Bachelor thesis
Computer Science

Radboud University

Security analysis of the IRMA app using SPARTA and fuzzing

Author:
Laurens Brinker
4361733

supervisor/assessor:
Erik Poll

Second assessor:
Aleks Kissinger

September 5, 2016

Abstract

This paper discusses a security analysis of the Android-application called *CardProxy*. This application is created by the IRMA (I Reveal My Attributes) project to be able to prove certain personal attributes (e.g. your age) that are stored on a smart card. The purpose of this research is not to check this specific application, but to see if the chosen methods can be used to analyze an application that handles sensitive data like CardProxy. Also, we did not focus on the underlying (cryptographic) security, but more on the security of the implementation. This means we will not analyze the security protocols that were used for purposes like encryption, but the usage of these protocols and implementations in the context of the CardProxy application. To clarify: Even if the cryptographic protocols behind IRMA are secure, a buggy implementation can still contain security flaws.

The information handled by the CardProxy can be very sensitive, therefore it is of great importance that the confidentiality of such data is guaranteed. There are multiple ways of analyzing an app's behaviors, but this research was primarily focused on an annotation based analysis for information flows. The specific tool that was used was SPARTA. Using this tool, we analyzed the source-code to investigate information-flow related security properties without needing to run the application. The second technique we wanted to use was dynamic program analysis. This is a process where one can provide and even automate some input and observe the application's behavior. The goal of this technique is to see if this will lead to unexpected (undesirable) behavior. Given that the CardProxy app is very security-sensitive, it is of great importance to investigate techniques to detect security flaws, or to prove the absence of certain types of flaws.

During the course of research we found that getting SPARTA to work with CardProxy was quite a problem because CardProxy is built using Gradle. We found that CardProxy contains some debug-code that should be removed because it logs the user's pin-code. SPARTA also successfully detects this code. We also concluded that using automated programs to do dynamic analysis is a bit more complicated because our NFC-traffic has to be emulated. Our research will primarily focus on the question if our chosen tools, with the focus on SPARTA, can provide a way to analyze applications like CardProxy.

Contents

Chapter 1

Introduction

IRMA is a privacy-friendly authentication solution. It can be used to provide attributes where they are requested. For instance, in order to access a liquor website one must be over the age of 18. The IRMA's CardProxy app functions as a proxy where personal information is retrieved from a personal smart-card via NFC and sent to the requester of the attribute over an internet-connection. So now arises the problem: How can we make sure that this application does as it proclaims? For instance, we would like this application to get into websites where we have to be above a certain age, but we definitely would not want sensitive data ending up on the internet.

There are multiple ways of checking that an application does not have unexpected or unwanted behavior. It is of great importance that we check that the app does not do things it is not allowed to and that sensitive information is only accessible to the parties that should have access to it. In this research:

- We gathered some tools which can be used for Android security analysis.
- We used a pluggable type-checker called SPARTA to analyze information flows (static analysis).
- We tried fuzzing to analyze the application's behavior (dynamic analysis).
- We concluded if the tools that were used provide a good way to check the implementation of applications handling personal information like CardProxy.

By doing this we would like to answer the following question: Using SPARTA and fuzzing, is it possible to analyze security-sensitive applications like CardProxy as a non-expert in these techniques?

Chapter 2

Background

This chapter gives some background information about certain aspects of my research. Section 2.1 discusses the distinction between malware and security flaws. Section 2.2 discusses the difference between static program analysis and dynamic program analysis. After this section the Checker-Framework and SPARTA, which are forms of Static program analysis, will be discussed in Section 2.3.1 and 2.3.2. As a form of dynamic program analysis we will talk about fuzzing in Section 2.3.4. To conclude extra information about IRMA will be given in Section 2.4 for those who are interested. For this research the particulars of IRMA are not important.

2.1 Malware or security flaw

Before we look at specific security analysis tools, it is important to recognize the difference between actual malware and security flaws. Malware is short for malicious software. Malware is designed to damage computer systems. Security flaws, however, are more subtle. They are not designed to directly cause damage to a system, but can be used for malicious actions. Often, this flaw is created by accident. For example, consider a banking application where the pin code is sent over internet without the pin code being encrypted. This is obviously a security flaw because eavesdroppers could read the pin code. But sending a message over the internet is not considered malware, and the fact that the message, in this case the user's pin code, is sent unprotected is just a matter of a poor implementation by the developer. Although this application is not considered malware, one would like to avoid applications where the implementation is not done correctly. So malware is intentionally malicious, whereas normal software can be insecure by accident (security flaw). Our research is primarily focused on finding security flaws.

2.2 Static- versus dynamic program analysis in a nutshell

The difference between static program analysis and dynamic program analysis is that static program analysis tests the source code or a dissected .apk file of a program. This means we can analyze the application without actually running it. Dynamic testing however requires the program to run so that the behavior of the application can be reviewed. We can also illustrate the difference as following:
With static program analysis we analyze the applications as if it were a white box. We know what is inside the box so that we can predict its behavior based on that knowledge. Dynamic program analysis generally analyzes an application as if it were a blackbox. We do not know what is inside, or we do not look inside the box but by examining its behavior given certain input / events we can check if there is any unexpected behavior. A simple example of this is providing malicious QR-codes to the CardProxy application to see if this leads to unwanted results.

2.3 Analysis tools

In this research we chose both a static program analysis technique (SPARTA) and a dynamic program analysis technique (Fuzzing) to see if this provides a good way to analyze our application. Section 3.1 will discuss what our alternatives were and why we chose to use these two techniques. The sections below will provide some background information about the Checker-Framework (which lies at the basis of SPARTA), SPARTA itself and fuzzing.

2.3.1 Checker-Framework

The Checker-Framework is used to create and check additional type-qualifiers for Java. This means that at compile time the framework will run additional type-checkers specified by the developer in means of annotations. For example, one can create an `@Encryption` type-qualifier. Using this type the developer can specify what functions and variables yield which type-qualifier. This way the Checker-framework can be used to check if specific variables are of that type-qualifier. For instance if it is specified that the input of the function `"sendPassword(String password)"` must send an encrypted password. We can annotate our method as following:
`"sendPassword(@Encrypted String password)"`.
The Checker-framework can then be used to check if the origin of the variable password is also of the `@Encryption` type.

In Java 8 one can use **JSR308** to provide the annotations. This is because JSR308 was incorporated in Java SE 8 with the purpose to be able to detect and apply annotations. Because JSR308 is not compatible with earlier versions of the Java SE, and Android uses Java 7, one needs to provide annotations in comments: `/*@Annotation*/` (see Figure 2.1 for a graphical representation of the relation between JSR308 and Checker-Framework).

2.3.2 SPARTA

SPARTA is a project created at the University of Washington. SPARTA is short for Static Program Analysis for Reliable Trusted Apps. Using the Checker-framework, SPARTA can be used to detect undesired information-flows in Android applications. More specifically it can also be used to verify that an application indeed contains no malicious information flows [Ernst et al., 2014]. SPARTA's input are annotated pieces of source-code, these annotations will be in the form of type-qualifiers created by using the Checker-framework. These type-qualifiers are necessary to prove a certain security property. These security properties that SPARTA is useful for are related to information flows. For instance, my GPS-location should not be send to the internet. Using the typequalifiers `@Sink` and `@Source` we can analyze information-flows and check if they uphold certain policies. With `@Sink` you specify where the information might go to and with `@Source` you specify where the information might come from. For example, suppose we have the following statements where we annotate an object "loc" that has a GPS location as source and internet as sink. This object will read GPS data and send it over the internet:

```
@Source(LOCATION)@Sink(INTERNET)String loc = readGPS();
sendToInternet(loc);
```

We can then use policies to review if the code above should be allowed or not. A policy specifies which flows are allowed to occur has the following form:
`[`*`SOURCE`*`] -> [`*`DESTINATION`*`]`

For instance, in the example above we could write the following policy:
`LOCATION -> INTERNET`

This policy means that information from the GPS-reader is allowed to flow to the internet. Because we have annotated `LOCATION` as source and `INTERNET` as sink the piece of code is allowed by our policy. If we would however change the policy to `LOCATION -> SMS` the SPARTA compiler would give a policy error because we would still have a `LOCATION -> INTERNET` flow but our policy file would only allow `LOCATION -> SMS` and disallow `LOCATION -> INTERNET` (example from [Poll, 2015]).

[Ernst et al., 2014] reports that they have analyzed 72 applications, in which they detected 57 Trojans. These Trojans were specifically designed to defeat typical malware analysis. According to the researchers they detected 96% of the malware that was related to information flow and 82% of all malware that was present in their particular sample set. For more information it is recommended visiting the SPARTA website:
`https://www.cs.washington.edu/sparta`

2.3.3 Relation between SPARTA and Checker-Framework

The relation between SPARTA and the Checker-Framework is as following: SPARTA is built on top of the Checker-Framework [Ernst et al., 2014]. SPARTA uses the Checker-Framework by creating two specific type-qualifiers. These type-qualifiers are, as stated on the previous page, `@Sink` and `@Source`. These type-qualifiers are used by SPARTA to analyze an application's information-flow via static program analysis. Both SPARTA and the Checker-Framework are used to analyze Java code by adding specific pluggable type-qualifiers. Pluggable in the sense of being able to add and delete owner-specified types into the type-system.

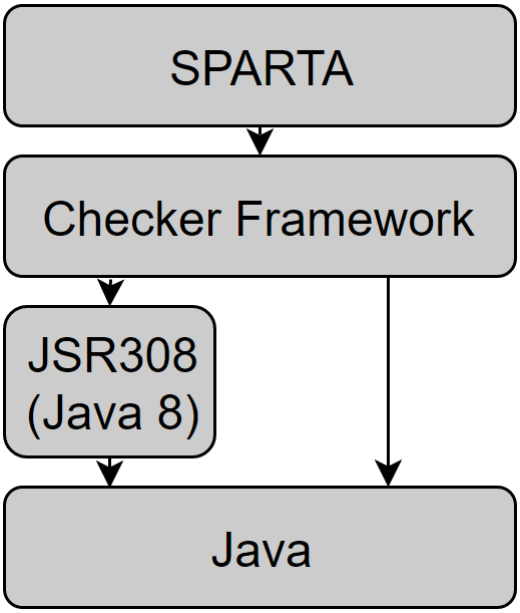

Figure 2.1: Relations illustrated (simplified)

2.3.4 Fuzzing

Fuzzing is a form of dynamic program analysis where we consider our program as a blackbox and start providing different input. Fuzzing is a technique used to uncover security vulnerabilities like Cross-site scripting, buffer-overflow, Denial of Service, etc. This analysis technique uses unexpected/-malformed data as program input [Garg, 2012]. Generally speaking we consider two categories of fuzzing programs:

1. **Mutation-fuzzing**. Also considered as "*dumb-fuzzing*", Mutation fuzzing is about changing existing input blindly. An example of mutation-fuzzing is bit-flipping. We can take normal input of the program and make small adjustments to the bits and see what kind of behavior this triggers in the program. This kind of fuzzing is called "dumb-fuzzing" because you don't need to have specific knowledge about the program you're testing. One needs not to be familiar with the protocols of the program.

