Static Data Flow Analysis for Vulnerable Permission Configurations of Android Applications

January 5, 2015

supervised by:
Prof. Dr. Sibylle Schupp
Declaration

I, Ernesto Isaac Ramírez Silva, declare that this Master’s project work titled, “Static Data Flow Analysis for Vulnerable Permission Configurations of Android Applications”, the work presented here, and the software source code are my own. I confirm that:

- I have written this document independently, and that I have not made use of any aid other than those acknowledged in this document.
- This work was done wholly during the Summer Term 2014 and Winter Semester 2014/2015 at this University.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this document is entirely my own work.
- I have acknowledged all main sources of help.
- I developed the software source code in my own, and if I used a third-party component, it is referred in this document.

Signed:

Ernesto Isaac Ramírez Silva
January 5, 2015,
Hamburg, Germany
Acknowledgements

I would like to express my thanks to the people who directly and indirectly participated in making this project possible.

First of all, I would like to thank my project advisor Prof. Dr. Sibylle Schupp\footnote{http://www.sts.tu-harburg.de/%7Eschupp/}, last accessed on January 5, 2015 for her invaluable support and helpful review throughout this academic work. Her lectures of “Verified Software Systems” and “Software Analysis” awakened my interest in formal methods behind IT Security. My participation in the summer semester 2014 “Software Analysis for Mobile Applications” seminar was the beginning of this adventure.

Additionally, I would like to express my deep appreciation towards M.Sc. Dragos Sbirlea\footnote{http://www.cs.rice.edu/ ds20/}, Phd. candidate at RICE university, for his advice in the implementation of the tool, particularly in the Permission Mapper. I also profited greatly from the remarks of M.Sc. Moti Cohen from the Efi Arazi School of Computer Science about my code and his answers for my queries about his master’s thesis. Thanks to M.Sc. Steven Arzt\footnote{https://ec-spride.sit.fraunhofer.de/forschungsgruppen/secure-software-engineering/staff/steven-arzt/}, last accessed on January 5, 2015 from the Software Engineering Research Group at the Darmstadt University for his suggestions about how to solve some issues with the Android source code. I am also grateful to M.A.Sc Kathy Au, Software Engineer at Google, graduated from the University of Toronto in Electrical and Computer Engineering, who answered my queries related to her master thesis (PSCOUT), and her answers gave me the idea for a paper’s topic.

Warm thanks go to Thomas Rahmlow\footnote{http://www.sts.tu-harburg.de/people/t.rahmlow/}, system administrator of the STS institute, who helped me to obtain the necessary software from IBM so that this project came true.

Last, but not least, I want to thank my girlfriend, family and friends for supporting me during this journey.

Ernesto Isaac Ramírez Silva

January 5, 2015
Contents

Declaration iii

Acknowledgements v

1. Abbreviations 1

2. Introduction 3
   2.1. Motivation ........................................ 3
   2.2. Objective ........................................ 7
   2.3. Contributions .................................... 7
   2.4. Outline .......................................... 8

3. Fundamentals 9
   3.1. Android security framework ..................... 9
       3.1.1. Application and components .................. 10
       3.1.2. Android Manifest file ........................ 15
       3.1.3. Specification of the Android Permission scheme .......................... 17
   3.2. Possible attacks to the Android permission system ............. 20
   3.3. Static Code Analysis ................................ 22
       3.3.1. Static Analysis Limitations .................. 25
       3.3.2. Program Representations used for Analysis ............. 26
   3.4. Data Flow Analysis (DFA) ......................... 27
       3.4.1. Program Representations used for Data Flow Analysis ............. 28
       3.4.2. Pointer and Alias Analysis ................. 30
       3.4.3. Data Flow Analysis for Java and Android ......... 31

4. Development 33
   4.1. Introduction ...................................... 33
   4.2. Prototype tool implementation .................... 33
       4.2.1. Preparing the analysis ...................... 34
       4.2.2. Processing the .apk file .................... 35
       4.2.3. Static data flow analysis ................. 42
       4.2.4. Permission map generation .................. 45
   4.3. Experiments ...................................... 48
   4.4. Performance Analysis ............................ 48

5. Final discussion 49
   5.1. Conclusion ...................................... 49
5.2. Future Work ........................................... 50
  5.2.1. Taint analysis ................................... 50
  5.2.2. Visual tool ....................................... 50
  5.2.3. Performance analysis .......................... 50
  5.2.4. Applications exposing the vulnerabilities .... 50

A. Hardware and software used .......................... 51
  A.1. Hardware ............................................ 51
  A.2. Software ............................................. 51

B. How to obtain the .apk files ........................... 53

C. Performance tuning for the JVM .................... 55
  C.0.1. JVM arguments of the CG generator ........... 55
  C.0.2. Categories of Java HotSpot VM Options ........ 56

D. Content of the CD ...................................... 61
# List of Figures

2.1. Internal Activity invocation or confused deputy .................................. 5  
2.2. Permission collusion by malevolent applications .................................. 5  
2.3. Application sharing user ID with a compromised application ............... 6  

3.1. Application and components ............................................................. 10  
3.2. Starting an implicit intent ................................................................. 13  
3.3. Android Manifest file (selected items) ............................................. 14  
3.4. Entities and relations in the Android permission scheme ...................... 18  
3.5. Android permissions ................................................................. 19  
3.6. Android permissions (refined) ....................................................... 20  
3.7. The nature of approximation: erring on the safe side ......................... 24  

4.1. The components of our prototype tool ............................................... 33  
4.2. Processing the .apk file ................................................................. 35  
4.3. XML SAX Parser Diagram ............................................................. 37  
4.4. Data flow analysis ....................................................................... 43  
4.5. IR_Stubs ......................................................................................... 46  
4.6. IR_NoStubs .................................................................................... 47
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.</td>
<td>List of Abbreviations</td>
<td>2</td>
</tr>
<tr>
<td>3.1.</td>
<td>Different configurations lead to different levels of vulnerability</td>
<td>22</td>
</tr>
<tr>
<td>4.1.</td>
<td>Output of the MisconfiguredAppAndroidManifest.xml</td>
<td>41</td>
</tr>
<tr>
<td>4.2.</td>
<td>Output of the MisconfiguredAppSharedUidAndroidManifest.xml</td>
<td>41</td>
</tr>
<tr>
<td>4.3.</td>
<td>WALA objects</td>
<td>42</td>
</tr>
<tr>
<td>4.4.</td>
<td>Call graph solvers</td>
<td>48</td>
</tr>
<tr>
<td>4.5.</td>
<td>Performance comparison</td>
<td>48</td>
</tr>
<tr>
<td>A.1.</td>
<td>List of hardware components</td>
<td>51</td>
</tr>
<tr>
<td>A.2.</td>
<td>List of software</td>
<td>52</td>
</tr>
</tbody>
</table>
Listings

4.1. MisconfiguredAppAndroidManifest.xml . . . . . . . . . . . . . . . . . . . . . 39
4.2. MisconfiguredAppSharedUidAndroidManifest.xml . . . . . . . . . . . . . . 39
4.3. IR .pdf file generation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.4. Call graph generation (selected illustrative instructions) . . . . . . . . . 43
List of Algorithms

1. Android Manifest Parser Algorithm ........................................... 38
2. Permission map generation ..................................................... 45
## 1. Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>What (it) stands for</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>APK</td>
<td>Android Package File</td>
</tr>
<tr>
<td>APP(s) or app(s)</td>
<td>Mobile Application(s)</td>
</tr>
<tr>
<td>AST(s)</td>
<td>Abstract Syntax Tree(s)</td>
</tr>
<tr>
<td>B, b</td>
<td>Bytes (1024 bits)</td>
</tr>
<tr>
<td>CF</td>
<td>Control Flow</td>
</tr>
<tr>
<td>CFG(s)</td>
<td>Control Flow Graph(s)</td>
</tr>
<tr>
<td>CG(s)</td>
<td>Call Graph(s)</td>
</tr>
<tr>
<td>CSV, csv</td>
<td>Comma-Separated Values</td>
</tr>
<tr>
<td>DDG(s)</td>
<td>Data Dependence Graph(s)</td>
</tr>
<tr>
<td>DEF-USE</td>
<td>Definition-Use Analysis</td>
</tr>
<tr>
<td>DF</td>
<td>Data Flow</td>
</tr>
<tr>
<td>DFA(s)</td>
<td>Data Flow Analysis(-es)</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphic User Interface</td>
</tr>
<tr>
<td>GB, Gb, gb</td>
<td>Gigabytes (1024 MB)</td>
</tr>
<tr>
<td>GC</td>
<td>Garbage Collection (Java system call)</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>ID</td>
<td>Identification or Identity (depending on the context)</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IMEI</td>
<td>International Mobile Station Equipment Identity</td>
</tr>
<tr>
<td>IMSI</td>
<td>International Mobile Subscriber Identity</td>
</tr>
<tr>
<td>IR(s)</td>
<td>Intermediate Representation(s)</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technologies</td>
</tr>
<tr>
<td>JAR</td>
<td>Java Archive (compressed file)</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>KB, Kb, kb</td>
<td>Kilobytes (1024 B)</td>
</tr>
<tr>
<td>LWP</td>
<td>Light-Weight Process</td>
</tr>
</tbody>
</table>

*Continued on next page*
## 1. Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>What (it) stands for</th>
</tr>
</thead>
<tbody>
<tr>
<td>MB, Mb, mb</td>
<td>Megabytes (1024 KB)</td>
</tr>
<tr>
<td>NP</td>
<td>Nondeterministic Polynomial time</td>
</tr>
<tr>
<td>PDG(s)</td>
<td>Program Dependence Graph(s)</td>
</tr>
<tr>
<td>PFG(s)</td>
<td>Program Flow Graph(s)</td>
</tr>
<tr>
<td>RAM</td>
<td>Random-Access Memory</td>
</tr>
<tr>
<td>SAX</td>
<td>Simple API for XML</td>
</tr>
<tr>
<td>SD</td>
<td>Secure Digital</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SIM</td>
<td>Subscriber Identity / Identification Module</td>
</tr>
<tr>
<td>SMS</td>
<td>Short Message Service</td>
</tr>
<tr>
<td>SSA(s)</td>
<td>Static Single Assignment(s)</td>
</tr>
<tr>
<td>TA</td>
<td>Taint Analysis</td>
</tr>
<tr>
<td>UID</td>
<td>User Identifier</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>WALA</td>
<td>IBM T.J. Watson Libraries for Program Analysis</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>

Table 1.1.: List of Abbreviations
2. Introduction

Nowadays, users of smartphones powered by the Android operating system interact on daily basis with mobile applications developed by third-parties. They take the security and privacy of these applications for granted without awareness of the fact that they can become a victim of a cyber-attack targeting to access or steal their personal information. The broad availability of online hacking tutorials and easy-to-use tools open the door to people who are willing to take up a new hobby in order to obtain confidential personal data exploiting the system’s vulnerabilities. Moreover, many applications may disclose the private information of their users if the developers did not set a proper configuration to the privacy settings. In this context, analysing the security properties of software using formal models plays a crucial role.

This project was inspired by the technical report “Automatic Detection of Inter-application Permission Leaks in Android Applications” by Sbîrlea et. al [1]. The present document explains my personal approach to assess the permission configuration of Android applications in order to detect vulnerabilities. I applied techniques derived from static data flow analysts to obtain a security report at the end.

2.1. Motivation

Mobile devices (mostly smartphones and tablets) have continually evolved over the last years. One breaking point was the introduction of the Apple iOS and Google Android operating systems, from that moment, the sales of personal smartphones have exploded. To have an idea of the numbers, more than 300 million devices were sold worldwide in the second quarter of 2010 alone[2]. In Germany, more smartphones exist than inhabitants in the country, which means that a significant part of the population owns more than one device[2]. One third of the Germans use their free-time to go online with their smartphones[3]. In recent years, these smartphones have turned into powerful devices that are basically a miniature version of a personal computer; they are not only capable to make phone calls and to send text/multimedia messages, but they also provide a communication and entertainment platform for users to use instant-messengers, send emails, surf the web, execute an unlimited number of apps, and play games. However, the ever increasing popularity and sophistication of mobile devices have also raised concerns about the privacy of users who operate them. These concerns have been exacerbated by the fact that it has become increasingly easy for users to install and execute third-party applications openly available in the online-stores [2]. On the market for mobile devices operating systems there is a clear leader: Android, which has become one of the most popular mobile platforms. According to recent surveys [3] between 52.5% to 70% of

all devices use this operating system. There are over 675,000 active mobile apps (which are updated regularly so new patches and versions are released) in the Google Play App Store and the growth rate of new mobile apps is over 12,000 per month; moreover, there are more than 100,000 active publishers who are publishing new mobile apps. One of the factors for such success is the opportunity for third-party providers to develop mobile applications based on the open API contained in the SDK distribution. The openness of the platform raises a security concern: malevolent developers can create their own certificates to distribute their applications. This is why most of the current security attacks against mobile devices target Android. There are more than 129,000 malicious apps created specifically to attack Android users (5,000 new malicious apps were detected in Q1 2012 alone). Such applications can be identified as malware. One of the adverse consequences of malware is the extraction of personal information. Given the broad range of applications available for mobile devices and their popularity, it is not surprising that they store an increasing amount of sensitive personal information about their users. To give some examples: the GPS receiver could reveal the exact location of the device at a specific point in time. The address book contains information about the people (friends, family, coworkers, etc.) that a user interacts with. Photos, notes, emails, the browsing history, and stored data from the apps can all contain private information. Thus, security mechanisms are required to properly protect sensitive data against malicious applications.

