, one widely used language – `C` – is highly vulnerable, because there are among others several library functions that manipulate buffers in an unsafe way [Larochelle and Evans, 2001; Wagner, Foster, et al., 2000]. Ganapathy et al. [2003] propose one approach to prevent such attacks. They use CodeSurfer [Horwitz, Reps, and Binkley, 1990; Reps et al., 1994], a tool to produce inter-procedural slices with the ability for forward and backward slicing from a program point. Next, they use the resulting slice from CodeSurfer to find declarations of buffers. This declarations are used to formulate constraints, which define the size of the buffers. Furthermore, the slice is analysed, if any of the statements break the constraint by accessing the buffer with a higher index as the constraint permits.

Furthermore, it is also possible to find dead code with static slicing [Benton, 2004]. Dead code is a code, which is never executed, because it does not have a connection to a potential entry point of the program. Thus, this code can be removed. Bergeron, Debbabi, Desharnais, et al. [2001] proposed a solution for finding and removing dead code based on the SDG. After building a SDG, dead code is defined as a not-connected sub-graph, because each of the not-connected sub-graphs does not have any data- or control-dependence edges to the entry point of the program [Bergeron, Debbabi, Desharnais, et al., 2001].

3.4 Security Analysis for Android

Modern mobile phone operating systems must deal with many different attack vectors. Beside Apple's iOS and Windows' mobile phone operating system, Android allows more access to system components by third-party applications [Lettner, Tschernuth, and Mayrhofer, 2012]. This can lead to vulnerabilities, but it can also lead to a higher security of Android [Shin et al., 2009].

Mobile phones have many different vulnerabilities, based on their used software and their included hardware components [Becher et al., 2011]. Becher et al. [2011] classify potential attack vectors into four categories.

Hardware-centric attacks target the mobile device itself. These are primarily attacks, which directly attack the physical hardware, and attacks on the smartcard communication.

Device-independent attacks target the communication channels of the device.

Software-centric attacks target vulnerabilities in installed applications.

User layer attacks contain every exploit that is not of technical nature. Many of today's mobile malware samples are not based on a technical vulnerability, but trick the user into overriding technical security mechanisms [*State of cell phone malware in 2007 (2007)*].

Thus, it is important to provide more information about an application and its potential risk to the user, to reduce user layer attacks. Beside the different attack vectors, the different attack channels are important as well. Guo, H. J. Wang, and Zhu [2004] discuss various ways in which a smartphone could be compromised. They defined three general attack channels.

- Attacks from the Internet [*Summer Brings Mosquito-Born Malware*].
- Infection from compromised PC during data synchronisation [*Windows Mobile-based Smartphones*].
- Peer smart-phone attack or infection [Labs, 2004; *SMS Killer*].

Attacks from the Internet are simple, and efficient user layer attacks, because in most cases the user downloads and installs applications from third party markets, or suspicious websites. Unfortunately, these sources include many malicious contents [Zhou, Z. Wang, et al., 2012].

Another interesting attack channel is the peer smart-phone attack. A common attack uses an infected phone to spread malicious content by sending SMS to all contacts in the address book. This malicious content can be an advertising message, a premium rate message, or a malformed SMS to enforce malfunctions at the receiving phone [*SMS Killer*].

Thus, it is important to detect malicious applications and to differentiate between malicious and benign applications [Felt, Finifter, et al., 2011]. A lot of related work deals with malware and detection strategies of malware. These works discuss current detection algorithms, and their improvements for detecting malicious applications. A few of them deal with the classification of applications into malicious

and benign applications, and with their security properties. These works primarily discuss security properties, and the behaviour of applications to define benign application. This master thesis contributes to the second category and also deals with security properties and classifying applications by these properties.

Related work for malware detection is discussed in a previous section. This section discusses related work regarding the analysis of applications, the classification of applications, and related work regarding defining security properties for applications and mobile operating systems.

To that effect, Grace et al. [2012] discuss issues with pre-installed applications on stock Android. Those applications leak private information, because of the way the private information is managed and accessed [*App Attack-Surviving the Explosive Growth of Mobile Apps*]. As opposed to Apple's iOS, Android does not use run-time approval for access to sensitive information, it uses a permission-based model, that grants a set of permissions to an application at install-time. Nevertheless, the problem of over-privileged applications is not limited to pre-installed applications. Many third party applications request too many permissions as well [Felt, Chin, et al., 2011]. Thus, it is essential to analyse the inter-process communication as well. Chin et al. [2011] and Felt, H. J. Wang, et al. [2011] analyse inter-process communication of Android applications. They concluded, that Android's message passing system supports the creation of rich, collaborative applications, but it also introduces the potential for attacks, if developers do not take precautions. Thus, developers must be aware of outgoing, and incoming communication risks. Potential outgoing communication risks are broadcast theft, data theft, result modification, and activity and service hijacking. Potential incoming communication risks are malicious activity and service launches, and broadcast injections [Chin et al., 2011].

One solution regarding information leaks, is to block the installation of potential unsafe applications [Enck, Ongtang, and McDaniel, 2009a]. They propose security rules to differentiate between malicious and benign applications and to only allow the installation of benign applications. The defined security rules are potentially able to defend against complex attacks, but not for malicious applications in general. This results from the fact, that many applications use the same set of permissions as malicious applications. A typical example is Facebook, which uses the permission for accessing the address book and the permission for web-access. This constellation can also be used from malicious applications to steal contact information from the phone. Thus, such malicious applications can not be detected by the proposed security rules.

Ongtang et al. [2012] propose a similar solution, to avoid information leaks. They developed a modified Android operating system, that allows developers of applications to define fine-grained security policies, for their application components. Thus, application components can not be hijacked or misused by other applications, if the security policies for the components are strong enough.

Dietz et al. [2011] contribute to the problem of information leaks with a framework, that tracks the call chain of inter-process communication on the device. To that end, they extended Android's Java runtime libraries and the IPC system. With this modification, applications can operate on the components of other applications in two different modes. The user can decide whether an application, which accesses components of another application, is allowed to also use the permissions of another application, or not. Therefore, with this mechanism, exploiting of over-privileged applications is not possible anymore.

Another proposal is to increase the sandboxing mechanism of the operating system, to isolate applications for different usage environments in different virtual machines. Andrus et al. [2011] propose an approach with so-called virtual phones. Each virtual phone is a lightweight Android environment, which is totally decoupled from other virtual phones. These virtual phones are configured on a PC and downloaded to a phone. On the phone the different virtual phones can not be modified anymore, and a regular application is used to switch through the installed virtual phone. Lange et al. [2011] propose a solution, that uses a microkernel instead of Android's monolithic architecture, because they think, that the primary security threat of Android is its monolithic kernel architecture.

3.5 Code Analysis Frameworks

The idea of reversing an Android application to analyse the code of the application is not new. In most cases the analyses is done to find malicious applications. The basic principle is to separate malicious applications from trustworthy applications by classifying them into categories. Furthermore, reversing an Android application can also be used to apply security analyses. Other frameworks can be used to add, or remove instructions of an application, and to repack and reinstall the modified application on the phone again. This section describes a small subset of those frameworks.

3.5.1 ADAM

ADAM is an automated system for evaluating the detection of Android malware. It provides an environment for the automatic evaluation of anti-virus products. This is done by gathering malware samples, and buildings variants of these malware samples by reverse engineering them, adding a method in the Smali-code, and repack them. The original malware samples, and their variants are used to evaluate commercial online and offline malware detection toolkits [Ching and John, 2012].

3.5.2 Andromaly

Andromaly is a framework, which tries to detect suspicious behaviour of the phone. The assumption is made that such a suspicious behaviour results from malicious applications. The basis of the detection process consists of real-time, monitoring, collection, and analysis of various system metrics, such as CPU consumption, number of sent packets through the Wi-Fi, number of running processes and battery level. After collecting the system metrics, a detection unit tries to find suspicious behaviour on the phone with the help of machine learning [Shabtai, Kanonov, et al., 2012].

3.5.3 Apktool

Apktool[3] is a framework to decompile APK files and then recompile them again. The framework does not decompile the program back to Java it uses the intermediate language Smali instead. Smali[4] is used by Apktool to represent the program. Smali is a low-level language which is loosely based on Jasmin's dedexer's syntax. Each Smali file represents a class which can be modified or analysed.

Singh and Garg [2012] use Apktool to automatically search for repackaged applications. Such applications are applications, which are reverse engineered and compiled, packaged, and signed again. Furthermore, Helfer and Lin [2012] and McClurg, Friedman, and Ng [2012] use Apktool to analyse Android applications in terms of their security. The first framework uses it to remove permissions from applications, and the second uses it to detect privacy leaks by adding taints to various instructions in the application. After recompiling and installing it, the taint is used to log the access to sensitive code snippets. Every time the application accesses such code snippets the user gets a notification.

The framework of this master thesis also uses the parser of Apktool to reverse engineer an application to search for information leaks.

3.5.4 Dex2jar

Dex2jar[5] is a framework to reverse engineer APK or DEX files to JAR files. Afterwards, JD-GUI[6] can be used to retrieve Java files from the archive. Nevertheless, the resulting Java files are not suitable for

[3] https://code.google.com/p/android-apktool/
[4] https://code.google.com/p/smali/
[5] https://code.google.com/p/dex2jar/
[6] http://java.decompiler.free.fr/?q=jdgui

applying code analysis techniques, because it is possible that the framework does not correctly parse every method. Furthermore, if an application should be also recompiled, then other frameworks would be better for code modifications than this one, because it is not easily possible to recompile a decompiled application.

This framework is used among others in Rhee, Hawon Kim, and Na [2012] for analysing the security of a mobile device management (MDM) systems and in Sharma et al. [2013] for analysing Android malware.

3.5.5 Ded

Ded is similar to Dex2jar. It is a reverse engineering framework, which decompiles DEX files to Java byte code, but also optimize them [Enck, Octeau, et al., 2011]. Listing 3.3 shows an optimisation of Ded in contrast to Dex2jar.

```
1  Decompiled and optimized with Ded      Decompiled with Dex2jar
2
3  double return_a_double(int var1) {     double return_a_double(int var1) {
4       double var2;                        long var2;
5       if(var1 != 1) {                     if(var1 != 1) {
6         var2 = 2.5D;                        var2 = 4612811918334230528L;
7       } else {                            } else {
8         var2 = 1.2D;                        var2 = 4608083138725491507L;
9       }                                   }
10
11      return var2;                        return (double)var2;
12    }                                   }
```

Listing 3.3: Ded vs. Dex2jar

3.5.6 Dedexer

Similar to Dex2Jar, Dedexer is able to decompile DEX files, but this framework decompiles the code to an assembly-like format. The format is similar to the Jasmin syntax, which is an assembler for the JVM, but contains Dalvik opcodes. Thus, the Jasmin compiler is not able to compile the generated Dedexer files.

3.5.7 ComDroid

ComDroid is a framework, which analyses the communication between Android applications. For that it uses Dedexer to decompile Android applications. Next it parses the result and logs potential component, and intent vulnerabilities. Such vulnerabilities are basically defined as sending and receiving mechanisms, without protection mechanisms. The protection mechanisms are defined through a permission map, and the sending and receiving mechanisms are Android's specific API calls. Data can be propagated by sending a broadcast, or starting an activity or service. These three components are also the only components for receiving information from other applications. If a vulnerability is found, then ComDroid issues a warning Chin et al. [2011].

3.5.8 Paranoid

Paranoid is a two-component system. Firstly, it records input data from applications on several phones and replays the situation on an emulator on a server. The first step ensures that the system image of the

mobile phone must not be modified, but dynamic taint analysis can be applied. Secondly, dynamic taint analysis is applied on the replica on the server. With this mechanism all data from a suspect source are tainted and tracked through the system [Portokalidis et al., 2010].

3.5.9 SAAF

SAAF is a static analyzer for Android's APK files. The main feature is the ability to calculate program slices for arbitrary method invocations and their corresponding parameters. SAAF will then calculate a slice for this so called slicing criterion and search for all constants which are part of that slice [*Static Android Analysis Framework* 2013].

This framework was developed during the development of our framework. Therefore, we were not able to use this framework for our purposes.

Chapter 4

APK Analyser

" The art of simplicity is a puzzle of complexity. "

[Douglas Horton]

This chapter describes the architecture and the limitations of the framework. The first section describes the general architecture of the framework, and its core components. The next section describes the tracking algorithm, the implementation properties, and the restrictions. Afterwards, the modules, which define the start, and end criteria of a slice, are described. The last section discusses the format of the generated results of the framework.

4.1 Architecture

The framework includes two parts. Firstly, the execution management, and secondly the execution environment. The execution management is used to handle Android-specific User Interface (UI) behaviours, because Android's UI design has certain properties, which must be considered for correct use.

Beside, the execution management the analysis of the framework is encapsulated in the execution environment. Basically, this part is built on top of an asynchronous task. The execution environment ensures that all parameters are correctly initialised, inspects the analysis, sends progress updates to the execution management, and generates the result logs.

Figure 4.1 illustrates the interaction between the two core parts. The following sections describe the architecture of the two parts, and its components in more detail.

4.1.1 Execution Management

The execution management primarily handles the user interaction with the application. Thus, this part encapsulates all UI classes, and defines the interfaces to the execution environment.

The UI of the framework is based on Android's Fragment architecture. Android uses Fragments to spilt the UI in small reusable modules. Those modules can be loaded from activities to interact with the user. The idea behind the Fragment activity concept is to reduce the development effort for different display sizes. Similar to an activity, a Fragment can exist in three states:

Resumed: The Fragment is visible.

Paused: Another activity is in the foreground, or has the focus, but the activity in which this Fragment lives is still visible.

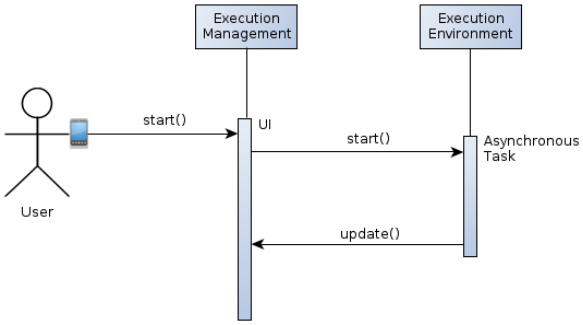

Figure 4.1: The figure shows the interaction of the two core parts of the Apkanalyser, the execution management, and the execution environment. The Execution management handles the complete interaction with the user, and the execution environment encapsulates the analysis in an asynchronous task.

Stopped: The Fragment is not visible.

The problem with these states is that Fragments and activities are re-created when they are paused and resumed. This always occurs when the user rotates the device or turns the screen off and on. To proper handle callbacks, the Fragment instances must retain state across activity re-creation. Additionally, the activity reference from the execution environment must be also updated.

Thus, to correctly handle notifications from the execution environment, the framework implements a special callback structure. Every time the activity changes its state, the context is updated. In our framework, this is done by registering a special Fragment, which is used for context retrieval. This Fragment is *detached* and *re-created* every time the activity is re-created. This Fragment connects the asynchronous task of the execution environment to the execution management. Additionally, it must hold a list where the task manager of the execution environment can register notifications. The notifications are forwarded to the UI thread, when the underlying activity gets ready. This must be done, because of the fact that Android recreates the activity after every UI event, and thus the application must wait until the UI is correctly rebuilt.

4.1.2 Execution Environment

In addition to the execution management, the application must also provide an execution environment to efficiently execute the framework and handle potential errors. This execution environment can be split into three parts:

- Module management
- Execution optimization
- Result logging

Module Management

A module defines sources, and sinks for the slicing algorithm, and is managed by the module factory. The module factory extends the enumeration class of Java. That means, that each module is a static, final instance of this class, and can be implicitly accessed by its name. This approach has advantages as well as disadvantages. An advantage of this approach is that the creation, and registration of a module is simple, because these steps are handled by the enumeration class.

A disadvantage of this approach is that the modules are created during the start of the application. Thus, all modules exist and consume memory, which is relatively limited on mobile phones, until the application is closed. Nevertheless, due to the architecture of our execution mechanism, the wasted memory is insignificant, because only the module classes themselves are allocated in the memory. The references to other classes only exist during an analysis. Thus, the garbage collector is able to free the memory of all classes, except the module classes.

Execution Optimization

Another concept of the execution environment is the execution optimizer. To improve the performance of the framework, certain code should not be analysed. For example, the linked Android libraries, or linked cryptographic libraries must not be analysed. This is done to improve the performance in terms of the run-time, and in terms of the accuracy of the analyser, because otherwise more false-positives occur.

The optimizer uses uses a blacklist, a whitelist, and a manifest analysis to examine relevant package names. The manifest analysis retrieves all package names of an application. Only classes that are part of one of the retrieved package names, and package names on the whitelist will be analysed.

Thus, foreign package names, which are not in the subset of the examined package names are not analysed. However, this method is used to remove third party libraries, and Android's libraries.

Result Logging

The logging architecture of the analyser consists of two parts. Firstly, the execution environment itself is able to write a general result file about the analysed application. This contains information from the manifest, as well as execution times, and calculated results. Secondly each module is able to write Smali-notated files, which represent the specific slice of the module.