2. **Generation-fuzzing**, however, is the opposite of "dumb-fuzzing". In Generation-fuzzing one needs to understand the program's protocol and adjust input based on the protocol. For instance, if our program would execute bank transactions we could perform generation fuzzing by analyzing the protocol, adjusting some protocol-stage and then check if we can induce malicious behavior by giving an input. In Generation-fuzzing this input is still related to the protocol's stage. For instance, in the IRMA CardProxy protocol one needs to scan a QR code to be able to verify for that specific instance (for example a liquor website). Because we know the protocol, and we know that there is a stage where the app scans a QR code, we could use generation fuzzing to provide malicious QR codes to the application to see if this leads to undesired behavior.

See [Sutton et al., 2007] for more information on fuzzing.

2.4 IRMA

I Reveal My Attributes, IRMA, is a project of the Radboud University in cooperation with SURFnet, TNO and SIDN. The project provides a way of personal authentication where an authenticator only shows information that is requested by the requester of the information, in most situations the service provider. For example, when you want to access a bar you have to show some sort of identification to prove that you are indeed over the age of 18 years. This means you would have to show an ID, passport or driver's license to the bouncer of that bar. The problem however is that these documents contain more privacy sensitive information that is not relevant for proving your age (e.g. a Social Security Number). The IRMA project has created a way where one can prove a certain personal attribute, like being over 18 years old, without having to disclose other sensitive information. This can make the identification process more privacy friendly [Alpár and Jacobs, 2013]. The original IRMA setup consists of the following elements: Every user has a smart card. Upon registering for IRMA, personal information like birth-date are checked by the authorities. This information is then stored on a personal smart card. When a requester, like a service provider, wants to check if an attribute holds, he holds an NFC-enabled device against the smart card. The software on that particular device is then able to read the smart card and deduce if a certain attribute holds.

There are multiple kinds of devices capable of communicating with the smart card. The biggest requirement is that an NFC-module is present in that particular device. The software that runs on these devices is developed by IRMA. For Android devices, IRMA has released a couple of applications:

- **CardProxy**. This application can be used by the normal users. For instance, I have a smart card with the attribute 18+. If I want to access a liquor site which has this attribute as a requirement. I can use this application to let the card communicate with the website. This is done as following: If I want to access this website with the required attribute, I can use the app to scan a QR-code on the website. I then place my smart card against the NFC-module of my device, the app will then read the smart card after a personal PIN-code has been entered and then communicate back to the website whether or not the user has successfully provided the attribute.

- **Verifier**. This application is typically used by the service providers. When one wants to check a certain attribute of a smart card, this application can be used to read the smart card to see if that specific attribute holds. For instance, if a bouncer at a bar wants to check the age of a customer, he can hold their smart card against his device (again with NFC module enabled) and the application will check if the attribute holds.

- **Management**. This application can be used by the user to view it's cards data. This application can also be used to change one's PIN-code.

- **CardEmu**. This is the most recent development of IRMA. Instead of storing information on a smart card, the device will hold the attributes. So in the future the user can use his device without the smart card to provide certain personal attributes.

For more information of the IRMA project it is recommended checking out their official website: `https://www.irmacard.org/` and their GitHub page for source-codes: `https://github.com/credentials`

Chapter 3

Prior research

This chapter discusses the work that has already been done in the field of Android application security analysis. This chapter first discusses [Fedler et al., 2013], a paper about typical Android antivirus followed by a paper about the research performed by the developers of SPARTA. Finally Section 3.1 contains a number of tools that are available to analyze the Android OS and Android applications. Sectoin 3.1 ends with a small overview with some practical data about the tools.

The first paper, [Fedler et al., 2013], is an assessment of the effectiviness of current antivirus apps. They also looked at the performance of these apps in scenarios that were slightly modified. For example, typical antivirus mainly relies on signature and heuristic analysis of known malware. But in this paper the authors also evaluated how well the tools can detect slightly modified malware with different signatures. The authors concluded that slightly altered malware is not detected well enough. They suggest that there is room for improvement for the signature and heuristic detection of current antivirus software. They claim it is still trivial for malware vendors to slightly modify their code and avoid detection. Their final conclusion is that on-device antivirus software is not sufficient to ensure the integrity of devices. This means that tools that do not rely on signatures and heuristics could possibly provide a better way to detect new malware.

The second paper, [Ernst et al., 2014], is written by the developers of SPARTA. This paper describes a high assurance app store that requires applications to be annotated by app-developers. The store's auditors will then verify the information flow properties and the annotations. Further on, the paper discusses a Red Team evaluation. The sponsor of SPARTA, DARPA, wanted to evaluate SPARTA. They hired 5 companies (red teams) to create both malicious and non-malicious applications. In total they created 72 Java applications, of which 57 were malicious. The results were that SPARTA

detected 96% of the malware. They also found that 19 applications had unjustified information flows. For instance, an application was allowed to load photo's from local-storage and it was allowed to send highscores to the internet. However, the application was definitely not allowed to send the photo's from local-storage over the internet to a server.

Another interesting chapter in the paper discusses a usability study. The first conclusion the researchers made is that when given an unknown application, the vast majority of time was spent reverse-engineering the application. Another conclusion was that the annotator only had to write 4% of annotations where they could have been written. The other 96% was either defaulted or inferred. As for learnability, the researches conducted a study involving 32 third-year Computer Science students. The students reported that the first annotations were the most time-consuming because they were still learning how to use SPARTA.

3.1 Available tools

We found a nice list [1] of Android security tools that could be interesting for our research sorted on Static, Dynamic and Fuzzing analysis. These categories are based on Section 2.2 where we explained the difference between static analysis and dynamic analysis.

3.1.1 Static analysis tools

1. **SPARTA**: Static program analysis to analyze information flows by annotating source-code. See Section 2.3.2 and Section 5 for more information (paper of creators: [Ernst et al., 2014]).

2. **AndroWarn**: Tool that statically analyses the bytecode of an application and generates a report based on information flow analysis. This tool will for example analyze: Geolocation information leakage, connection interfaces information leakage (WiFi details etc.), telephony services abuse, information flow interception (like capturing calls / video's) and external memory operations. For more information check out the AndroWarn GitHub page [2].

3. **DiDFail**: Research prototype that is based on Soot and FlowDroid. The principle of DiDFail consists of two phases: Given a set of applications the tool determines the information flows that are enabled on an individual level. This means that the tool starts by analyzing the

[1] https://github.com/ashishb/android-security-awesome
[2] https://github.com/maaaaz/androwarn

applications independently. Then, in the second phase, the tool enumerates the dangerous information flows in the set applications. This means that the tool can be used to analyze information flows between different apps. In [Burket et al., 2015] one can find more information about this tool.

4. **FlowDroid**: Also works with the Source and Sink principle. Performs a static program analysis. Additionally they use a model of the Android's lifecycle to allow the analysis to handle callbacks invoked by the Android framework. The developers also state that context, flow, field and object-sensitivity allows the analysis to reduce the amount of false positives. For the paper see [Arzt et al., 2014].

5. **Amandroid**: Analyses inter-component control and information flows. We consider components as modules of the device like storage and SMS. The tool converts an application's bytecode into an intermediate representation on which static analysis can be performed. The tool is able to illustrate the flows by creating an inter-component information flow graph and data dependency graph which can be used to detect explicit information flows. For more information see [Wei et al., 2014].

6. **Smalisca**[3]: Tool that analyses .smali files. Smali is similar to assembly and multiple .smali files make up the final .apk file of an application. Smalisca analyses the .smali files and provides a graphical overview of the relationships in terms of information flows between classes and modules of the application.

7. **DroidSafe**: Tool that again analyses information flows by analyzing bytecode or an APK of the application. It already contains a verified core of 98% of the API calls in Android applications. They also provide an Eclipse IDE plugin [4].

3.1.2 Dynamic analysis tools

1. **IntelliDroid**: This tool generates specific input for Android applications. This is the opposite of random fuzzing where we generate random inputs and observe the output. IntelliDroid is also capable of determining the order in which input has to be provided. Another interesting part is that IntelliDroid uses static analysis to identify and generate inputs in such a way that the dynamic analysis only has to execute 5% of the application's code. With this tool one can analyze information flows. But dynamic analysis also makes it easier to check other properties of a applications. For example, the ability to DoS or

[3] `https://github.com/dorneanu/smalisca`
[4] `http://mit-pac.github.io/droidsafe-src/`

crash the application are generally found using dynamic analysis. The paper of IntelliDroid: [Wong and Lie, 2016].

2. **Android DBI**: Enables you to see I/O logging and debug information of an application. It also provides a way to inject your own code into the application. The toolkit basically provides several ways to "hack" the application. In a slide show of the toolkit the developer also talks about NFC fuzzing. Instead of fuzzing with an actual card the developer suggests simulating NFC traffic by pushing data to the NFC stack. For more information check out the GitHub page [5].

3. **DroidBox**: This tool observes what an application is doing during execution. For instance, it monitors if there are files being read/written or sniffing the network packets that are sent to/from the application. The tool uses an emulator to run the application.

4. **TaintDroid**: For this tool one needs custom-built firmware for his/her device. TaintDroid tracks when private information is handled by applications and monitors if this information leaves the phone. [Enck et al., 2014]

5. **Android-Hooker**: Android-Hooker is an open-source project for dynamic program analyses of Android applications. It provides tools and applications that can be used to automatically intercept and modify any API calls made by a targeted application. Android-Hooker uses ElasticSearch and Kibana to give a graphical visualization of the results. Source: `https://github.com/AndroidHooker/hooker`.

3.1.3 Fuzzing

Although typical fuzzing is also dynamic analysis the webpage (see footnote 1 on the previous page) considered it a different sub-category.

1. **IntentFuzzer**: Android intents are objects that are passed between components of the Android device. The IntentFuzzer performs static analysis to determine the structure of the intents and uses this to create inputs for fuzzing. This fuzzer is intended to find vulnerabilities in the Android OS [Sasnauskas and Regehr, 2014].

2. **Radamsa**: Radamsa is a black-box fuzzer. It does not require information about the application. The tool takes input for an application, for example "aaa" and creates variations to this input like "aaaa". The tool is especially useful to see how the application reacts to weird input. Radamsa's usage for Android applications is limited since the tool is mainly for Windows and Linux programs [6].

