Thomas Brüggemann

Master Thesis
im Fach Information Systems

Automated Information Privacy Risk Assessment of Android Health Applications

Themensteller: Prof. Dr. Ali Sunyaev

Vorgelegt in der Masterprüfung
im Studiengang Information Systems
der Wirtschafts- und Sozialwissenschaftlichen Fakultät
der Universität zu Köln

Köln, September 2016

Contents

Index of Abbreviations

API	Application Programming Interface
APK	Android Application Package
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IP	Internet Protocol
JAR	Java Archive
mHealth	Mobile Health
NFC	Near Field Communication
P3P	Platform for Privacy Preferences
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
XML	Extensible Markup Language

List of Tables

List of Figures

1. Introduction

1.1 Problem Statement

The market for mobile phone and tablet applications (apps) has grown extensively since recent years.[1] It has become increasingly easier for companies or even single developers to create unique apps that reach millions of users around the planet via digital app stores. This market growth affected mobile health (mHealth) apps as well. More and more mHealth apps are available that support the users in resolving their health-related issues and that try to remedy health-related information deficiencies.

But receiving personal health-related information yields information privacy risks to users. Users are asked to expose personal health-related information, e.g. information on disease symptoms or medical appointments in order to receive a tailored app that fits their needs.[2] It remains however unclear how and where the vulnerable user information is sent, processed and stored.[3]

The information about these privacy related practices of app providers and their offered apps should be stated in the privacy policy document provided by the app provider.[4] Processing these privacy policies requires a higher level of education and time to read through large bodies of text, in order to find the relevant information. Additionally, the important information is hidden in legal language or is insufficiently addressed, if at all.[5] Aside from data usage beyond the control of the users, it is also challenging to assess what kind of private information an app asks for, prior to the app usage. Users have to download the apps of interest and try them out, before it becomes clear what health-related information is processed by the app and in which way. This leads to low comparability between apps. When users are looking for specific functionality in a mHealth app, it is challenging to find the app that offers the desired functionality at an acceptable information privacy risk. Even if users would pursue the task of finding and comparing mHealth

---

1 See for this and the following sentence Enck et al. (2011), p. 1.

2 See Chen et al. (2012), p. 2.

3 See He et al. (2014), p. 652.

4 This paragraph follows Dehling, Gao, Sunyaev (2014), p. 11.

5 See Pollach (2007), p. 104.

apps of similar functionality, the high volume of apps available in the app stores[6] makes it laborious to review all of them by hand. One way to assess information privacy risks of the large amount of mHealth apps is to automate the review process of each individual app. The assessment automation can be done by downloading and analyzing the source code of each app and by tracing data leaks. Static code analysis is used in the field of informatics to analyze application source code and detect faults or vulnerabilities.[7] It is yet unclear how and to what degree the concepts of static code analysis and information privacy risk assessment can be combined in order to automate app assessment. A static code analysis could, in theory, be used to automatically assess some of the information privacy risks that mHealth apps pose. Previous research has not shown how and to what degree the combination of static code analysis and information privacy risks assessment is feasible in the field of mHealth app information privacy risk assessment and therefore the aim of this study is to explore the possibilities of static code analysis for information privacy risk assessment. This leads to the research question: How and to what degree can the information privacy risks of mHealth apps be automatically assessed? The 'degree' refers to the amount and the level of detail that information privacy risk factors can be automatically assessed.

The automated process furthermore can help to drastically reduce the effort of reviewing each individual app and can enhance the information experience users receive while looking for mHealth apps. Additionally, it exposes new possibilities for research in the information privacy risks area. The research could be conducted on providing solutions and best practices for further enhancing the information privacy risks communication of apps.

1.2 Objectives

The main objective of this study is to ascertain how and to what degree the assessment of information privacy risk factors for mHealth apps can be automated. In order to reach this objective, the following sub-objectives have to be met.

The first sub-objective is to extract information privacy risk factors from the infor-

---

6 See Enck et al. (2011), p. 1 and https://web.archive.org/web/20160602091827/http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, visited 07/13/16.

7 See Baca, Carlsson, Lundberg (2008), p. 79.

mation privacy practices that Dehling, Sunyaev (2016) identified and that are relevant for automated information privacy risk assessment. As a second sub-objective we will develop strategies to identify the information privacy risk factors within the source code of mHealth apps via static code analysis. This is necessary since it is yet unclear how and to what degree the static code analysis can help to identify information privacy risk factors of mHealth apps. Finally, we will evaluate how well the automated information privacy risk assessment tool can identify information privacy risk factors in comparison to two human reviewers. In order to fully ascertain the degree static code analysis can identify information privacy risk factors, a manual review of the results of the static code analysis is necessary.