4.2 Register Tracker

The Register Tracker is the core of the framework. It includes the static slicing algorithm for Android applications. Static slicing is a well-known technique that can be used for tracing variables. The concept of static slicing is described in Section 2.5.

In the context of this work, variables are traced on top of the Dalvik bytecode. This section describes the slicing criteria of the framework for slicing Android applications and which trade-offs are made.

4.2.1 Slicing Android Applications

The algorithm, which is used for the framework, is based on forward slicing, but has certain slight modifications to fulfill the requirements of slicing Dalvik bytecode in an efficient way. Additionally, to classify applications in various security categories, only explicitly influenced variables are added to the slice. This results from the fact that if an implicitly influenced variable is added to the slice, then the implemented detection mechanism does not work. The reason for that is that if a specific register is traced through the program, and if implicitly influenced registers are also added to the slice of the specific register, then our algorithm will potentially wrongly detect sinks, because of the definition of implicitly influences. For example, if an influenced register is part of a condition, then the influenced register implicitly influence all statements of the resulting branch.

The register tracker class implements the slicing algorithm and is build on top of the parser of the Apktool framework (see Section 3.5.3). The parser of this framework reads the DEX file and parses the code based on its instruction formats (Section 2.3 describes the Dalvik bytecode in more detail). Each Dalvik instruction format, is represented as a Java class in our framework. The slicing algorithm

works on top of these classes. Thus, for identical, or similar instructions with different formats, these instructions must be brought together and different instructions with identical instruction formats must be separated to handle all instructions efficiently.

Furthermore, to correctly handle the register-based architecture of Dalvik, certain criteria to handle the different instructions, the sources, and the sinks must be defined. The source of a slice is basically a single register, which is observed by a module. Thus, the module (The modules are discussed in Section 4.3) inspects the code until a suitable source is found, and then it starts the register tracker, based on the found register. However, it is possible that the so-called start condition consists of a specific set of instructions in a specific order. This condition can be fulfilled by the register tracker itself. If a module is based on such a start condition, then it starts a slice at every possible occurrence of the first instruction of the starting set, and checks if the built slice fulfills the criteria of the starting set. If it is part of the slice, the register tracker is reseted and the slice with the correct start condition can start. The starting conditions of every module are described in more detail in Section 4.3.

To properly handle class, and method changes, the register tracker must provide a structure to store influenced classes, their influenced methods and their influenced variables. The slice itself is a set of influenced classes and each class holds the class-specific elements, like methods, instance variables, and static variables. Additionally, each method is divided into several branches and each branch consists of an influenced register set. If the method changes, then the register tracker will create a newly empty method, with one root branch for this method. In addition, if the current method has some influenced instructions, then this method will be added to the containing class, and each influenced class is added to the slice when the class changes.

The slicing algorithm works on these branches, but the detection patterns of the module operate on the complete slice with its class structure.

To decide if a single instruction is added to the slice or not, the algorithm consists of a set of criteria to handle different instruction classes. An instruction class is a set of instructions, which are not identical in terms of their format, but their operation. For example, Dalvik consists of several different move operation, where each move operation basically is a data-flow from one register to another one. The instructions can be divided into the following instruction classes.

- Benign statements
- Assignment operations
- Conditions and jumps
- Method invocations
- Return statements
- Operations on instance variables
- Operations on static variables
- Operations on arrays

Benign statements are all statements that do not have any influence on the slice. Obviously, `nop` is such an instruction, and beside *nop*, instructions for exception handling are not part of the slice. We consider it unlikely that the code we analyse uses implicit information flow, like those generated by exceptions. The complete instructions list of this category consists the following instructions:

- `check-cast`

- `monitor-enter, monitor-exit`
- `throw`
- `nop`

Assignment operations are simply all operations that assign a register. Many instructions fall in this category. Compare, arithmetic, binary and object generation instructions are in this category, as well as typical assignment and move operation. The algorithm must decide if a statement is added to the slice or not. This decision is based on the definition of the slicing algorithm of Section 2.5.

The previously defined slicing criterion $C = (p, V)$ consists of a program point p, and a list of variables V. For Dalvik, V is a list of registers. Thus, the slice S with respect to C is a set of statements, which are influenced by V. A statement is added to S if the rules for $inSlice(n)$ are fulfilled.

To update the registers in the influenced set $Infl(n)$ the algorithm must consider two rules. Firstly, if the intersection of the reference set $REF(n)$ and the incoming set $in(n)$ is not the empty set, then the used register will be added to the influenced set. Secondly, if the influenced set contains the used register and the intersection of the referenced set $REF(n)$ of the current instruction with the incoming set $in(n)$, is the empty set then the register will be killed and not further tracked. Listing 4.1 shows these two rules.

```
1  Infl(n) = Infl(n) ∪ v|v ∈ DEF(n) ∧ (REF(n) ∩ in(n) ≠ {})
2  Infl(n) = Infl(n) \ v|v ∈ S_C ∧ (REF(n) ∩ in(n) = {})
```

Listing 4.1: These two rules are applied to properly handle assignment instructions.

Operations on instance variables are based on `put` and `get` operations. Instance variables are similar to assignment operations, but with one additional property - they have a specific scope and lifetime. Obviously, it is impossible that a variable is able to violate the limitation of the scope or lifetime. Nevertheless, the algorithm must be aware of this, because a typical instance variable is generated in the constructor and then it is reused in different methods and classes. To handle instance variables, the register tracker has a set of influenced instance variables for each influenced class. Adding variables to the slice is based on the rules of assignment operations. In addition, the slice can grow very fast if instance variables are tracked. Therefore, it is possible to deactivate the tracking of instance variables in our framework.

Operations on static variables are similar to operations on instance variables, but it is not sensible to track static variables because they are very often used for debug-level flags. This fact can lead to a large slice, because if an influenced register is logged, the register of the debug-level flag will be influenced and this results in a slice where every occurrence of the specific debug-level flag is also in the slice. Therefore, the algorithm is able to track static variables, but this feature is deactivated by default.

Operations on arrays are similar to operations on static or instance variables. For static slicing, it is not easily possible to compute the index of a specific element or the element itself in an array. Thus, it is hard for static slicing algorithms to only track the specific element in an array. A common approach, which is also used in this framework, is to handle arrays as a single variable and not as a container. The result is that the complete array and every operation on the array will be tracked, and not only the specific element in the array and operations on that element.

Method invocations are slightly more complex than assignments of variables, and operations on them. If an influenced register is passed as an argument to a method, then the following criteria must be considered:

- The register is potentially not the same in the invoked method.
- The correct method must be found.

The first n registers of a method are local registers and the first parameter register is the n^{th} register. For tracking a parameter register, the algorithm calculates the offset of the parameter register and adds it to the slice.

However, a method can be invoked on a base-type, on an interface or on a derived class. Unfortunately, during a static analysis it is not easily possible to find a specific class based on an interface or a base-class, because for that the instantiation of the object must be found first. To find the correct method of the invocation, the register tracker uses a slight simplification. The tracker inspects each method in every class that fits to the signature of the invoke instruction. Thus, if the invocation is done on an abstract method, then the tracker will slice all implementations of the abstract method, which are found.

Return statements are the logical sequel of method invocations. If a register of an invoked method is influenced, and the method has a return value, then the register of the return value must also be tracked in the calling method.

However, a register, which is returned from a method, is stored in a hidden register first. A special move instruction retrieves it and stores it in an accessible register. Therefore, a return statement is identically handled to an assignment instruction, with the addition that the algorithm must be aware of the scope.

Conditions and jumps are the most complex code structures for static slicing. Basically, the slicer inspects the instructions in the order as defined in the bytecode. If a condition or an unconditional jump occurs, then the slicer calculates the offset address, where the new branch starts, and adds the new branch to the tracking set. After the branch instruction, a recalculation of all branches is done, when the newly added branch starts before the current instruction. That means, if the calculated offset is negative.

It is also possible that more than one branch is active at the same time. Each branch lives until a return, or unconditional jump statement occurs. Nevertheless, by design it is not possible that two branches are identical, but it is possible that two branches visit the same instructions. Figure 4.2 illustrates this behaviour. The first branch starts at instruction number 1 and inspects each instruction until instruction number 3. At instruction number 3, a new branch is added, which starts at instruction number 6. Next, the first branch continues with instruction number 4 and 5. Instruction number 5 is an unconditional jump, which closes the first branch and creates a new branch, which contains the instructions from 3 to 5. This branch is immediately recalculated, because it starts before the current instruction and the subsequent branches are updated. Thus, the second branch is updated, because this is the branch, which is generated by instruction number 3. At instruction number 6 the second branch becomes active until instruction number 8 returns to the calling method. In this example branch 2 is a subset of branch 0, and thus they visit the same instructions, but they are not identical.

4.2.2 Issues with Static Slicing

Furthermore, to be able to slice Dalvik bytecode, various issues must be considered. These issues result from the Dalvik architecture, and the slicing algorithm. This section discusses the following issues, and how we considered these issues in the framework:

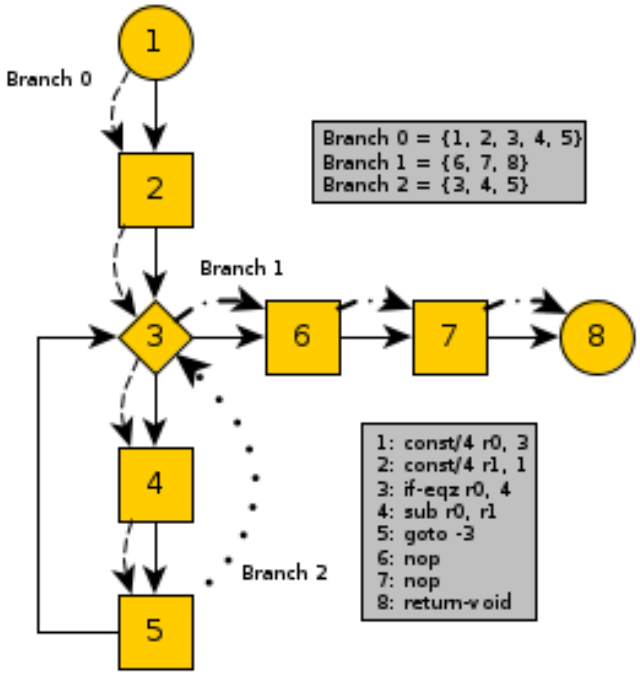

Figure 4.2: Illustration of the Generation of Branches for the Register Tracker

- Concurrency
- Aliasing
- Callback mechanisms
- Native calls
- Exceptions and reflections

Concurrency: Android applications frequently use concurrency, because Android's application concepts enforce this. Each application on Android has its own UI thread. This handles the callbacks from the operating system when an interaction with the application occurs. Nevertheless, Android restricts the runtime for every UI thread. That means, that each callback to a user input must not run longer than 5 seconds and each callback to a broadcast receiver must not run longer than 10 seconds [*Android Developer Guide - ANR* 2013]. Thus, it is not possible to execute a long-running operation on the UI-thread.

To circumvent this issue, threads must be used. This can be done by using Java threads or Android's specific concurrency classes – services, and asynchronous tasks. The algorithm must be aware of the concurrency and must be able to correctly detect information-flow in threaded applications.

The detection of Java threads is simple, because Java threads are based on two steps:

1. Initialise the runnable object.
2. Trigger the system to start the thread.

This two-step execution is simple to trace, because all data on which the thread operates on are passed to the thread by step one, which is typically the constructor of the object. In addition, the second step can

not be tracked, because the invoked method differs from the executed method. The connection between these two methods is done by the system, and thus can not be tracked. However, threaded applications are still correctly sliced because of the initialisation step. The slicer recognise the use of the influenced registers, and continues at this point.

Unfortunately, by the nature of Android's concurrent components, the algorithm is not able to correctly trace registers through such components. That means, that the slicer miss some influenced registers. The reason for that is that the the start method can be parametrised, an these parameter variables can not be traced. Similar to Java's thread architecture, the invoked method name differs from the executed method name. Thus, parameters from the starting method can not be tracked through the system, and so the trace of the relevant registers are lost at this point.

Aliasing: Aliasing describes a situation in which data can be accessed through different names in the program. Thus, modifying the data through one name implicitly modifies the values associated with all aliased names. For static slicing that means that an aliased name can potentially modify influenced values without being tracked. We are aware of the problem of aliasing, but we did not considered it in the framework, because our manual analyses of applications showed that for the purpose of the framework it is not necessary to deal with aliasing, and because of the limited time for this work. This is also described in Section 5.

Callback mechanisms: Beside the issue with different Android components, the algorithm has some problems with callback architectures, which is a special form of aliasing. The problem with common callback mechanisms in an application is similar to the problem with Android's concurrent components. In both cases the method name of the invocation differs from the executed method name. This is only true for callbacks on base-types and interfaces. If a class invokes the callback method on the class in which the method is implemented, then the previously defined invocation rules are applied.

The behaviour of callbacks on interfaces and base-types is discussed with the code from Listing 4.2. The example consists of two classes, one implements an interface and the other one creates an instance of this class and stores the reference to the class in an instance variable, which has the interface as type. In addition, a method invokes the callback method from the interface on the instance variable. If register three is in the tracking set, then this register will be a parameter for the method *callback* from the type *Lat/iaik/thesis/MyCallbackInterface*. Unfortunately, the variable, on which the reference to the object is stored, is not in the slice and even if it is in the slice the implementation of the method *callback* can not be found, because the correct class is not known. The correct class will be known, if for each variable the instantiation is stored. However, this is not feasible, because of the consumed memory of this list.

Nevertheless, to be able to handle such callbacks the algorithm inspects every method in each class, which implements the specific interface or base-type. Thus, it is possible that the slice contains too many methods, because if something like a command pattern is implemented then every command has a method with the same signature. The modules must consider such things for the definition of the detection patterns.

```
1   .class Lat/iaik/thesis/AClass
2   .super Ljava/lang/Object
3   .interfaces null
4
5   .instance variables
6     interface -> Lat/iaik/thesis/MyCallbackInterface;
7
8   .method <init>
9     ...
10    new-instance  v0, Lat/iaik/thesis/MyClass;
```

```
11      invoke-direct {v0}, Lat/iaik/thesis/MyClass;-><init>()V
12      iput-object p0, v0, Lat/iaik/thesis/AClass;->interface
13      ...
14
15    .method someMethod()
16      ...
17      iget-object p0, v0, Lat/iaik/thesis/AClass;->interface
18      invoke-interface {v0, v1, v3}, Lat/iaik/thesis/MyCallbackInterface;->
           callback(Ljava/lang/String; Ljava/lang/String);V
19      ...
20
21    .end class
22
23    .class Lat/iaik/thesis/MyClass
24    .super Ljava/lang/Object
25    .interfaces Lat/iaik/thesis/MyCallbackInterface
26
27    .method callback(Ljava/lang/String; Ljava/lang/String)
28      ...
29      new-instance v1, Ljava/io/File;
30      invoke-direct {v1, p1}, Ljava/io/File;-><init>(Ljava/lang/String);V
31      invoke-virtual {v1}, Ljava/io/File;->delete();V
32      ...
33
34    .end class
```

Listing 4.2: This piece of Smali code illustrates the issues of callback mechanisms.

Native calls: Android supports native libraries for computation-expensive code fragments. Unfortunately, native libraries are outside the scope of the slicer. They are written in a different language, which are compiled to native code. Thus, to also analyse native libraries a completely different parser and analyser is needed. In this case the analyser is restricted by its architecture, which is designed for Dalvik bytecode and such potential gaps must be accepted in the slice.

Exceptions and reflections: In addition, Java has two additional language elements which are not considered by the slicer – exceptions and reflection. Firstly, exceptions are used for error propagation through the program. The challenges of exceptions are:

- The enormous amount of different exception classes, and thus throw statements in the code.
- The arbitrary place of the catch block.
- The possibility to nest catch blocks.

For a static slicer it is not possible to detect if a program misuses regular exceptions for information forwarding. Thus, if an application forces a null-pointer exception to start malicious code, which is hidden in the code of the exception handler, then the static slicer does not notice it. Because of the nature of exceptions, the whole code of each catch block must be added to the slice to detect such implicit data-flow. Unfortunately, this increases the slice and adds a lot of code to the slice that is not potentially connected to it. Thus, as discussed before, our slicer is not able to slice exceptions.

Reflection is the ability of a computer program to examine and modify the structure and behavior (specifically the values, meta-data, properties and functions) of an object at runtime [J. Malenfant and Demers, 1996]. The complexity in slicing the reflection mechanism is the design of this mechanism

itself. With the help of reflection a class can be instantiated, and used based on a string that represents the class. Thus, to slice this mechanism the slicer must be able to analyse all strings in the program to understand which class is built, and which method is used. Furthermore, it would be possible that the strings, which represent the classes and methods, are encoded, and decoded at runtime. Such mechanism would efficiently blockade static program slicing.