Sometimes the attacks are performed in an indirect manner. Security vulnerabilities in normal applications act as an outpost for malware, to trigger stronger attacks. One misconfiguration in the security setting can leave the door open for attackers.

The Android operating system uses a permission-based, access-control model to protect personal user information from attacks. One Android-specific feature is the exchange of information between components (a communication mechanism called Intent) of the same app or different apps. The previous mechanism may reduce the protection level offered by the classic permissions models. There are some attacks that target specifically the Intents for malicious purposes:

1. **Confused Deputy**: Misconfigured apps allow third apps to call interactive components by unauthorised callers, which in turn access permission-protected information or execute protected actions. Figure 2.1 shows the attack.

2. **Intent spoofing**: Is an Android-specific form of confused deputy. Apps communicate to other apps even thought they are not meant to. This attack exploits the lack of necessary security configuration, allowing the invocation of internal Activities by external apps.

3. **Permission Collusion**: An app that only has access to a restricted set of permis-
2.1. Motivation

Figure 2.1.: Internal Activity invocation or confused deputy [1].

Figure 2.2.: Permission collusion by malevolent applications [1].

sions executes a permission escalation, or augmentation attack by invoking another collaborating app through Intents. To perform this attack, malevolent developers could deceive users to install another collaborating malware app that is used to compromise privacy. The malicious cooperation process is shown in Figure 2.2.

4. Attack on sibling apps: Targets compromised apps (when the developer’s certificate has been stolen, or by exploiting other vulnerabilities, i.e. when the developers store secret authentication keys in their Android apps [6]) sharing the same user ID in order to access the permission-protected information. The attack is presented in the Figure 2.3, it has a low probability to be seen in practice due to its narrow applicability.

The three before mentioned attacks have in common the exploiting of the Intent mechanism to obtain unauthorised access to private information. They are named permission-leak attacks and can be detected by using static analysis techniques [1]. Even though the current Android permission mechanism is considered to be robust, it still presents a number of limitations. To mention some of them: there is no routine for validation at installation time, that the permissions declared and requested by an app are actually
needed at runtime (this causes that the app is over privileged with respect to the APIs it indeed uses). The user is the one who decides to grant permissions or not during the installation of an app, but the permissions themselves provide little contextual information about how the sensitive APIs are utilised by the app. For example, it is difficult for an average user to deduce that an app with the permission to access the Internet, as well as to read the phone SIM card, could expose its private contact’s list stored on the SIM card by using a service. Apps can even collude with one another to effectively gain permissions they were not explicitly granted (a threat that is more likely to happen if apps are over-privileged), by exploiting the many inter-process communication mechanisms provided by Android [7], as presented in the Figure 2.2.

Most users do not possess the technical knowledge necessary to understand what each permission means [8, 9] and naively grant them without hesitation [9]. In fact, Au et al. [10] show that users often ignore the security related information (if presented) and grant all the requested permissions blindly. For the previous reason, providing fine-grained information about the use of Android permissions is necessary for users to make well informed decisions. It may also improve the effectiveness of the install-time permission mechanism [5]. Studies have shown that the Android permission mechanism is not effective as a protection mechanism due to its lack of transparency and suggest allowing users to grant specific individual permissions [11], blocking and sanitising sensitive information [12], designing an app verification mechanism [13], and analysing apps to report over-privilege of declared permissions [14]. Previous case studies [8, 9] have reported that comprehension of permissions is limited mainly due to the “presentation” of the permissions to the users and not due to the protection mechanism itself [7]. Motivated by this situations, some efforts have been done to create secure applications or even secure systems, such as the Blackphone [7] which extend the Android operating system by implementing stronger security mechanisms.

Figure 2.3.: Application sharing user ID with a compromised application [1].

https://www.blackphone.ch, last accessed on January 5, 2015
2.2. Objective

The ultimate goal of this project is the implementation of a prototype tool capable of discovering automatically configuration and byte code permission vulnerabilities in Android applications by using static analysis techniques given an arbitrary android application (.apk file) as input.

This tool may be used by Android users which care about their privacy, developers willing to verify the security of their application before releasing it, and security professionals assessing the security of a mobile application [1].

To reach the goal set, the following tasks need to be performed:

1. Construct the application blocks that will compose the backbone of the project. We will focus on the development of the XML parser and the generation of the permission’s map.

2. Interpret the statements of the theory from the static data flow analysis in order to perform a practical implementation.

3. Overcome the lack of documentation and illustrative examples to develop our own source code.

4. To build our prototype tool on top of existing reliable work by extending their functionality with our own ideas.

2.3. Contributions

The contributions of this work can be summarized as follows:

1. The implementation of the XML SAX parser that extracts security permissions from the application’s Android manifest file. We analyse the extracted information in order to detect the existence of configurations that may lead to different levels of vulnerabilities [1], and generate the corresponding user-friendly report.

2. The explanation of the sequence of steps needed to perform the static data flow analysis of the Android source code, more specifically, backwards def-use analysis.

3. The algorithm for the generation of a permission’s map independent of the level of Android API analysed, based on the Android’s source code.

4. The extension of the WALA 1.3.5 framework by the addition of some self-implemented methods: we modified the class com.ibm.wala.viz.PDFViewUtil, adding the method generateIRpdf because the original implementation for IR .pdf generation has a compulsory call to a .pdf viewer, which interrupts the flow of our analysis.

5. The log files that show the results of the executions.
6. The combination of parameters for the JVM that increase the performance of the prototype tool.

7. The prototype tool itself, which delivers the final security assessment of the Android application.

8. There is a lack of real life examples about how to use the WALA framework. This work can be now referenced as a tutorial for similar projects. It provides a practical example of the use of static analysis to detect software vulnerabilities of Android applications.

2.4. Outline

This work is structured as follows: in Chapter 3, the theoretical foundations for Android security, and static analysis techniques and algorithms are laid, with an emphasis on the methods and structures we used. The actual implementation of the tool, which includes the explanation of all its components as well as our contributions, is presented in Chapter 4. Then, in Chapter 4.3, performance comparisons of the different CG solvers are shown. Finally, Chapter 5 discusses the results providing open questions that offer room for further research. Additionally, the Appendixes provide useful information about the hardware and software used in the project, how to obtain the .apk files for the analysis, and the list of JVM arguments used for the CG generator.
3. Fundamentals

In the first part of this chapter the Android security framework is introduced from a formal point of view. We continue by discussing some common static data flow analysis techniques and describing its data structures. Finally, we state the importance of formal models for computer security, with a focus on static code analysis, more specifically, in the topic of taint analysis.

3.1. Android security framework

The background information that is presented in this section comes mostly from the paper “A Formal Model to Analyze the Permission Authorization and Enforcement in the Android Framework” by Shin et al. [15]; moreover, many concepts and definitions are taken directly from the online documentation (API guides) of Android.

Android is the open mobile operating system based in Java developed by Open Handset Alliance (OHA). It deals with the security worries by implementing a permission-based security policy at each mobile device and limiting the action’s range of installed apps by enforcing permissions. Permissions are defined as unique character strings distinguishable from each other. When a permission is assigned to an operation or/and a resource object, the app must be granted the permission in order to execute the desired operation on the object. The Android framework provides a set of default permissions defined within the class `android.Manifest.permission`, and it also allows us to either extent the existent permissions or even define new permissions (both methods are just valid in the context of our application). The new permissions should be declared in an application via the `android Manifest`, and introduced into a system at the moment of installing the application. Authorization of the permissions is performed by the user when the application is being installed. An application requests the acceptance for a set of permissions required to complete its task prior to be installed on a device. A summarized list (not showing all the fine-grained elements) of requested permissions appears on the screen so that the user is aware of the possible actions of the app. Only if the user agrees, the application is installed, otherwise, the process is stopped. All of the requested permissions are then granted to the application. As a result of this approach, the system implements the main security principle of Android, namely, applications can perform operations that would affect other parts of the system only when they are permitted to do so.

While the above mentioned security countermeasure, which is also referenced as the Android permission scheme, has already been implemented and used in Android’s based products, the security of Android by itself has not been rigorously studied. Just a few studies focus on the formal specification of the Android permission scheme. To explain how Android works, we present the formal model of the Android permission framework proposed by Shin et al. [15]. This work describes the permission’s scheme.
specifying entities and relationships (which has also been used to represent Role-Based Access Control (RBAC) policy [10] and its variations), using this model helps us to understand the scheme clearly, and it can be used as a reference model for those who want to express and evaluate various security properties of the scheme. It can be extended by broadening the specification or restricted by adding additional constraints.

The next step is to provide a state-based model, which includes the behavior specification of permission authorization and the interactions between application components, composed of operations that one component performs on the other. It also explains how security requirements can be obtained from the definition of the system and written in logical formulae. From the specification of the Android scheme defined by Google, some sentences that mention security requirements of the permission scheme can be obtained. Although these security conditions are not explicitly stated, it is possible to identify them from natural language sentences, and translate them into logical representations (predicates) written in terms of the state-based specification elements previously defined.

For given security requirements, the specification is verifiable. That means that it is possible to mathematically confirm diverse security-related properties, and verify the security of the system. After the identification of the security requirements, the security can be verified or logically confirmed by utilizing a theorem prover. With the help of a formal tool to perform the mechanically checked proofs, it can be confirmed that the permission scheme satisfies the security requirements. For this model, Coq, the French interactive theorem prover, was used. Such tools facilitate theoretical confirmation of security properties when the system is updated, when there are new security requirements, or when more constrains are added to the model. This may help to ascertain the security level of the Android permission scheme; considering security certification standards (e.g., Common Criteria) requires formal descriptions for higher-level assurance.

### 3.1.1. Application and components

Any Android application signed and published by a developer contains sub-components of four types: activity, service, content provider, and broadcast receiver as shown in Figure 3.1. Each specific type serves a distinct purpose and has an individual lifecycle that defines how the sub-component is created and destroyed. An **activity** works actively in the foreground of the device screen interacting with the user. For example, an email
app might have one activity that presents a screen for reading emails, another activity showing a list of incoming emails and another activity to compose a new email. Although all the activities within the email app form together a cohesive user experience, each one works independently of the others. For this reason, another app is able to start any of these activities (if the email app allows it by granting the corresponding permissions). For example, a camera app can start the activity in the email app that composes a new mail, so the user can share a picture by sending an email. A service works in the background without an user interface (GUI screen) to perform work for remote processes or to perform long-running operations. For example, a service might transmit data over the wireless network without blocking user interaction with an activity or might play music in the background while the user is playing a game or surfing the internet. Another component, such as an activity, can start the service and let it run in the background or connect to it in order to exchange information. A content provider supplies data storage for our applications. You can store the data in different containers: on the web, in a SQLite database, the file system, or any other persistent storage location your app can access. Through the content provider, other apps can read or even modify the data (provided that the content provider allows it). For example, the Android system provides a content provider that manages the user’s contact information. As such, any app holding the required permissions can consult part of the content provider (such as ContactsContract.Data) to read and write information about a specific contact. Content providers are also pretty useful for reading and writing data that is private to your app and is not shared to others. For example, the Note Pad sample app uses a content provider to save notes. A broadcast receiver helps application components inter-communicate by responding to system-wide broadcast announcements. Many broadcasts are originated from the system, for example, a broadcast announcing that the screen has been turned off, the battery is getting low, or a photo was just captured. Apps can also initiate broadcasts, for example, to inform other apps that some data has been downloaded to the device and is available for them to process. Although broadcast receivers do not have a GUI screen, like an activity, they may alert the user by displaying a status bar notification when a broadcast event occurs. More commonly, though, a broadcast receiver is just a “gateway” to other application components and is intended to perform a very minimal amount of work. For instance, when an event happens, it might trigger a service to carry out some work.

A unique feature of the Android system design is that each component can be instantiated and executed separately while interacting with others, and can even be started by other apps as needed. For example, if you want the user to capture a picture with the device camera, there is probably another app that does that task and your app can just call it, instead of developing an activity to capture a photo yourself (this accelerates the development of apps). You do not need to include or even refer to the source code from the camera app. Instead, you can simply initiate the activity in the camera app that captures a picture. After finishing the external activity, the picture is even returned to your app so you can make use of it. This is totally transparent to the user: it seems as if the camera is actually an element of your app.

When an app or the system executes a component, it starts the process for the complete
app (if it is not already in execution) and instantiates the Java classes needed for the component to work. For example, if your application starts the activity in the camera app that captures a picture, that activity runs in the process that belongs to the camera app, not in your app’s process. Therefore, unlike apps on most other systems, Android apps do not have a single entry point (for example, there is no java main() function).

One app cannot directly activate a component from another app because the system executes each app in an independent process with file permissions that restricts access to other apps. However, the Android operating system can do it. In order to start a component in another app, you have to send a message to the system that notifies your intent to start a particular component. The system then starts the component for you.