1.3 Structure

This work is structured in five chapters and the content can be summarized as follows. The previous sections of this chapter motivated the goals of this thesis by outlining the knowledge gap of how and to what degree information privacy risks assessment of mHealth apps can be automated.

In section 2, we[8] will give an overview over the current state of the research and lay the foundations of this thesis. We will first highlight research regarding information privacy risk assessments and its current limitations in section 2.1. Following in chapter 2.2, we will outline research regarding static code analysis. We take special interest in further possibilities static code analysis offers for interdisciplinary research, between technical possibilities and business informatics. The data base for this thesis, the information privacy practices identified by Dehling, Sunyaev (2016), will be introduced in chapter 2.3.

Chapter 3 focuses on the implementation and evaluation of an automated information privacy risk assessment tool . We will explain the three phases of the implementation of the automated information privacy risk assessment tool in detail in chapter 3.1. The following chapter 3.2 explains the process of evaluation of the automated information privacy risk assessment tool performance in comparison to two human researchers.

The next chapter, chapter 4, will present the results of the implementation and eval-

[8] Although this work is done by a single author, 'we' and 'our' respectively will be used throughout this thesis.

uation of the automated information privacy risk assessment tool . Chapter 4.1 contains results of the three implementation phases and will also point out the challenges that occurred within the implementation phases. The results of the evaluation of the automated information privacy risk assessment tool are presented in chapter 4.2.

The thesis closes with the 'Discussion' chapter 5. In chapter 5.1 we will summarize the key findings of this work and will present the contributions this work offers to current research in chapter 5.2. The limitations this thesis is under will be presented in chapter 5.3. The next chapter 5.4 will give an outlook towards the topics that future research could cover in relation to this work. Finally we will conclude the thesis with a conclusion chapter 5.5.

2. Combining Source Code Analysis with Information Privacy Risk Assessment

mHealth apps have been examined in various research studies that aim at providing insights for developers as well as users into how private information is processed. Privacy issues are the most impactful user complaints while using mobile apps.[9] Especially mHealth apps deal with sensitive and vulnerable user information and therefore pose a high information privacy risk. [10] This encourages research to address information privacy risks.

2.1 Information Privacy Risk Assessment

Information privacy is defined as the circumstance that people are able to control if, how and where information and knowledge about themselves is acquired, stored and processed.[11] In addition to mere control over the users private information, society and social structure must establish an information regulation architecture that allows people to sustainably enforce their information privacy rights.[12] An information privacy risk is therefore, by implication, a threat or a circumstance that disables users to enforce control over their private information. Threats to information privacy in digital services can happen at application level (e.g. by sharing users' personal information with third-parties), as well as at communication level (e.g. by using an unencrypted data connection).[13]

Research focus has been put on the technical side of information privacy breach. It has been analyzed, to what degree the data storage in internal Android log files or on the memory card within a phone or tablet as well as data connections to the Internet pose threats to users information privacy.[14] Special focus has been put on unencrypted Hypertext Transfer Protocol (HTTP) connections that send out user information to either the app providers servers or third party services. The assessment has been carried out

---

9 See Khalid et al. (2015), p. 5.

10 See Kumar et al. (2013), p. 33.

11 See Fischer-Hübner (1998), p. 421-422.

12 See Solove (2002), p. 1115.

13 See Fischer-Hübner (1998), p. 423-427.

14 For this and the next sentence, see He et al. (2014), p. 645-646, 649.

using the network analysis tool *Wireshark*[15] that logs all HTTP traffic.[16] Technical evaluation of mobile apps even goes further into the topics of decompilation to analyze device identification or geolocation data leaks.[17] In conclusion, decompilation is a widely used assessment technique for information privacy risks and data leaks.

In informatics and software development contexts, static code analysis has been used to analyze source code[18] and provide feedback on coding styles to the users while programming[19] or "to find defects in programs"[20]. Static code analysis provides a fast way to analyze source code,[21] which makes it suitable to automate the assessment of large datasets. A further benefit of using static code analysis to retrieve information from software is that the software does not need to be executed during the analyzation process. This additionally supports the development of fast performing assessment tools that are suitable for application on large datasets of source code since there is no need to wait for the application runtime to execute the software.