4.3 Modules

The register tracker contains the slicing architecture, and the modules specific start and success patterns. These so-called sources, and sinks define specific code fragments in an application, to define the start, and end criteria of a slice. Currently, our framework includes the following modules.

Root: Analysis the application for the existence of a root-check. This is important for security critical applications, because otherwise such applications have additional security threats, like no private folder to privately store session information.

IO: The IO module has two fields of application. Firstly, it classifies applications by their use of the external file system, and secondly it supports the other modules to improve the detection capabilities of the other modules.

SMS: This module tries to classify SMS sniffers and catchers. In addition, it classifies the application by the registration type of the SMS application – static or dynamic. In combination with the IO module, it tries to find applications that control the phone through SMS.

Crypto: The crypto module tries to find code for cryptographic operations. The crypto module is basically built to examine the possibilities of the framework for the analysis of cryptographic code.

Each module is derived from a base class. This base class handles all slicer instances for a module. Furthermore, it handles method and class changes, and the reset of a module after an inspection. This is important to avoid wasting memory, because all modules are created by a factory, which is designed as an enumeration class. Enumeration classes have the disadvantage to store all modules in static instances of the class. This architecture keeps the modules in the memory.

4.3.1 Root

The first module is the root module. This module tries to find code fragments that do a root-check. Basically, applications do not have superuser access, but it is possible to circumvent this security mechanism by unlocking the root user on the phone. Normally, this is done by exploiting a security leak of the operating system (OS). Unfortunately, the user-based permission system of the Android system is one of the most important security features and if this feature is circumvented by the root user, then applications do not have private folders. Thus, all other applications are able to access all data of every installed application. Additionally, applications are able to access and install low-level services – for example their own kernel modules. The superuser access is a threat for the data of installed applications.

An application can use different methods to detect whether a phone is rooted or not. After a deep manual analysis of applications with implemented root checks, we found out that at least one of the following checks is used to classify a mobile phone as rooted or not in the analysed applications.

- Check the existence of the *Superuser.apk*, which is a root-user manager.
- Check the existence of the *su* command.

- Execute the *su* command and check if the shell throws an exception.
- Check the build-tag of the phone.

The first and the last check are simple. The first test checks if an application with the name *Superuser.apk* in */system/app/* exist and the last one checks if Android's build-tag contains the string '*test-keys*'. Each of these checks can be written in one line of Java code. Listing 4.3 shows these two lines of code. The second check is more complex because there are different places where the *su* application can be installed. Each directory in the *PATH* environment variable can hold it. However, the basic idea is the same as for the *Superuser.apk* file, which searches for a file at a specific location on the file system.

```
1  if ( new File(``/system/app/Superuser.apk'').exist() ) {
2    ...
3  }
4
5  if ( Build.TAGS.contains(``test-keys'') ) {
6    ...
7  }
```

Listing 4.3: This source code example shows two root-checks. The first line shows the Superuser.apk check, and the second line shows the build-tag check.

The third check tries to execute the *su* application and checks whether it was able to execute it or not. Listing 4.4 shows the code for this check.

```
1  try {
2    Runtime.getRuntime().exec(``su'')
3  } catch (IOException ex) {
4    // Could not find, or execute the command.
5  }
```

Listing 4.4: This source code example shows how to check if a phone is rooted or not, by executing the su command.

Additionally, there are some other techniques to classify Android phones as rooted or not, but none of the manually analysed applications used one of these mechanisms. Some of these checks are:

- Check the correct mapping, and mount properties of the partitions.
- Check the existence of applications, which are only available on rooted phones, like *tcpdump*.
- Check the file and folder permissions, for system files, and folders, which are not modifiable by the user.

The root module analyses applications for the commonly used detection code snippets. For detecting the *Superuser.apk*, or the *su* binary file, the application must check if the specific file exists. Next, executing a shell command is basically a method invocation with specific parameters. The last check searches for an access of Android's build-tag and a belonging check for its correctness. The phone is rooted if the build-tag contains the value *test-keys*.

4.3.2 IO

The IO module evaluates whether an application tries to modify the file system or not. Modifications can be done by:

- Creating a file.
- Writing to a file.
- Deleting a file.

These operations are done by nearly all file browser applications. It does not make sense to use these operations to make an assumption about the security of an application, but it is possible to classify applications by their file system behaviour. Therefore, the module tries to find operations on the file system and classifies the application into the following categories:

- NONE – The application does not use files.
- WRITER – The application creates files and writes to it.
- DELETER – The application removes files.

Nevertheless, this module is more useful in combination with other modules, like the SMS module. If these two modules are combined, then the analyser is able to potentially detect simple SMS command architectures. That means, that if an application receives an SMS and if the slice includes an IO operation, then the application will have a SMS command architecture, or is a backup application, which stores the SMS on the file system.

4.3.3 SMS

The SMS module tries to classify applications by its SMS behaviour. Applications receive SMS, which are distributed by Android's broadcasting mechanism, for different reasons. Most of the applications are simple messengers, which show the received message to the user, but some other applications use short messages to remotely control the phone. Additionally, applications are not only able to receive and read SMS they are also able to abort the broadcast. Thereby, no other application receives the SMS. Such applications are potentially malicious, but the classification in malicious and trustworthy applications can not be done by this detection system, because this classification mainly depends on the context in which the application is installed, and used.

Thus, this module classify applications by the type of registration and by the behaviour of the receiver into the following categories, where 1 is the highest and 3 the lowest category.

1. DYNAMIC
   (a) CATCHER
   (b) SNIFFER READER
   (c) SNIFFER DUMMY
2. STATIC
   (a) CATCHER
   (b) SNIFFER READER
   (c) SNIFFER DUMMY

3. NONE

Android has two different registration types for broadcast receivers. The common way is a static registration in the Android manifest. Beside this one, dynamic receivers can be registered during runtime. The problem with dynamic receivers is that they can not be determined through Android API calls. Therefore, we think that dynamic receivers may do more suspect things than static receivers. The module tries to find the registration process, which is for static receivers a simple manifest lookup, and for dynamic receivers, the module searches for the correct registration pattern. Afterward, the determined receiver is classified into one of the above categories.

If more than one receiver is found, then the application is classified based on the category of the highest receiver, which is found in the application. Dynamic receivers are more suspect and potentially more dangerous than static receivers, so the categories for dynamic receivers are higher than the categories for static receivers. After the registration type classification, the receiver is classified by its behaviour. A catcher aborts the broadcast, a sniffer reader extracts at least the body, or the origin address, and a sniffer dummy receives the SMS but does not extract data from it. This classification principal is shown in Figure 4.3. The category with the thick edges is the category in which the framework classifies such applications.

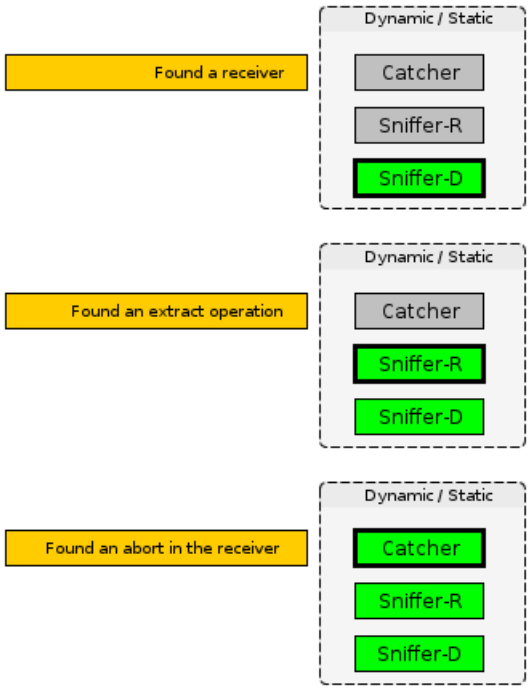

Figure 4.3: SMS Classification Hierarchy

This figure illustrates the SMS classification hierarchy. If a receiver is found, then it is a sniffer dummy. If an extract operation is found, then it is a sniffer reader. If an abort operation is found, then it is a catcher.

This module does not inspect the further usage of the SMS message. If the application extracts the body and executes a command based on the body, then this will not have any effect on the classification. It is possible to combine this module with other modules to detect this behaviour.

4.3.4 Crypto

The crypto module tries to find the use of cryptographic algorithms. The entry point of the slice of this module is an initialisation of the crypto class. This class takes the cryptographic algorithm identifier as initialisation parameter. Thus, the source is the initialisation string, which defines the cryptographic algorithm, and the initialisation statement of the crypto class.

Afterwards, the module can track the operations on this class. At the end of a cryptographic operation the method `doFinal` must be called. Therefore, the defined sink for this module is the `doFinal` method, which finalises a cryptographic operation. Listing 4.5 shows a code example that can be sliced with this module.

```
1  public static byte[] encrypt(Key key) throws Exception {
2    Cipher aes = Cipher.getInstance("AES/ECB/PKCS5Padding");
3    aes.init(Cipher.ENCRYPT_MODE, key);
4    byte[] ciphertext = aes.doFinal("This is the plaintext".getBytes());
5
6    return ciphertext;
7  }
```

Listing 4.5: This source code example shows how to encrypt data in Java.

Nevertheless, the detection rate of this module is limited by the design of the slicer and common implementation of cryptographic operations. The chosen source, and sink is one possible way to encrypt data. However, if it is differently implemented, then the slicer do not detect this cryptographic code.

4.4 Generated Results

Basically, the framework is designed to run directly on the device. The framework should analyse applications and present results for users for deciding if an application should be removed or kept and used on the device. Additionally, for technical experts a more detailed result should be stored for later inspection. Thus, the analyser stores the slice for later inspection and makes assumptions about the analysed application and presents clear statements to the user.

4.4.1 User Notifications

Each module generates a result string which is shown to the user. The following enumerations show the different result strings of the different modules.

For the root module only two possible result strings exist. This results from the fact that an application can check for root and we find it, or not. Therefore, the result of the checks is one of the following strings:

- A root check can not be found. Either the application does not check if the phone is rooted, or the application uses an uncommon technique for determining if a phone is rooted or not.
- The application has at least one check. The more checks the application implements the higher the chance to find an indication that the phone was rooted.

The IO module searches for file operations and describes if an application modifies the file system. It is possible that more than one result string fits for the result of the check. Thus, all fitting result strings are concatenated together. The possible result strings for the IO module are the following ones:

- The application deletes files on the phone.
- The application creates files and writes to them.
- The application does not touch the file system.

The SMS module analysis operations that are based on incoming short messages. Similar to the IO module it is possible that more than one result string fits the application. Furthermore, the registration type is not included in the result string, because it is assumed that a typical user does not know the differences between these categories.

- The application aborts the received SMS, so that no other application receives the SMS.
- The application reads the origin address from the SMS.
- The application extracts the message from the SMS object.
- The application receives SMS, but it neither extracts the message nor the origin address form the SMS.
- The application is not able to receive SMS.

4.4.2 Technical Experts

For technical experts the analyser contains a more detailed result writer. This result writer creates a general file, and for each module a detailed file. The general file contains the classification results of each module, the requested permissions of the application and the timings, to analyse the runtime of the analyser.

Each executed module also generates a file, which contains the slice of all registered trackers. It is possible that a module registers more than one tracker, where each tracker starts with a different start pattern. Thus each slice of each tracker must be written into the file. To structure the output of all tracker, each tracker must register a section for the file. These sections are sequentially written to the file. The used layout of the result file is based on the Smali notation, to be able to use Smali's syntax highlighting in supported editors for inspection.

Chapter 5

Evaluation

" We must reinforce argument with results. "

[Booker T. Washington]

The evaluation is done to verify the detection patterns of every module. We do not know the quality of our analyser. Thus, we built an evaluation basis, by manually analysing, and classifying applications for each module. After our manual classification, we evaluated the detection rate of every module. Furthermore, we evaluated the timing of each module based on a Nexus S.

The complexity of an evaluation is the evaluation basis. An evaluation basis must fulfil certain properties, to make an assumption about the correctness of a framework. In our case, such properties are the amount of sample applications, the amount of different implementation mechanisms, and the knowledge of the correct classification category.

During our manual analysis of applications, and the definition of the slicing criteria, we gained among others some in-depth knowledge about the implementation techniques of several security mechanisms. Therefore, we built evaluation bases consisting of manually built applications, and downloaded applications, which we manually analysed. These evaluation bases are assigned to the modules. Thus, each module has its unique evaluation basis.

Furthermore, each module is tested with a wider basis. We tested our framework on the basis of 3709 arbitrary applications and 1260 malware samples. To retrieve these applications, we used two other frameworks. We retrieved the arbitrary set of applications by our non-official market crawler framework, and the malware samples were retrieved from Malgenome during another master's thesis. Thus, we are able to make assumptions about the behaviour of malicious applications, and about the behaviour of an arbitrary set of applications. Unfortunately, it is not possible to manually analyse all of these 4969 applications to verify the results of our implementation. Therefore, we have another approach to interpret the results of the two large application sets. We use the knowledge of the evaluation basis in combination with a selectively manual analysis of the larger application sets to make an assumption about the correctness of the results.

The following sections describe the evaluation basis, and the evaluation results for each module. Consequently, each of the following sections is divided into a section that discusses the evaluation basis, the evaluation of the arbitrary set, and a section that discusses the evaluation of the malware set. Furthermore, we discuss possibilities to combine modules to improve the classification. Finally, in the last section we discuss the timing of the framework.

5.1 IO

The IO module checks whether an application accesses public files on the external storage or not. This is done by finding variables that hold the path to the external storage and additionally, by finding all IO operations associated to these variables. With this module we focused on modification access, because then it is possible to filter applications by their permissions in the manifest. Furthermore, we assume that the public folder only contains content that should be publicly available for other applications. Therefore, we also assume that a read-access is tolerated by the user. For our analysis, we defined a modification access as one of the following actions. Either the application tries to write data, or the application tries to delete some data on the external storage by overwriting, or deleting it. Thus, we classify these applications as potentially dangerous, because it could be possible to hijack components of such applications to modify the external storage or to completely wipe it.

Beside a standalone analysis, it is also possible to use this module in combination with other modules. The idea, of such a combined analysis, is to increase the detection rate of the modules. We use this method to increase the detection rate of the root, and IO module. The resulting evaluation of the combined analysis are described in the sections of the appropriate modules.

5.1.1 Evaluation Basis

The evaluation basis of the IO module consists of manually written applications, downloaded file-browser applications, and applications, which do not modify the file-system. The manually written applications are either write, or delete a file, whereas the downloaded applications write, and delete files. Thus, in Table 5.1 the number of classified applications is higher than the application set. The applications without an IO operation holds the permission for modifying the external storage, but does not modify it. These applications are used to evaluate the detection of false-positives.

Table 5.1: Evaluation of the IO Module with the Manual Evaluation Set

	Classified (#)
Applications with write operations	12 (12)
Applications with delete operations	12 (12)
Applications without an IO operation	20 (20)
Application set	40

The table shows the evaluation results of the evaluation basis for the IO module. The manual application set of 40 applications is classified in three different categories. The sum of these three categories is higher than the total amount of applications, because an application can have write, as well as delete operations on the external directory. Thus, some applications are counted twice. Furthermore, the number in braces is the basis for each category.

The evaluation of the IO module with its evaluation basis showed that our module correctly classifies all applications from the evaluation basis. We think that this outcome primarily results from two different properties. Either our defined sources and sinks are precisely enough to correctly classify applications with, and without an IO operation, or our manual evaluation set does not reflect all possibilities.

Therefore, the evaluation basis shows that the IO module correctly classifies common implementation types. Although we are aware of that this module has a specific error rate. A possible approach to circumvent the detection mechanisms of this module is to use system applications. Android has specific system applications to read, write, and delete files. To read a file, *cat* can be used. This is an application that reads the content of a file and sends it to the current output stream. To write some content to a file, *echo* can be used. *Echo* is an application that writes data from the input stream to the output stream,

which can be a file. Finally, to delete a file, the system provides the so-called *rm* application. Thus, an application could open a shell and read, write, and delete files with the help of these specific system applications.

5.1.2 Evaluation of the Arbitrary Set

The evaluation of the arbitrary set of applications showed that more than the half of the applications in our test-set write, or delete data on the external storage. Table 5.2 shows the result of the evaluation. We assume that this result is a consequence of the random property of the evaluation set. A brief manual analysis of the evaluation set showed that the set includes many file browser applications.

Table 5.2: Evaluation of the IO Module with the Arbitrary Set

	Classified (#)	Classified
Writes files	1210/1458	82.99%
Deletes files	1015/1458	69.62%
Modify external storage	1458/3709	39.31%
Do not modify external storage	2251/3079	60.69%
Application Set	3709	100%

The results of the evaluation of the IO module with the arbitrary set is separated into two sections. The first section shows the separation between write, and delete applications. As explained before, the sum of these two categories can be higher than the total classified applications. The next section shows the classification into applications that modifies the external storage and those that do not modify it. The sum of this section is the total amount of applications for this evaluation.

Furthermore, the result of our analysis is visualised in Figure 5.1. The intersection of the sets of write- and delete-accesses to the external storage holds the largest part of the classified applications. Thus, we assume that we have at least 767 file browser applications, or applications that primarily operate on the file system in our evaluation set.