[5] `https://github.com/crmulliner/adbi`
[6] `https://github.com/aoh/radamsa`

3. **MFFA (for Android OS)**: MFFA stands for Media Fuzzing Framework for Android. The principle of this tool is to generate media files that are corrupt but structurally allowed. In other words, the tool creates malformed media files that are accepted by Android. This tool is also mainly to detect vulnerabilities in the Android OS and standard applications like Media Players, Web Browsers, etc. [7].

3.1.4 Overview

The table below will be a brief overview of the tool with some possible relevant information. In the Gradle support entry SPARTA was classified as IND (indirect) because the Checker-Framework has Gradle support enclosed in its manual. The entries with a YES * mean that it does not matter how the application is built. For example, some tools only use the .apk file to do Static Program Analysis (they can dissect the .apk file).

Table 3.1: Small practical overview

Tool	Creator	Gradle support	Last Update
SPARTA	University of Washington	IND	Aug 25, 2016
DiDFail	Will Klieber	YES *	Feb 1, 2016
FlowDroid	Steven Arzt	YES *	Aug 10, 2016
Amandroid	Argus Laboratory	YES *	Jun 16, 2016
Smalisca	Victor Dorneanu	YES *	Jun 22, 2015
DroidSafe	MIT CRS	NO	June 13, 2016
IntelliDroid	Michelle Wong	YES *	Aug 23, 2016
Android DBI	Collin Mulliner	YES *	Jun 13, 2015
DroidBox	Patrik Lantz	NO	Sep 25, 2015
TaintDroid	Intel Labs and Penn State University	NO	Feb 6, 2014
Android-Hooker	AMOSSYS	YES *	Aug 9, 2016
IntentFuzzer	iSECpartners	N/A	Dec 10, 2015
Radamsa	B. Archer and Darkkey	N/A	Jul 20, 2016
MFFA	Alexandru Blanda	N/A	Aug 17, 2015

[7] https://github.com/fuzzing/MFFA

Chapter 4

Research

This chapter discusses the choices we made with respect to our chosen research and the tools to do this research with. We will begin by explaining the goal of our research in Section 4.1 and the initial choices we made. Section 4.2 introduces the context of the researcher so that the readers will understand his field of expertise prior to the research. Section 4.2 will explain the purpose of our research. Section 4.3 discusses what research with respect to the CardProxy we did before analyzing it with SPARTA and fuzzing in terms of "regular" malware and manual analysis.

4.1 Research question

Chapter 1 introduced our research question briefly. Now we would like to elaborate it a bit more.
The purpose of this research is to see if static program analysis and dynamic program analysis can be used to detect malicious code and to prove that certain malicious or vulnerable code is not present. More specifically, we will examine if SPARTA and fuzzing are sufficient to analyze applications like CardProxy that handle sensitive data. It is of great importance for the techniques used to be easy to use. The last property is also why we will explain my background in Section 4.2. Someone who wants to analyze an application should be able to do this without requiring extensive background knowledge of the tool. Especially if we consider the concept of [Ernst et al., 2014] where we have an app store where the applications provided are required to be already annotated by the developers to prove that the application has no insecure information flows. For a developer it should not be a burden to have to learn how to use SPARTA.

Why CardProxy

CardProxy is not the latest application of the IRMA project. Their latest app, CardEmu, does not use the smart-card to store the attributes but it uses the device itself as storage for attributes. However, from an information-flow perspective the CardProxy application is very interesting to analyze because it uses the smart-card. Sensitive data is requested from the device, read from the smart-card and fed back to the requester. This means that the implementation of this application must be of great quality because an error in the implementation can cause big problems in the exchange of sensitive data.

The source-code of the CardProxy app can be found at: `https://github.com/credentials/irma_android_cardproxy`

4.2 My background and expertise

For this research it is also important to briefly inform the reader about the researcher. This way, the reader will understand the background of the researcher and the scope of his capabilities with respect to the tools. It is especially useful because the research question implicitly asks if a person with knowledge of programming, but not of the specific tools, can still use them effectively to analyze his or her applications. My background at the time of starting the research was as following:

- Student Computer Science,
- Familiar with the basics of Gradle,
- Proficient with Java programming,
- Familiar with Android developing.
- **Not** familiar with the specifics of Java compilers.
- **Not** familiar with SPARTA.
- **Not** familiar with fuzzing.

4.3 Initial investigations of CardProxy

4.3.1 "Ordinary" Android antivirus analysis

When checking an application for malware one of the first logical steps is to analyze the application with antivirus software for Android. These software usually calculate the application's signature by hashing the program and checking this signature in their database of known malware. But as [Fedler et al., 2013] correctly conclude it is easy to circumvent this by changing the bits of the program a bit so that you would have the same (malicious) program, but with a different checksum. They also remark that common antivirus software is not able to dynamically analyze the applications so that downloading exploits/malware at runtime can go unnoticed. Nevertheless, it can never hurt to check the applications with antivirus software.

AndroTotal [1] is a great website where one can upload an .apk file of the application and let it be checked by various Android antivirus software. Below you can see the result of the AndroTotal test. [2]

Sample SHA-256	57511b4c9299037cc8a999ab640e8dfc1461a94a13426ca4d790961da755adff
Sample SHA-1	992e20f136517505d7d17852221aa37767e51ca3
Sample MD5 7	37ead3d706b9229a2b08837ef080780f
Detections	0/6

The 6 tools used are, similar to [Fedler et al., 2013]:

1. AVG Mobile
2. Bitdefender
3. Comodo Security Solutions
4. ESET
5. TrustGo inc
6. McAfee Mobile Security

For CardProxy, all the tests came back negative (no threats detected). An important notion to bear in mind is that these anti-virus software only detect malware. Security-flaws are generally not considered malicious by these tools. See Section 2.2 for more information about the difference between malware and security flaws.

[1] `https://andrototal.org/`

[2] `https://andrototal.org/sample/57511b4c9299037cc8a999ab\`
`640e8dfc1461a94a13426ca4d790961da755adff`

4.3.2 Checking the code "by hand"

Previous chapter discussed an automated way of virus detection of the application. Now we will talk about some more in-depth analysis of the code. If you have access to the source code of a project it is also good to just check the code "by hand". Obvious buggy code can possibly detected this way. When going through the source code of CardProxy we quickly saw something that could be considered undesirable. The code was still the debug version. Although adding debug statements to code is not malicious, one must make sure to remove these lines of code before releasing the application. In the case of CardProxy the following piece of code immediately caught our attention:

```
Log.i(TAG, "PIN entered:  " + dialogPincode);
```

This statement logs the PIN-code of a user to the console. The console can be read by every application. Obviously this statement is highly insecure as it leaks the PIN of the user. Surely this is code that is not intended to be released. If we could for example use SPARTA to detect this before the application gets released it would prevent the leakage of sensitive data. Other than the undesired debug code we found no obvious malicious/vulnerable code in the CardProxy repository.

Chapter 5

SPARTA

This chapter discusses SPARTA. Section 5.1 will briefly compare SPARTA policies and Android permissions. Section 5.2 discusses which information flows properties we would like to analyze. Then, Section 5.3 discusses the main approach of SPARTA and the approach I followed. Section 5.4 discusses the Inference-framework and the principle behind Stub files to determine annotations. Then, Section 5.5 will briefly list the steps needed to take to get the tool working. Finally we will talk about the results of running the tool in Section 5.6.

5.1 Android permissions vs SPARTA policies

Before installing an application the Android OS shows you a message box with information about permissions that a specific application is requesting. An example:

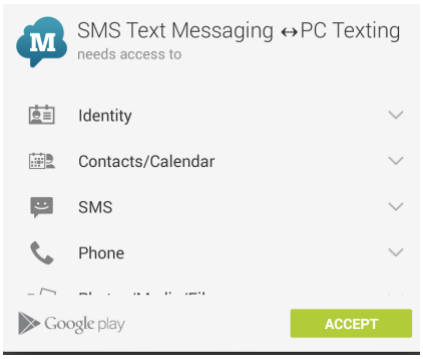

Figure 5.1: Example of Android app store permissions

Because Android makes every application run in their own Sandbox (their own environment with different permissions), Android can specify which specific permission an application can use. These permissions are the ones

specified by the developers and agreed by the user upon accepting the permissions notification. One thing to notice is that these permissions are the device's components and corresponding APIs that an application is allowed to use. So for the example above the application could access the SMS component and the Contact list. These permissions however do not specify if we allow data to flow between these allowed components. This is where SPARTA's policies come into play. With these policies we can specify what information-flows between components we allow. We will discuss these policies in the next section.

5.2 Information flow properties to analyze

It is kind of trivial, but surely we would want to analyze information-flows that deal with privacy sensitive information. This means that we both have to analyze how the information comes into the application and how it leaves. In our case our main input is data obtained via NFC, keyboard and camera and the output is mainly to internet or local storage. Especially how data leaves the application is very important to analyze since this is the place where our data is "released into the wild". However, this is not all. When analyzing information flows it is not sufficient to just analyze how the information flows into and out of the device, we also need to look at how the information is handled internally. As noted in Section 4.3.2 logging a password to a debugging console is not desired. Therefore, we must also check that sensitive information does not leak to places like consoles, memory, shared-preferences and so on.

Because, as Section 5.3.1 will further elaborate, SPARTA works with white-listing instead of black-listing it is hard to specify how we will prohibit certain information flows. However, if we tackle the problem from another perspective, we first disallow every information flow (by providing zero allowed flows) and then check for each flow whether or not it is allowed. This is the approach SPARTA recommends (see section 5.3.1).

5.3 Approach

5.3.1 SPARTA's approach

First of all it is really useful for someone with no experience in the field of annotations to fiddle around with them. I highly recommend using the Checker-framework first because they have made a really easy step-by-step tutorial to go through a couple of practice cases. I personally used the Eclipse plugin they have created. The full tutorial can be found at `http://types.cs.washington.edu/checker-framework/tutorial/`.

To get started with SPARTA the first step would be to install SPARTA. See Section 5.4 for more specific information. The tool's creators describe in their tutorial the following basic approach for using SPARTA:

1. Infer annotation types. Using the Inference framework we can automatically annotate our source-code (we will give more information in Section 5.4).

2. Run the Information-flow checker. This will yield all information flows as errors because we have not specified a flow policy yet.

3. Create a file called flow-policy to store all the policies which we would like to allow. This is one of the strengths of SPARTA because instead of disallowing certain flows, we must specify which flows we do allow. This manner is a white-listing approach which follows the principle of least privilege. This privilege states that we only allow actions that are necessary and nothing more. So we would rather block an allowed action than allow a malicious action.

4. The next step is to start white-listing certain information flows. For example, if we would run the tool and get an error where the policy `LOCATION -> INTERNET` is detected as prohibited. If we would like to allow it we must be able to justify this information flow. An information flow is justified when the developer can show the (potential) verifiers that the information flow is necessary and poses no threat to the users. If the flow can be justified it can be added to the flow-policy file. The error should now disappear.