Three out of the four component types (activities, services, and broadcast receivers) are started by an asynchronous message known as intent. Intents link individual components to each other at runtime by setting an inter-procedural communication channel for data interchange, whether the component is part of your app or another. An intent is defined by an Intent object, which declares a message to activate either a specific type of component or a specific targeted component (an intent can be either implicit or explicit, respectively, the difference will be explained in the next paragraphs). In the case of activities and services, an intent defines the action to perform, for example, to “send” something or to “view”, and may also specify the URI of the data to act on (among other parameters that the component being created might request), for example, an intent might transport a request for an activity to display a picture or to open a website. In some cases, an activity can be started to receive a result. If do so, the activity returns the result wrapped with an Intent, for example, an app can create an intent to let the user select a personal contact from the phone’s address book and have it returned to the app, in such case, the return intent contains an URI pointing to the selected contact. In the case of broadcast receivers, the intents are simpler: they only define the announcement being broadcast, for example, one intent send a broadcast which only includes a known action string “battery starts charging” indicating that the device battery has started charging. The last component type, content provider, instead of being activated by intents, it is started when it receives a request from a content resolver. The content resolver handles all direct transactions with the content provider acting as a middleware, so that the component that is performing transactions with the provider does not need to call it directly and instead invokes methods on the ContentResolver object. This creates a layer of abstraction between the content provider and the component requesting data for security purposes.\footnote{http://developer.android.com/guide/components/fundamentals.html, last accessed on January 5, 2015}

Intents facilitate intercommunication between components in several ways, there are different methods for starting each type of component:

- An activity can be started (or assign to it a new task) by passing an intent to \texttt{startActivity()} or \texttt{startActivityForResult()}, if it is expected a result from the activity when it finishes. The activity obtains the result as an additional intent object in the activity’s \texttt{onActivityResult()} callback. This allows the calling app to read the

\footnote{http://developer.android.com/guide/components/fundamentals.html, last accessed on January 5, 2015}
3.1. Android security framework

return code and any additional information (key/value pairs) returned by the called Activity. The intent describes the activity to start and carries with it any necessary parameters.

- A service can be started to perform a one-time operation (such as download a file) or command new instructions to an already ongoing service by passing an intent to `startService()`. If the service is designed with a client-server interface, another component can bind to it by passing an intent to `bindService()`. The intent describes the service to start or binds and carries with it any necessary parameters.

- A broadcast to other apps can be started by passing an intent to methods like `sendBroadcast()`, `sendOrderedBroadcast()`, or `sendStickyBroadcast()`.

- A query to a content provider can be executed by calling the method `query()` on a `ContentResolver` object.

There are two types of intents: explicit and implicit. **Explicit intents** specify the target component to start by fully-qualified class name. An explicit intent is typically used to start another *internal component* of an app, because the class name of the activity or service wanted to start is known by the developer. For example, create a new activity to provide feedback after an user action or initiate a service to obtain a file from the Internet in the background. The explicit intent feature is mostly used in intra-application communication, but can be useful as well for inter-application communication, and its existence is the root cause of some vulnerabilities [1]. When an explicit intent to start an activity or service is created, the Android system immediately starts the app component declared in the Intent object. **Implicit intents** have a broader scope, they do not call
3. Fundamentals

Android Manifest XML file

<manifest>
  <permission>
    <uses-permission>
      <application>
        <activity>
          <intent-filter>
            <service>
              <receiver>
                <provider>
                  android:sharedUid
                  android:permission
                </provider>
              </receiver>
            </service>
          </intent-filter>
          android:permission
        </activity>
      </application>
    </uses-permission>
  </permission>
</manifest>

Figure 3.3.: Android Manifest file (selected items)

a specific component, but instead define a general action to perform, which allows an external component from another app to handle it. For example, one app wants to show the user a location on a map, so it can use an implicit intent to start another capable app to show the specified location, based on geographic coordinates sent by the calling app. Android allows creation of implicit intents by specifying a triple (action, data type, category) and any Activity registered to receive those parameters through an intent filter will be able to receive the intent. When an implicit intent is created, the Android system looks up the appropriate component to start by comparing the contents of the intent to the intent filters declared in the Android manifest file of other apps installed on the device. If the intent fully matches a declared intent filter, the system starts that component and gives it the Intent object. If multiple intent filters are compatible, the system shows a dialog so the user can select which app to start.

Figure 3.3 illustrates how an implicit intent is sent through the system to start another activity: 1) Activity A creates an Intent with an action description and forwards it to startActivity(). 2) The Android System searches among all the installed apps for an intent filter that matches the intent. When a match is located, 3) the system starts the matching activity (Activity B) by invoking its onCreate() method and forwarding it the intent.

An intent filter is an expression declared in an app’s Android manifest file that states the type of intents that the component receives. For instance, by declaring an intent filter for an activity, you allow other apps to directly start your activity with a specific intent. In a like manner, if an activity does not have any declared intent filters, then it can only be started via an explicit intent. To ensure that an app is secure, it is necessary to use always an explicit intent when starting an app service (so is the intended one) and do not declare any intent filters for services. Sending an implicit intent to start a service is a security treat because the app cannot be sure about which service will reply to the intent, and the user cannot see which service it starts because services do not present a GUI.

3.1. Android security framework

3.1.2. Android Manifest file

The general deployment format of an Android app is an .apk file. Compiled codes of a program (byte code) and binary resources (such as icons, pictures, videos, etc.) are archived together in a Zip-compatible format, and signed with the developer certificate. The package also includes an XML file, called the Android manifest file (AndroidManifest.xml). Figure 3.3 presents its structure. In this file an app must declare all its components to inform the system that the component exists before it can be started by the Android system. This is accomplished by using XML tags named elements. Only the <manifest> and <application> elements are required, they each have to exist and must be unique. Most of the others can be declared many times or not at all (although at least some of them have to exist for the manifest to accomplish anything meaningful, that is, defining an app). An element can contain other elements, for example, the XML element <application> might contain a set of XML sub-elements for components. The elements used to declare app components are: <activity> for activities, <service> for services, <receiver> for broadcast receivers, and <provider> for content providers. Activities, services, and content providers that are included in the source code but are not declared in the manifest file, are not visible to the system, and, consequently, can never run.

All values are set through attributes. All attribute names begin with an android: prefix. Most of them are optional. However, there are some that have to be specified for an element to accomplish its purpose. For truly optional attributes, it is necessary to use the documentation as a guide because it states what happens in the absence of a specification or mentions a default value.

Some examples of attributes are the following:

- In the root element <manifest> we can find the attribute package, which represents a full Java-language-style package name for the app. The name is ought to be unique, it may contain uppercase or lowercase letters ('A' through 'Z'), numbers, and underscores ('_'). However, individual package name parts may only begin with letters. To avoid conflicts with other developers, one should use Internet domain ownership in reverse as the basis for package names. For example, apps published by Google begin with com.google. One should also never use the com.example namespace when publishing apps. The package name serves as a unique identifier for the app, it is also the default name for the app process and the default task affinity of an activity. Something important to take into consideration is: once an app has been published, the package name cannot be changed. The package name defines the app’s identity, so if it is changed, then it is considered to be a different app and users of the previous version cannot update to the new version.

- Another attribute of the <manifest> element is the android:sharedUserId or android:sharedUid. It represents the name of a Linux user ID (defined as a string of characters) that will be shared with other apps. By default, Android assigns each app its own unique user ID. However, if this attribute is set to the
exact same value for two or more apps, they will all share the same ID. Provided that they are also signed by the same certificate, meaning that they were developed by the same author. An app with the same user ID can access each other’s stored data and, if desired, run in the same system process.

- The `android:name` attribute of the `<activity>` element specifies the fully qualified class name (such as, “com.example.project.MyActivityExportDefault”) of the Activity subclass. However, as a shortcut, if the first character of the name is a period (for example,”.MyActivityExportDefault”), it is appended to the package name specified in the `<manifest>` element. There is no default. The name must be specified. Once an app has been published, the developer should not change this name (unless it have been set `android:exported="false"`).

- Another attribute of the `<activity>` element is the `android:exported`, which represents whether or not the activity can be started by components of other apps (“true” if it can be, and “false” if not). If “false”, the activity can be started only by components of the same app or other apps with the same user ID. The default value depends on whether the activity contains `intent filters` or not. The presence of at least one filter implies that the activity is intended for external use, so the default value is “true”. On the other hand, the absence of any filters means that the activity can be invoked only by specifying its exact class name. This implies that the activity is intended only for application-internal use (since other developers would not know the class name). So in this situation, the default value is “false”.

Apart from declaring the app’s components, the Android manifest pursues other objectives. Among these additional functions, there is one important in security terms: to identify any user permissions that the app requires, such as read/write-access to the user’s contacts or Internet connectivity. To succeed in doing this task, the manifest file specifies the following authorization-related information:

- List of permission declarations: An app can declare a new security permission using the `<permission>` element. Developers can define their own permissions by means of this element. The permission can then be used to limit access to specific components or features of this app or others; it is added to a system at the time of installation. Such permissions are called *declared-permissions*.

- List of permissions expected to be granted: An app lists the permissions needed to be granted in order for it to operate correctly, by using the `<uses-permission>` element. The permissions are requested at the installation time of the app. The installer determines whether or not to grant the requested permissions by checking the authorities that signed the application’s certificates and, in some cases, asking the user. If the user’s intervention is required, a summary of the permissions is shown on the screen. The user either accepts the installation or cancels it. Allowing installation of the app means granting all of the requested permissions as well. If the permissions are granted, the app is able to access the protected features. If not,
its attempts to use those features will simply fail without sending any notification to
the user. Normally a permission failure will throw a SecurityException back to the
application. Different terms are defined for the requested permissions at the time
of installation and for the permissions after being granted: requested-permissions
and use-permissions of the app, respectively.

- List of permissions used for protection: The android:permission attribute is part
  of the <application> element and the component elements. A permission is an
  imposed restriction limiting access to data on the device or to a part of the code. The
  limitation is set to protect critical code and data that could be misused to modify or
  compromise the user’s experience. Each permission is identified by a unique text label
defined by Android (listed in android.Manifest.permission) or declared by other
applications. Often this label indicates the action that is restricted (for example:
android.permission.ACCESS_NETWORK_STATE). If a permission name is as-
signed to the attribute, access to the app or the component requires the permission.
If the attribute is set by the <application> element, all of its sub-components are
protected by the permission as well. If a permission is assigned to the attribute
in the component element, access to the component requires the permission set by
the component, but the application-level permission enforcement setting is ignored.
A feature can be protected by at most one permission. The permissions used to
protect an application or a component are called enforce-permissions of the app or
the component.

- A really important subelement of the <activity> element is the <intent-filter>,
which specifies the different types of Intents that a component (activity, service,
or broadcast receiver) can communicate to. An intent filter declares the capa-
bilities of its parent component: what actions an activity or service can perform
and what types of broadcasts a receiver can handle. It allows the component to
receiving intents of the advertised type, while rejecting those that are not meaning-
ful for the component. It is pretty important to understand that the intent-filter
mechanism does not guarantee security by itself and it is meant only as a loose
binding between Activities and Intents; any Activity with an intent-filter can still
reply to an explicit Intent in which case the intent-filter is ignored. The presence
of this subelement, however, changes the behavior of the security-related exported
attribute, as detailed above. It has been found that many developers overlook the
security-related implications when using intent-filters [1].

3.1.3. Specification of the Android Permission scheme

This subsection formally specifies the Android permission framework by identifying sys-
tem components and describing their interaction. It is also shown, how permission-based
restrictions on the component relationships can be imposed in the model. The Android
permission framework is represented by an entity-relationship model (shown in Figure
3.4), which has also been used to represent Role-Based Access Control (RBAC) policy
[16] and its variations. The three major entities in this model are as follows:
3. Fundamentals

Figure 3.4.: Entities and relations in the Android permission scheme [15].

- **APPS**, the set of apps
- **COMPS**, the set of app components
- **PERMS**, the set of permissions

At the top of Figure 3.4 the structuring elements of the model are presented. The following is a more in-depth clarification of the cardinality constraints in the model:

- An app contains one or more components, and the components are integrated to a system when the app is installed. Therefore, in the COMPOSE relation, the app is `composedOf` one or more components and each component `composes` one app.

- An app declares some permissions or none. Declared permissions by an app are integrated to a system when the app is installed. Therefore, in the DECLARE relation, each permission must be `declaredBy` an app, while an app optionally `declares` multiple permissions.

- The remaining USE, AENFORCE, and CENFORCE relations are optional N-to-M relationships. A component can enforce some permissions. An app can use or enforce some permissions too. Correspondingly, a permission can be enforced by a component, or used or enforced by an app. The use and/or enforcement relations are not mandatory.

A permission may be required to execute a restricted operation on a protected resource object. Therefore, the entity permission has a relation with both objects and operations. The relationship determines what type of permission is needed and when the permission has to be checked. Figure 3.5 shows the relations between permissions, objects, and operations. Observe that apps and components correspond to entities in the diagram, since the objective is to specify the relations between them. The first model is extended by adding a new entity and relation:
3.1. Android security framework

- **OPS**, the set of operations with an **EXWITH** relationship that shows the permissions enforced on operations. It is a mapping of N operations *executedWith* a M number of permissions.

This means that when we declare a permission in an app, it includes automatically a relation with a set of objects and a relation with a set of operations. The information for the former relationship (**AENFORCE** and **CENFORCE**) can be gathered from the Android manifest, while the information for the latter relationship, **EXWITH**, can be extracted from source codes of the Android framework and apps (concrete implementation). For example, when an activity component declares its enforce-permission in the Android manifest file, it does not mean that the permission is always checked any time an action is performed on the activity. Yet, when the activity is started, the permission checking mechanisms embedded in security relevant API calls (e.g., by passing an Intent to *startActivity()* or *startActivityForResult()*) are triggered. Google’s documentation briefly describes which operations lead to the permission enforcement. The relation with operations can also be created when an app explicitly invokes check permission methods (e.g., *ContextWrapper.checkPermission(String permission, int pid, int uid]*) in its code. The relations in Figure 3.5 can be refined and regrouped in terms of components, as presented in Figure 3.6. The new optional N:M relationship, **ENFORCE** is interpreted as **AENFORCE** when a component does not enforce permissions by itself (the app enforce the permissions for it), or **CENFORCE** when it does.