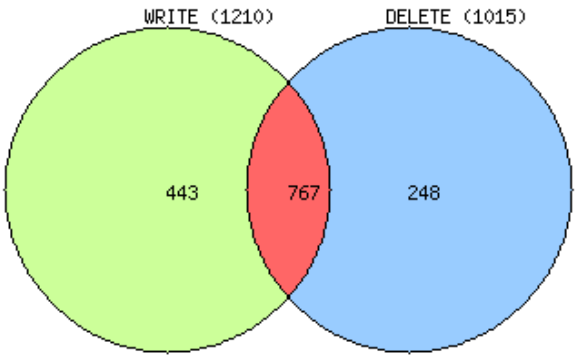

Figure 5.1: Intersection of the IO Categories Write, and Delete

Conclusion

Through the standalone analysis of the IO module, we got in-depth details about the behaviour of the applications in the arbitrary application set, and information about the performance of the IO module. In combination with a manual analysis, we found out that many applications in our arbitrary set are file-browser applications. Furthermore, we are able to separate applications that use the external storage and those they do not use it.

However, this module is primarily designed to support the other modules by their classification. The sections of the root, and SMS module show that we are able to improve the root, and SMS module with the IO module.

5.1.3 Malware Evaluation

The second evaluation set consists of malicious applications only. Table 5.3 shows the result of this evaluation. The evaluation shows that the number of malicious applications with IO operations on the external storage and those without such operations are nearly equal. Thus, a general assumption about the file-usage behaviour of malicious applications can not be made. Nevertheless, an additionally manual analysis showed that the usage of the external storage depends on the malware families. Specific malware families do not modify the external storage and other malware families have the possibility to write, and to delete files on the external storage.

Table 5.3: Evaluation of the IO Module with the Malgenome Set

	Classified (#)	Classified
Writes files	530/579	91.54%
Deletes files	507/579	87.56%
Modify external storage	579/1260	45.95%
Do not modify external storage	681/1260	54.05%
Application set	1260	100%

Similar to the evaluation result of the arbitrary set, the result is also separated into two sections. The first one divides those applications that modify the external storage into writer, and deleter applications.

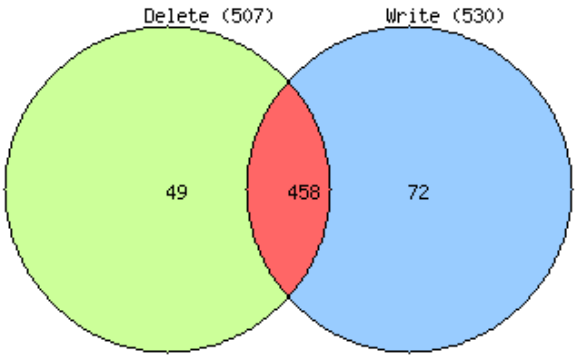

Figure 5.2: Intersection of the IO Categories Write, and Delete for the Malgenome Evaluation

The largest part of the malware, which is able to modify the external storage, has the ability to write as well as to delete files. We assume that this results from the malicious nature of a malware. A malware with both functionalities is more flexible, and efficient. Figure 5.2 illustrates the intersection of the classified categories.

Conclusion

Similar to the evaluation with the arbitrary set of application, the detection of IO operations makes limited sense only. In this scenario we are able to differentiate between malicious application that use the external storage, and those they do not use it. Thus, this module is able to make a pre-separation of the malicious application set for an additionally manual analysis. Our brief manual analysis showed that the use of the external storage primarily depends on the malware family. Nevertheless, the module is designed to cooperate with other modules and a standalone usage is less conclusive.

5.2 Root

The root module checks if an application includes some root-checks in its code or not. We defined in Section 4.3 the detection mechanisms for the root module in detail. In summary, common root checks are the following ones:

- Check the existence of the *Superuser.apk* file.
- Check the existence of the *su* command.
- Execute the *su* command to check whether the command is executable with the default environment variables.
- Check Android's *BUILD* tag.

Our framework classifies applications, based on these four checks. Thus, either an application has at least one of these checks and is classified as an application that does root-checks to ensure the integrity of the phone, or it has not such a check. If it is classified as an application without a root-check, then either it is an application without a root-check, or it has an uncommon root-check, which our framework is not able to detect.

5.2.1 Evaluation Basis

To evaluate the root module, a good evaluation basis must be found. Unfortunately, it is not easily possible to find applications with root-checks by special key-words, or tags. Thus, we did a comprehensive survey of container-, and banking-applications. These applications should have root-checks to ensure the integrity of the phone [Lange et al., 2011]. Nevertheless, only a few applications have implemented a root-check.

Therefore, our evaluation basis consists of the applications from our comprehensive survey that use at least one root-check, and own sample applications, which are based on our manual analysis of applications with root-checks. To also detect false-positives we built a set of applications without any root-checks as well. This set consists of the remaining applications from our comprehensive survey, as well as manually built applications.

Table 5.4 shows the results of the manual evaluation set. The applications with root-checks were correctly detected, but there are also four wrongly detected applications. The problem is that the module does not check, whether the application opens a root-shell, or does a root-check. This issue can be fixed by combining the root module with the IO module.

Table 5.4: Evaluation of the Root Module with the Manual Evaluation Set

	Classified (#)
Applications with root-checks	24 (20)
Applications without root-checks	16 (20)
Application set	40

The table shows the evaluation of the root module with the corresponding evaluation basis. This module classifies an application either as an application that has at least one root-check, or non. Furthermore, the number in braces is the basis for each category.

The following sections describe the different evaluation results, based on the arbitrary, and the malware application set. Furthermore, combined results for further improvements are discussed.

5.2.2 Evaluation of the Arbitrary Set

Our evaluation on the wider set of arbitrary applications showed that only approximately 5% of these applications contain at least one of the four defined root-checks. Table 5.5 shows this result in more detail.

We assume that the small amount of applications, with a detected root-check in our test-set, primarily results from two different properties. Firstly, the applications in our test-set are arbitrarily retrieved from different markets. Because of this random property, many applications function properly without an existing root-check. A manual analysis showed that the arbitrary set also includes wallpaper, messenger, and simple notifier applications. Obviously, these types of applications do not need a root-check. Secondly, our manual analysis also showed that hardly any applications implemented a root check.

Table 5.5: Evaluation of the Root Module with the Arbitrary Set

	Classified (#)	Classified
Applications with root-checks	199/3709	5.37%
Applications without root-checks	3510/3709	94.63%
Application set	3709	100%

The table shows the evaluation of the root module with the arbitrary set of applications. It separates the 3709 applications of the arbitrary set into applications with and without root-checks.

Nevertheless, 5% of our test-set contains a root-check, and the evaluation of our manual evaluation basis showed that we are able to detect the defined root-checks, but the result can also include false-positives. Therefore, detected applications can also use the privileges of the root user on the phone to gain more permissions for their applications. The next section discusses a method to differentiate between root-checks and potential root-usages in an application.

Combined Results

The combined evaluation combines the root module with the IO module. The idea behind this is to improve the performance of the root module, by separating applications that open a root-shell. A usable root-shell can be created by executing the superuser binary and retrieving the corresponding input and output streams. In contrast to this, one of the defined root-checks tries to execute the superuser binary, and to check whether the system throws an exception or not.

Thus, we used the IO module to differ between applications that implement a root-check and those that use root to gain more permissions for their applications. The IO module has the possibility to find file operations in an application. These file operations can be divided into read-, write-, and delete-access. Our assumption is that if an application uses the privileges of the root user, the application needs an input-, and an output-stream for the created root-shell. These streams are derived from Java's IO architecture and therefore, they are IO operations that our IO module is able to detect.

Table 5.6 shows the combined result of the IO module and the root module. For this analysis we used the 5% of applications, which are classified as applications with root-checks.

Table 5.6: Evaluation of the Root Module in Combination with the IO Module

	Classified (#)	Classified
Applications with IO operations in the slice of the root-check	145/199	72.86%
Applications without IO operations in the slice of the root-check	54/199	27.14%
Application set	199	100%

The table separates the applications with root-checks into applications with IO operations in the slice of the root-check and those without ones.

145 applications are classified in one or more IO-categories, but 54 applications do not have IO operations related to their root-check. Therefore, based on our previous assumption, these 54 applications are applications that really do a root-check. Furthermore, we did a brief manual analysis of the applications with root-checks, and IO operations. The manual inspection showed that these applications are primarily file-browser applications, which use root to gain more permissions for their applications.

Conclusion

The analysis in combination with our manual inspection showed that our framework is able to find code that is used to make root-checks. It also showed that the detector for the root-checks also wrongly detect some applications. Nevertheless, the wrongly detected applications contains code that uses the root user for opening a root-shell and these applications can be filtered with the combined approach. Thus, this module is able to find root-checks, and the usage of the *su* command to open a root-shell.

5.2.3 Malware Evaluation

Our second wider evaluation basis consists of malware only. Therefore, we assume that our framework finds more applications, which do root-checks, or open root-shells. The evaluation showed that this assumption was correct. Table 5.7 shows the result of the evaluation, where 452 applications contain a root-check.

Nevertheless, we still have the problem that the detected applications do not make a root-check, but rather use root to open a root-shell. In this specific set of applications, we assume that many of these applications generate a root-shell to execute their malicious code. The combined result helps us to differentiate between these two types.

Combined Results

Similar to the arbitrary set of applications, the combined result is used to differentiate between applications with root-checks, and applications with root-shells. Table 5.8 shows the result of this evaluation. We found out that 436 of 452 applications contingently use root-shells for executing code. For this specific applications we can conclude that they use a root-shell for malicious purpose. Furthermore, 16

Table 5.7: Evaluation of the Root Module with the Malgenome Set

	Classified (#)	Classified
Applications with root-checks	452/1260	35.87%
Applications without root-checks	808/1260	64.13%
Application set	1260	100%

The table shows the separation into applications with and without root-check, based on the malicious application set.

applications have a root-check, but do not use it for executing malicious code. In these applications, a manual inspection showed that the root-check is used to control the behaviour of the malicious application. We found out that some malware families check if a phone is rooted or not. If it is not rooted, then such a malware tries to root the phone.

Table 5.8: Evaluation of the Malgenome Set with the Root Module in Combination with the IO Module

	Classified (#)	Classified
Applications with IO operations in the slice of the root-check	436/452	96.46%
Applications without IO operations in the slice of the root-check	16/452	3.54%
Application set	452	100%

The previously classified applications that include a root-check are separated into applications that have an IO operations in the slice of the root-check and those without.

Conclusion

With the analysis of the malware set, we showed that malicious applications as well as benign applications make root-checks. Obviously, the motivation is differently. A manual analysis showed that malicious applications use root-checks to classify mobile phones into rooted, and non-rooted phones. Furthermore, we found out that these malicious applications classify the mobile phone to know if a root-exploit must be executed or not, and whether an executed root-exploit was successful or not. Finally, we showed that our framework is able to find applications, which try to execute commands as root, but it can not differentiate between malicious and benign applications.

5.3 SMS

The SMS module tries to classify the different types of SMS receiver. It is possible to register a receiver either statically in the manifest file, or dynamically in the code. Statically registered receivers can be easily found through Android's API calls. Thus, such applications can be easily listed as applications that receive incoming SMSs. Unfortunately, third-party applications are not able to easily detect dynamically registered receivers in other applications. These applications hide the functionality to receive incoming SMSs. Furthermore, the influence on the incoming SMSs from an application should be also classified, because it is possible that an application only counts the arrival, reads the body or the address of the SMS, or aborts the SMS. The SMS module classifies each application into one of seven categories, to separate between the registration types and the type of usage of the incoming message.

5.3.1 Evaluation Basis

To evaluate this module, we built an evaluation basis with 145 different applications. These applications were downloaded from Android's official market, and from 3rd party markets. 117 applications contain at least one SMS receiver and 28 applications are without an SMS receiver, to also evaluate false positives. The applications are not equally distributed over all categories, because it is not easily possible to find applications, which fit to the criteria of a category. This results from the fact that we can not specifically search for applications with dynamically registered receivers in it. Thus, many applications must be reverse engineered until an application with a suitable dynamically registered SMS receiver is found.

Furthermore, applications with complex programming constructs are not included in the set. Complex programming constructs are reflections and libraries. Both can not be sliced with the design of the analyser. Thus, it is not sensible to include applications with such code in the manual evaluation set. Additionally, we did not find any applications that use one of these constructs to handle incoming SMSs, during our analysis of this module.

Table 5.9: Evaluation of the SMS Module with the Manual Evaluation Set

Category	Classified (#)	Classified
Dynamic Catcher	13 (16)	81.25%
Dynamic Sniffer - Reader	34 (35)	97.14%
Dynamic Sniffer - Dummy	6 (6)	100%
Static Catcher	12 (12)	100%
Static Sniffer - Reader	34 (41)	82.93%
Static Sniffer - Dummy	7 (7)	100%
None	28 (28)	100%
Complete result without None	106 (117)	90.6%
Complete result with None	134 (145)	92.41%

The SMS module seperates the applications into seven categories. An SMS can be aborted, read, or only counted, and the broadcast receiver can be statically, or dynamically implemented. Thus, an SMS applications is classified into a dynamic, or static SMS application of the specific type. The number in braces is the manually defined number of applications in each category.

Table 5.9 shows the detection rate of our manual evaluation set for each category, and an overall detection rate. 11 applications were not correctly detected. These 11 applications were classified as applications with SMS receiver, but they were classified in a wrong category. The reason for this is that all of these applications use Android's service architecture to handle receiving SMSs. Unfortunately, our slicer is currently not able to correctly slice over Android's service architecture. This results from issues with the IPC architecture in combination with static slicing. Nevertheless, our framework found the SMS receiver, but did not correctly classify the applications. Therefore, the wrong result is only the type of a receiver, but it is not the possibility to receive an SMS. Thus, it does not forge the number of applications with and without an SMS receiver.

5.3.2 Evaluation of the Arbitrary Set

In our arbitrary set of applications, our framework found 1205 applications with a registered SMS receiver. Table 5.10 shows this result in detail. A manual analysis showed that 1889 applications have the permission for receiving an SMS. We assume that this gap of 684 applications exist, because of one of the following reasons.

- The application does not contain a receiver and is over-privileged, because the developer did not care about Android's permission model.
- The receiver is not publicly accessible and consequently not tracked by our framework.
- Our framework could not find the receiver.

Felt, Chin, et al. [2011] also describe issues with over-privileged applications. Thus, we assume that our framework is correct, except a minimal error rate with services in combination with SMS receiver. Nevertheless, this error rate only has an effect on the categories and not on the detection of SMS receivers itself.

Table 5.10: Evaluation of the SMS Module with the Arbitrary Set

Category	Classified (#)	Classified
Dynamic Catcher	172/3709	4.64%
Dynamic Sniffer - Reader	351/3709	9.46%
Dynamic Sniffer - Dummy	54/3709	1.46%
Static Catcher	204/3709	5.50%
Static Sniffer - Reader	311/3709	8.38%
Static Sniffer - Dummy	113/3709	3.05%
None	2504/3709	67.51%
Applications with SMS receiver	1205/3709	32.49%
Application set	3709	100%

The table shows the results of the evaluation of the SMS module with the arbitrary set.

In addition to the gap of 684 applications, we classified the applications into seven categories. A manual analysis showed that the two catcher categories primarily consists of two different types of applications. Firstly, typical third-party messenger applications, which abort the message to avoid that the default messenger also retrieves the SMS and notifies the user. Secondly, applications that use SMSs as control commands. Such control commands are not shown to the user and thus, such applications abort the incoming SMSs. Furthermore, our manual analysis showed that typical sniffer applications retrieve the address, and the body to store it into a private database, or to show detailed notifications to the user. Typical dummy sniffer are also applications that notifies the user that an SMS arrived.

Furthermore, we also figured out that we can not correctly detect all applications in the correct category, because some sniffer are also abort the incoming SMS. Nevertheless, we also showed this with the evaluation basis, and thus, it is not astonishing.

Combined Results

Furthermore, it is possible to combine the SMS and the IO module. The idea of this combined result is to filter applications that use SMSs as remote commands. Our assumption is that if an application receives an SMS and if the message of an SMS has a relation to an IO operation, then this application will use SMSs to control the phone. This assumption is based on the manual analysis of applications with remote wipe functionality.

Table 5.11 shows the result of this combined evaluation and Figure 5.3 illustrates the result. A brief manual analysis showed that the 181 classified applications can be divided into two different categories. Either the applications modify the file system, based on the message of the SMS, or the applications hide the SMS by aborting the broadcast and storing the SMS into a private database. The first category is our

Table 5.11: Evaluation of the SMS Module in Combination with the IO Module

File Operation Type	Classified (#)
Read	119/1205
Write	140/1205
Delete	61/1205
Application set	1205

The table shows the separation of the applications with an SMS slice into the different IO operations in the slice. The sum of all IO operations can be higher than the application set, because each application can also include more than one type of IO operation.

predicted SMS command architecture and the second category consists of SMS blocking applications, which are used to avoid messages from specific addresses.