Our study will use the benefits of static code analysis and apply them to the assessment of mHealth information privacy risks. It is yet unclear if static code analysis is a viable tool to analyze and identify information privacy risk factors. We will use the comprehensive privacy-risk-relevant information privacy practices that Dehling, Sunyaev (2016) identified[22] and try to implement static code analysis strategies to identify those risks automatically. This will be a vital addition to current research, since there is yet no holistic approach to apply static code analysis to information privacy risks detection that takes an ample amount of information privacy risk factors into account.

---

15 https://web.archive.org/web/20160710141347/https://www.wireshark.org/, visited 07/13/16

16 See He et al. (2014), p. 649.

17 See Mcclurg (2012), p. 1, 5., Enck et al. (2011), p. 1. and Mitchell et al. (2013), p.6-7.

18 See Haris, Haddadi, Hui (2014), p. 5.

19 See Bardas, Others (2010), p. 10.

20 Bardas, Others (2010), p. 1.

21 See Bardas, Others (2010), p. 5.

22 See Dehling, Sunyaev (2016), p. 8-17.

2.2 Static Code Analysis

TODO: NICHT NUR WAS ANDERES MIT SCA UMSETZEN, SONDERN WAS KANN ICH WEITER DAMIT MACHEN.

2.3 Relevant Information Privacy Risk Factors

TODO: GEGENREVIEW VON MANUEL

For this thesis, we will use the set of information privacy practices extracted from literature, the Platform for Privacy Preferences (P3P) guide[23] and app reviews by Dehling, Sunyaev (2016) as a source to derive information privacy risk factors from. [24] Information privacy practices are common practices of informing users about the information privacy practices of an app that app providers should follow in order to achieve higher levels of transparency. A hierarchy of information privacy practices is formed by clustering the information privacy practices by their content aspects. The top level of the hierarchy are 'Content' and 'Practice'. The 'Content' hierarchy level contains sub-hierarchy branches that express information privacy practices concerning the handling of information content, the information collection content, and meta content about information collection, information on offered privacy controls, and information on what purpose the information privacy practices was collected for. The sub-hierarchy branches of the top-level 'Practice' contain for instance information on the existence of dispute resolution practices or access rights practices of the users.

We argue that if a information privacy practices is a circumstance that the user should be informed about, an information privacy practices expresses an information privacy risks to the app user. But since not all of the enlisted information privacy practices express or imply an information privacy risks to app users, we review and extract the information privacy practices that are relevant in terms of posing and expressing a potential information privacy risk. We will further limit the information privacy practices by excluding information privacy practices that are known to be technically infeasible to detect via static code analysis. An example for such an exclusion is the information privacy practices 'InformationRetentionContent', which captures, if an app provider carries out a

---

[23] https://web.archive.org/web/20160616160213/http://www.w3.org/TR/P3P/, visited 06/16/2016

[24] For this and the following sentence, see Dehling, Sunyaev (2016), p. 1-2.

certain information retention policy or not.[25] This is a feature that is undetectable by static code analysis and beyond the scope of app source code analysis. An analysis of the app providers backend system would be necessary to ensure that the collection information is retained according to the app providers policy promise.

We include a full list of all information privacy practices in Appendix A including detailed comments on the technical limitations, if any, towards the static code analysis detection of each information privacy practices and wether they express a risk or not.

The following information privacy practices were identified as relevant to the static code analysis and further inspection within this thesis:

The complete hierarchy *CH2* 'InformationSecurityContent' can be analysed via static code analysis including the information privacy practices 'SecurityDuringProcessingContent', 'SecurityDuringStorageContent' and 'SecurityDuringTransferContent'. Partially supported will be the hierarchy *CH3* 'InformationSharingContent'. Analysis will be applied to the containing information privacy practices *CH33* 'SharingWithAdvertiserContent', *CH34* 'SharingWithAggregatorContent', *CH35* 'SharingWithAnalystContent', *CH36* 'SharingWithDeliveryContent', *CH37* 'SharingWithGovernmentContent', *CH38* 'SharingWithOtherUsersContent', *CH310* 'SharingWithPublicContent' and *CH312* 'SharingWithUserAuthorizedContent'. The hierarchy *CH4* 'InformationStorageContent' is relevant and can be analysed via static code analysis, as well as the hierarchies *CI21* 'EnvironmentSensorContent', *CI22* 'LocationSensorContent', *CI23* 'UserSensorContent' and all their coherent sub-hierarchies. For the hierarchy *CI24* 'SoftwareUseSensorContent' only partial support for the sub-hierarchies *CI242* 'CookiesContent' and *CI243* 'SurveysContent' are feasible to be analyzed by static code analysis. With the exception of one information privacy practices in the hierarchy level *CI31* 'InformationFormContent' all other information privacy practices are relevant for this thesis: *CI311* 'AudioInformationContent', *CI312* 'ImageInformationContent', *CI314* 'TextInformationContent' and *CI315* 'VideoInformationContent'. The next hierarchy level *CI32* 'IdentifierContent' is fully relevant and all coherent sub-hierarchies will be analyzed. More difficult to analyze via static code analysis will be the hierarchy level *CI33* 'OperationalContent', because only two information privacy practices were identified as relevant to static code