The main advantage of construct a formal model such as the one just explained, is that it forms a verifiable specification. The use of the above model was demonstrated by finding a security flaw in the Android operating system [17]. The vulnerability was reported to Google and they wrote the fix.
3. Fundamentals

3.2. Possible attacks to the Android permission system

Most persons believe that the main goal of information security is stopping unauthorised entities from accessing confidential information. When it comes to personal data protection, the coined term is privacy. Even though the creation of a new user account in most websites (in our case, through an app) requires the reading of the privacy policies by the users, this hardly occurs in practical scenarios. People just introduce their personal data without examining the policies thoroughly [18]. One important aspect to consider is what information is provided and how to protect it. As our world becomes more and more accessible to others through the smartphones with internet access we are now faced with new challenges in regards to protecting our personal information. Trusting the developers of applications does not necessarily translate to more security, the lack of mandatory standards in the software development industry provides security breaches that attackers can exploit in order to steal personal data. Even though security mechanisms are used, a wrong implementation does not protect the information effectively. The data stored in some apps is a target of people who attack these systems to obtain the valuable information. It is important to know the main threats that we are exposed to in order to identify risk situations where our personal information can be compromised or disclosed [19].

The analysis performed by Xu et al. [5] among 51 malware/spyware app families and over 110,000 free apps demonstrates that malicious apps request on average 6.5 Android permissions and 64 unique permissions in total. They designed and built Permlyzer, a general-purpose framework to automatically analyse the uses of requested permissions in Android apps. Malicious apps often show distinct features in permission use. To evaluate Permlyzer capability in exposing and characterizing these features (meaning, the final intention of the attackers), they first manually determined the malicious behaviour of these rogue apps. After that, they used Permlyzer to analyse the permission use in each app and to discover the link between aggregated permission uses of a malware/spyware...
category and its malicious behaviour. They categorised the identified malicious behaviour into four categories (some malicious apps exhibit behaviour that fall into more than one category). The resulting categorisation is by no means a comprehensive one, which encapsulates all the possible threats, but it does cover a large part of the malicious behaviour observed in Android apps [20] at the time of writing the article. The categories are:

1. **Collect and Send**: Malicious apps collect permission-protected data (i.e., mobile device ID, device model, IMEI/IMSI, pictures, geographic location, personal SMS, contact’s list, etc.) from the Android device and then send the (sometimes encrypted) information to a remote server (normally sent through SMS or Internet), which stores/processes the stolen sensitive data. Almost the half (47%) of the malicious apps exhibit this behaviour.

2. **Obtain a Financial Gain**: The main goal of malicious apps developers is to monetize their attacks against mobile devices. This is achieved in three different ways (based on observed behaviour): a) apps send normal SMS messages to predetermined or fetched premium rate numbers or services to charge unwitting users; b) apps silently (in a background service) issue multiple HTTP requests to promote a specific website, so they profit from paid advertisement; and c) apps that register the victim’s phone number to a charged subscription service by sending the phone number in an undesired SMS message to the service. More than the half (57%) of the malicious apps exhibit this behaviour.

3. **Annoy Users**: A small percentage of the apps affect smartphone users by performing annoying actions without user awareness. These operations include displaying unsolicited advertisements, draining the battery, modifying the set configuration of the phone (e.g. changing the wallpaper, etc.). A small part (6%) of the malicious apps exhibit this behaviour.

4. **Monitor Smartphone Activities**: Some apps monitor the victim smartphone activities such as content in SD card storage, network activities, etc. to discover patterns of use and create profiles. The information can be obtained by apps without GUI, which send the analysis to a server. Some (11%) of the malicious apps exhibit this behaviour.

All Android apps displaying a GUI contain at least one screen (Activity), which means that attacks exploiting vulnerabilities related to Activities affect a majority of apps. The vulnerabilities identified by Sbîrlea et al. [1] have as common factor the existence of information flows that permit child Activities to communicate with authorised parents. Unfortunately, these flows are employed by malevolent apps to access permission-protected information without explicitly declaring the corresponding necessary permission.

The permission-leak attacks were already introduced in the Section 2.1. The success of these attacks relies on certain specific misconfigurations. Such misconfigurations are a combination of implicitly public Activities (callable by unexpected callers) and bad
Fundamentals

configuration of Activity permissions, which produces a failure to enforce the ownership of permissions on callers for Activities that return permission-protected data. If it is not tested that those Activities check for permissions of their callers, some trustworthy applications could be considered vulnerable (false positives), as Sbîrlea et al. [1] found in previous work.

Attacks on misconfigured applications occur when another malicious application is installed on the mobile device, so it is able to exploit its vulnerable flows. Table 3.1 lists the different parameter combinations that can lead to security vulnerabilities or information leaks. If an application sets one of the configurations listed as high risk, then any application existing on the same device can spawn it by exploiting the misconfiguration.

When Activities are designed with the feature of being invoked by unknown apps, developers have two options to ensure that the callers own a valid set of permissions: a) declaratively (in the Android manifest file, by using the permission attribute of the Activity, which can only be used to enforce a single permission) or b) dynamically (by calling the Android API function `checkCallingPermission(String permission)`). Both approaches are based on the model defined in the Section 3.1.3. The safest approach is to completely disable outside access to internal Activities that may leak protected information (as described in Sections 3.1.1 and 3.1.2), that is, avoiding using any of the parameter combinations listed in Table 3.1.

<table>
<thead>
<tr>
<th>Activity configuration</th>
<th>Application configuration</th>
<th>Consequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exported</td>
<td>Intent-filter</td>
<td>Callers accepted</td>
</tr>
<tr>
<td>&quot;true&quot; present</td>
<td>any</td>
<td>all</td>
</tr>
<tr>
<td>&quot;true&quot; absent</td>
<td>any</td>
<td>all</td>
</tr>
<tr>
<td>&quot;false&quot; present</td>
<td>set</td>
<td>from same developer &amp; UID</td>
</tr>
<tr>
<td>&quot;false&quot; absent</td>
<td>set</td>
<td>from same developer &amp; UID</td>
</tr>
<tr>
<td>default</td>
<td>present</td>
<td>any</td>
</tr>
<tr>
<td>default</td>
<td>absent</td>
<td>set</td>
</tr>
</tbody>
</table>

Table 3.1.: Different configurations lead to different levels of vulnerability [1].

As the security problems originate from the interaction of many apps installed in the same mobile device, there have been some research on the topic of their interactions. A representative work is the Scandroid project, which shows how a security model evolves as more apps are installed into an Android system, adding more entities, relations and variables to the analysis [21].

3.3. Static Code Analysis

This section is organised in a similar fashion as in the work of Cohen [22]. We took the introduced concepts and broadened their definitions with information contained in the classic text books on this field [23, 24]. We present a condensed review of the main
techniques and data structures from the area of static code analysis that are used in our work, such techniques exist for a few decades now, and since their creation, they have evolved greatly. These techniques have been employed successfully in finding general software bugs (i.e. crash, deadlock, livelock, infinite loops, etc.) \[25, 26\], and more specifically, for finding security vulnerabilities (flaws) \[27, 28\].

In a nutshell definition, static code analysis of software means automatically extracting behavioural information without executing the program, that is, prior to run time and independent on any particular input and delivery. The result of the analysis is a sound, finite, and approximate calculation of the program execution semantics.

The academia had witnessed a significant increase in research of software bug detection static techniques and dynamic software protection in the last couple of decades. The main goal of bug detection static techniques is to detect security flaws prior to the deployment of software into the production environment. To achieve that, the application’s source code is analysed statically and verified for different types of software bugs, ranging from general coding errors (i.e. infinite loops, null pointer dereference, etc.) to security vulnerabilities (such as data leakage). On the other hand, dynamic software protection is meant to protect applications at runtime. In this case, the idea behind is to adapt the application’s behaviour or environment in a way that will provide real-time protection against unexpected external threats. The application manipulation is usually carried out by code instrumentation, where guarding code is inserted prior to sensitive methods execution. There are some hybrid methods which incorporate the two techniques together, intended to overcome the inherited limitations of the use of a single protection mechanism \[22\].

In this project, we focus on the static code analysis. However, there is a problem with its automation. The informal definition of Rice theorem \[29\] states that:

> “every non-trivial semantic property of a program (in a universal programming language) is undecidable.”

Decidability means:

> “there exists an algorithm that takes as input an arbitrary program and an arbitrary property and decides (automatically) whether the property holds.”

Rice theorem implies that: precise, automated analysis is generally impossible. Program analysis therefore compute approximative answers. Approximation means: program analysis says that program \(p\):

- \(\cdots\) has property \(\phi\) but it might not have it;
- \(\cdots\) does not have property \(\phi\) but in reality, it might have.

It depends on the semantic question and on the application whether over-approximation or under-approximation is safe:

\[\text{http://santos.cis.ksu.edu/schmidt/Scuela03/home.html}, \text{last accessed on January 5, 2015}\]
3. Fundamentals

Figure 3.7.: The nature of approximation: erring on the safe side [23].

- Ex.: “does variable x have constant value?” When in doubt: no.

- Ex.: “is pointer p null?” When in doubt: yes.

The static compile-time techniques offered by the research field of program analysis are mainly used for predicting safe and computable approximations to the set of variables’s values or program’s behaviours arising dynamically at run-time when executing a software program. One of the up-to-date applications of such techniques is the software validation, implemented to reduce the likelihood of malicious or unintended behaviour especially if is a piece of code that was developed by a third-party [23].

Providing approximative answers in order to remain computable is a common theme behind program analysis approaches. In general, it is expected to obtain a larger set of true possibilities (including cases that may never actually occur during the normal execution) after performing the analysis. However, we prefer more precise or safe answers. This example of safe approximation is presented in the Figure 3.7.

Static analysis tools present one major limitation. Due to the fact that some semantic questions asked about source code are undecidable, common techniques implement algorithms that perform approximations and heuristics to answer queries such as: How does the program behaves?, How does its structure look like?, which are the properties of the entities or the relation between them? and other more formal questions about correctness, efficiency and the generation of information. In approximative approaches, one distinguishes four results:

- **True positives, true negatives**: do not represent a problem

- **False positives**: are the defects that are reported, but they are not actual defects. They set off a false alarm, which is a warning for something that is not really a bug
  - Analysis sound, but imprecise
3.3. Static Code Analysis

- In general, not avoidable (consequence of the approximation)

- **False negatives**: are defects in the code that were not reported. The consequence is a missed alarm, it is a true bug that was not caught.
  - Analysis unsound
  - Typically not acceptable (they mean a flaw in the analysis)

Most static analysis frameworks and tools (including the ones cited and used in this project) usually provide a sound analysis, meaning there should not be false negatives but for the trade-off of generating more false positives [22]. For a practical, useful analysis, the big challenge is to keep the number of false positives low. The reason for the previous statement is because analyses with high false positives are trivial to write. We know that software analysis of a program is *decidable*, so we ought to focus our efforts in its soundness. We should remember the concept of *soundness* in the context of program analysis: a logical system has the soundness property if and only if its inference rules prove only formulas that are valid (correct, in the sense that they are valid sequences preserving the truth) with respect to its (truth-table) semantics [30]. The word “sound” derives from the Germanic term “Sund” as in “Gesundheit” (meaning health). Thus to affirm that an argument is sound means, following the etymology or the concept, to state that the argument is healthy [3].

With respect to soundness, program analysis provides the following approximation results:

- **Pessimistic approximation**: refuse programs that possess the desired property (“false positive”). Analyses may return false positives.

- **Optimistic approximation**: accept programs that do not possess the desired property (“unsound”).

3.3.1. Static Analysis Limitations

The validity of results obtained using static analysis tools is a hot research topic. For example, in a review of the experimental results obtained with the tool “FindBugs” researchers found that at least 30% percent of the warnings provided by the tool were irrelevant and could be omitted. One major point stated on their paper is the discussion about the fact that categorising defect warnings into false positives and true positives oversimplifies the problem, and that inconsistent and obviously bogus coding mistakes might not have adverse impact on the functionality of the program. A more important and interesting research area is the creation of static analysis techniques that suppress or de-prioritise true defects effectively with minimal impact, and highlight defects with significant influence in the result of the program’s execution [31].

Using automated tools is an attractive approach to perform the systematic assessment of the security of a program in a less labor intensive way. However, even the best of

---

[https://www.princeton.edu/ achaney/tmve/wiki100k/docs/Soundness.html, last accessed on January 5, 2015]
these tools have limitations in the types of vulnerabilities that they are able to identify. In the paper written by Kupsch and Miller [32], they attempt to assess and quantify the effectiveness of two of the best automated source code vulnerability detection tools in the market at that time by comparing the results of the tools to the results of an in-depth manual code review of the same system.

They concluded that the static analysis tools are not powerful enough to detect all the vulnerabilities caused by design flaws or were not present in the compiled code because the tools did not have a deep enough understanding of the system, therefore there is a need for manual vulnerability assessment performed by a skilled human. Moreover these tools provided an enormous amount of alerts in the form of false positives that had “a serious impact on the effectiveness of the analysis”. Although the automated tools are not perfect, they do provide value over a human to discover certain implementation bugs or defects such as information leaks, which cause privacy concerns. In the end, a trained operator with experience is still required to ensure the correctness of the results provided by the tools, especially when the analysis should include how to fix the problem. Human expertise is crucial to make any automated tool work better.