5. One must try to avoid adding policies to the policy file that are too permissive. For example `ANY -> INTERNET`. This flow is in most cases too permissive because ANY contains too much sources we cannot account for. Avoiding policies that are too permissive requires additional manual annotating of the code. This is because ANY often originates from SPARTA's default flow qualifiers (see Table 5.1). Some

good examples of policies can be found in the Sparta manual [1].

6. When all errors generated by SPARTA have been fixed by creating a policy file that describes all allowed flows, we can guarantee that the application only contains the information flows that are specified in the flow policy. A good practice is to check if there are flows in the policy file that should not be allowed. In step 4. we discussed that an information-flow policy should be justifiable so this is just a double check. In [Ernst et al., 2014] the authors propose an app-store where all applications are validated. In this situation it is the store's administrator's task to validate that the policies drafted by the app-developers are indeed justified.

   Another approach, yet not specifically discussed in the SPARTA manual, can be the following: Consider an application where we would like to verify that we only allow certain information flows. The first step of this alternative approach would be to write down policies that should be allowed. For instance, `SMS -> INTERNET`. The next step would be to run the SPARTA tool to check if the application will be accepted given the predetermined policies. When there are errors encountered we can choose between three things. The first thing would be that the error was caused by code that had not been inferred by the Inference Framework (see Section 5.4). We should now add our own annotations. The second possibility is to alter the source-code of the application to make sure the piece of code that caused the error will be fixed. For instance, if we had a piece of code that did some logging and the policies did not allow this. This piece of code can be deleted and the error should disappear. As a third possibility we could follow SPARTA's original approach and extend the policy file.

5.3.2 My approach

The goal of SPARTA's approach is to prove that an application only contains information-flows that should be allowed. Our goal is different. Instead of wanting to prove that CardProxy is a valid application, our goal was to analyze the information flows that SPARTA found. Obviously, proving that CardProxy handles sensitive data correctly would be the ultimate goal, but unfortunately due to some restrictions (which will be discussed in Section 5.4 and Section 7) we did not have time to do this. Because SPARTA initially lists all information flows as forbidden, we can still use this to deduce critical information about the information flows of the application. So my approach was to get SPARTA working with CardProxy, detect all information flows and finally analyze these information flows **by hand** to see if there are some

[1] `http://types.cs.washington.edu/sparta/current/sparta-manual.pdf`

information-flows that should not be allowed. This means we will analyze the given information-flows by hand and will **not** draft policies. *This means that we will also check if the logging of the password will be detected by SPARTA.*

5.4 Inference framework and Stub files

The Inference framework is a tool that can be used to automatically annotate source code. The Inference framework automatically infers `@Source` and `@Sink` types for fields, method parameters and return types. For the Inference framework it is important that the code of the APIs is already known. For instance, the Inference framework relies on Android's main APIs to be annotated by developers. According to [Ernst et al., 2014] the developers only has to annotate 4% of the places annotations can be placed. The rest is inferred or defaulted.

When you run SPARTA, the tool will give an error where it says that it found two types (`@Source and @Sink`) but required another pair. For example:

```
found   : @Sink({}) @Source("ANY") String
required: @Sink("INTERNET") @Source({}) String
```

Here a defaulted String was found with zero allowed sinks ({}) and ANY possible sources. The String however requires to have as sink "internet" and source "nothing". This usually happens when an object is defaulted and used in a method which is detected to have other (required) sources and sinks. A second possible error is that a flow is found, but forbidden:

```
found: @Sink("ANY") @Source("ANY") JsonObject
forbidden flows:
    ANY -> ANY
```

In this example SPARTA detected a flow that is not yet in the policy file. Therefore SPARTA will yield this flow as an error.

So how does SPARTA know what types of `@Source and @Sink` a certain piece of code is? There are two ways how SPARTA determines the types to code. The first way are so called stub files. Because Android works with a lot of library calls, the Information flows checker needs to know the effect of that call. Stub files basically include an annotation summary for the API methods. An example taken from the SPARTA manual:

```
package android.telephony;

class TelephonyManager {
    public @Source(READ_PHONE_STATE) String getLine1Number();
    public @Source(READ_PHONE_STATE) String getDeviceId();
}
```

This example will tell SPARTA how to interpret a piece of code where these methods are being called. Annotating Stub files for all APIs is almost impossible as these also contain external APIs. Therefore SPARTA has a second way of determining the types of the code: defaulting. Because SPARTA has no information about the types the code has it will consider the code as permissive as possible by defaulting it with types like `@Sink(ANY)` and `@Source(ANY)`:

Table 5.1: Default information-flow qualifiers for unannotated types from the SPARTA manual

Location	**Default Flow Type**
@Source(α)	@Source(α) @Sink(ω), ω is the set of sinks allowed to flow from all sources in α
@Sink(ω)	@Source(α) @Sink(ω), α is the set of sources allowed to flow to all sinks in ω
Method parameters	@Source(ANY) @Sink({})
Method receivers	@Source(ANY) @Sink({})
Return types	@Source({}) @Sink(ANY)
Fields	@Source({}) @Sink(ANY)
null	@Source({}) @Sink(ANY)
Local variables	@Source(ANY) @Sink({})

5.5 Installation of SPARTA

The next section discusses the general approach for installing the SPARTA tool. We reference to other links and manuals which will provide more information and possible solutions for user-specific errors. The last step will be discussed in Section 5.6 because this was the crucial step in getting SPARTA to work in our research.

1. Install Java JDK 1.7. Installing the Java JDK 1.7 is very important because we want to analyze code for Android. Android uses Java 1.7.
2. Install Ant. We need Ant to compile the SPARTA repository.
3. Install Android SDK.
4. Install Checker-Framework. [2]
5. Install and build the SPARTA tool. [3]
6. Set up environment variables. See appendix A.1.
7. Install Gradle.
8. Adjust build.gradle. See appendix A.2 for the full file and Section 5.5 for more information.

[2] http://types.cs.washington.edu/checker-framework/current/checker-framework-manual.html#installation

[3] http://types.cs.washington.edu/sparta/current/sparta-manual.html#sec%3Ainstall

5.6 Results of SPARTA

Section 3.1.4 briefly discussed the support of different tools for Gradle. But Gradle is a big part of this research because our chosen application is built using Gradle. Gradle is a system to build applications and it no longer uses the build.xml file one would typically see in older projects. It is required that SPARTA can analyze applications built with Gradle.

In the Checker-Framework's manual there is brief section that explains how to use the framework in a Gradle environment. There is no information about Gradle in the SPARTA manual but since the SPARTA is built on top of the Checker-Framework [Ernst et al., 2014] we thought getting it to work would work the same as the Checker-Framework. This turned out to be only partially true.

Before analyzing the CardProxy application we tested some examples with both the Checker-Framework and later with SPARTA to get familiar with the tools. The examples given worked great and after a while we felt confident that we could start analyzing the CardProxy application. However, the test cases we used were all using Ant as build tool. CardProxy, built using Gradle, was more difficult to setup. The first step was to try to get CardProxy working with the Checker-Framework since there was some information about that available. However, we encountered some weird errors:

```
warning: unknown enum constant ElementType.TYPE_USE
warning: unknown enum constant ElementType.TYPE_PARAMETER
```

These element-types are needed to use the `@Sink` and `@Source` annotations.

Although the manual of the Checker-Framework is quite extended, the section about Gradle is very limited. Therefore we could not find a suitable solution to fix this error. Also, because this problem was quite specific we could not find a suitable answer on the internet. Our last step was to get in contact with the developers of the tool. We had an email conversation with S. Millstein, one of the developers of SPARTA, and she referred to a GitHub issue [4]. This GitHub issue contained a comment where an adaption of the build.gradle file was suggested. Using this adapted version of the build.gradle file we managed to get the Checker-Framework working with out Gradle project. The next step for our research was to get SPARTA working with CardProxy. Because I am not an expert in the field of Java compilers and Gradle the help of one of de developers was again needed.

[4] `https://github.com/typetools/checker-framework/issues/564#issuecomment-200927221`

With some instructions of the developer we managed to get SPARTA working with Gradle by adding the following to the build.gradle file (for the full build.gradle file, see Appendix A.2.):

```
sparta fileTree(dir: "$System.env.SPARTA_CODE", include: ['
    *.jar'])
if (typecheck) {
    compile fileTree(dir: "$System.env.SPARTA_CODE", include
        : ['*.jar'])
}
gradle.projectsEvaluated {
    tasks.withType(JavaCompile).all { JavaCompile compile ->
    compile.options.compilerArgs = [
            '-processor', 'sparta.checkers.FlowChecker',
            '-processorpath', "${configurations.sparta.
                asPath}:${configurations.checkerFramework.
                asPath}",
            '-AflowPolicy=/home/laptop/Documents/Bachelor-
                Scriptie/Irma-Application/irmaproxychecker/
                Flow-Policy/policy.flow'
    ]
}
```

This adaption basically lets Gradle execute some tasks upon building the application with custom processor arguments to run SPARTA. After adjusting the build.gradle accordingly we successfully managed to run SPARTA. The outcome was a list of all information flows given as errors. This was expected since, as discussed in Section 5.2, SPARTA will return all information flows as invalid when there are no policies specified. The next step would be to annotate the code. This is when some problems occurred. First of all, the Inference-framework (see Section 5.4) did not work with our project. This is probably because the Inference- framework does not support Gradle. However we cannot verify this because there is no extensive manual for the inference framework. The inability to use the Inference-framework had as consequence that we could not automatically annotate a lot of code (roughly 96% [Ernst et al., 2014]).

The second problem was that CardProxy uses a lot of external APIs. This means that if we were to annotate our full project we would also have to annotate the APIs in the form of stub-files (Section 5.4). Because my knowledge of the APIs was not enough to annotate them the choice was made to only use SPARTA as a tool to list all information flows instead of proving that the program does not contain specific unwanted flows. This is the reasoning behind using our approach as discussed in Section 5.3.2.

5.6.1 Running SPARTA

To run SPARTA one must first go to CardProxy's main directory and execute the following two commands in a terminal:

1. Assuming the Environment Variables (See appendix A.1.) are set using ∼/.bash_profile (`$vim ∼/.bash_profile`, copy the variables and adjust accordingly to your setup) one must first execute the `$source ∼/.bash_profile` command.

2. The command to run the tool: `$gradle compileReleaseJavaWithJavac -Ptypecheck=true`

For the full output see the `output_sparta.txt` file on GitHub [5].