Nevertheless, this is only a subset of applications with SMS commands, because our framework is not able to slice IPC architectures. Furthermore, we only detect commands that result in an IO operation, and consequently other SMS commands are not detected during our evaluation. Thus, we conclude that our result only contains a small subset of applications with SMS command functionality.

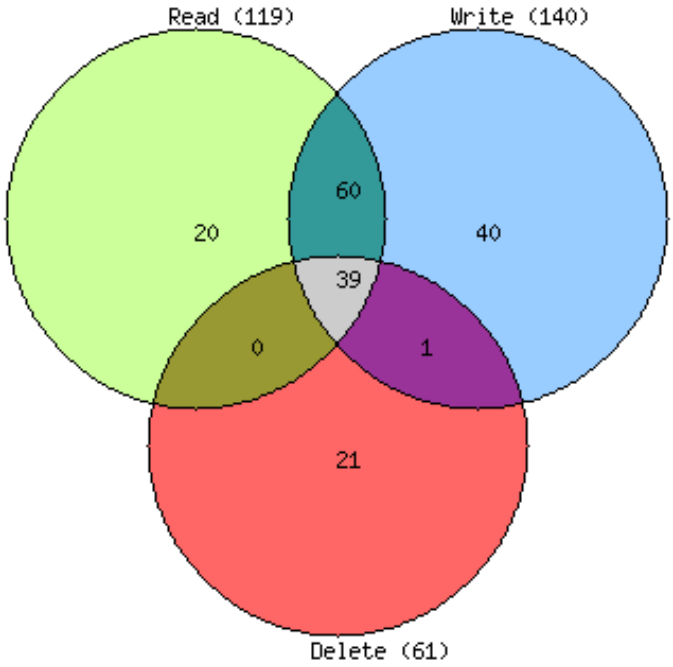

Figure 5.3: Intersection of the SMS and IO Module

Conclusion

We showed that beside the detection categories, other criteria are important as well. It is important to know what the application does with the message, beside of the knowledge about the category of SMS applications. If an application is able to modify the system, based on a receiving short message, then this will be more dangerous as an application, which only aborts the broadcast. Furthermore, we found out that some applications from the evaluation set are blocking applications. These applications extract the origin addresses of the incoming short messages and block those messages that the user is not allowed

to see. If the user defines the blocking rules, then such applications work correct, but if another person defines the rules, then it will be a malicious behaviour.

5.3.3 Malware Evaluation

Another dataset for the evaluation of the SMS module is retrieved from the Malgenome project. Table 5.12 shows the result of this application set. Similar to our other arbitrary evaluation set, many applications do not contain the permission to handle incoming SMS. Furthermore, the malware set also contains over-privileged applications. 22 applications have the SMS permission in the manifest, but neither a static nor a dynamic receiver is detected by our framework. The reasons for such applications are discussed in the evaluation section of the arbitrary set.

Table 5.12: Evaluation of the SMS Module with the Malgenome Malware Set

Category	Classified (#)	Classified (%)
Dynamic Catcher	116/1260	9.21%
Dynamic Sniffer - Reader	3/1260	0.24%
Dynamic Sniffer - Dummy	11/1260	0.87%
Static Catcher	287/1260	22.78%
Static Sniffer - Reader	57/1260	4.52%
Static Sniffer - Dummy	3/1260	0.24%
None	783/1260	62.14%
Applications with SMS receiver	477/1260	37.86%
Application set	1260	100%

The table shows the results of the evaluation of the SMS module with the malware set.

The primary categories are dynamic, and static catchers. We expected this result, because we think that malicious applications try to catch SMS for their own purpose. This can be done to sniff mobile TANs, to make electronic depository transfers without the knowledge of the user [Helmut, 2013]. Furthermore, we think that the other categories result from another type of malicious application. These applications are malicious applications with the possibility to read incoming SMS. This type of application can read and forward incoming SMS without the knowledge of the user.

Combined Results

Interestingly, the malware evaluation set does not use IO operations in combination with incoming SMS. Nevertheless, the applications in our malware set access the file system. This shows the result of the evaluation of the IO module, but they do not include SMS commands to directly modify the file system, based on incoming SMS. We think that this method of controlling a phone is not common for a malware and thus, it is not included in our malicious sample set.

Conclusion

We found malware that try to intercept the SMS communication in our malicious application set. Furthermore, the applications, which are classified as dummy sniffer, are wrongly classified. A manual inspection showed that these application forward the intent to a service, which we can not be tracked with our framework. Nevertheless, we correctly found the applications and only wrongly classified it. Malicious applications, which only reads the applications, are either wrongly classified and they also

abort the broadcast, or they only extract the content and probably forward the retrieved information through another communication channel, like the Internet.

5.4 Crypto

Our cryptographic module is based on the idea of a bachelor thesis, which also used static analysis for classifying Android applications. The basic idea of our approach is to find the generation of a cipher and track the cipher until it is finished. Therefore, we ignore cryptographic code that is not used to encrypt, or to decrypt data in the code.

For the definition of our detection parameters, we used our experiences of the bachelor thesis as well as a brief manual analysis of encryption applications. Consequently, we defined a set of cryptographic algorithms that we want to find. For this module we defined AES, DES and its variants, RSA, Blowfish, and Twofish as algorithms we want to find.

We used this small sample set of algorithms to evaluate the usability of this approach. Nevertheless, we are aware of the fact that other detection mechanisms are potentially better than this approach. Furthermore, we also know that our framework does not perfectly fit for cryptographic analysis, because of the following facts.

- Our slicer is not able to track registers through libraries. Unfortunately, cryptographic algorithms are also implemented in native libraries to improve the performance for encryption, and decryption.
- Cryptographic libraries for Java use many loose bindings, between the creation process and the cipher itself. Consequently, we can not always find a connection between the creation of a cipher and the usage.
- Furthermore, cryptographic algorithms can also be implemented by themselves.

5.4.1 Evaluation Basis

Because of the experimental approach of the crypto module, the evaluation basis consists of manually built, and downloaded applications. We manually built 20 application that use encryption and downloaded 20 applications without any encryption in it. Table 5.13 shows the result of the manual evaluation set.

Table 5.13: Evaluation of the Crypto Module with the Manual Evaluation Set

	Classified (#)
Applications with cryptographic operations	14 (20)
Applications without cryptographic operations	20 (20)
Application Set	40

The table shows the evaluation of the crypto module with the corresponding evaluation basis. The number in braces shows the amount of applications for each category.

From 20 applications, we correctly detected 14 applications and for 6 applications we could not find a connection between the creation of the cipher and the finalisation of the cipher. This results from the loose binding, when factories are used.

Nevertheless, we also run the module of the two wider evaluation sets. We know of the weaknesses of our module, but we also know that the module is able to find the usage of cryptographic algorithms.

Thus, we get additional information about the applications, which we can combine with the previous analyses.

5.4.2 Evaluation of the Arbitrary Set

Our evaluation of the wider arbitrary set showed that 1191 applications use one of the defined cryptographic algorithms. Table 5.14 shows the results of the evaluation. Unfortunately, our result does not assume anything about the completeness of the detected set of applications.

Nevertheless, we know that the detected applications really use one of the defined ciphers. This results from our very limited definition of the start- and end-condition of the slicer. Furthermore, this knowledge is more interesting in combination with the root module.

Table 5.14: Evaluation of the Crypto Module

	Classified (#)	Classified
Applications with cryptographic operations	1191/3709	32.11%
Applications without cryptographic operations	2518/3709	67.89%
Application set	3709	100%

The table shows the evaluation of the crypto module with the arbitrary set of applications.

Combined Results

Beside the standalone analysis, we can also combine the results of the crypto module with other modules. An interesting approach of the cryptographic module is the usage of this module in combination with the root module. If we combine these two modules, we are able to make some assumptions about the relation of integrity checks and cryptographic operations.

Figure 5.4 illustrates the result of the combined evaluation. We combined the results of the crypto module with the results of the combined analysis of the root module with the IO module. The result shows that applications with IO operations in the root-check more often use cryptographic code in the application as the applications without IO operations in the root-check. Less than the half of the applications without IO operations in the root-check also use cryptographic operations in the application. Consequently, we can assume that these applications are security related applications, which want to ensure a secure environment for their application on the phone.

Conclusion

The cryptographic module by its own is not really useful, because we do not get additional information about an application, except that it uses a cryptographic algorithm. With the results of a standalone execution, all detected applications must be reverse engineered to understand the usage of the cryptographic operations in the applications. Thus, we manually analysed a subset of the result, to argue from this selective set of applications to all applications. Our brief analysis showed that the applications uses cryptographic operations for file encryption, key derivation, and payment verification. Furthermore, Google's advertising library also uses encryption. It seems that this library encrypts sensitive information before it sends it to a server.

In contrast to the standalone result, we can see the relation between cryptographic operations, and root-checks in the combined result. More than the half of the applications, which are detected by the root module, use cryptographic operations. We can assume that the largest part of the intersection are security relevant applications. A brief manual analysis showed that the intersection also holds some file-browser

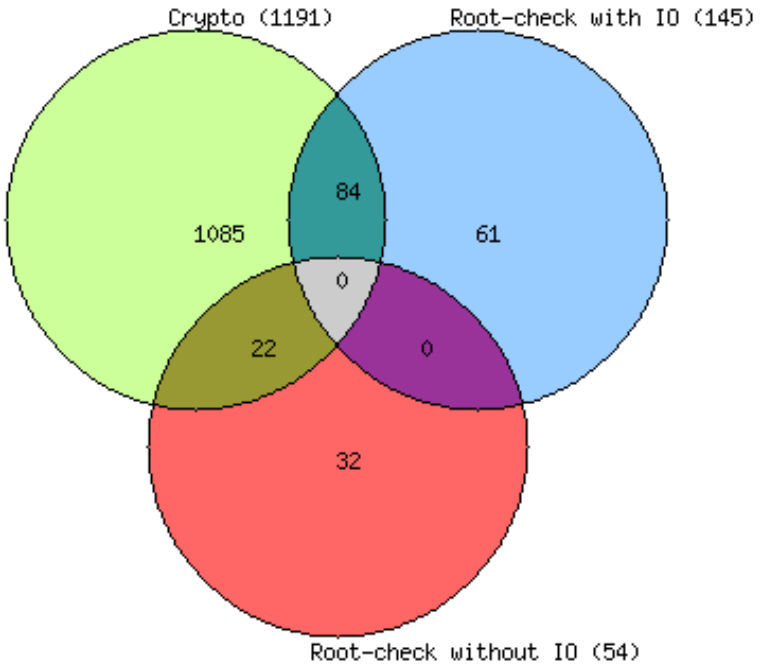

Figure 5.4: Intersection of the Root and Crypto Module

applications, which use root-shells and cryptography. Thus, we found out that this combination primarily occurs by file-browser, and security relevant applications.

5.4.3 Malware Evaluation

The evaluation of the malware set showed that nearly the half of the malware includes cryptographic code. Table 5.15 shows the result of this evaluation. Beside the use in advertising libraries, it seems to us that some malware hides data, or code with the help of cryptographic algorithms. A manual analysis showed that it depends from the malware family. Some malware families only use cryptographic operations in combination with advertising libraries, and other families use malware for encryption, and signature creation and verification as well. The last type of applications seems to be trojan horses. This assumption is made, because we compared the applications of our manual analysis with an online virus database. This database categorised these applications as trojan horses.

Table 5.15: Evaluation of the Crypto Module

	Classified (#)	Classified
Applications with cryptographic operations	600/1260	47.62%
Applications without cryptographic operations	660/1260	52.38%
Application set	1260	100%

The table shows the evaluation of the crypto module with the malware set.

Combined Results

Similar to the combined result for the arbitrary set, we try to gain some information about the relation between cryptographic operation and the usage of root-checks, or root-shells, with the combination of

the crypto module and the root module. Figure 5.5 illustrates the combined result. Nearly all malware applications that use the root user to gain more permissions also use cryptographic algorithms in their applications. 84.63% of the malware applications with IO operations in the root-check also use cryptographic operations. We assume that these malware use root in combination with cryptographic operations to hide data or code, and to modify their application's binary. Rastogi, Chen, and Jiang [2013] and You and Yim [2010] also describe malware with polymorphic capabilities. Furthermore, 62.5% of the applications without IO operations in the root-check also use cryptographic operations. The behaviour of malicious applications that use root-checks is discussed in the root module. To sum it up, it could be that an application uses the root-check to classify the phone as rooted, or not. Such applications are able to root the phone and thus, to also use cryptographic operations to hide data or code.

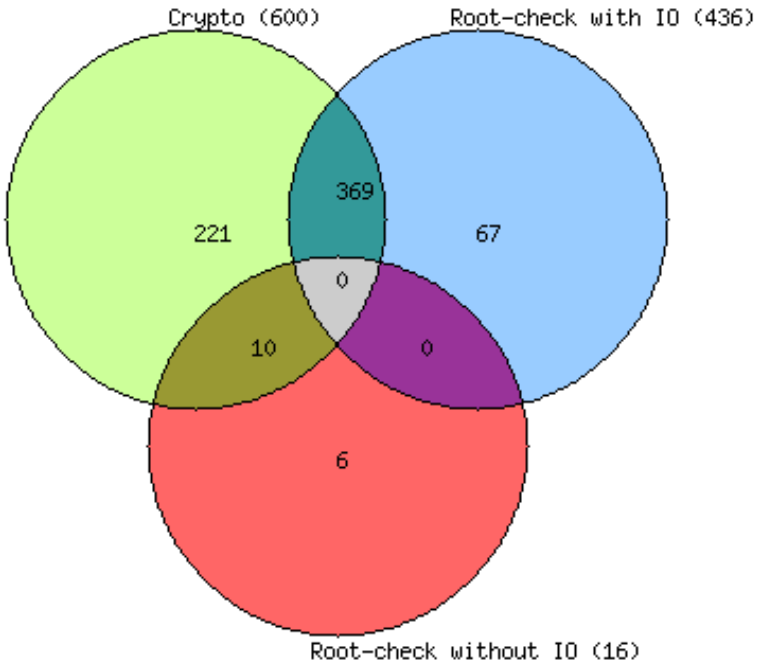

Figure 5.5: Intersection of the Root and Crypto Module, Based on the Malware Set

Conclusion

Although the experimental approach of this module, we got interesting information about the behaviour of malicious applications. We found out that the malicious applications does not really differ by their use of cryptographic operations to benign applications. Some malware families use cryptographic operations in combination with advertising libraries. During our analysis we found out that malicious applications use Chinese advertising libraries more often than benign applications. Nevertheless, this can also be a result of our reduced manual analysis, or a property of our malicious application set. Furthermore, malicious applications use cryptographic code for signature creation and verification. If we compare the results with virus databases, then these applications will be classified as trojan horses. Additionally, it seems to be that some of the malicious applications use encryption to hide data.

5.5 Timing

One important factor for the usability of a framework is the execution time. We measured the timing of our framework, based on the wider arbitrary evaluation set and a Nexus S. Table 5.16 shows the result of

the timing evaluation.

Table 5.16: Timing of the Modules

Module	Average Time	Median
ROOT	11.78 sec	5.01 sec
IO	27.80 sec	5.23 sec
SMS	10.44 sec	2.91 sec
CRYPTO	17.50 sec	6.10 sec

The execution of each module is on average faster than half a minute and the median of each module is below ten seconds. Furthermore, there are some differences between the execution time of the different modules. This results from two influence factors. Firstly, we can not control background operations on the phone. Nevertheless, with our large evaluation basis, we believe that the influences of the operating system do not really impact the average time of execution. Secondly, the random nature of the evaluation set also influences the result. We do not exactly have the same amount of applications for every category of each module. Consequently, some modules produce a longer slice, which takes more time. This is also the reason for the significant gap between the IO module and the other modules.

Chapter 6

Concluding Remarks

" Everything has an end, only a sausage has two. "

[German, Danish, and Dutch proverb.]

This master's thesis tries to efficiently analyse Android applications directly on the device. The idea of this approach is to learn something about the security functions, and protection mechanisms of Android applications. Furthermore, we want to separate applications by their use of data.

For example, we classify applications in different SMS categories, where the categories define the type of usage. Thus, we separate applications into applications that abort SMS broadcasts, read the message or the address, or only count that a message arrived. With this method of classification we learned which type of application does which operation. We found out that, beside malicious application, benign messaging applications typically abort the SMS broadcast. We think that the messaging applications abort the broadcast to avoid that Android's default messaging application also receives the message. Furthermore, we found out that benign applications, which extract the origin address or the message of a SMS, are typically either a message blocking application, or a control application. An additional analysis showed that we are able to find applications that use SMS for controlling the phone. We can conclude on this behaviour of an application, if the application has an IO operation in the slice of the SMS.