3.3.2. Program Representations used for Analysis

An analysis is not executed directly on the source code of the analysed program, in practice it is typically performed on an suitable intermediate representation of the code. This interim representation is also known as internal representation, which is composed of data structures with a variety of shapes: they might be linear (i.e. a sequence of quadruples), hierarchical (i.e. abstract syntax trees), or general non-linear such as graphs. The latest may capture hierarchical abstractions of control flow (as in call graphs) or linear abstractions of control flow (as in control flow graphs) [24]. The following structures serve to illustrate these representations:

- **Intermediate Representation (IR)**: is the central data structure used to represent the instructions of a particular method. Compilers and standard static analysis techniques usually transform the method’s instructions of the analysed code from the source programming language (Java, Byte code, etc.) into an equivalent representation in some mid-level language (in the case of WALA, it is similar to JVM byte code). This new representation based in SSA registers, which eliminates the stack abstraction by relying instead on a set of symbolic registers, is used in order to simplify the analysis phases, because the IR is supposed to be simpler and probably more suitable for analysis. The IR organises method’s instructions in a CFG of basic blocks, as typical found in compiler textbooks. The IR in the WALA framework is immutable: that means that it does not support program transformation, and does not provide support for automatic code generation from an IR (other frameworks may provide this feature). The philosophy behind is that the IR could be used in a pass that generates analysis data, to later be used in support of some other independent tool, such as a compiler or an IDE. Typically, an analysis
3.4. Data Flow Analysis (DFA)

Data flow analysis is one of the core techniques used in source code analysis for all kind of software applications. It statically evaluates information about the data flow for each program point in the program being analysed in order to discover useful information from programs without executing them (“data” means variables and their values, with a focus on their definitions and uses along all possible paths). This computed information must be a safe approximation of the run time behaviour (focusing on the desired properties) of the analysed code during each possible execution of that program point on all possible inputs (“flow” refers to the control flow of a procedure represented through a control flow graph). The purpose of DFA is to determine at a specific point in a program the set of all the possible objects that can reach that selected point: constants, variables, instances of an object defined in a prior instruction, etc. (“analysis” in this context means computing data flow information for all program points including small facts about a program obtained by static computation and over-approximation). One can distinguish different terms associated with DFA for different levels of scope in the domain of imperative languages [24]:

- **Local DFA**: Across statements but confined to a maximal sequence of statements with no control transfer other than fall through (i.e. within a basic block) or the analysis of a single statement. In practical analyses, this form has little relevance due to its narrow scope.

- **Global (intra-procedural) DFA**: Across basic blocks but confined to a function / procedure or the analysis across statements in a function / procedure. Effectively, the basic blocks for such analyses consist of a single statement on a per method basis ignoring function calls.

- **Inter-procedural DFA**: Across functions / procedures boundaries, by taking into account method calls on the entire program code. In this type of DFA, the calling

context of a procedure influences DF information and for more precision, such an
analysis should be context sensitive by being aware of the actual method caller (the
context) and the chain of calls. In an inter-procedural non-sensitive DFA, all call
contexts are the same (default context).

Data flow analyses are specialised, they can be characterised in many types, depending
on the algorithms applied in them to compute the desired information (which influence
their precision), rules of summarisation, formulation of the analysis, etc. [24], but some
DFAs share theoretical properties and therefore form classes. We present a small classi-
fication with the families that are relevant to the project described in this document:

- **Flow sensitivity**: DFA is in most of its implementations flow sensitive in the sense
  that it computes point-specific information. This means that the analysis considers
  the control flow of the program. Flow-sensitive pointer analysis is polynomial or
  NP-hard [34]. In some cases, like alias analysis, flow insensitive analyses are also
  commonly used, in this situation the CFG solution (either per program point or
  for the whole program) is ignored.

- **Context sensitivity**: inter-procedural DFA can be either context sensitive or con-
text insensitive. The difference was explained some paragraphs above. In general
terms, carrying on fully context sensitive analysis is very inefficient because it con-
sumes a lot of computational resources such as RAM, causing performance issues.
Therefore most practical algorithms employ a limited amount of context sensitivity
(usually constraining the analysis of call strings or containers). Context insensitive
DFA is also very common and it is used mainly when the precision of the result is
not as important as the speed of the analysis.

- **Propagation**: depending on the type of analysis, the transversal direction (how
  the program paths are examined) can be forward, backward or even bidirectional.

### 3.4.1. Program Representations used for Data Flow Analysis

The best-known representation class of DFAs are sets of program entities such as variables
or expressions satisfying the given property. Bit vectors were the first structures to
represent these sets. There are also data flow analyses that are no bit-vector based such
as access paths, which require summarisation. The most usual representations for intra-
procedural DFA are PFGs, CFGs, PDGs and SSAs whereas inter-procedural DFA use a
combination of CGs (normally PFGs or CFGs). Even though ASTs can and have been
employed for DFA, they are not commonly used since they do not exhibit CF explicitly
[24]. The following is a selection of structures used in our project:

- **Abstract Syntax Tree (AST)**: is a tree representation that holds the abstract
  syntactical structure of a program’s source code, the AST is generated by parsing
  the source code written in a programming language. The syntax is “abstract” be-
  cause it does not represents every specific detail appearing in the original syntax
3.4. Data Flow Analysis (DFA)

(i.e. grouping parentheses are implicit in the tree structure). This is the basic data structure used for all software analysis techniques, as well as compilation techniques. Programming languages are usually described as context-free grammars, which represent the language’s statement structure. These grammars can be represented as a tree, where each node represents an expression or a construct occurring in the analysed source code [35].

• **Control Flow Graph (CFG):** An intra-procedural CFG is a directed graph that represents the transfer of control in a given program where nodes represent regions of source code and edges represent the possible execution paths a program can follow. “Intra-procedural” means that the CFG describes a single method or procedure; the term is usually omitted. This structure is pretty useful for program traversal, understanding program dependencies between statements, etc.. Normally, a CFG is relatively straightforward to extract from the source code, even when only the binary code is available. For each standard flow construct of a language, there exists a well-defined CF (sub-)graph:

  - **IF statement** (two-way branch): a node for the test clause and two children (blocks of code), representing the THEN statement and the ELSE statement.
  - **CASE statement** (n-way branch): a node for the test clause and n children (blocks of code), representing one branch each
  - **WHILE statement** (repetition if a condition is fulfilled): a node for the loop header (test clause), the CFG for the body of the loop cycle, and two exiting edges (one back to the loop header, one to outside the loop)
  - **Sequence of instructions**: single edges

Normally, the nodes of a CFG are basic blocks of code. A basic block is the longest sequence of instructions that is always executed together before a flow construct appears (it must be the very last). If the test clause of an IF-statement is formed by a compound expression, then it is broken across two separated blocks. It has one unique entry point (the first instruction) and one unique exit point (the last instruction of the block). For practical reasons, depending on the implementation, some CFGs introduce two virtual nodes: entry and exit (and link them correspondingly to the first and last basic block).

• **Call Graph (CG):**

In this document we talk about a static call graph, which is a directed graph that represents all calling relations (caller-callee) between different methods in the analysed program. Each subroutine or method in the source code is depicted as a node, and an edge denotes a call from method A to method B that is labeled by the call sites [24]. CGs do not only contain caller-callee relationships, but might also incorporate a chain of method calls leading to a specific method. This is called the context. For a precise CG construction is necessary to perform a “CG analysis” as follow: for each method, a new node is created for each possible call stack (chain
of method calls) that ends in the analysed method. The computation of this CG is an undecidable problem, and therefore existing algorithms use approximations as stated previously. CG construction is much more complicated than CFG construction for several reasons, just to mention some examples: polymorphism (dynamic dispatch), overloading of methods, recursive functions (would cause cycles in the CG).

- **Definition-Use Graph:**
  A classical property computed in static analysis is reaching definitions. The aim of this analysis is to determine, at a given program point, which value assignments might have been occurred and not overwritten by another instruction, when program execution goes through this analysed point along some path in the data flow. Each assignment or calculation instruction creates a new value (called “live” definition). This definition is later used in future calculations. The result of DEF-USE yields the data structure Definition-Use graph. The main application of this intermediate representation is to locate, for the desired live definition, its future uses (forward traversal analysis), or at a given use in an specific program point, which definitions may reach that point (backward traversal analysis). This graph is normally built per method, similarly as the CFG, and is very useful for data flow analysis.

  DEF-USE pairs capture direct data dependencies. Those dependences can be represented in a data dependence graph. A DEF-USE pair connects a definition of a variable with all the program points where that definition is used. A DDG is a directed graph where the nodes are basic blocks of code (the same as in the CFG) and edges represent DEF-USE pairs.

3.4.2. **Pointer and Alias Analysis**

There are a couple more of analyses that were used in this project, although they have no much relevance, we still need to apply them to have sound results. Pointer analysis (also know as Points-to analysis) comes to answer the question to which object or area in the memory (set of locations) a pointer or a program variable points to. This analysis is important because the application’s memory content changes depending on the previously run instructions (or the context). The precision of points-to analysis is directly influenced by the precision of the CG, for example, to get the most precise result, the CG has to be context sensitive and all the performance optimisations must be removed. We distinguish two types of pointer analyses:

- **May analysis**: if it determines the locations a pointer variable may point to.

- **Must analysis**: if it determines the locations a pointer variable must point to.

Alias analysis is a closely related analysis: for pairs of variables it determines which locations they may or must share. This analysis answers the question raised when it is needed to know if two pointers point to the same area in memory. Similar to the
other analyses, it is also affected by different configurations and precision levels (if it considers flow-sensitivity, inter-procedural DFA, context-sensitivity, etc.). These two last described analyses are among the most advanced, and also most complicated. Most existing algorithms use some sort of approximation [34].

3.4.3. Data Flow Analysis for Java and Android

The paper “Sound and Precise Malware Analysis for Android via Pushdown Reachability and Entry-Point Saturation” by Liang et al. [4] presents a condensed review of the research on the field of DFA for Java and Android in recent years. We summarise their work as a wrap up for this Chapter:

- **Static analysis for Java**
  
  Scalable and Precise, context-sensitive points-to analysis for Java programs, which implement the object-oriented paradigm, has been an open problem for decades. The major part of the existing research focused their efforts on finite-state abstractions for Java such as the k-CFA algorithm [47], limited object sensitivity, and their variants. The IBM WALA framework is a static analysis set of open source libraries designed to support different configurations of points-to analysis. By using the libraries, one can compute which exceptions a specific method can throw, but the precise match between exceptions and their corresponding handlers is not guaranteed [4].

  In works on the area of exception flows [36–39], the analysis is usually based on context-insensitivity or limited context-sensitivity. This has as consequence that the contexts where an exception is thrown cannot be differentiated or that the handlers handling an exception cannot be precisely determined. Paddle [41] and Spark [40] both implement imprecise exception analysis. Soot [42] relies on an independent exception analysis developed by Fu et al. [43], which is not based on points-to analysis and not integrated into the tool. Bravenboer and Smaragdakis suggest to combine exception flow analysis [44] and pointer analysis in order to improve precision and decrease the analysis run time in their DOOP framework [45]. The latter authors have conducted extensive comparison of different options for poly-variance. Their approach provides a more precise and efficient exception-flow analysis than the mentioned Spark, Paddle, and Soot, with respect of points-to and exception-catch links under certain metrics [45, 46].

- **Static analysis for Android**
  
  Analysing apps written for this operating system is challenging. As stated in the Section 3.1, Android is based on Java, so it presents most of the issues mentioned in the prior paragraphs. The general problem inherited from Java of analysing object-oriented applications, where the state of the art is finite-state based, remains to be solved. More specifically, long-established analysis approaches like k-CFA [47] and its many derived versions explicitly or implicitly finitize the stack during abstraction. Existing analysers divide exception-handling points and dynamic return

---

[34] Reference to be added.
[35] Reference to be added.
[36] Reference to be added.
[37] Reference to be added.
[38] Reference to be added.
[39] Reference to be added.
[40] Reference to be added.
[41] Reference to be added.
[42] Reference to be added.
[43] Reference to be added.
[44] Reference to be added.
[45] Reference to be added.
[46] Reference to be added.
3. Fundamentals

points among a finite number of abstract return contexts. For this reason, the analyser becomes unable to distinguish the context when two dynamic return points are mapped to the same abstract context. Moreover, a new domain-specific challenge is added by programming following the Android paradigm: asynchronous interleaved execution of event-driven actions with multiple entry points.

In the Sections 2.1 and 3.2 we introduced the concept of malware tailored to Android and its negative effects. Liang et al. [4] affirm that the static tools used to detect malware normally rely upon an existing analytic framework, for example, Woodpecker depends on Soot [49], CHEX [48] depends on WALA. Other ad hoc techniques suggested to cope with the problem combine dynamic execution to eliminate paths not appearing at run time [50] or implement heuristics to abort the unsound paths [48]. One issue with this strategy is that the inherent imprecision of the underlying analytic framework is translated as an imprecise malware analysis. Another issue is that many static malware detection tools make a trade-off for the sake of efficiency: they go unsound so they can handle the large number of permutations generated while analysing multiple entry points. Often this has as unintended consequence that the static analyser is no longer able to prove the absence of malicious behaviours.
4. Development

4.1. Introduction

For the purpose of analysing the permission configurations of Android apps, more specifically, in order to detect their security vulnerabilities, we implemented a prototype tool. The inputs to our tool are:

1. *An Android app (.apk file):* developed in Java for the Android operating system. We evaluate Java byte code, and therefore we do not require the source code, we use a decompiler to extract the relevant input data for our tool, and

2. *The Android operating system’s source code (.jar file):* must be the same API level as the analysed Android app.