5.6.2 Output SPARTA

```
/home/laptop/Documents/Bachelor-Scriptie/Irma-Application/
    irmaproxychecker/src/org/irmacard/androidcardproxy/
    ProtocolResponseSerializer.java:33: error: [method.invocation
    .invalid] call to getAPDU() not allowed on the given receiver
    .
                obj.addProperty("apdu", Hex.bytesToHexString(src
                    .getAPDU().getBytes()));

  found    : @Sink("INTERNET") @Source("NFC") ProtocolResponse
  required: @Sink("ANY") @Source("ANY") ProtocolResponse


/home/laptop/Documents/Bachelor-Scriptie/Irma-Application/
    irmaproxychecker/src/org/irmacard/androidcardproxy/
    ProtocolResponseSerializer.java:34: error: [forbidden.flow]
                return obj;
                       ^
  flow forbidden by flow-policy
  found: @Sink("ANY") @Source("NFC") JsonObject
  forbidden flows:
    NFC -> ANY
```

The output above is a typical example of some errors that are easy to understand for a developer of the application but a bit harder for an outsider. In the first error the method `getAPDU()` is recognized by SPARTA. After doing some research it turned out to be that APDU, short for Application Protocol Data Units, are the data units transferred from and to a smartcard via NFC. This method retrieves information from NFC traffic. Because the method getAPDU() is recognized by SPARTA, SPARTA labels this method as `@Source(NFC)`. The Object "obj" is detected by SPARTA to be sent over the internet in a different part of the code. For this reason the method

[5] https://github.com/LaurensBrinker/irmaproxychecker/blob/master/output_sparta.txt

is labeled `@Sink(INTERNET)`.

This is information that can be very useful for someone analyzing or even developing the application. The person analyzing the code would see the error above and notice that it concerns a sensitive piece of code. He could then choose to take a closer look at this part of the code to see if there are potential vulnerabilities.

The next error to discuss is the following:

```
/home/laptop/Documents/Bachelor-Scriptie/Irma-Application/
    irmaproxychecker/src/org/irmacard/androidcardproxy/
    MainActivity.java:622: error: [argument.type.incompatible]
    incompatible types in argument.
                Log.i(TAG, "PIN entered: " + dialogPincode);
                                             ^
  found   : @Sink({}) @Source("ANY") String
  required: @Sink("WRITE_LOGS") @Source({}) String
```

This is exactly the piece of code we detected by hand to be undesirable. SPARTA detected this information flow because there is an information flow to the WRITE_LOGS. The task for the user of SPARTA is now to analyze if the data being written to the logs is actually undesirable. In this case we would like to prevent our pin-code being written to the logs. Chapter 7 discusses the conclusions with regard to SPARTA. An important part of the conclusions will be about how well we can express information flows using SPARTA's language.

Chapter 6

Fuzzing

This chapter discusses the second analyzing technique we wanted to use on the CardProxy application. NFC and QR-codes are very interesting inputs for CardProxy because the NFC-traffic contains the sensitive data and the QR-code starts the verification protocol. This is the reason why we focused on these parts for fuzzing. However, because the focus of this research was on SPARTA, the fuzzing part had received a lower priority. We did experiment with some fuzzing and analysis-tools, but due to some restrictions we did not get any proper results. This chapter will therefore briefly discuss the limitations we faced while trying fuzzing.

First of all we tried to provide some wrong input ourselves via manual fuzzing. Our primary focus was the QR-code that CardProxy scans verify attributes:

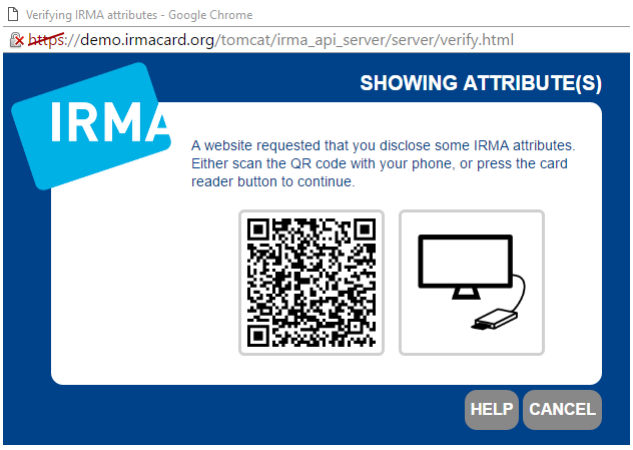

Figure 6.1: IRMA QR-example

We chose to begin with QR-codes because these are very tangible. First, a regular QR-scanner application was used to determine the URLS that were used by the IRMA demo's. One of the URLs we found was: `https://demo.irmacard.org/tomcat/irma_api_server/api/v2/verification/rfFAK2vYRSjkeYxVHCv\YRN7kmEChlkxOWX1UHAKQa50`. We first created our own QR-code that would contain a malicious URL (like redirecting to "www.bad.com"). (Un)fortunately this was detected by IRMA.

Another thing was trying to use expired QR-codes. We simply made some QR-codes that would redirect to an expired url. IRMA uses tokens to identify QR codes and after expiration these tokens will become invalid. CardProxy has incorporated checks to prevent the application from accepting malicious QR-codes. One thing that was not desired was that CardProxy will crash when an invalid URL or expired token would be provided. Generally one would like to avoid program crashes because it can contain critical information about the program, the host's system or some memory dumps that can be exploited. There was no case of a possible DoS (Denial of Service) attack because the QR-code only crashed the application and it had no impact on the server. Apart from this there were no critical flaws found by hand.

The next step was to try some tools that automated fuzzing. The tools that were discussed in Section 3.1 were not really suitable for this research. MFFA and IntentFuzzer are mainly used for detecting vulnerabilities in the Android OS and Radamsa is primarily focused on programs for Linux and Windows. The next step was to check the "ordinary" dynamic program analysis tools. Android-Hooker caught our attention. Android-Hooker's developers provided an example application. The application was a simple BlackJack game. After using the tool to analyze the application, it was found that the app calls API methods like `getDeviceid()`, `getSimState()`, `getNetworkOperator()`, `getLine1Number()`, etc [1]. These API calls are obviously not necessary for the game and can contain some private information. Although this analysis is a pure blackbox (we only see what goes in and what goes out in terms of API calls), it does give a good idea what information is obtained and how information enters and leaves the application.

Then we tried to get CardProxy running with Android-Hooker. One of the requirements for Android-Hooker is having the Substrate Framework installed on the device used for analysis. This framework requires your device to be rooted (Android equivalent of Jailbreaking).

[1] `https://developer.android.com/reference/android/telephony/TelephonyManager.html`

After doing this and getting CardProxy ready to analyze, we found that Android-Hooker does not yet support external interactions. This includes NFC. In [Mulliner, 2012] the author suggests emulating NFC traffic to be able to perform fuzzing. However, emulating NFC traffic is quite complicated because it a form of dynamic binary instrumentation (DBI). Android DBI (see Section 3.1) is a method of analyzing an application on binary level. This happens by injecting code into the normal instruction stream (execution path). Due to the advanced knowledge required to use DBI this technique was pushed to our Future Work.

Chapter 7

Conclusions

This chapter discusses the conclusions we made with respect to our research. Section 7.1 will briefly discuss information flows in general. Then, Section 7.2 discusses our conclusions with respect to the SPARTA tool. As third, Section 7.3 will be a short conclusion about the fuzzing part of our research. Finally, Section 7.4 will be a final conclusion and a brief reflection of the research performed in this paper.

7.1 Information flows in general

In terms of security flaws (See chapter 2.1) malicious information flows can be considered as one of the most dangerous. One of the main reasons is that these flaws can be unbeknownst to the user of the application. The idea to analyze information flows via static program analysis is really great. We also concluded that it is hard to formulate information flows that we want and do not want to allow. For example, it is easy to say that a password should not end up in the log files. But consider a weather application. To provide accurate weather information this application could use your GPS location. So we would like to allow the flow `GPS-data -> Internet`. However, if this application also supports advertisements, one would like to prevent these ads to obtain your GPS data. For this reason, it is really hard to determine concrete information flows because there are a lot of parameters to consider (although for this specific case SPARTA offers a solution, see the next section). Nevertheless, information flows can give still give a good indication about the general behavior of an application. Analyzing flows can also give great insight into which flows we would like to block or allow. After detecting possible information flows it should be a bit easier to understand which flows can pose a security threat like ones that can cause information leakage. Section 7.3 will discuss the usage of information flows in SPARTA.

7.2 SPARTA

7.2.1 Expressivity

The first important question to ask if SPARTA's language (sources and sinks) is expressive enough. Is the existing list of sources and sinks adequate enough for apps like CardProxy? Independent of the approach that was followed in our research we came to the following conclusion: Not entirely. Although with the sources and sinks SPARTA has provided you can cover a lot (the sources and sinks in appendix A.3. cover all modules, like SMS and GPS, that an Android application can interact with), there is still an aspect which cannot be done. At least not without extensive knowledge of the tool. Consider the following example app which can be seen as an abstract version of CardProxy:

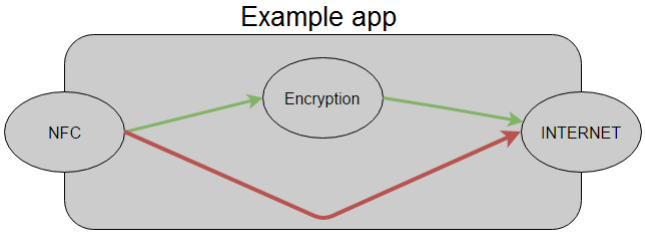

Figure 7.1: Example application simplified

This application has two ways of sending NFC data to the internet. The first way is via a method that first encrypts the data (desired), the second one is via a route that has no encryption (undesired). As discussed in Section 5.2 the information flows that we would like to allow depend on whether or not it has passed through an "encryption module" within the application. In the SPARTA manual there is a brief section dedicated to using parameters in the flow policy file. For example the following policy is mentioned:
`FILESYSTEM("/home/user") -> INTERNET("mydomain.org").`
This way one can specify that the application should only allow data taken from the directory "/home/user" to flow to the webpage "mydomain.org" on the internet. This construction can work for the example we gave in Section 7 about our GPS data flowing to the internet. Using this construction we could write the policy:
`ACCESS_FINE_LOCATION -> INTERNET("weather.com")`
to only allow the user's GPS data flowing to the weather.com website. We tried to use the same construction for our NFC data (figure 7.1). But it did not work. This is because the parameters are only applicable in the flow policy file. SPARTA can recognize that data has been taken from a certain directory and flows to a certain webpage, but it can not detect that data has flowed through an internal encryption module. It requires the analyzer

to annotate in the code what parameters will be attached to certain Sinks and Sources. To our knowledge this does not work (yet).

The same applies to our USER_INPUT problem where the pin-code was written to WRITE_LOGS. SPARTA can not automatically determine if some input is a pin-code or some information that is not dangerous to log.

There is one workaround that could be used for both issues, although it is not optimal. The workaround would be to suppress specific warnings with proper justification. This way we could prohibit the flow `USER_INPUT -> WRITE_LOGS` by not adding this flow to the policy file, but if need to use this flow we can suppress warnings for individual errors (by using `@SuppressWarnings`, see Checker-Framework manual).