Beside SMS applications, we also inspected another security relevant property of mobile phones. For some kinds of applications, it is relevant to know whether a device is rooted or not. This results from the fact that on rooted phones applications have less protection mechanisms against attackers, and malicious applications. Thus, it is sensible for security relevant applications – like banking, business, or video streaming applications – to check whether a phone is rooted or not and to prevent the execution if it is rooted. Our framework tries to determine if applications implement such security checks or not. Our analysis showed that many applications do not check if a phone is rooted or not. Furthermore, the applications, which check if the phone is rooted or not must be separated into two categories. The first category consists of those applications, which make root-checks for protecting application's data, and the second category includes the applications, which also try to use an existing root user for their own purposes. We were able to partially filter these two categories by analysing the IO operations in the slice of the root-check.

Our third evaluation setup tries to use the framework to find cryptographic code in applications. Unfortunately, the results of this evaluation were not as good as from the two former ones. Our evaluation basis, which consists of manually written applications, showed that the complexity of cryptographic libraries can not be easily sliced with our approach. This results from the loose binding between the creation of a cipher and the usage of the cipher. For example, a cipher can be initialised, and then retrieved by a factory. Such code constructs can not be sliced, because from the perspective of the slicer the creation of the cipher does not have a static relation to the usage of the created cipher. Nevertheless,

we were able to detect cryptographic code and with a manual analysis we were able to partially classify the applications.

All things considered, we contributed to the understanding of security mechanisms of Android with our framework. We primarily discussed issues with the SMS channel on Android, and the issues with rooted phones. Thus, our framework helps to better understand these security vulnerabilities.

Chapter 7

Outlook

" It's very easy to predict the future. People do it all the time. What you can't do, is get it right. "

[Don Norman, The Front Desk, BBC Video, 1995.]

We precisely discussed the capabilities and restrictions of this framework. Nevertheless, there is still space for improvement. For example, we made an experimental setup for detecting cryptographic code in applications. We found out that we are able to partially classify cryptographic code with our framework. A possible improvement of our framework would be to improve the detection rate and to extend the framework to detect other cryptographic operations as well. Another security relevant property of an application is the storage of passwords. Our framework should fit the requirements to slice password fields and to detect whether an application uses a key derivation function, or not. Furthermore, another possible extension would be to track if sensitive information – like passwords – are sent over the Internet.

Beside the improvement of the modules, the static slicer itself should be improved as well. There are some functionalities that would improve the quality of the slicer, but were not implemented because of the limited amount of time. It would be possible to parse a complete string, which is built from the string builder class, to learn more about string parameters. Furthermore, a solution for more complex programming constructs must be found. Java reflections, and IPC calls are examples for such programming constructs.

Appendix A

Acronyms

AES Advanced Encryption Standard
AID Android ID
ANR Application Not Responding
API application programming interface
APK Android application package
CFG Control Flow Graph
DES Data Encryption Standard
DEX Dalvik Executable
DVM Dalvik Virtual Machine
GID Group ID
ID Identifier
IO Input-Output
IPC Interprocess Communication
JAR Java archive
JVM Java Virtual Machine
MDM Mobile Device Management
OS Operating System
PDG Program Dependence Graph
RSA Rivest, Shamir und Adleman
SDG System Dependence Graph
SMS Short Message Service
TAN Transaction Authentication Number
TISSA Taming Information-Stealing Smartphone Applications
UI User Interface
UID User ID
URI Uniform Resource Identifier
VM Virtual Machine
WALA T. J. Watson Libraries for Analysis

Appendix B

Class Diagrams

Figure B.1 shows the class diagram of the complete framework. Each of this sections of the framework is described in Section 4. The framework itself is separated into two sections:

- The execution management.
- The execution environment.

The first section of the framework handles the execution of the framework on a phone. This consist of selecting an application and handling the slicer. The slicer itself runs for a longer period of time and all potential events, which change the environment, must be considered during this period. Such events are suspends, rotations and applications switches.

The next section handles the execution of the framework with the selected modules. It does some performance improvements which decreases the amount of code the slicer must analyse. Otherwise the inspection would potentially take too long for a sensible execution on the phone. Additionally, this section handles the logging process. The execution environment has a general result writer for protocolling general information about the application, the execution time and all assigned modules. Furthermore, each module has one result writer with at least one section. Each slicer from a module gets its own section and the result writer writes all sections with Smali notation to the file.

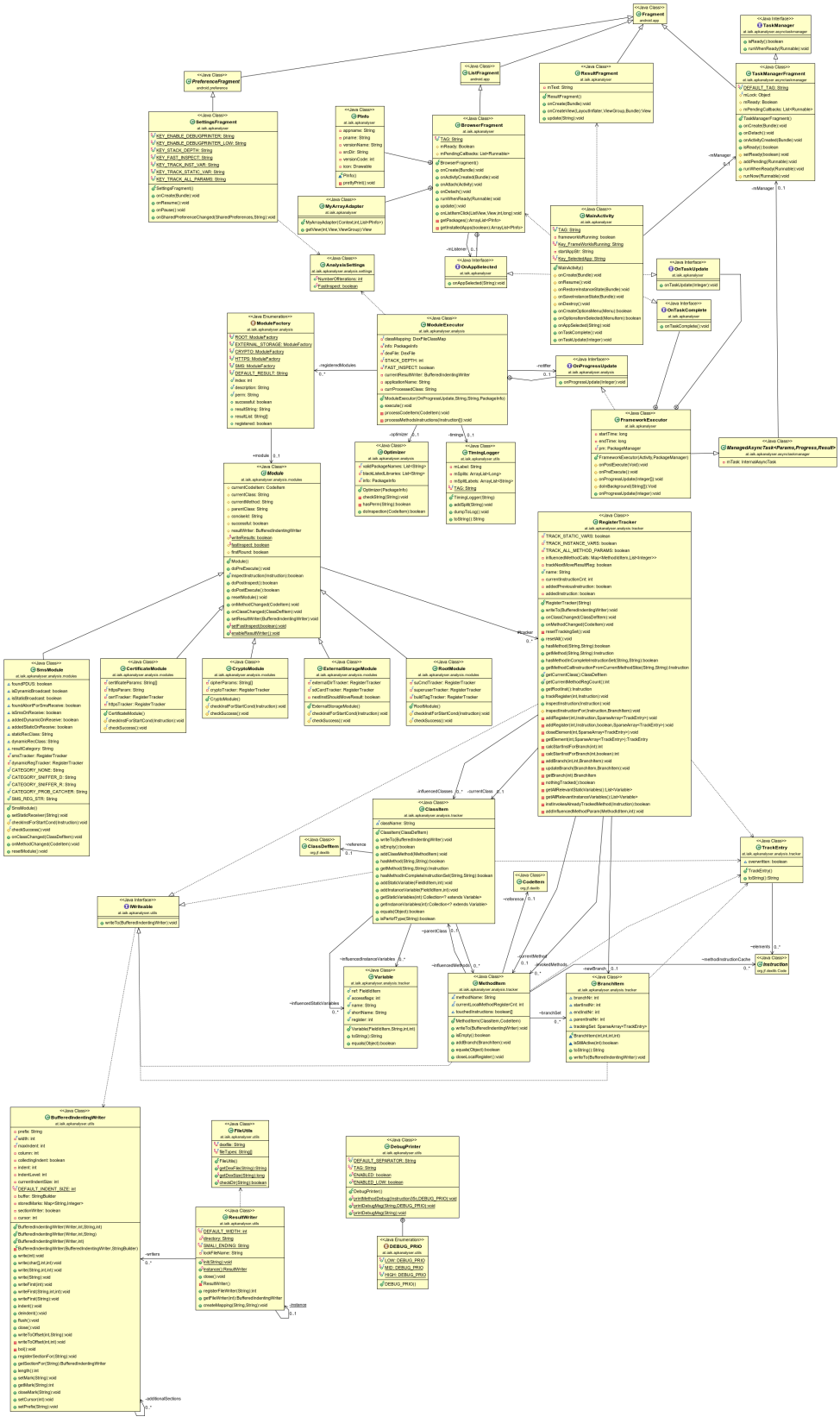

Figure B.1: The complete framework architecture including the analyser and the execution environment on the device.

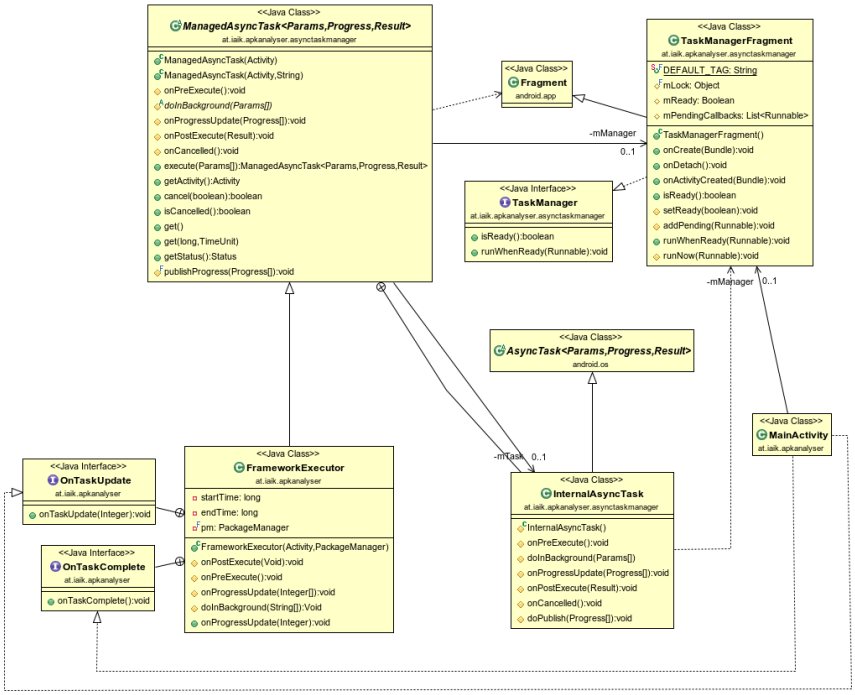

Figure B.2: This class diagram shows the callback mechanism of the asynchronous task, which executes the framework.

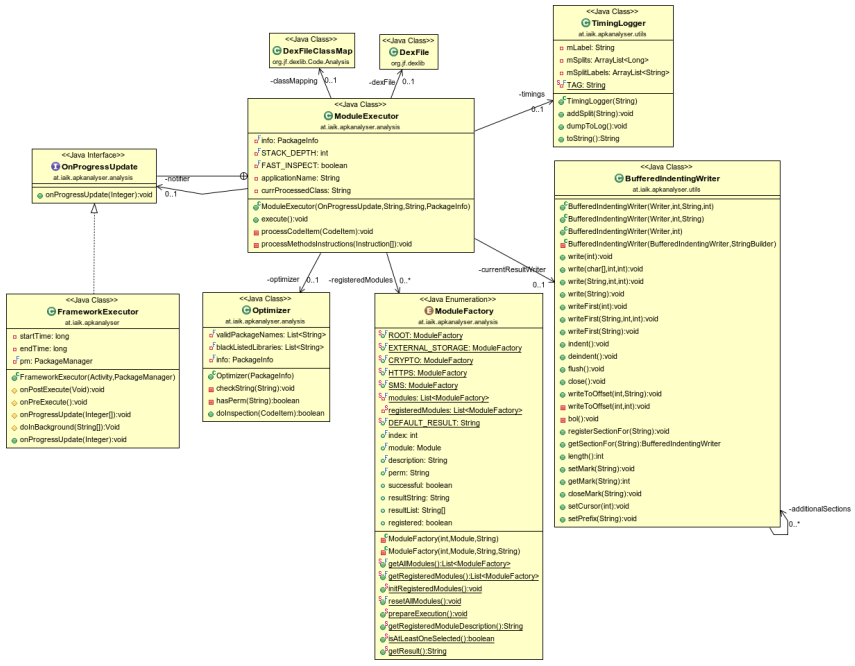

Figure B.3: This class diagram shows the three categories of the execution environment, the module executor and the entry point.

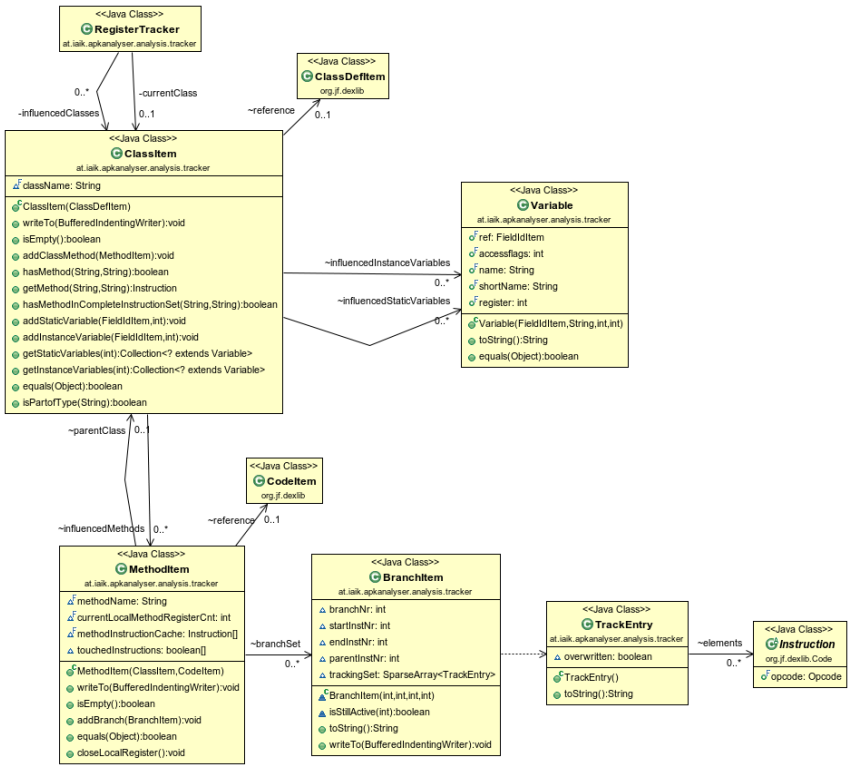

Figure B.4: Class Diagram: Slicing Architecture

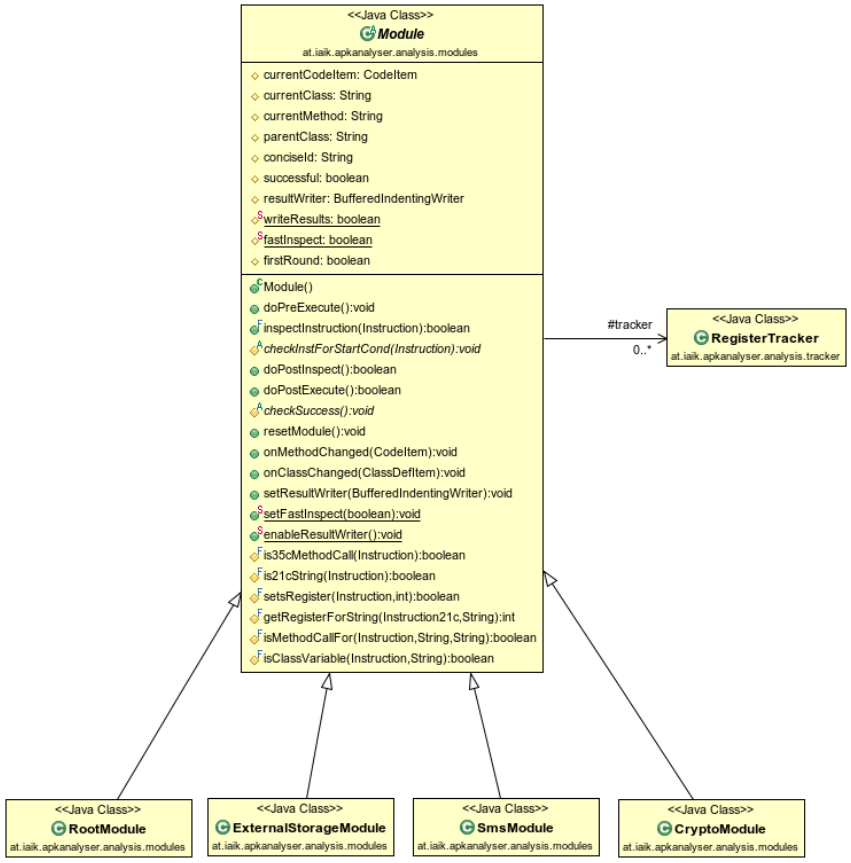

Figure B.5: Class Diagram: Module Architecture

Appendix C

Dalvik Opcodes

Example opcode: *move vA, vB*

- *move* is the base opcode.

  Opcodes can have additional suffixes for indicating different register sizes.

  The suffix *wide* indicates that it operates on 64 bits.

  The suffix *from16* indicates that the opcode has a 16-bit register reference as a source.
- *vA* is the destination register which must be in the range of 0 to 255.
- *vB* is the source register which must be in the range of 0 to 255.