This chapter presents the contribution of this Master’s project work, meaning, here is presented the actual implementation of the prototype tool, which includes the explanation of all its components. Our analysis is based on the concepts and techniques from the theory introduced in the Chapter 3 and our own heuristics, which work together to overcome the problems we encountered during the implementation. We will provide a review of the different stages of the development step by step, explaining the relevant terms along the way, with emphasis on our additions. By implementing the tool, we aim to reach the goal set in the Section 2.2.

4.2. Prototype tool implementation

The Figure 4.1 shows the general block diagram of the tool. The further subsections describe the components that constitute the tool in more detail.

The Appendix A provides the list of hardware and software that we employed. We will refer frequently to those external tools.

![Figure 4.1: The components of our prototype tool](image-url)
4. Development

4.2.1. Preparing the analysis

First of all, we need an Android app to analyse. In the Appendix it is explained how to obtain its corresponding .apk file.

Secondly, it is indispensable to have the Android operating system’s source code (.jar file) to perform the static data flow analysis. The first approach was to work with the android.jar archive packed with the Android SDK. However, this strategy failed, because this is just an stub JAR file, used chiefly for type-checking. This reduced version has its advantages: it is quicker to load since it is much smaller, which also means that it requires less RAM. Unfortunately, for the static data flow analysis, this means that our call chains will end abruptly before reaching the expected definitions.

We asked academic researchers, authors of papers regarding Android static DFA, how could we obtain the complete version of the Android source code. In the end, we got three different alternatives, all of them valid:

1. Sven Arzt from the Software Engineering Research Group at the Darmstad University:

“The Android platform JAR files available in the Sable group on Github are built from real phones or emulators and thus include complete implementations instead of the stub JARs shipped with the official Android SDK. However, you usually only need those complete JAR files if you want to analyse the Android framework itself. If you are only working with normal Android apps, you can use the normal JAR files included in the SDK. These stub JAR files are also quicker to load since they are much smaller which also means that they require less memory in Soot. I would thus recommend using the stub JAR files from the SDK unless you have a use case that really requires you to have full implementations.

When I built my last full JAR file (Android 4.4), I copied the odex files from my phone to my computer, converted them from odex to dex (there are tools on the Internet) and then decompiled those dex files to Java class files and then packaged those as a JAR. Be aware that there is some loss involved, i.e., you will not be able to recover each and every class. This is more of a best-effort approach.”

2. Dragos Sbirlea from RICE university:

“Please build the Android OS and extract all jars that get created with the Android libraries from the SDK. This is why your call chains end (probably). You can unzip the jar and examine which class files are inside, compared to what you would expect from analysing the source files in Eclipse and seeing the actual call chains.

Try using the ones in the Android source code. Build Android and use the compiled as jars/class files.”
4.2. Prototype tool implementation

3. Kathy Au, Software Engineer at Google:

"You will need to get the Android source code and follow the instructions to build it on your machine in order to get the jar files for PScout to analyse.

For the last solution, some high requirements exist: 16 GB RAM, 100 GB free disk space (source code classes weight about 8.5 GB), 64 bit environment for APIs above 2.3 and several compiling tools

We chose to use the Android.jar file from the Github site of the Sable group because the Android containing the full implementation of the API level 10 (Android version 2.3) was available. This version is the same used by Sbirlea et al. in [1].

4.2.2. Processing the .apk file

In the Figure 4.2 is illustrated how we process the .apk file of the analysed Android apps. There are two subprocesses: decompile and extract resources.

- **Decompile**

  To evaluate the Android app, we need to obtain its original source code. In order to do so, we convert the Java byte code packed in the .apk file (.class files) into readable Java classes (.java files) and generate a .jar file with them.
We use the *dex2jar* decompiler to decompile the byte code. To run the tool, we need to execute its binary in a command-line console following this call template:

```
sh d2j-dex2jar_PATH android-apk_PATH
```

For example:

```
sh ../dex2jar-0.0.9.15/d2j-dex2jar.sh
../android apks/com.dropbox.android.apk
```

After the execution, the generated output is the decompiled application (.jar file). The result from the previous command is the *com.dropbox.android-dex2jar.jar* file.

- **Extract resources**

  We presented the Android permissions in the Section 3.1.2 and their formal specification was explained in the Section 3.1.3. Later, the Section 3.2 describes how malware apps try to exploit vulnerabilities by using attacks such as the ones described in Section 2.1. The Table 3.1 presented Android app configurations that are particularly prone to leak private information.

  In order to detect vulnerabilities in the permission configurations, we have to examine the declared permissions stored in the Android Manifest.xml file of the analysed app. The right branch of the Figure 4.2 summarises our approach to obtain the information that we need.

  To start our analysis, we extract the Android Manifest.xml file from the .apk package by using a XML resource extractor (apktool). The original manifest file contained in the .apk app has a *binary format*, the apktool converts the binary Android Manifest.xml into an actual readable XML file.

  To run the XML extractor, we need to execute its binary in a command-line console following this call template:

  ```
apktool_PATH d android-apk_PATH
```

  The *d* argument means that we want to decompress the XML file. An example of execution is:

  ```
/apktool d ./../android apps/com.dropbox.android.apk
```

  The output result from the previous command is the textual *AndroidManifest.xml* file. We can take a quick look at the declared permissions stored in it by executing the following command:

  ```
cat com.dropbox.android/AndroidManifest.xml | grep permission
```
The next step in our process is to locate in a formal approach the declared permissions. In the work of Xu et al. [5], it is proposed a scheme to automatically explore the use of permissions associated to the functionality of an Android app. The identification of declared permissions is achieved by parsing the AndroidManifest.xml file and interpreting the meta-information stored in it. Sbirlea et al. [1] follow a similar process.

The source code of the parsing process for the AndroidManifest.xml file was not openly available, so we implemented our own parser. There exist different XML parsing techniques, we decide to implement a Stream Based Parser (event based), more specifically, a Push Parser. This type of parser reads sequentially through the document, when the parser encounters the elements that we want to extract, it notifies the application through callback methods (listener objects) so we can use the values found. SAX XML parser is one such example of push parser, we chose this parser because, in our opinion, it was the most appropriate to implement our analysis.

We gathered information to write our SAX parser from several online sources: [3] [4]

Algorithm 1: Android Manifest Parser Algorithm

<table>
<thead>
<tr>
<th>Input: XML Manifest (AndroidManifest.xml)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Declared Permission’s report (file with the format .csv)</td>
</tr>
<tr>
<td>read XMLFile;</td>
</tr>
<tr>
<td>for each relevantElement in XMLFile do</td>
</tr>
<tr>
<td>extract desiredAttribute</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>for desiredAttributeFound do</td>
</tr>
<tr>
<td>compare desiredAttributeFound to vulnerableConfigurations</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>write CSVFile;</td>
</tr>
</tbody>
</table>

and we took elements that produce our own version, which is composed by the Java classes SAXXMLParser, SAXXMLHandler, AndroidManifestXMLParser, ContentsTest and CSVFile. The Figure 4.3 shows our parser diagram.

The elements and their attributes (refer to the Section 3.1.2 for more information) extracted from the Android manifest are the following:

- `<manifest>`: android:sharedUid
- `<activity>`: android:name, android:exported, <intent-filter>

The Algorithm 1 illustrates our implementation. The actual values from the extracted attributes are compared to the vulnerable configurations from the Table 3.1. The output of the parsing is a .csv file containing the declared permission’s report, including the result of the assessment of vulnerabilities. The name of the file is generated following the pattern:

“OutputAnalysis_” + manifestFileName + “_” + myDateFormat.format(currentDate).toString() + “.csv”

One example of such file is:

OutputAnalysis_MisconfiguredAppAndroidManifest.xml_20141103_0544.csv

In order to be on the safe side, ensuring that our SAX XML parser works as intended, we created two test configuration files: MisconfiguredAppAndroidManifest.xml (Listing 4.1) and MisconfiguredAppSharedUidAndroidManifest.xml (Listing 4.2). The output of the processing of them are the Tables 4.1 and 4.2 which demonstrates that our implementation is correct, producing sound analyses.

---

4.2. Prototype tool implementation

Listing 4.1: MisconfiguredAppAndroidManifest.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.android.app.
  MisconfiguredAppAndroidManifestApplication">
  <uses-permission android:name="android.permission.VIBRATE" />
  <application android:name="MisconfiguredAppAndroidManifestApplication">
    <activity android:name="MyActivityExportDefault">
      <intent-filter>
        <action android:name="com.zxing.SCAN" />
        <category android:name="category.DEFAULT" />
      </intent-filter>
    </activity>
    <activity android:name="MyActivityExportDefaultWithoutIntentFilter" />
    <activity android:name="MyActivityExportTrue" android:exported="true">
      <intent-filter>
        <action android:name="com.zxing.SCAN" />
        <category android:name="category.DEFAULT" />
      </intent-filter>
    </activity>
    <activity android:name="MyActivityExportTrueWithoutIntentFilter" android:exported="true" />
    <activity android:name="MyActivityExportFalse" android:exported="false">
      <intent-filter>
        <action android:name="com.zxing.SCAN" />
        <category android:name="category.DEFAULT" />
      </intent-filter>
    </activity>
    <activity android:name="MyActivityExportFalseWithoutIntentFilter" android:exported="false" />
  </application>
</manifest>
```

Listing 4.2: MisconfiguredAppSharedUidAndroidManifest.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.android.app.
  MisconfiguredAppSharedUidAndroidManifestApplication"
  android:sharedUid="uidIdentifier">
  <uses-permission android:name="android.permission.VIBRATE" />
  <application android:name="" MisconfiguredAppSharedUidAndroidManifestApplication">
    <activity android:name="MyActivityExportDefault">
      <intent-filter>
        <action android:name="com.zxing.SCAN" />
        <category android:name="category.DEFAULT" />
      </intent-filter>
    </activity>
    <activity android:name="MyActivityExportDefaultWithoutIntentFilter" />
    <activity android:name="MyActivityExportTrue" android:exported="" />
    <activity android:name="MyActivityExportFalse" android:exported="" />
  </application>
</manifest>
```
true">
  <intent-filter>
    <action android:name="com.zxing.SCAN" />
    <category android:name="category.DEFAULT" />
  </intent-filter>
</activity>
<activity android:name="MyActivityExportFalseWithoutIntentFilter" android:exported="false" />
</application>
</manifest>
### 4.2. Prototype tool implementation

#### Activity configuration

<table>
<thead>
<tr>
<th>Name</th>
<th>Exported</th>
<th>Intent-filter</th>
<th>SharedUid</th>
<th>Callers accepted</th>
<th>Risk level</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyActivityExportDefault</td>
<td>null</td>
<td>present</td>
<td>any</td>
<td>all</td>
<td>HIGH</td>
</tr>
<tr>
<td>MyActivityExportDefaultWithoutIntentFilter</td>
<td>null</td>
<td>absent</td>
<td>any</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>MyActivityExportTrue</td>
<td>“true”</td>
<td>present</td>
<td>any</td>
<td>all</td>
<td>HIGH</td>
</tr>
<tr>
<td>MyActivityExportTrueWithoutIntentFilter</td>
<td>“true”</td>
<td>absent</td>
<td>any</td>
<td>all</td>
<td>HIGH</td>
</tr>
<tr>
<td>MyActivityExportFalse</td>
<td>“false”</td>
<td>present</td>
<td>any</td>
<td>null</td>
<td>HIGH</td>
</tr>
<tr>
<td>MyActivityExportFalseWithoutIntentFilter</td>
<td>“false”</td>
<td>absent</td>
<td>any</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

Table 4.1.: Output of the MisconfiguredAppAndroidManifest.xml

---

<table>
<thead>
<tr>
<th>Name</th>
<th>Exported</th>
<th>Intent-filter</th>
<th>SharedUid</th>
<th>Callers accepted</th>
<th>Risk level</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyActivityExportDefault</td>
<td>null</td>
<td>present</td>
<td>set</td>
<td>all</td>
<td>LOW</td>
</tr>
<tr>
<td>MyActivityExportDefaultWithoutIntentFilter</td>
<td>null</td>
<td>absent</td>
<td>set</td>
<td>from same developer &amp; UID</td>
<td>LOW</td>
</tr>
<tr>
<td>MyActivityExportTrue</td>
<td>“true”</td>
<td>present</td>
<td>set</td>
<td>all</td>
<td>null</td>
</tr>
<tr>
<td>MyActivityExportTrueWithoutIntentFilter</td>
<td>“true”</td>
<td>absent</td>
<td>set</td>
<td>from same developer &amp; UID</td>
<td>null</td>
</tr>
<tr>
<td>MyActivityExportFalse</td>
<td>“false”</td>
<td>present</td>
<td>set</td>
<td>all</td>
<td>LOW</td>
</tr>
<tr>
<td>MyActivityExportFalseWithoutIntentFilter</td>
<td>“false”</td>
<td>absent</td>
<td>set</td>
<td>from same developer &amp; UID</td>
<td>LOW</td>
</tr>
</tbody>
</table>

Table 4.2.: Output of the MisconfiguredAppSharedUidAndroidManifest.xml
4.2.3. Static data flow analysis

The entire static data flow analysis is implemented on the WALA framework, which is provided as an open-source project by the IBM T.J. Watson Research Centre. WALA contains a set of static analysis libraries written in Java and is used to analyse different Java / JavaScript programs. It provides the data structures and implements the algorithms introduced in the Section 3.3.