---

[25] See Dehling, Sunyaev (2016), p. 8.

analysis: *CI333* 'LocationContent' and *CI335* 'OnlineContactsContent'. Finally the hierarchy *CI34* 'UserDetailsContent' is partially relevant, namely the information privacy practices *I341* 'DemographicsContent', *CI343* 'HealthContent', *CI344* 'IdeologicalContent', *CI345* 'PreferencesContent' and *CI346* 'UserDeviceContent'.

All relevant information privacy practices and their static code analysis identification strategies will be explained further in chapter 3.1.3 of this thesis.

3. Implementation and Evaluation of an Automated Information Privacy Risk Assessment Tool

3.1 Implementation of an Automated Information Privacy Risk Assessment Tool

Our implementation of an automated information privacy risk assessment tool is structured in three phases. In the first phase, Android APK files need to be downloaded to acquire the foundation of a static code analysis: the source code. While APK files are binary representations of source code, it is necessary, in a second phase, to decompile to binary code back into actual source files. The third phase is the analysis phase, where the information privacy risk assessment takes place.

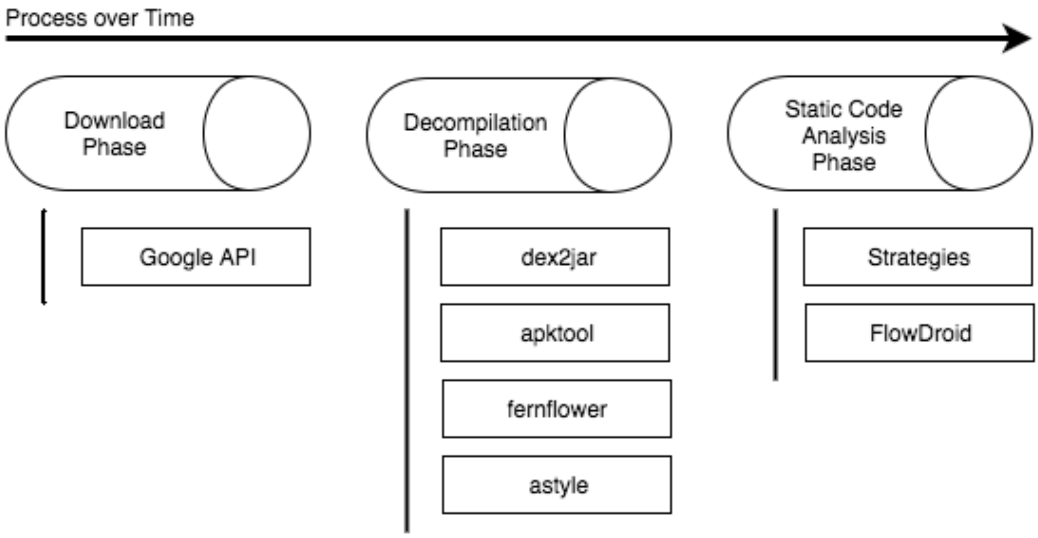

Fig. 3-1: Diagram of implementation phases of the automated information privacy risk assessment tool over time.

Figure 3-1 shows the implementation phases over time including the tools used within each phase. The tools will be described in greater detail within the following chapters.

3.1.1 Download Phase

The download phase is the first of the three implementation phases and comprises the acquisition of Android APK files. The APK files hold the necessary Java source code that we will perform the static code analysis on. Since this thesis emphases on Android mHealth apps, we used the repository database of Xu, Liu (2015) as our main app data

source.[26] Xu, Liu (2015) list mHealth apps from the Apple AppStore[27] and Android PlayStore[28] and update their repository quarterly by scraping the app stores. The list contains information for example on the app's identifier, category in the app stores, description, email address of the developer, price and the user rating of the app.