7.2.2 Gradle support

As discussed in Section 5.6, SPARTA's Gradle support is very limited. Using SPARTA to analyze a Gradle project is really complicated right now and external tools like the Inference-framework do not work properly in the Gradle environment. As discussed in Chapter 5.4, the Inference-framework combined with defaulting generally accounts for 96% of the annotations in a program. Luckily, during the e-mail conversation with one of the developers she mentioned that proper Gradle support is coming in the future. And although the difference between using SPARTA on an Ant-built application and one built by Gradle is really important, it is unfortunately not clearly mentioned in SPARTA's manual.

7.2.3 Final conclusion about using SPARTA to analyze CardProxy

In [Ernst et al., 2014] the authors say that getting SPARTA to work and getting used to SPARTA's policies is the hardest part and consumes most time. This can be confirmed. Especially considering the fact that the tool was completely new for me and the fact that getting SPARTA to work with Gradle was even more complicated. However, although this research did not use SPARTA like it was intended by the developers (we used SPARTA to detect information flows not to prove certain information flow properties), SPARTA can still be very useful in analyzing privacy sensitive applications. Even though the pin-code to logs "error" was detected by hand, there is also a bit of luck involved. For example if some code is spread over muliple source files it can be hard to find a vulnerability. SPARTA can provide an automated systematic way to detect these kinds of problems, even the ones that can appear as innocent on first sight.

This is because SPARTA for example traces the Source back to the origin and will then show the analyzer where data can come from (and flow to). I am also certain SPARTA will be able to find more complicated errors when everything is working correctly.

As for the problem with the "encryption-module" or detecting different types of USER_INPUT I suggest the following (if not already possible): Use the Checker-Framework and SPARTA's parameter principle (Section 7.2.1) to allow a user of SPARTA to create "custom" Sources and Sinks. For example, assume we have a method called `nfcToEncrypted(){...}` which takes NFC data and encrypts it. I propose the following conceptual technique: Use the Checker-Framework to annotate the method as the custom type "NFC_ENCRYPTED". So we would get `@Source(NFC) @NFC_ENCRYPTED nfcToEncrypted(){...}`. We could now specify in our policy file that we do allow the flow `NFC_ENCRYPTED -> INTERNET`. This way we can enforce that the only NFC data that will be send over the internet will come frome the encryption module. I think this could provide users that do not know all the details of SPARTA a way to write and use "their own" Sources and Sinks. Maybe it is not necessary to introduce custom sources and sinks by just adding an extra declassification method. Consider an encryption module that can enforce that data is classified as `@Source(NFC) @Sink(ANY)`. By doing this we can make some data only available after it has been encrypted (declassify). After the data has been encrypted we can allow it to flow to anything (hence `@Sink(ANY)`).

To summarize: SPARTA's principle is really great and after SPARTA will be more compatible with Gradle it could be a great tool not only detect information flows, but also prove that an application does only contain the flows that are allowed by the policy flow file.

7.3 Fuzzing

Unfortunately there was not enough time left to extensively explore the field of fuzzing. Therefore, we cannot make a concrete conclusion about the technique. Chapter 6 discussed two ways we wanted to do fuzzing. The first was by hand in terms of trying out unexpected QR-codes. The only result we got was the application crashing. For CardProxy this is only a DoS for your own device, and it does not affect the server. If it did, it would be a serious problem because crashing a server by entering some malformed QR-codes would be a big problem.

Although the fuzzing tools that were discussed in Section 3.1.3 were not suitable for CardProxy (see Chapter 6), we tried to use Android-Hooker. Android-Hooker is a dynamic program analysis tool that can be used to automate input and observe the behavior of an application. As discussed in Chapter 6, the tool worked pretty good for the example application (the blackjack app) the developers gave. But due to the tool not being able to handle NFC traffic CardProxy could not be analyzed.

The main purpose of the tool however is to observe the behavior of the application, but we wanted to use the tool in such a manner that we could also automate some input. Due to the NFC-problem we faced with this tool we moved the usage of Android-Hooker to our future work.

7.4 Final conclusion and small reflection

In our introduction chapter the following research question was stated: "Using SPARTA and fuzzing, is it possible to analyze security-sensitive applications like CardProxy as a non-expert in these techniques?" Section 7.2.3 and 7.3 concluded that the principle of the tools / techniques are useful for applications like CardProxy, but in practice they were not yet sufficient for someone without extensive knowledge of SPARTA because of the lack of support for Gradle. For Ant we found that SPARTA worked good for the examples given by the developers.

Although SPARTA was used in a way that was still useful for our research. We would have liked to use SPARTA with CardProxy to prove that it contains only the flows we allow instead of just analyzing the flows (see Section 5.3 for the difference between these two approaches). If this research were to be repeated the biggest change would be seeking help from the developers earlier. Too much time was spent trying to get the tool working ourselves and searching for the answer. In SPARTA's case, we initially thought that the problem was our own setup and that the manual would be sufficient to get the tool working. However, after getting help from the local department in Nijmegen and after contacting the developers in Washington it turned out that the answer to the problem was something we would have probably never found by ourselves.

Assuming that we did not run into any problems running SPARTA an interesting question would be what our policy file would look like. Although we cannot give a specific policy file, we guess that the file will not be large. This is because the CardProxy does not offer more features apart from scanning the QR-code and authenticating (providing the attributes).

Therefore we predict that the ideal main sources and sinks used in our policies would be NFC, INTERNET, USER_INPUT and CAMERA. I will definitely keep an eye on SPARTA because I have the feeling I only grasped the surface of the tool and that when the tool is fully working I could use its full potential.

For the fuzzing part of our research question we would have liked more time to explore it some more. As Chapter 7.3 discussed, NFC is essential for the CardProxy application and because we could not emulate our NFC traffic yet we did not have enough results to make a substantiated conclusion about fuzzing the CardProxy app.

Acknowledgement

Special thanks to S. Millsteiner for helping me out with the SPARTA tool.

Chapter 8

Future work

This chapter will discuss some points that would be interesting to perform in future work. These points will be actions we would have liked to perform but due to the time were not able to and possible improvements and alternatives for our research.

1. Creating a better integration of SPARTA and the Inference-Framework with Gradle. At this moment the Gradle support for these tools is not sufficient. In order for applications like CardProxy that use Gradle to make use of SPARTA a better integration is necessary.

2. When doing the Checker-Framework tutorial we also used the Eclipse plugin. This turned out to be very easy to use. A future work could be to integrate the Checker-Framework and SPARTA into AndroidStudio as AndroidStudio is the current standard IDE and support for Eclipse will be stopped [1].

3. The first option would be to prove that CardProxy does indeed have certain allowed information flows. As discussed in our conclusion due to problems getting SPARTA working with CardProxy we did not have time to fully annotate the code. But another factor would be to get the Inference-Framework to work, because annotating everything by hand will cost a lot of time.

4. It would also be interesting to check the CardEmu application because it is more recent than CardProxy and will actually be the standard application for the IRMA project in the future. CardEmu will also be more of a challenge because instead of getting the credentials from a smart-card they need to be taken from local storage. This will add more potential risky information flows. The only problem is that CardEmu also uses Gradle and NFC, so the problems we had with CardProxy will also occur when analyzing CardEmu.

[1] `https://developer.android.com/studio/index.html`

5. Focus more on dynamic program analysis. This research was primarily focused on SPARTA. A future work could be to explore dynamic program analysis a bit more together with automated fuzzing. Because NFC and internet are the most important input and output for CardProxy we would focus our analysis on those two information flows. Considering the solution that we can simulate NFC traffic in order to analyze the application using Android-Hooker (see Chapter 6 and [Mulliner, 2012]).

Bibliography

[Alpár and Jacobs, 2013] Alpár, G. and Jacobs, B. (2013). Credential design in attribute-based identity management. In *Bridging distances in technology and regulation, 3rd TILTing Perspectives Conference*, pages 189–204.

[Arzt et al., 2014] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices*, 49(6):259–269.

[Burket et al., 2015] Burket, J., Flynn, L., Klieber, W., Lim, J., and Snavely, W. (2015). Making didfail succeed: Enhancing the CERT static taint analyzer for Android app sets.

[Enck et al., 2014] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2014). Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5.

[Ernst et al., 2014] Ernst, M. D., Just, R., Millstein, S., Dietl, W., Pernsteiner, S., Roesner, F., Koscher, K., Barros, P. B., Bhoraskar, R., Han, S., et al. (2014). Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1092–1104. ACM.

[Fedler et al., 2013] Fedler, R., Schütte, J., and Kulicke, M. (2013). On the effectiveness of malware protection on android. *Fraunhofer AISEC, Berlin, Tech. Rep.*

[Garg, 2012] Garg, P. (2012). Fuzzing: Mutation vs. generation. `http://resources.infosecinstitute.com/fuzzing-mutation-vs-generation/`.

[Mulliner, 2012] Mulliner, C. (2012). Powerpoint: Dynamic binary instrumentation on android. Lecture slides Systems Security Labs (Northeastern University).

[Poll, 2015] Poll, E. (2015). Powerpoint: Information flow for Android apps. Lecture slides Software Security (Master course at Radboud University).

[Sasnauskas and Regehr, 2014] Sasnauskas, R. and Regehr, J. (2014). Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, pages 1–5. ACM.

[Sutton et al., 2007] Sutton, M., Greene, A., and Amini, P. (2007). *Fuzzing: brute force vulnerability discovery*. Pearson Education.

[Wei et al., 2014] Wei, F., Roy, S., Ou, X., et al. (2014). Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM.

[Wong and Lie, 2016] Wong, M. Y. and Lie, D. (2016). Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*.