<table>
<thead>
<tr>
<th>Theoretical structure</th>
<th>WALA object</th>
<th>Object type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method signature</td>
<td>Method Reference</td>
<td>com.ibm.wala.types.MethodReference</td>
</tr>
<tr>
<td>Call graph</td>
<td>Call graph</td>
<td>com.ibm.wala.ipa.callgraph</td>
</tr>
<tr>
<td>Control flow graph</td>
<td>Control flow graph</td>
<td>com.ibm.wala.cfg</td>
</tr>
<tr>
<td>Intermediate representation</td>
<td>IR</td>
<td>com.ibm.wala.ssa.IR</td>
</tr>
<tr>
<td>Pointer analysis</td>
<td>PointerAnalysis</td>
<td>com.ibm.wala.ipa.callgraph.propagation.PointerAnalysis</td>
</tr>
<tr>
<td>Static Single Assignment</td>
<td>SSA</td>
<td>com.ibm.wala.ssa</td>
</tr>
</tbody>
</table>

Table 4.3.: WALA objects

The Table 4.3 shows the relation between the theoretical data structures and the actual implementation existing in WALA.

We extended the WALA framework by modifying the class `com.ibm.wala.viz.PDFViewUtil`, adding the method shown in the listing 4.3 because the original implementation has a compulsory call to the .pdf viewer, which interrupts the flow of our analysis. For an explanation about how to replace the original java class, refer to the Appendix D.

```java
/**
 * Generate the PDF for the IR
 *
 * @throws IllegalArgumentException if ir is null
 */

public static void generateIRpdf(IClassHierarchy cha, IR ir, String pdfFile, String dotFile, String dotExe) throws WalaException {
    if (ir == null) {
        throw new IllegalArgumentException("ir is null");
    }
    Graph<? extends ISSABasicBlock> g = ir.getControlFlowGraph();
    g = CFGSanitizer.sanitize(ir, cha);
    DotUtil.dotify(g, null, dotFile, pdfFile, dotExe);
    return;
}
```

Listing 4.3: IR .pdf file generation
4.2. Prototype tool implementation

The Listing 4.4 is a selection of the commands used in order to generate the CG for our DFA. The Figure 4.4 depicts our process to generate the CG.

Listing 4.4: Call graph generation (selected illustrative instructions)

```java
1 AnalysisScope represents code to be analyzed. Build an AnalysisScope
   which represents the set of classes to analyze. In particular, we
   will analyze the contents of the appJar jar file and the Java
   standard libraries.
2 AnalysisScope scope = AnalysisScopeReader.
   makeJavaBinaryAnalysisScope(INPUT_JAR_FILE,exclusionsFile);
3
4 Invoke WALA to build a class hierarchy for name resolution, etc. Build a
   class hierarchy representing all classes to analyze. This step will
   read the class files and organize them into a tree.
5 IClassHierarchy cha = ClassHierarchy.make(scope);
6
7 Create a name representing the method whose IR we will visualize
8 MethodReference mr = StringStuff.makeMethodReference(methodSig)
   ;
9
10 Resolve the method name into the IMethod, the canonical representation
    of the method information.
11```
4. Development

```java
IMethod m = cha.resolveMethod(mr);
if (m == null) {
    Assertions.UNREACHABLE("could not resolve " + mr);
}

// what are the call graph entrypoints? **** all application methods entry points NOT JUST MAIN methods
AllApplicationEntrypoints aep = new AllApplicationEntrypoints(
    scope, cha);
entrypoints = aep;

// Build the IR and cache it. This is from the TH, not the call graph!
// ******
IR ir = cache.getSSACache().findOrCreateIR(m, Everywhere.
    EVERYWHERE, options.getSSAOptions());
if (ir == null) {
    Assertions.UNREACHABLE("Null IR for " + m);
    log.error("Null IR for " + m);
} else {
    log.debug(ir.toString());
}

// builds call graph via pointer analysis different builders
CallGraphBuilder builderRTA = Util.makeRTABuilder(options,
    cache, cha, scope);
log.info("building call graph RTA...");
start = System.currentTimeMillis();
// cgRTA = builderRTA.makeCallGraph(options, null);
end = System.currentTimeMillis();
// log.info("took " + (end-start)/1000/60 + "min");
// log.info("stats for call graph RTA...");
// log.info(CallGraphStats.getStats(cgRTA));

// *** Method created by me
PDFViewUtil.ghostviewIR(cha, ir, mr.getName() + "_" +
    PDF_FILE_IR, DOT_FILE_IR, dotExe);
log.info("pdf IR done");

// Create the PDF file for the CG
log.info("total number of cg nodes: " + cg.getNumberOfNodes()
    ());
gCF = pruneForAppLoader(cg);
log.info("gCF number of application nodes: " + gCF.
    getNumberOfNodes());

log.info("building Pointer Analysis...");
// PointerAnalysis pa = builderRTA.getPointerAnalysis();
// PointerAnalysis pa = builder1NCFA.getPointerAnalysis();
log.info("done");