Opcode & Format	Mnemonic / Syntax
00 10x	nop
01 12x	move vA, vB
02 22x	move/from16 vAA vBBBB
03 32x	move/16 vAAAA, vBBBB
04 12x	move-wide vA, vB
05 22x	move-wide/from16 vAA, vBBBB
06 32x	move-wide/16 vAAAA, vBBBB
07 12x	move-object vA, vB
08 22x	move-object/from16 vAA, vBBBB
09 32x	move-object/16 vAAAA, vBBBB
0a 11x	move-result vAA
0b 11x	move-result-wide vAA
0c 11x	move-result-object vAA
0d 11x	move-exception vAA
0e 10x	return-void
0f 11x	return vAA
10 11x	return-wide vAA
11 11x	return-object vAA
12 11n	const/4 vA, #+B
13 21s	const/16 vAA, #+BBBB
14 31i	const vAA, #+BBBBBBBB
15 21h	const/high16 vAA, #+BBBB0000

16 21s	const-wide/16 vAA, #+BBBB
17 31i	const-wide/32 vAA, #+BBBBBBBB
18 51l	const-wide vAA, #+BBBBBBBBBBBBBBBB
19 21h	const-wide/high16 vAA, #+BBBB000000000000
1a 21c	const-string vAA, string@BBBB
1b 31c	const-string/jumbo vAA, string@BBBBBBBB
1c 21c	const-class vAA, type@BBBB
1d 11x	monitor-enter vAA
1e 11x	monitor-exit vAA
1f 21c	check-cast vAA, type@BBBB
20 22c	instance-of vA, vB, type@CCCC
21 12x	array-length vA, vB
22 21c	new-instance vAA, type@BBBB
23 22c	new-array vA, vB, type@CCCC
24 35c	filled-new-array {vC, vD, vE, vF, vG}, type@BBBB
25 3rc	filled-new-array/range {vCCCC . . . vNNNN}, type@BBBB
26 31t	fill-array-data vAA, +BBBBBBBB
27 11x	throw vAA
28 10t	goto +AA
29 20t	goto/16 +AAAA
2a 30t	goto/32 +AAAAAAAA
2b 31t	packed-switch vAA, +BBBBBBBB
2c 31t	sparse-switch vAA, +BBBBBBBB
2d 23x	cmpl-float vAA, vBB, vCC
2e 23x	cmpg-float vAA, vBB, vCC
2f 23x	cmpl-double vAA, vBB, vCC
30 23x	cmpg-double vAA, vBB, vCC
31 23x	cmp-long vAA, vBB, vCC
32 22t	if-eqz vAA, +BBBB
32 22t	if-nez vAA, +BBBB
32 22t	if-ltz vAA, +BBBB
32 22t	if-gez vAA, +BBBB
32 22t	if-gtz vAA, +BBBB
32 22t	if-lez vAA, +BBBB
44 23x	aget vAA, vBB, vCC
45 23x	aget-wide vAA, vBB, vCC
46 23x	aget-object vAA, vBB, vCC
47 23x	aget-boolean vAA, vBB, vCC
48 23x	aget-byte vAA, vBB, vCC
49 23x	aget-char vAA, vBB, vCC
4a 23x	aget-short vAA, vBB, vCC
4b 23x	aput vAA, vBB, vCC
4c 23x	aput-wide vAA, vBB, vCC
4d 23x	aput-object vAA, vBB, vCC
4e 23x	aput-boolean vAA, vBB, vCC
4f 23x	aput-byte vAA, vBB, vCC
50 23x	aput-char vAA, vBB, vCC
51 23x	aput-short vAA, vBB, vCC
52 22c	iget vA, vB, field@CCCC
53 22c	iget-wide vA, vB, field@CCCC

54 22c	iget-object vA, vB, field@CCCC
55 22c	iget-boolean vA, vB, field@CCCC
56 22c	iget-byte vA, vB, field@CCCC
57 22c	iget-char vA, vB, field@CCCC
58 22c	iget-short vA, vB, field@CCCC
59 22c	iput vA, vB, field@CCCC
5a 22c	iput-wide vA, vB, field@CCCC
5b 22c	iput-object vA, vB, field@CCCC
5c 22c	iput-boolean vA, vB, field@CCCC
5d 22c	iput-byte vA, vB, field@CCCC
5e 22c	iput-char vA, vB, field@CCCC
5f 22c	iput-short vA, vB, field@CCCC
60 21c	sget vA, field@BBBB
61 21c	sget-wide vA, field@BBBB
62 21c	sget-object vA, field@BBBB
63 21c	sget-boolean vA, field@BBBB
64 21c	sget-byte vA, field@BBBB
65 21c	sget-char vA, field@BBBB
66 21c	sget-short vA, field@BBBB
67 21c	sput vA, field@BBBB
68 21c	sput-wide vA, field@BBBB
69 21c	sput-object vA, field@BBBB
6a 21c	sput-boolean vA, field@BBBB
6b 21c	sput-byte vA, field@BBBB
6c 21c	sput-char vA, field@BBBB
6d 21c	sput-short vA, field@BBBB
6e 35c	invoke-virtual {vC, vD, vE, vF, vG}, meth@BBBB
6f 35c	invoke-super {vC, vD, vE, vF, vG}, meth@BBBB
70 35c	invoke-direct {vC, vD, vE, vF, vG}, meth@BBBB
71 35c	invoke-static {vC, vD, vE, vF, vG}, meth@BBBB
72 35c	invoke-interface {vC, vD, vE, vF, vG}, meth@BBBB
74 3rc	invoke-virtual/range {vCCCC ... vNNNN } meth@BBBB
75 3rc	invoke-super/range {vCCCC ... vNNNN } meth@BBBB
76 3rc	invoke-direct/range {vCCCC ... vNNNN } meth@BBBB
77 3rc	invoke-static/range {vCCCC ... vNNNN } meth@BBBB
78 3rc	invoke-interface/range {vCCCC ... vNNNN } meth@BBBB
7b 12x	neg-int vA, vB
7c 12x	not-int vA, vB
7d 12x	neg-long vA, vB
7e 12x	not-long vA, vB
7f 12x	neg-float vA, vB
80 12x	neg-double vA, vB
81 12x	int-to-long vA, vB
82 12x	int-to-float vA, vB
83 12x	int-to-double vA, vB
84 12x	long-to-int vA, vB
85 12x	long-to-float vA, vB
86 12x	long-to-double vA, vB
87 12x	float-to-int vA, vB
88 12x	float-to-long vA, vB

89 12x	float-to-double vA, vB
8a 12x	double-to-int vA, vB
8b 12x	double-to-long vA, vB
8c 12x	double-to-float vA, vB
8d 12x	int-to-byte vA, vB
8e 12x	int-to-char vA, vB
8f 12x	int-to-short vA, vB
90 23x	add-int vAA, vBB, vCC
91 23x	sub-int vAA, vBB, vCC
92 23x	mul-int vAA, vBB, vCC
93 23x	div-int vAA, vBB, vCC
94 23x	rem-int vAA, vBB, vCC
95 23x	and-int vAA, vBB, vCC
96 23x	or-int vAA, vBB, vCC
97 23x	xor-int vAA, vBB, vCC
98 23x	shl-int vAA, vBB, vCC
99 23x	shr-int vAA, vBB, vCC
9a 23x	ushr-int vAA, vBB, vCC
9b 23x	add-long vAA, vBB, vCC
9c 23x	sub-long vAA, vBB, vCC
9d 23x	mul-long vAA, vBB, vCC
9e 23x	div-long vAA, vBB, vCC
9f 23x	rem-long vAA, vBB, vCC
a0 23x	and-long vAA, vBB, vCC
a1 23x	or-long vAA, vBB, vCC
a2 23x	xor-long vAA, vBB, vCC
a3 23x	shl-long vAA, vBB, vCC
a4 23x	shr-long vAA, vBB, vCC
a5 23x	ushr-long vAA, vBB, vCC
a6 23x	add-float vAA, vBB, vCC
a7 23x	sub-float vAA, vBB, vCC
a8 23x	mul-float vAA, vBB, vCC
a9 23x	div-float vAA, vBB, vCC
aa 23x	rem-float vAA, vBB, vCC
ab 23x	add-double vAA, vBB, vCC
ac 23x	sub-double vAA, vBB, vCC
ad 23x	mul-double vAA, vBB, vCC
ae 23x	div-double vAA, vBB, vCC
af 23x	rem-double vAA, vBB, vCC
b0 12x	add-int/2addr vA, vB
b1 12x	sub-int/2addr vA, vB
b2 12x	mul-int/2addr vA, vB
b3 12x	div-int/2addr vA, vB
b4 12x	rem-int/2addr vA, vB
b5 12x	and-int/2addr vA, vB
b6 12x	or-int/2addr vA, vB
b7 12x	xor-int/2addr vA, vB
b8 12x	shl-int/2addr vA, vB
b9 12x	shr-int/2addr vA, vB
ba 12x	ushr-int/2addr vA, vB

bb 12x	add-long/2addr vA, vB
bc 12x	sub-long/2addr vA, vB
bd 12x	mul-long/2addr vA, vB
be 12x	div-long/2addr vA, vB
bf 12x	rem-long/2addr vA, vB
c0 12x	and-long/2addr vA, vB
c1 12x	or-long/2addr vA, vB
c2 12x	xor-long/2addr vA, vB
c3 12x	shl-long/2addr vA, vB
c4 12x	shr-long/2addr vA, vB
c5 12x	ushr-long/2addr vA, vB
c6 12x	add-float/2addr vA, vB
c7 12x	sub-float/2addr vA, vB
c8 12x	mul-float/2addr vA, vB
c9 12x	div-float/2addr vA, vB
ca 12x	rem-float/2addr vA, vB
cb 12x	add-double/2addr vA, vB
cc 12x	sub-double/2addr vA, vB
cd 12x	mul-double/2addr vA, vB
ce 12x	div-double/2addr vA, vB
cf 12x	rem-double/2addr vA, vB
d0 22s	add-int/lit16 vA, vB #+CCCC
d1 22s	rsub-int/lit16 vA, vB #+CCCC
d2 22s	mul-int/lit16 vA, vB #+CCCC
d3 22s	div-int/lit16 vA, vB #+CCCC
d4 22s	rem-int/lit16 vA, vB #+CCCC
d5 22s	and-int/lit16 vA, vB #+CCCC
d6 22s	or-int/lit16 vA, vB #+CCCC
d7 22s	xor-int/lit16 vA, vB #+CCCC
d8 22b	add-int/lit8 vAA, vBB #+CC
d9 22b	rsub-int/lit8 vAA, vBB #+CC
da 22b	mul-int/lit8 vAA, vBB #+CC
db 22b	div-int/lit8 vAA, vBB #+CC
dc 22b	rem-int/lit8 vAA, vBB #+CC
dd 22b	and-int/lit8 vAA, vBB #+CC
de 22b	or-int/lit8 vAA, vBB #+CC
df 22b	xor-int/lit8 vAA, vBB #+CC
e0 22b	shl-int/lit8 vAA, vBB #+CC
e1 22b	shr-int/lit8 vAA, vBB #+CC
e2 22b	ushr-int/lit8 vAA, vBB #+CC

Table C.1: Dalvik's Opcode List [*Bytecode for the Dalvik VM* 2007]

Bibliography

Andreas, Floemer [2013]. *Android-Versionsverteilung: Jelly Bean auf über 50% aller Geräte*. Nov. 2013. http://www.androidnext.de/news/android-versionsverteilung-jelly-bean-auf-ueber-50-aller-geraete/ (cited on page 2).

Andrews, Keith [2012]. *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*. Graz University of Technology, Austria. Oct. 22, 2012. http://ftp.iicm.edu/pub/keith/thesis/ (cited on page xiii).

Android Developer Guide - Activity [2013]. June 2013. http://developer.android.com/reference/android/app/Activity.html (cited on page 12).

Android Developer Guide - ANR [2013]. June 2013. http://developer.android.com/training/articles/perf-anr.html (cited on page 51).

Android Developer Guide - Context [2013]. June 2013. http://developer.android.com/reference/android/content/Context.html#MODE_WORLD_WRITEABLE (cited on page 16).

Android Developer Guide - Manifest [2013]. June 2013. http://developer.android.com/guide/topics/manifest/(activity-element.html%20%7C%20receiver-element.html%20%7C%20provider-element.html%20%7C%20service-element.html) (cited on page 17).

Android Developer Guide - Permission's Protection Level [2013]. June 2013. http://developer.android.com/guide/topics/manifest/permission-element.html (cited on page 17).

Android Developer Guide - Provider [2013]. June 2013. http://developer.android.com/guide/topics/manifest/provider-element.html (cited on page 13).

Android Developer Guide - Receiver [2013]. June 2013. http://developer.android.com/guide/topics/manifest/receiver-element.html (cited on page 12).

Android Developer Guide - Service [2013]. June 2013. http://developer.android.com/guide/components/services.html (cited on page 12).

Android Kernel Development [2013]. Oct. 2013. http://source.android.com/devices/index.html (cited on page 10).

Android System Architecture [2013]. Feb. 2013. http://upload.wikimedia.org/wikipedia/commons/thumb/a/af/Android-System-Architecture.svg/1000px-Android-System-Architecture.svg.png (cited on pages xiii, 11).

Andrus, Jeremy et al. [2011]. "Cells: a virtual mobile smartphone architecture". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pages 173–187 (cited on page 41).

Barrera, David et al. [2010]. "A methodology for empirical analysis of permission-based security models and its application to android". In: *Proceedings of the 17th ACM conference on Computer and communications security*. ACM. 2010, pages 73–84 (cited on pages 4, 35).

Becher, Michael et al. [2011]. “Mobile security catching up? revealing the nuts and bolts of the security of mobile devices”. In: *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE. 2011, pages 96–111 (cited on page 40).

Becker, Arno and Marcus Pant [2010]. *Android 2*. 2nd edition. dpunkt.verlag, 2010. ISBN 9783898646772 (cited on pages 13, 15).

Bender, Waldemar [2010]. “Entwicklung eines IF-MAP Clients für die Android Plattform”. *Bachelorarbeit im Studiengang Angewandte Informatik in der Abteilung Informatik der Fakultät IV an der Fachhochschule Hannover* (2010) (cited on page 18).

Benton, Nick [2004]. “Simple relational correctness proofs for static analyses and program transformations”. *ACM SIGPLAN Notices* 39.1 (2004), pages 14–25 (cited on page 40).

Beresford, Alastair R et al. [2011]. “MockDroid: trading privacy for application functionality on smartphones”. In: *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*. ACM. 2011, pages 49–54 (cited on page 38).

Bergeron, Jean, Mourad Debbabi, Jules Desharnais, et al. [2001]. “Static detection of malicious code in executable programs”. *Int. J. of Req. Eng* 2001 (2001), pages 184–189 (cited on page 40).

Bergeron, Jean, Mourad Debbabi, Mourad M Erhioui, et al. [1999]. “Static analysis of binary code to isolate malicious behaviors”. In: *Enabling Technologies: Infrastructure for Collaborative Enterprises, 1999.(WET ICE’99) Proceedings. IEEE 8th International Workshops on*. IEEE. 1999, pages 184–189 (cited on page 38).

Blasing, Thomas et al. [2010]. “An android application sandbox system for suspicious software detection”. In: *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*. IEEE. 2010, pages 55–62 (cited on page 24).

Bugiel, Sven et al. [2011]. “Practical and lightweight domain isolation on android”. In: *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM. 2011, pages 51–62 (cited on page 15).

Burguera, Iker, Urko Zurutuza, and Simin Nadjm-Tehrani [2011]. “Crowdroid: behavior-based malware detection system for android”. In: *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM. 2011, pages 15–26 (cited on page 34).

Butler, Margaret [2011]. “Android: Changing the mobile landscape”. *Pervasive Computing, IEEE* 10.1 (2011), pages 4–7 (cited on page 10).

Bytecode for the Dalvik VM [2007]. 2007. `http://source.android.com/devices/tech/dalvik/dalvik-bytecode.html` (cited on page 93).

Center, IBM T.J. Watson Research [2011]. *T. J. Watson Libraries for Analysis (WALA)*. Mar. 2011. `http://wala.sourceforge.net/wiki/index.php/Main_Page` (cited on page 37).

Chin, Erika et al. [2011]. “Analyzing inter-application communication in Android”. In: *Proceedings of the 9th international conference on Mobile systems, applications, and services*. MobiSys ’11. Bethesda, Maryland, USA: ACM, 2011, pages 239–252. ISBN 978-1-4503-0643-0. doi:10.1145/1999995.2000018. `http://doi.acm.org/10.1145/1999995.2000018` (cited on pages 3, 12, 15, 16, 41, 43).

Ching, Lee Pak and LUI Chi Shing John [2012]. “ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-Virus Systems”. *DIMVA12* (2012). `http://dspace.lib.cuhk.edu.hk/handle/2006/398721` (cited on pages 34, 42).

Cohen, Fred [1984]. *Computer Viruses - Theory and Experiments*. 1984. http://web.eecs.umich.edu/~aprakash/eecs588/handouts/cohen-viruses.html (cited on pages 32, 33).

Cohen, Fred [1988]. "On the implications of computer viruses and methods of defense". *Comput. Secur.* 7.2 (Apr. 1988), pages 167–184. ISSN 0167-4048. doi:10.1016/0167-4048(88)90334-3. http://dx.doi.org/10.1016/0167-4048(88)90334-3 (cited on page 33).