Appendix A

Appendix

A.1 Environment Variables

Environment Variables are necessary to run SPARTA. The following variables were used in this research. The Environment Variables were stored in ∼/.bash_profile. If you are using the same file, remember to do `$source ∼/.bash_profile` every time you boot your laptop/PC.

```
export CHECKERFRAMEWORK=/home/laptop/Documents/Bachelor-Scriptie
    /sparta-sparta-1.0.2/checker-framework-1.9.11
export PATH=${CHECKERFRAMEWORK}/checker/bin:$PATH
export JAVA_HOME=/usr/lib/jvm/default-java
export PATH=$JAVA_HOME/bin:$PATH
export ANDROID_HOME=/home/laptop/Documents/Bachelor-Scriptie/
    android-sdk-linux
export PATH=$ANDROID_HOME/platform-tools:$PATH
export PATH=$ANDROID_HOME/tools:$PATH
export ANDROID_SDK=/home/laptop/Documents/Bachelor-Scriptie/
    android-sdk-linux
export SPARTA_CODE=/home/laptop/Documents/Bachelor-Scriptie/
    sparta-sparta-1.0.2
export CHECKER_INFERENCE=/home/laptop/Documents/Bachelor-
    Scriptie/checker-framework-inference
```

A.2 build.gradle file

This build.gradle file is necessary to run the SPARTA information-flow checker with the CardProxy application. `https://github.com/LaurensBrinker/irmaproxychecker/blob/master/build.gradle`

```
apply plugin: 'com.android.application'
apply plugin: 'maven'

version="0.8"
group="org.irmacard"
```

```
ext.targetJavaVersion = JavaVersion.current().isJava7() ?
    JavaVersion.VERSION_1_7 : JavaVersion.VERSION_1_8


buildscript {
    System.properties['com.android.build.gradle.
        overrideVersionCheck'] = 'true'
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.5.0'
    }
}

repositories {
    mavenLocal()
    maven {
        url "https://credentials.github.io/repos/maven2/"
    }

    // Use this to get minimal android library
    maven {
        url "https://raw.github.com/embarkmobile/zxing-android-
            minimal/mvn-repo/maven-repository/"
    }
    mavenCentral()
}

android {
    compileSdkVersion 19
    buildToolsVersion "19.1"

    sourceSets {
        main {
            manifest.srcFile 'AndroidManifest.xml'
            java.srcDirs = ['src']
            resources.srcDirs = ['src']
            res.srcDirs = ['res']
            assets.srcDirs = ['assets']
        }
    }

    lintOptions {
        // BCProv references javax.naming
        // CredentialsAPI references java.awt and java.swing
        disable 'InvalidPackage'
    }
}

configurations {
    if (targetJavaVersion.isJava7()) {
        checkerFrameworkJavac {
```

```
            description = 'a customization of the Open JDK javac
                compiler with additional support for type
                annotations'
        }
    }
    checkerFrameworkAnnotatedJDK {
        description = 'a copy of JDK classes with Checker
            Framework type qualifers inserted'
    }
    checkerFramework {
        description = 'The Checker Framework: custom pluggable
            types for Java'
    }
    sparta
}

dependencies {
    ext.checkerFrameworkVersion = '1.9.11'
    ext.jdkVersion = JavaVersion.current().isJava7() ? 'jdk7' :
        'jdk8'
    checkerFrameworkAnnotatedJDK "org.checkerframework:${
        jdkVersion}:${checkerFrameworkVersion}"
    checkerFrameworkJavac "org.checkerframework:compiler:${
        checkerFrameworkVersion}"
    checkerFramework "org.checkerframework:checker:${
        checkerFrameworkVersion}"
    compile "org.checkerframework:checker-qual:${
        checkerFrameworkVersion}"

    sparta fileTree(dir: "$System.env.SPARTA_CODE", include: ['
        *.jar'])
    if (typecheck) {
        compile fileTree(dir: "$System.env.SPARTA_CODE", include
            : ['*.jar'])
    }
    compile 'com.loopj.android:android-async-http:1.4.3'
    compile 'com.android.support:support-v4:19.1.0'
    compile 'com.google.code.gson:gson:2.2.2'

    // zxing QR code libraries
    compile 'com.embarkmobile:zxing-android-minimal:2.0.0@aar'
    compile 'com.embarkmobile:zxing-android-integration:2.0.0
        @aar'
    compile 'com.google.zxing:core:3.0.1'

    compile "org.irmacard.android:irma_android_library:0.9.1"
    compile 'net.sf.scuba:scuba_sc_android:0.0.7-irma'
    compile 'net.sf.scuba:scuba_smartcards:0.0.7-irma'

    ext.checkerFrameworkVersion = '1.9.13'
    ext.jdkVersion = JavaVersion.current().isJava7() ? 'jdk7' :
        'jdk8'
    checkerFrameworkAnnotatedJDK "org.checkerframework:${
        jdkVersion}:${checkerFrameworkVersion}"
```

```
}


def typecheck = project.properties['typecheck'] ?: false
allprojects {
if (typecheck) {
    gradle.projectsEvaluated {
        tasks.withType(JavaCompile).all { JavaCompile compile ->
        compile.options.compilerArgs = [
                '-processor', 'sparta.checkers.FlowChecker',
                '-processorpath', "${configurations.sparta.
                    asPath}:${configurations.checkerFramework.
                    asPath}",
                '-AflowPolicy=/home/laptop/Documents/Bachelor-
                    Scriptie/Irma-Application/irmaproxychecker/
                    Flow-Policy/policy.flow'
                // uncomment to turn Checker Framework errors
                    into warnings
                //'-Awarns',
                //"-AprintErrorStack"
        ]
            compile.options.compilerArgs += ['-source', '7', '-
                target', '7']
            options.bootClasspath = "${configurations.
                checkerFrameworkJavac.asPath}:" + System.
                getProperty("sun.boot.class.path") + ":" +
                options.bootClasspath


            options.fork = true
            options.forkOptions.jvmArgs += ["-Xbootclasspath/p:$
                {configurations.checkerFrameworkJavac.asPath}"]
        }
    }
    }
}

task wrapper(type: Wrapper) {
    gradleVersion = '2.2'
}
```