log.info("total number of cg nodes: " + cg.getNumberOfNodes()
    ());
log.info("checking graph integrity");
```
4.2. Prototype tool implementation

**Algorithm 2**: Permission map generation

**Data**: Methods signatures

**Result**: Permissions map

begin

predecessors;

**foreach** methodSignature in MethodsSignatures **do**

predecessors = getPredecessorNodesOf(methodSignature);

end

end

execute DefUseAnalysis;

generate permissionMap write resultsFile;

---

65  GraphIntegrity.check(cg);
66  log.info("integrity check done");

The Figures 4.5 and 4.6 are taken from the generated IR .pdf files.

We implemented a performance tuning for the JVM in order to decrease the generation time of the CG. The parameters used are explained in the Appendix C.

4.2.4. Permission map generation

Once the CG has been generated, we can perform our data flow analysis. We look for specific API method calls based on their method signature.

We describe our targets points as a three tuple \(<c, m, p>\), where \(c\) is an Android class, \(m\) is a method inside that class and \(p\) is the permission.

The permissions changing from version to version of the Android API level implementation. Therefore is important to develop a generic algorithm applicable to all the versions. Such algorithm is the one presented in Algorithm 2.
Figure 4.5.: IR_Stubs.
Figure 4.6.: IR_NoStubs.
4.3. Experiments

We construct the CG with different solvers. The Table 4.4 presents a summary of the features of each of them.

<table>
<thead>
<tr>
<th>Theoretical name</th>
<th>WALA object</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTA</td>
<td>Util.makeRTABuilder</td>
<td>RTA operates on bytecodes, some of which may be proved dead during SSA construction and hence not represented in the SSA IR. This can lead to undesirable behaviors like call sites returned by the RTA call graph that do not appear in the SSA IR.</td>
</tr>
<tr>
<td>0CFA</td>
<td>Util.makeZeroCFABuilder</td>
<td>context-insensitive, class-based heap</td>
</tr>
<tr>
<td>01CFA</td>
<td>Util.makeZeroOneCFABuilder</td>
<td>context-insensitive, allocation-site-based heap</td>
</tr>
<tr>
<td>0-1-CFA</td>
<td>Util.makeZeroOneContainerCFABuilder</td>
<td>0-1-CFA with object-sensitive containers</td>
</tr>
<tr>
<td>k-CFA</td>
<td>Util.makeNCFABuilder</td>
<td>call-string context sensitivity, with call-string length limited and a context-sensitive allocation-site-based heap abstraction.</td>
</tr>
<tr>
<td>1NCFA</td>
<td>Util.makeVanillaNCFABuilder</td>
<td>call-string context sensitivity, with call-string length limited and a context-sensitive allocation-site-based heap abstraction. Standard optimizations in the heap abstraction like smushing of strings are disabled.</td>
</tr>
</tbody>
</table>

Table 4.4.: Call graph solvers

4.4. Performance Analysis

We compared the execution time for the generation of the CG with different solvers. The Table 4.5 presents a summary of our analysis results.

<table>
<thead>
<tr>
<th>Solver</th>
<th>Normal JVM</th>
<th>JVM with tuning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTA</td>
<td>1568</td>
<td>1462</td>
</tr>
<tr>
<td>0CFA</td>
<td>1568</td>
<td>1462</td>
</tr>
<tr>
<td>01CFA</td>
<td>1568</td>
<td>1462</td>
</tr>
<tr>
<td>0-1-CFA</td>
<td>1568</td>
<td>1462</td>
</tr>
<tr>
<td>k-CFA</td>
<td>1568</td>
<td>1462</td>
</tr>
<tr>
<td>1NCFA</td>
<td>1568</td>
<td>1462</td>
</tr>
</tbody>
</table>

Table 4.5.: Performance comparison
5. Final discussion

This final chapter concludes the project work. Here, we review the project objective and summarise the main results. During the project, we left out some interesting issues due to time constraints. These issues are stated in the future work section.

5.1. Conclusion

We manage to fully implement the XML parser to analyse the declared permissions of Android apps. The output of our process is presented as a .cvs file that includes the grade of severity of the configuration found, remarking if there is a vulnerable configuration. The generation of the permission map was partially fulfilled.

The interpretation of the abstract theoretical statements from the static data flow analysis was successful. However, a lot of background knowledge is required in order to perform a practical implementation. The programming of the analysis may look easy but it is not. One have to own a deep understanding of what want to be achieved with the execution of each instruction.

Although the lack of documentation and illustrative examples to develop our own source code was always an annoying issue, we manage to implement our own versions of analysis from scratch. This project can be used now as a reference for other students.

Our prototype tool does represent our own ideas. We learnt how to use the existing framework and we even extended their original functionality with our own methods.

As an addition, we finishing the project with the following lessons learnt:

– It takes a long time to find solutions alone. We realised that most of the work is done in research groups, with continual collaboration among the members. The amount of knowledge is difficult to handle for a single person.

– The valuable experience were gained in trial-and-error approach because of the lack of open available documentation on the subject. A lot of effort was put into come up with our own implementations.

– The collaboration with the authors of papers is possible if the right questions are asked nicely. In the beginning it was difficult to get our queries answered, especially when the question had a wide scope. When we manage to formulate the questions in an almost yes / no format, we got prompt answers.

– If the work collaboration flows smooth, write a paper together becomes a reality, especially when you propose new interesting ideas to the authors of the papers.
5.2. Future Work

Time has been a limiting factor and many ideas and solutions were not fully implemented even though they have been examined in thought and some of them also in practice. We mention here what could be additionally performed to investigate interesting areas or lead to clarifications or enhancements of the developed work.

In our opinion this project can be extended in interesting and exiting directions. The following are some of our suggestions to improve the capabilities of the prototype tool:

5.2.1. Taint analysis

The next module that can be implemented, is the one which performs the taint analysis by combining the permissions from the permission map and the vulnerable information flows detected after analyse the Android source code.

5.2.2. Visual tool

Implement a GUI to ease the execution of our project and provide a graphical interpretation of the reports.

5.2.3. Performance analysis

To better evaluate the performance of the presented solutions, to have a better understanding of how the properties affect the DFA.

5.2.4. Applications exposing the vulnerabilities

Develop one Android app including examples for each of the attacks scenarios mentioned in this project. Explain their features and how many vulnerabilities they have.
A. Hardware and software used

A.1. Hardware

The project work was developed using a laptop computer with the components described in the table [A.1]

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Apple MacBook Pro (Early 2011)</td>
</tr>
<tr>
<td>Operating system</td>
<td>OS X 10.8.5 (12F45)</td>
</tr>
<tr>
<td>Processor</td>
<td>Intel Core i7 (2.3 GHz, 4 cores)</td>
</tr>
<tr>
<td>RAM</td>
<td>8 GB (1333 MHz DDR3)</td>
</tr>
<tr>
<td>Graphic card</td>
<td>Intel HD Graphics 3000 (512 MB)</td>
</tr>
</tbody>
</table>

Table A.1.: List of hardware components

A.2. Software

For specific installation instructions, please refer to the website of the developer provided in the URL field of the Table [A.2] Apart from the tools mentioned there, it is necessary to have installed a .pdf viewer like Adobe Acrobat Reader and a spreadsheet processor, such as Microsoft Excel, to open the .cvs files.
### A. Hardware and software used

<table>
<thead>
<tr>
<th>Software</th>
<th>Version</th>
<th>URL</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>dex2jar</td>
<td>0.0.9.15</td>
<td><a href="http://code.google.com/p/dex2jar/">http://code.google.com/p/dex2jar/</a></td>
<td>Java decompiler</td>
</tr>
<tr>
<td>Eclipse</td>
<td>4.2 (Juno)</td>
<td><a href="http://projects.eclipse.org/projects/eclipse">http://projects.eclipse.org/projects/eclipse</a></td>
<td>The IDE needs several plugins. Refer to the WALA documentation about the requisites.</td>
</tr>
<tr>
<td>Java</td>
<td>J2SE 1.6.24 (64 bits)</td>
<td><a href="http://java.com">http://java.com</a></td>
<td>With special initialisation parameters (described in Appendix C).</td>
</tr>
<tr>
<td>WALA</td>
<td>1.3.5</td>
<td><a href="http://wala.sourceforge.net/wiki/index.php/Main_Page">http://wala.sourceforge.net/wiki/index.php/Main_Page</a></td>
<td>Problems with latest unreleased version. The release 1.3.4 works as well, but has some java 1.6 compatibility problems.</td>
</tr>
<tr>
<td>Graphviz</td>
<td>2.36.0</td>
<td><a href="http://www.graphviz.org">http://www.graphviz.org</a></td>
<td></td>
</tr>
<tr>
<td>Android</td>
<td>2.3 (API 10)</td>
<td><a href="http://developer.android.com/index.html">http://developer.android.com/index.html</a></td>
<td>Is needed the full compiled version of the source code, not the stub version which is normally packed with the SDK.</td>
</tr>
</tbody>
</table>

Table A.2.: List of software
B. How to obtain the .apk files

APKs are the installation files (binary distribution form) for Android apps. When you download and install any Android application from an online store to your Android device, you get nothing but an .apk file. There are several ways to obtain the .apk files from the Google Play Store. We choose to use the approach suggested in techbeasts.com due to its simplicity.

The steps to follow are the following:

1. Go to the site [http://play.google.com](http://play.google.com) and look up for your desired APP.
2. Click on the description of the APP that you are looking for, in order to enter to its detailed webpage.
3. Copy the URL shown in the navigation bar, which should probably include something like "com.appname.creator".
4. Now open the site: [http://apps.evozi.com/apk-downloader](http://apps.evozi.com/apk-downloader) and paste the copied URL into the text box "Package name or Google Play URL", and click on the button "Generate Download Link".
5. After this step, you will get some information about the APP such as its file size, version, etc., then you have to click on the button "Click Here To Download com.appname.creator". The download should begin immediately.

Once you have downloaded the .apk file of the desired application to analyse, you can use it as input to our static code analysis tool and assess its security.

C. Performance tuning for the JVM

The generation of the CG for the Android APIs is a resource-intensive, time-consuming process. The more precise the solver is, the more extensive and intensive the analysis becomes. In this situation, the performance of the JVM is a critical factor. As demonstrated in [51], passing different arguments to the JVM has a direct impact on the overall performance of the application. We gathered information for the JVM performance tuning from sources such as Oracle (Java’s developer), IBM, Apple, and external sites. In this appendix, we present a list of arguments for the JVM, limited to the ones used for the execution of our static data flow analysis, and their respective meaning.

C.0.1. JVM arguments of the CG generator

- -server -d64 -XX:+PrintVMOptions -Xconcurrentio -XX:+UseConcMarkSweepGC
- XX:+CMSIncrementalMode -XX:+CMSIncrementalPacing -XX:+UseParNewGC
- XX:+CMSGarbageCollection -XX:+CMSIncrementalPacing -XX:+UseParNewGC
- XX:+CMSParallelRemarkEnabled -XX:SoftRefLRUPolicyMSPerMB=1
- XX:ParallelGCThreads=4 -Xmx6G -Xms6G -XX:+DisableExplicitGC
- XX:NewSize=1536M -XX:MaxNewSize=1536M -javaagent:classmexer.jar
- XX:+UseTLAB -XX:ThreadStackSize=256K -XX:+UseStringCache
C. Performance tuning for the JVM

-XX:+UseCompressedStrings -XX:+OptimizeStringConcat -XX:+UseCompressedOops
-verbose:gc -XX:+PrintGCDetails -XX:+PrintTenuringDistribution
-XX:NewSizeThreadIncrease=32K -XX:MaxPermSize=1536M
-XX:+UseLargePages -XX:+ExitOnLargePageFailure

C.0.2. Categories of Java HotSpot VM Options

There are many initialisation parameters that can be passed to the JVM. Standard options for the Java HotSpot VM mean that they are recognised by Windows, Unix, OSX, Solaris and Linux. Some parameters might vary in other architecture/OS/JVM version. Some of the more common non-standard options are omitted here, but can be viewed by passing the -X option to java. The following are mainly non-standard options recognised by the Java HotSpot VM under the development environment presented in the appendix A:

- Parameters that start with -X are non-standard (there is no guarantee to be supported on all JVMs implementations), and are subject to modification or elimination without previous notice in subsequent releases of the JDK.
- Parameters that are set with -XX are not stable and are subject to modification without previous notice.

Depending on the type of parameters, they may need additional values to set their configuration properly. These are known as options, and they fall into the following classification:

- **Boolean options** are turned on with -XX:+<option> and turned off with -XX:-<option>. Disa

- **Numeric options** are set with -XX:<option>=<number or size>. Numbers are always integer values. When size in bytes is specified as a parameter in an option it can include ‘g’ or ‘G’ for gigabytes ‘m’ or ‘M’ for megabytes, and ‘k’ or ‘K’ for kilobytes: for example, 4194304 -meaning bytes-, 4096k, 4096K, 4m, and 4M are all equivalent values.

- **String options** are set with -XX:<option>=<string>, are usually used to specify a file, a path, or a list of commands

The options below are loosely grouped into categories: a) **Behavioral options** change the basic behavior of the JVM, b) **Garbage First (G1)** are garbage collection options, c) **Performance tuning options** are knobs which can be used to tune JVM performance and d) **Debugging options** generally enable tracing, printing, or output of JVM information.

**Behavioral Options**

- **-server**: It optimizes the Just-In-Time (JIT) compiler by performing continuous execution of byte code to trade slower startup time for faster runtime
performance over extended periods.

- `-d32` and `-d64`: Specify whether the program is to be run in a 32 or 64-bit environment. The major advantage of a 64-bit Java implementation is to be able to create and use more Java objects than its 32-bit counterpart.

- `-XX:+UseCompressedOops`: Enables the use of compressed pointers (object references represented as 32 bit offsets instead of 64-bit pointers) for optimized 64-bit performance with Java heap sizes less than 32 GB.

**Garbage Collection: General Settings**

- `-XX:-DisableExplicitGC`: By default calls to `System.gc()` are enabled, these are known as *explicit* GC calls. Using this parameter disables explicit calls to `System.gc()`. Note that the JVM still performs garbage collection when necessary.

- `+UseConcMarkSweepGC`: Flag to turn on the *Concurrent Low Pause Collector* GC. This algorithm is optimized for multi-CPU computers, it is used to minimize the GC pause. The concurrent GC still pauses the JVM and uses parallel GC to clean up short-lived objects. However, it cleans up long-lived objects from the heap using a background thread running in parallel with other JVM threads. The concurrent GC drastically reduces the GC pause, but managing the background thread does add to the overhead of the system and reduces the total throughput. This collector may deliver better response time properties for the application (i.e., low application pause time). It is a parallel and mostly-concurrent collector and can be a good match for the threading ability of an large multi-processor systems. The concurrent collector is used to collect the tenured generation and does most of the collection concurrently with the execution of the application. Typically applications which have a relatively large set of long-lived data (a large tenured generation), and run on machines with two or more processors tend to benefit from the use of this collector. This option has an effect only on multiprocessor computers.

- `-XX:+UseParNewGC`: A parallel version of the young generation copying collector is used with the concurrent collector. Parallel Minor Collection Options for multiprocessor machines, enables multi threaded young generation collection.

- `-XX:+CMSParallelRemarkEnabled`: Reduce remark pauses.

- `-XX:ParallelGCThreads=<n>`: Sets the number of garbage collection threads in the young and old parallel garbage collectors. The default value varies with the platform on which the JVM is running. The “n” should be the same as the number of processor cores available. For example, if there are two dual-core processors, enter: `-XX:ParallelGCThreads=4`. 
- **-XX:+CMSIncrementalMode**: Enables incremental mode. Note that the concurrent collector must also be enabled (with -XX:+UseConcMarkSweepGC) for this option to work. The incremental mode is meant to lessen the impact of long concurrent phases by periodically stopping the concurrent phase to yield back the processor to the application. This feature is useful when applications that need the low pause times provided by the concurrent collector are run on machines with small numbers of processors (e.g., 1 or 2).

- **-XX:+CMSIncrementalPacing**: Enables automatic pacing. The incremental mode duty cycle is automatically adjusted based on statistics collected while the JVM is running.

### Garbage Collection: Memory Usage

- **-Xms<size>, -Xmx<size>, -Xmn<size>**: Specific parameters pertain to allocation of RAM memory to the JVM, namely for setting the minimum memory (-Xms), maximum memory (-Xmx), and the size of the “young generation” memory space for short-lived objects (-Xmn): if the application generates lots of new objects, the performance might improve GCs dramatically by increasing this value. The “young generation” size should almost never be more than 50% of heap. To prevent attempts to reallocate memory, -Xms and -Xmx must be set to the same value, generally, a number that is roughly 3/4 of the total RAM is preferred: i.e., for a server with 8 GB RAM, the preferred setting is a minimum of 6 GB.

- **-XX:NewSize=<size>**: Sets the default size for the Eden generation of allocated objects. The default value is 640K. (The -server flag increases the default size to 2 MB).

- **-XX:MaxNewSize=<size>**: Set -XX:NewSize and -XX:MaxNewSize to the same value. The number should be 1/4 of the size chosen for -Xms.

- **-XX:MaxPermSize=<size>**: Modifies the size of the permanent generation. The default value is 32 MB.

- **-javaagent:classmexer.jar**: To initiate the Classmexer agent. Classmexer is a simple Java instrumentation agent that makes available some convenience methods for measuring the memory usage (number of bytes occupied) of Java objects, including “sub-objects” or objects referred to by the “main” object(s) passed in, from within an application. A copy of the referenced .jar must be in the working directory (the directory from which the application is started).

### Threading options

- **-XX:NewSizeThreadIncrease=<size>**: Specifies how much to increment the young object space size per active thread. This option may be useful in
regulating an increased allocation rate due to increased threads. The default increment is 16 (KB).

- **-XX:ThreadStackSize=</size> or -Xss</size>:** Changes the thread stack size from the operating system default size. On 64-bit systems, the call stack for each thread is allocated 1 MB of memory space. Most threads do not use that much space. By using the flag with a smaller value, for example 256 KB, the stack decreases its size to allow more threads. 64 KB is the least amount of stack space allowed per thread.

- **-XX:+UseTLAB:** Enables a thread-local allocation buffer so that heavily threaded applications can be allocated more efficiently, greatly increasing allocation performance. For Java 1.4.2, this option is on by default on multiprocessor computers and in OS X Server. For J2SE 5.0, it is on for all configurations.

- **-Xconcurrentio:** to use LWP based synchronization instead of thread based synchronization and some other internal options, it generally helps programs with many threads, particularly on Solaris. Certain applications can speed up by over 40%.

**Performance Options**

- **-XX:+UseLargePages:** Tells the HotSpot VM to allocate the Java heap using large (2 MB) pages instead of default (4 KB) pages. This allocated memory is wired memory, and therefore the requested amount must be available as free physical memory. This option is available in OS X v10.6 and later on machines running the 64-bit OS X kernel, like Ultrasparc CMT systems. In some situations, performance can be improved by using large page sizes.

- **-XX:+ExitOnLargePageFailure:** Tells the HotSpot VM to exit if large page support is not enabled or available.

- **-XX:+UseStringCache:** Enables caching of commonly allocated strings.

- **-XX:+UseCompressedStrings:** Use a byte array for Strings which can be represented as pure ASCII. (Introduced in Java 6 Update 21 Performance Release).

- **-XX:+OptimizeStringConcat:** Optimize String concatenation operations where possible. (Introduced in Java 6 Update 20).

- **-XX:SoftRefLRUPolicyMSPerMB=1:** It adjusts the amount of time (integer values representing milliseconds) that softly reachable objects will remain alive after the last time they were referenced. The default value is one second of lifetime per free megabyte in the heap.
Debugging Options

- `-XX:+PrintGCDetails`: To print more detailed logging information of the GC.

- `-verbose:gc` and `-Dsun.rmi.dgc.client.gcInterval`: Print real-time information about the GC status, so they are useful in determining garbage collection issues while running a program.

- `-XX:+PrintVMOptions`: Prints the list of options passed to the JVM actually in use.

- `-XX:+PrintTenuringDistribution`: Prints tenuring age information for allocated objects in the young generation.
D. Content of the CD

The project is composed from a set of files that are explained now.

- **Studienarbeit (Project’s folder):** is the Eclipse project that can be imported into a workspace including our own developed classes for the processing of .apk files (extracting the properties of the app) and static data flow analysis.

- **Our Classes for extending WALA (folder):** in order to perform our analysis, it was added by us some specific functionality: the original implementation has a compulsory call to the .pdf viewer, which interrupts the flow of our analysis, read the Section 4.2.3 for more information. The modified WALA class is: `com.ibm.wala.viz.PDFViewUtil`, namely the method `generateIRpdf`. To use our version, copy this class the folder where the normal WALA framework class is: `com.ibm.wala.viz`, which belongs to the project `com.ibm.wala.core`.

- **Test Scenarios for Android Apps (folder):** contains some .apk files corresponding to the Android App that were used in the project and their decompiled versions. There are as well, one Android app’s manifest for each of the .apk files. The analyses results are the .csv files.

- **Data Flow Analysis (folder):** contains the generated .pdf files for data structures such as Call Graphs, Intermediate Representations and Type Hierarchies. It includes as well the temporal dot files.

- **Jars and Libraries (folder):** are the external .jar libraries used in the project: Android Source Code, Class Meter, Log4J and JLex sample java application.

- **Logs (folder):** stores the execution logs for the experiments that we conducted.

- **Studienarbeit EIRS.pdf (pdf File):** is this report in electronic format.
Bibliography