Cooperation, Microsoft. *Windows Mobile-based Smartphones*. http://www.microsoft.com/windowsmobile/smartphone/default.mspx (cited on page 40).

Cyrus, Peikari et al. *Summer Brings Mosquito-Born Malware*. http://www.informit.com/articles/article.asp?p=327994%5C&seqNum=1 (cited on page 40).

Davi, Lucas et al. [2011]. "Privilege escalation attacks on android". In: *Information Security*. Springer, 2011, pages 346–360 (cited on pages 3, 5, 35).

Developers, Android [2011]. "What is android?" *ht tp://developer. android. com/guide/basics/what-is-android. html* 2 (2011) (cited on page 11).

Dietz, Michael et al. [2011]. "QUIRE: Lightweight Provenance for Smart Phone Operating Systems." In: *USENIX Security Symposium*. 2011 (cited on page 41).

Egele, Manuel et al. [2011]. "PiOS: Detecting privacy leaks in iOS applications". In: *Proceedings of the Network and Distributed System Security Symposium*. 2011 (cited on page 37).

Ehringer, David [2010]. "The dalvik virtual machine architecture". *Techn. report (March 2010)* (2010) (cited on page 18).

Enck, William, Peter Gilbert, et al. [2010]. "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones". In: *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. 2010, pages 1–6 (cited on pages 4, 24, 36).

Enck, William, Damien Octeau, et al. [2011]. "A Study of Android Application Security." In: *USENIX security symposium*. 2011 (cited on pages 13, 43).

Enck, William, Machigar Ongtang, and Patrick McDaniel [2008]. "Mitigating Android software misuse before it happens" (2008) (cited on page 1).

Enck, William, Machigar Ongtang, and Patrick McDaniel [2009a]. "On lightweight mobile phone application certification". In: *Proceedings of the 16th ACM conference on Computer and communications security*. ACM. 2009, pages 235–245 (cited on page 41).

Enck, William, Machigar Ongtang, and Patrick McDaniel [2009b]. "Understanding android security". *Security & Privacy, IEEE* 7.1 (2009), pages 50–57 (cited on pages 12, 14, 16).

Felt, Adrienne Porter, Erika Chin, et al. [2011]. "Android permissions demystified". In: *Proceedings of the 18th ACM conference on Computer and communications security*. ACM. 2011, pages 627–638 (cited on pages 10, 17, 41, 70).

Felt, Adrienne Porter, Matthew Finifter, et al. [2011]. "A survey of mobile malware in the wild". In: *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM. 2011, pages 3–14 (cited on page 40).

Felt, Adrienne Porter, Kate Greenwood, and David Wagner [2011]. "The effectiveness of application permissions". In: *Proceedings of the 2nd USENIX conference on Web application development*. USENIX Association. 2011, pages 75–86 (cited on page 35).

Felt, Adrienne Porter and David Wagner [2011]. "Phishing on mobile devices". *University of California, Berkeley* (2011) (cited on page 35).

Felt, Adrienne Porter, Helen J Wang, et al. [2011]. “Permission Re-Delegation: Attacks and Defenses.” In: *USENIX Security Symposium*. 2011 (cited on pages 13, 14, 41).

F-Secure. *SMS Killer*. http://www.f-secure.com/v-descs/sms.shtml (cited on page 40).

F-Secure [2012]. *Mobile Threat Report Q2 2012*. Aug. 2012. http://www.f-secure.com/weblog/archives/MobileThreatReport_Q2_2012.pdf (cited on page 32).

Fuchs, Adam P, Avik Chaudhuri, and Jeffrey S Foster [2009]. “SCanDroid: Automated security certification of Android applications”. *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/~ avik/projects/scandroidascaa* (2009) (cited on page 37).

Ganapathy, Vinod et al. [2003]. “Buffer overrun detection using linear programming and static analysis”. In: *Proceedings of the 10th ACM conference on Computer and communications security*. ACM. 2003, pages 345–354 (cited on page 39).

Gibler, Clint et al. [2012]. “AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale”. In: *Trust and Trustworthy Computing*. Edited by Stefan Katzenbeisser et al. Volume 7344. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pages 291–307. ISBN 978-3-642-30920-5. doi:10.1007/978-3-642-30921-2_17. http://dx.doi.org/10.1007/978-3-642-30921-2_17 (cited on pages 35, 37).

Goasduff, Laurence and Christy Pettey [2011]. *Gartner Says Worldwide Mobile Phone Sales Grew 35 Percent in Third Quarter 2010; Smartphone Sales Increased 96 Percent*. Gartner. 2011. http://www.gartner.com/newsroom/id/1466313 (cited on page 32).

Grace, Michael et al. [2012]. “Systematic detection of capability leaks in stock Android smartphones”. In: *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*. 2012 (cited on page 41).

Guo, Chuanxiong, Helen J Wang, and Wenwu Zhu [2004]. “Smart-phone attacks and defenses”. In: *HotNets III*. 2004 (cited on page 40).

Helfer, Jonas and Ty Lin [2012]. “Giving the User Control over Android Permissions” (2012) (cited on page 42).

Helmut, Reimer [2013]. “ESET Secure Authentication: Sicherer Zugang zu VPN und Outlook Web App”. German. *Datenschutz und Datensicherheit - DuD* 37.7 (2013), pages 479–480. ISSN 1614-0702. doi:10.1007/s11623-013-0194-y. http://dx.doi.org/10.1007/s11623-013-0194-y (cited on page 72).

Hornyack, Peter et al. [2011]. “These aren’t the droids you’re looking for: retrofitting android to protect data from imperious applications”. In: *Proceedings of the 18th ACM conference on Computer and communications security*. ACM. 2011, pages 639–652 (cited on page 38).

Horwitz, Susan, Thomas Reps, and David Binkley [1990]. “Interprocedural slicing using dependence graphs”. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.1 (1990), pages 26–60 (cited on page 39).

Hypponen, M. *State of cell phone malware in 2007 (2007)* (cited on page 40).

Islam, Zak [2012]. *Google Play Matches Apple’s iOS With 700,000 Apps*. Oct. 2012. http://www.tomsguide.com/us/Google-Play-Android-Apple-iOS,news-16235.html (cited on page 10).

J. Malenfant, M. Jacques and F.-N. Demers [1996]. “A Tutorial on Behavioral Reflection and its Implementation” (1996). http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/malenfant/malenfant.pdf (cited on page 53).

Kim, Hyojun, Nitin Agrawal, and Cristian Ungureanu [2012]. "Revisiting storage for smartphones". *ACM Transactions on Storage (TOS)* 8.4 (2012), page 14 (cited on page 16).

Krinke, Jens [2003]. *Advanced Slicing of Sequential and Concurrent Programs*. Apr. 2003 (cited on page 25).

Kroop, Sebastian [2008]. "Evaluierung der Android-Plattform anhand einer Referenzanwendung" (2008) (cited on pages 12, 13).

Kuhn, Tobias, Thomas Ritter, and Bernhard Mitschang [2012]. "Das Sicherheitskonzept von Android" (2012) (cited on page 16).

Labs, Kaspersky [2004]. *Viruses move to mobile phones*. 2004. http://www.kaspersky.com/news?id=149499226 (cited on page 40).

Lam, Monica et al. [2006]. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 2006 (cited on pages 25, 27).

Landesman, Mary [2012]. *What is a Virus Signature?* 2012. http://antivirus.about.com/od/whatisavirus/a/virussignature.htm (cited on page 33).

Lange, Matthias et al. [2011]. "L4Android: a generic operating system framework for secure smartphones". In: *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM. 2011, pages 39–50 (cited on pages 41, 65).

Larochelle, David and David Evans [2001]. "Statically Detecting Likely Buffer Overflow Vulnerabilities." In: *USENIX Security Symposium*. Washington DC. 2001, pages 177–190 (cited on page 39).

Lawton, G. [2008]. "Is It Finally Time to Worry about Mobile Malware?" *Computer* 41.5 (May 2008), pages 12–14. ISSN 0018-9162. doi:10.1109/MC.2008.159 (cited on page 32).

Leavitt, Neal [2005]. "Mobile phones: the next frontier for hackers?" *Computer* 38.4 (2005), pages 20–23 (cited on page 32).

Lettner, Michael, Michael Tschernuth, and Rene Mayrhofer [2012]. "Mobile platform architecture review: Android, iPhone, Qt". In: *Computer Aided Systems Theory–EUROCAST 2011*. Springer, 2012, pages 544–551 (cited on page 40).

Lo, Raymond W, Karl N Levitt, and Ronald A Olsson [1995]. "MCF: A malicious code filter". *Computers & Security* 14.6 (1995), pages 541–566 (cited on page 38).

Mahaffey, K and J Hering. *App Attack-Surviving the Explosive Growth of Mobile Apps* (cited on pages 17, 41).

"Malicious mobile threats report 2010/2011" [2011] (2011) (cited on page 32).

McCaskill, Mary K [1998]. "Grammar, Punctuation, and Capitalization". *A Handbook for Technical Writers and Editors. NASA SP-7084*. 20 (1998). http://www.eknigu.org/get/L%5C_Languages/LEn%5C_English/McCaskill.Grammar,%20punctuation,%20and%20capitalization.%20A%20handbook%20for%20technical%20writers%20and%20editors.

McClurg, Jedidiah, Jonathan Friedman, and William Ng [2012]. "Android Privacy Leak Detection via Dynamic Taint Analysis" (2012) (cited on page 42).

Milošević, Nikola [2013]. *History of Malware*. 2013. http://cryptome.org/2013/02/malware-history.pdf (cited on page 31).

Mishra, Umakant [2012]. *Methods of virus detection and their limitations*. 2012. http://ssrn.com/abstract=1916708 (cited on page 33).

Myers, Andrew C and Barbara Liskov [1997]. “A decentralized model for information flow control”. In: *ACM SIGOPS Operating Systems Review*. Volume 31. 5. ACM. 1997, pages 129–142 (cited on page 39).

Nagamine, Kathy [2012]. *Android Marks Fourth Anniversary Since Launch with 75.0% Market Share in Third Quarter, According to IDC*. 2012. http://www.idc.com/getdoc.jsp?containerId=prUS23771812#.URtLEnxQBgE (cited on page 10).

Newsome, James and Dawn Song [2005]. “Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software” (2005) (cited on page 39).

Nolan, Godfrey [2012]. *Decompiling Android*. 2nd edition. Apress, 2012. ISBN 9781430242482 (cited on pages 18, 19).

One, Aleph [1996]. “Smashing the stack for fun and profit”. *Phrack magazine* 7.49 (1996), page 365 (cited on page 39).

Ongtang, Machigar et al. [2012]. “Semantically rich application-centric security in Android”. *Security and Communication Networks* 5.6 (2012), pages 658–673 (cited on page 41).

Orthacker, Clemens et al. [2012]. “Android Security Permissions–Can we trust them?” In: *Security and Privacy in Mobile Information and Communication Systems*. Springer, 2012, pages 40–51 (cited on pages 12, 17, 34, 35).

Ottenstein, Karl J and Linda M Ottenstein [1984]. “The program dependence graph in a software development environment”. In: *ACM Sigplan Notices*. Volume 19. 5. ACM. 1984, pages 177–184 (cited on page 24).

Paller, G [2012]. *Dalvik opcodes*. 2012 (cited on pages 19, 22).

Paul, Ryan [2007]. “Why Google chose the Apache Software License over GPLv2 for Android”. *Ars Technica,(November 06, 2007)* (2007) (cited on page 10).

Pistoia, Marco et al. [2007]. “A survey of static analysis methods for identifying security vulnerabilities in software systems”. *IBM systems journal* 46.2 (2007), pages 265–288 (cited on pages 38, 39).

Portokalidis, Georgios et al. [2010]. “Paranoid Android: versatile protection for smartphones”. In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACSAC ’10. Austin, Texas: ACM, 2010, pages 347–356. ISBN 978-1-4503-0133-6. doi:10.1145/1920261.1920313. http://doi.acm.org/10.1145/1920261.1920313 (cited on page 44).

Rao, Bharat and Louis Minakakis [2003]. “Evolution of mobile location-based services”. *Communications of the ACM* 46.12 (2003), pages 61–65 (cited on page 32).

Rastogi, Vaibhav, Yan Chen, and Xuxian Jiang [2013]. “Evaluating Android Anti-malware against Transformation Attacks” (2013) (cited on page 76).

Reps, Thomas et al. [1994]. *Speeding up slicing*. Volume 19. 5. ACM, 1994 (cited on page 39).

Rhee, Keunwoo, Hawon Kim, and Hac Yun Na [2012]. “Security Test Methodology for an Agent of a Mobile Device Management System” (2012) (cited on page 43).

Scheid, Julian [2012]. “Kapitel 1 Sicherheit mobiler Geräte-Schutzmaßnahmen, Angriffsarten & Angriffserkennung auf Android”. *Ausgewählte Themen der IT-Sicherheit* (2012), page 7 (cited on page 14).

Schmidt, A.-D. et al. [2009]. “Smartphone malware evolution revisited: Android next target?” In: *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*. Oct. 2009, pages 1–7. doi:10.1109/MALWARE.2009.5403026 (cited on page 32).

Shabtai, Asaf, Yuval Fledel, et al. [2010]. “Google android: A comprehensive security assessment”. *Security & Privacy, IEEE* 8.2 (2010), pages 35–44 (cited on page 1).

Shabtai, Asaf, Uri Kanonov, et al. [2012]. “"Andromaly": a behavioral malware detection framework for android devices”. *J. Intell. Inf. Syst.* 38.1 (Feb. 2012), pages 161–190. ISSN 0925-9902. doi:10.1007/s10844-010-0148-x. http://dx.doi.org/10.1007/s10844-010-0148-x (cited on page 42).

Sharma, Kriti et al. [2013]. “Malware Analysis for Android Operating”. In: *8th Annual Symposium on Information Assurance (ASIA’13)*. 2013, page 31 (cited on page 43).

Shin, Wook et al. [2009]. “Towards formal analysis of the permission-based security model for android”. In: *Wireless and Mobile Communications, 2009. ICWMC’09. Fifth International Conference on*. IEEE. 2009, pages 87–92 (cited on page 40).

Singh, Ankush Kumar and Shree Garg [2012]. “Detection of Repackaged Smartphone Applications On Android” (2012) (cited on page 42).

Snelting, Gregor, Torsten Robschink, and Jens Krinke [2006]. “Efficient path conditions in dependence graphs for software safety analysis”. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15.4 (2006), pages 410–457 (cited on page 39).

Spreitzenbarth, Michael [2013]. “Dissecting the Droid: Forensic Analysis of Android and its malicious Applications Sezierung eines Androiden: Forensische Analyse von Android und dessen schadhaften Applikationen” (2013) (cited on page 1).

Static Android Analysis Framework [2013]. 2013. https://code.google.com/p/saaf/ (cited on page 44).

The twenty most critical internet security vulnerabilities. www.sans.org/top20 (cited on page 39).

Tonella, Paolo [2005]. “Reverse engineering of object oriented code”. In: *Proceedings of the 27th international conference on Software engineering*. ACM. 2005, pages 724–725 (cited on page 27).

Wagner, David and R Dean [2001]. “Intrusion detection via static analysis”. In: *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*. IEEE. 2001, pages 156–168 (cited on page 39).

Wagner, David, Jeffrey S Foster, et al. [2000]. “A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities.” In: *NDSS*. 2000, pages 2000–02 (cited on page 39).

Weiser, Mark [1981]. “Program slicing”. In: *Proceedings of the 5th international conference on Software engineering*. IEEE Press. 1981, pages 439–449 (cited on page 24).

Wotawa, Franz [2002]. “On the relationship between model-based debugging and program slicing”. *Artificial Intelligence* 135.1–2 (2002), pages 125–143. ISSN 0004-3702. doi:http://dx.doi.org/10.1016/S0004-3702(01)00161-8. http://www.sciencedirect.com/science/article/pii/S0004370201001618 (cited on pages 25, 29).

You, Ilsun and Kangbin Yim [2010]. “Malware obfuscation techniques: A brief survey”. In: *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*. IEEE. 2010, pages 297–300 (cited on page 76).

Zeller, Andreas [2009]. *Why programs fail: a guide to systematic debugging*. Access Online via Elsevier, 2009 (cited on page 27).

Zhou, Yajin and Xuxian Jiang [2012]. “Dissecting Android Malware: Characterization and Evolution”. In: *Security and Privacy (SP), 2012 IEEE Symposium on*. May 2012, pages 95–109. doi:10.1109/SP.2012.16 (cited on pages 1, 6, 32, 34).

Zhou, Yajin, Zhi Wang, et al. [2012]. “Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets”. In: *Proceedings of the 19th Annual Network and Distributed System Security Symposium*. 2012 (cited on pages 34, 40).

Zhou, Yajin, Xinwen Zhang, et al. [2011]. “Taming information-stealing smartphone applications (on Android)”. In: *Trust and Trustworthy Computing*. Springer, 2011, pages 93–107 (cited on pages 32, 38).

Index