A.3 List of SPARTA's sources and sinks

The Sinks and Sources as taken from the FlowPermission.java file (Dir = /sparta/src/sparta/checkers/quals)

```
/**
 * This special constant is shorthand for all sources, that is,
     the data can
 * come from any possible source. Using this constant is
     preferred to
 * listing all constants, because it's future safe.
 */
ANY(T.BOTH),

/**
 * The following are special permissions added by SPARTA Make
     sure that
 * whatever permission you add is not the same as any permission
      already
 * added.
 */

CAMERA_SETTINGS(T.BOTH), DISPLAY(T.SINK), FILESYSTEM(T.BOTH),
    RANDOM(T.SOURCE), READ_TIME(
        T.SOURCE), // WRITE_TIME is an Android Permission, but
            read time
                   // isn't
USER_INPUT(T.SOURCE),
WRITE_LOGS(T.SINK), // READ_LOGS is an Android Permission, but
    there is no
                    // WRITE_LOGS
DATABASE(T.BOTH), // This is an Android database that could be
    any of the
                  // Content database.
SYSTEM_PROPERTIES(T.BOTH), // This is for java.lang.System
MEDIA(T.SOURCE),
READ_EMAIL(T.SOURCE),
WRITE_EMAIL(T.SINK),
WRITE_CLIPBOARD(T.SINK),
READ_CLIPBOARD(T.SOURCE),
SPEAKER(T.SINK), // Physical speaker / headphones
SENSOR(T.SOURCE), //See android.hardware.Sensor
PACKAGE_INFO(T.BOTH), //For data from/to android.content.pm.
    PackageManager

/**
 * These are old sources or sinks that may or may not be of use
 */
PHONE_NUMBER(T.SOURCE),
SHARED_PREFERENCES(T.BOTH),
ACCELEROMETER(T.SOURCE),

/**
```

```
 * The following permissions are temporary and implemented now
     in a simple
 * way for an upcoming engagement.
 */

REFLECTION(T.BOTH), // The caller of the invoke method should
    have this
                    // permission.
INTENT(T.BOTH),
BUNDLE(T.SOURCE),
PROCESS_BUILDER(T.BOTH), // The ProcessBuilder variable should
    have this
                         // permission.
PARCEL(T.BOTH),
SECURE_HASH(T.BOTH), // Use only for one way hashes (MD5 for
    example)
CONTENT_PROVIDER(T.BOTH),

// Allows read/write access to the "properties" table in the
    checkin
// database, to change values that get uploaded.
ACCESS_CHECKIN_PROPERTIES(T.BOTH),
// Allows an app to access approximate location derived from
    network
// location sources such as cell towers and Wi-Fi.
ACCESS_COARSE_LOCATION(T.SOURCE),
// Allows an app to access precise location from location
    sources such as
// GPS, cell towers, and Wi-Fi.
ACCESS_FINE_LOCATION(T.SOURCE),
// Allows an application to access extra location provider
    commands
ACCESS_LOCATION_EXTRA_COMMANDS(T.SOURCE),
// Allows an application to create mock location providers for
    testing
ACCESS_MOCK_LOCATION(T.SOURCE),
// Allows applications to access information about networks
ACCESS_NETWORK_STATE(T.SOURCE),
// Allows an application to use SurfaceFlinger's low level
    features
ACCESS_SURFACE_FLINGER(T.BOTH),
// Allows applications to access information about Wi-Fi
    networks
ACCESS_WIFI_STATE(T.SOURCE),
// Allows applications to call into AccountAuthenticators.
ACCOUNT_MANAGER(T.SOURCE),
// Allows an application to add voicemails into the system.
ADD_VOICEMAIL(T.SINK),
// Allows an application to act as an AccountAuthenticator for
    the
// AccountManager
AUTHENTICATE_ACCOUNTS(T.BOTH),
// Allows an application to collect battery statistics
BATTERY_STATS(T.SOURCE),
```

```
// Must be required by an AccessibilityService, to ensure that
    only the
// system can bind to it.
BIND_ACCESSIBILITY_SERVICE(T.BOTH),
// Allows an application to tell the AppWidget service which
    application can
// access AppWidget's data.
BIND_APPWIDGET(T.BOTH),
// Must be required by device administration receiver, to ensure
     that only
// the system can interact with it.
BIND_DEVICE_ADMIN(T.BOTH),
// Must be required by an InputMethodService, to ensure that
    only the system
// can bind to it.
BIND_INPUT_METHOD(T.BOTH),
// Must be required by a RemoteViewsService, to ensure that only
     the system
// can bind to it.
BIND_REMOTEVIEWS(T.BOTH),
// Must be required by a TextService (e.g.
BIND_TEXT_SERVICE(T.BOTH),
// Must be required by an VpnService, to ensure that only the
    system can
// bind to it.
BIND_VPN_SERVICE(T.BOTH),
// Must be required by a WallpaperService, to ensure that only
    the system
// can bind to it.
BIND_WALLPAPER(T.BOTH),
// Allows applications to connect to paired bluetooth devices
BLUETOOTH(T.BOTH),
// Allows applications to discover and pair bluetooth devices
BLUETOOTH_ADMIN(T.BOTH),
// Required to be able to disable the device (very dangerous!).
BRICK(T.SINK),
// Allows an application to broadcast a notification that an
    application
// package has been removed.
BROADCAST_PACKAGE_REMOVED(T.SINK),
// Allows an application to broadcast an SMS receipt
    notification
BROADCAST_SMS(T.SINK),
// Allows an application to broadcast sticky intents.
BROADCAST_STICKY(T.SINK),
// Allows an application to broadcast a WAP PUSH receipt
    notification
BROADCAST_WAP_PUSH(T.SINK),
// Allows an application to initiate a phone call without going
    through the
// Dialer user interface for the user to confirm the call being
    placed.
CALL_PHONE(T.SINK),
// Allows an application to call any phone number, including
```

```
    emergency
// numbers, without going through the Dialer user interface for
    the user to
// confirm the call being placed.
CALL_PRIVILEGED(T.SINK),
// Required to be able to access the camera device.
CAMERA(T.SINK),
// Allows an application to change whether an application
    component (other
// than its own) is enabled or not.
CHANGE_COMPONENT_ENABLED_STATE(T.SINK),
// Allows an application to modify the current configuration,
    such as
// locale.
CHANGE_CONFIGURATION(T.SINK),
// Allows applications to change network connectivity state
CHANGE_NETWORK_STATE(T.SINK),
// Allows applications to enter Wi-Fi Multicast mode
CHANGE_WIFI_MULTICAST_STATE(T.SINK),
// Allows applications to change Wi-Fi connectivity state
CHANGE_WIFI_STATE(T.SINK),
// Allows an application to clear the caches of all installed
    applications
// on the device.
CLEAR_APP_CACHE(T.SINK),
// Allows an application to clear user data
CLEAR_APP_USER_DATA(T.SINK),
// Allows enabling/disabling location update notifications from
    the radio.
CONTROL_LOCATION_UPDATES(T.SINK),
// Allows an application to delete cache files.
DELETE_CACHE_FILES(T.SINK),
// Allows an application to delete packages.
DELETE_PACKAGES(T.SINK),
// Allows low-level access to power management
DEVICE_POWER(T.SOURCE),
// Allows applications to RW to diagnostic resources.
DIAGNOSTIC(T.BOTH),
// Allows applications to disable the keyguard
DISABLE_KEYGUARD(T.SINK),
// Allows an application to retrieve state dump information from
    system
// services.
DUMP(T.SOURCE),
// Allows an application to expand or collapse the status bar.
EXPAND_STATUS_BAR(T.SINK),
// Run as a manufacturer test application, running as the root
    user.
FACTORY_TEST(T.NONE),
// Allows access to the flashlight
FLASHLIGHT(T.SINK),
// Allows an application to force a BACK operation on whatever
    is the top
// activity.
```

```
FORCE_BACK(T.SINK),
// Allows access to the list of accounts in the Accounts Service
GET_ACCOUNTS(T.SOURCE),
// Allows an application to find out the space used by any
    package.
GET_PACKAGE_SIZE(T.SOURCE),
// Allows an application to get information about the currently
    or recently
// running tasks.
GET_TASKS(T.SOURCE),
// This permission can be used on content providers to allow the
     global
// search system to access their data.
GLOBAL_SEARCH(T.BOTH),
// Allows access to hardware peripherals.
HARDWARE_TEST(T.BOTH),
// Allows an application to inject user events (keys, touch,
    trackball) into
// the event stream and deliver them to ANY window.
INJECT_EVENTS(T.SINK),
// Allows an application to install a location provider into the
     Location
// Manager
INSTALL_LOCATION_PROVIDER(T.SINK),
// Allows an application to install packages.
INSTALL_PACKAGES(T.SINK),
// Allows an application to open windows that are for use by
    parts of the
// system user interface.
INTERNAL_SYSTEM_WINDOW(T.SINK),
// Allows applications to open network sockets.
INTERNET(T.BOTH),
// Allows an application to call killBackgroundProcesses(String)
    .
KILL_BACKGROUND_PROCESSES(T.SINK),
// Allows an application to manage the list of accounts in the
// AccountManager
MANAGE_ACCOUNTS(T.BOTH),
// Allows an application to manage (create, destroy, Z-order)
    application
// tokens in the window manager.
MANAGE_APP_TOKENS(T.SINK),
//
MASTER_CLEAR(T.NONE),
// Allows an application to modify global audio settings
MODIFY_AUDIO_SETTINGS(T.SINK),
// Allows modification of the telephony state - power on, mmi,
    etc.
MODIFY_PHONE_STATE(T.SINK),
// Allows formatting file systems for removable storage.
MOUNT_FORMAT_FILESYSTEMS(T.BOTH),
// Allows mounting and unmounting file systems for removable
    storage.
MOUNT_UNMOUNT_FILESYSTEMS(T.BOTH),
```

```
// Allows applications to perform I/O operations over NFC
NFC(T.BOTH),
// This constant was deprecated in API level 9. This
    functionality will be
// removed in the future; please do not use.
// Allow an application to make its activities persistent.
PERSISTENT_ACTIVITY(T.SINK),
// Allows an application to monitor, modify, or abort outgoing
    calls.
PROCESS_OUTGOING_CALLS(T.BOTH),
// Allows an application to read the user's calendar data.
READ_CALENDAR(T.SOURCE),
// Allows an application to read the user's call log.
READ_CALL_LOG(T.SOURCE),
// Allows an application to read the user's contacts data.
READ_CONTACTS(T.SOURCE),
// Allows an application to read from external storage.
READ_EXTERNAL_STORAGE(T.SOURCE),
// Allows an application to take screen shots and more generally
     get access
// to the frame buffer data
READ_FRAME_BUFFER(T.SOURCE),
// Allows an application to read (but not write) the user's
    browsing history
// and bookmarks.
READ_HISTORY_BOOKMARKS(T.SOURCE),
// This constant was deprecated in API level 16. The API that
    used this
// permission has been removed.
READ_INPUT_STATE(T.SOURCE),
// Allows an application to read the low-level system log files.
READ_LOGS(T.SOURCE),
// Allows read only access to phone state.
READ_PHONE_STATE(T.SOURCE),
// Allows an application to read the user's personal profile
    data.
READ_PROFILE(T.SOURCE),
// Allows an application to read SMS messages.
READ_SMS(T.SOURCE),
// Allows an application to read from the user's social stream.
READ_SOCIAL_STREAM(T.SOURCE),
// Allows applications to read the sync settings
READ_SYNC_SETTINGS(T.SOURCE),
// Allows applications to read the sync stats
READ_SYNC_STATS(T.SOURCE),
// Allows an application to read the user dictionary.
READ_USER_DICTIONARY(T.SOURCE),
// Required to be able to reboot the device.
REBOOT(T.SINK),
// Allows an application to receive the ACTION_BOOT_COMPLETED
    that is
// broadcast after the system finishes booting.
RECEIVE_BOOT_COMPLETED(T.SOURCE),
// Allows an application to monitor incoming MMS messages, to
```

```
    record or
// perform processing on them.
RECEIVE_MMS(T.SOURCE),
// Allows an application to monitor incoming SMS messages, to
    record or
// perform processing on them.
RECEIVE_SMS(T.SOURCE),
// Allows an application to monitor incoming WAP push messages.
RECEIVE_WAP_PUSH(T.SOURCE),
// Allows an application to record audio
RECORD_AUDIO(T.BOTH),
// Allows an application to change the Z-order of tasks
REORDER_TASKS(T.SINK),
// This constant was deprecated in API level 8. The
    restartPackage(String)
// API is no longer supported.
RESTART_PACKAGES(T.SINK),
// Allows an application to send SMS messages.
SEND_SMS(T.SINK),
// Allows an application to watch and control how activities are
     started
// globally in the system.
SET_ACTIVITY_WATCHER(T.SINK),
// Allows an application to broadcast an Intent to set an alarm
    for the
// user.
SET_ALARM(T.SINK),
// Allows an application to control whether activities are
    immediately
// finished when put in the background.
SET_ALWAYS_FINISH(T.SINK),
// Modify the global animation scaling factor.
SET_ANIMATION_SCALE(T.SINK),
// Configure an application for debugging.
SET_DEBUG_APP(T.SINK),
// Allows low-level access to setting the orientation (actually
    rotation) of
// the screen.
SET_ORIENTATION(T.SINK),
// Allows low-level access to setting the pointer speed.
SET_POINTER_SPEED(T.SINK),
// This constant was deprecated in API level 7. No longer useful
    , see
// addPackageToPreferred(String) for details.
SET_PREFERRED_APPLICATIONS(T.SINK),
// Allows an application to set the maximum number of (not
    needed)
// application processes that can be running.
SET_PROCESS_LIMIT(T.SINK),
// Allows applications to set the system time
SET_TIME(T.SINK),
// Allows applications to set the system time zone
SET_TIME_ZONE(T.SINK),
// Allows applications to set the wallpaper
```

```
SET_WALLPAPER(T.SINK),
// Allows applications to set the wallpaper hints
SET_WALLPAPER_HINTS(T.SINK),
// Allow an application to request that a signal be sent to all
    persistent
// processes
SIGNAL_PERSISTENT_PROCESSES(T.SINK),
// Allows an application to open, close, or disable the status
    bar and its
// icons.
STATUS_BAR(T.SINK),
// Allows an application to allow access the subscribed feeds
// ContentProvider.
SUBSCRIBED_FEEDS_READ(T.SOURCE),
//
SUBSCRIBED_FEEDS_WRITE(T.SINK),
// Allows an application to open windows using the type
    TYPE_SYSTEM_ALERT,
// shown on top of all other applications.
SYSTEM_ALERT_WINDOW(T.SINK),
// Allows an application to update device statistics.
UPDATE_DEVICE_STATS(T.SINK),
// Allows an application to request authtokens from the
    AccountManager
USE_CREDENTIALS(T.SOURCE),
// Allows an application to use SIP service
USE_SIP(T.BOTH),
// Allows access to the vibrator
VIBRATE(T.SINK),
// Allows using PowerManager WakeLocks to keep processor from
    sleeping or
// screen from dimming
WAKE_LOCK(T.BOTH),
// Allows applications to write the apn settings
WRITE_APN_SETTINGS(T.SINK),
// Allows an application to write (but not read) the user's
    calendar data.
WRITE_CALENDAR(T.SINK),
// Allows an application to write (but not read) the user's
    contacts data.
WRITE_CALL_LOG(T.SINK),
// Allows an application to write (but not read) the user's
    contacts data.
WRITE_CONTACTS(T.SINK),
// Allows an application to write to external storage.
WRITE_EXTERNAL_STORAGE(T.SINK),
// Allows an application to modify the Google service map.
WRITE_GSERVICES(T.SINK),
// Allows an application to write (but not read) the user's
    browsing history
// and bookmarks.
WRITE_HISTORY_BOOKMARKS(T.SINK),
// Allows an application to write (but not read) the user's
    personal profile
```

```
// data.
WRITE_PROFILE(T.SINK),
// Allows an application to read or write the secure system
    settings.
WRITE_SECURE_SETTINGS(T.SINK),
// Allows an application to read or write the system settings.
WRITE_SETTINGS(T.SINK),
// Allows an application to write SMS messages.
WRITE_SMS(T.SINK),
// Allows an application to write (but not read) the user's
    social stream
// data.
WRITE_SOCIAL_STREAM(T.SINK),
// Allows applications to write the sync settings
WRITE_SYNC_SETTINGS(T.SINK),
WRITE_TIME(T.SINK),
// Allows an application to write to the user dictionary.
WRITE_USER_DICTIONARY(T.SINK),
```