We used the repository database to loop over the available mHealth app listings and included only the apps that were available for free, indicated by a price of $0.00. As soon as the package name of an app is gathered, the download of the APK file can begin. While there is no official source to download APK files for Android apps, a multitude of websites exist that host copies of APK files to download for free. Unfortunately, all of these websites implement mechanisms that make it impossible to browse and download the APK files programatically within a download script. Instead, we used a open source Python implementation of an undocumented Google PlayStore API.[29] The undocumented part of the Google API allows users to download APK files. Even though the project has not been maintained for four years, the software is still in working order. The Python script authenticates to the Google API via the hardware ID of an Android smartphone or tablet and pretends to request data from this smartphone or tablet, even though the requests are sent from a desktop computer. We used a real Android tablet to detect its hardware ID and authenticate the Google API requests with this hardware ID. The main issue that has to be taken care of during the download phase is not to run into Google API limitations. Google allows an API user to only request a certain amount requests per time unit. After this limit is exceeded, the requests will just return a HTTP error code and no APK file will be downloaded. In order to work around this circumstance, we ran our download script multiple times, always until the Google API returns error codes. We then waited a couple of hours and tried the download script again, which would pick up the download process where it had stopped on the last run.

---

[26] This paragraph follows Xu, Liu (2015).

[27] https://itunes.apple.com, visited 07/13/16

[28] https://play.google.com, visited 07/13/16

[29] https://web.archive.org/web/20140920061441/https://github.com/egirault/googleplay-api, visited 05/12/16.

3.1.2 Decompilation Phase

In order to decompile the amount of APK files available, it is necessary to automate the process. The automation script[30] uses a chain of tools to gather access to the source code files from an APK file. The tools used to decompile the APK files follow closely the tools described and used by Enck et al. (2011).[31]

In a first step, we use the tool *dex2jar* to extract the JAR file from the APK file. The JAR file contains the java bytecode representations of the app which is just one part of the contents of an APK file. The next step is to extract resource files, such as the *Android Manifest* file from the APK file. The *Android Manifest* contains meta information about the app in a structured XML format.[32] The meta information include the package name of the app, the permissions the app requests, for example camera usage, internet access or geolocation usage. The *Android Manifest* file is therefore an important indicator at a high level view on activities within a given app. In order to extract the *Android Manifest* file from the APK file along with other resources such as images, icons, xml files or other files used within the app, we use the *apktool*[33]. *apktool* is a frequently updated Android reverse engineering tool that is used to extract resources from APK files. Another important part of the extraction of resource files is retrieving the layout and localization files. These files include information on the user interface components used within the app as well as text content for labels and text fields. The text content will be used to train machine learning algorithms to classify features of the app, further described in the analysis phase section below.

At the core of the decompilation process is the usage of *fernflower*[34]. *fernflower* is the recommended java decompiler by Enck et al. (2011). They used the tool to decompile a test sample of apps and gained a significantly higher code recovery rate than by using other decompiling tools.[35] An obstacle in decompiling java source code is obfuscation.

---

30 https://github.com/thomasbrueggemann/AIPRAT/blob/master/decompile/decompile.sh, visited 05/17/16.

31 See Enck et al. (2011), p. 5.

32 See for this and the next sentence Xu (2013), p. 7 and Shabtai, Fledel, Elovici (2010), p. 331.

33 https://web.archive.org/web/20160617041519/http://ibotpeaches.github.io/Apktool/, visited 06/17/16.

34 https://web.archive.org/web/20150808204302/https://github.com/fesh0r/fernflower, visited 05/24/16.

35 See Enck et al. (2011), p. 6.

Java developers can make use of a security feature called obfuscation that aims at hiding away the logic of java classes by renaming classes, variables and method names and disassembling the code into pieces that are difficult to read for an human interpreter. The idea is to make it more difficult to retrieve and make sense of the original source code by decompiling the byte code. *fernflower* uses a renaming approach by assigning every obfuscated class with a new naming pattern. Member variables and methods will be automatically renamed and therefore provide an easier and more unique way of reading the source code. Optionally the decompilation process can use an automatic code formatting tool called *astyle*[36] to format the source code. This helps humans to read the source code files more easily, since the formatting and indentation of all source code files is identical and therefore very structured. Formatting the source code will help in the evaluation phase of this thesis to support the manual inspection the source code for information privacy risks by human researchers.

The expected result of the decompilation phase is a directory named after the package name of a given app that contains the resource files, including the *Android Manifest* and the decompiled source code of the app. The decompilation will be performed in order of file size. The smaller the APK file the earlier it will decompiled. This is due the fact that only APK files to a certain size can be analysed by the dataflow analysis tool, described in section 3.1.3.

3.1.3 Static Code Analysis Phase

The static code analysis phase is the main analysis phase of the thesis and uses the output of the previous decompilation phase to perform the static code analysis. The static code analysis tool is implemented as a Java software project, since the used analysis libraries are implemented in Java and Android source code is written in Java too. The output of the static code analysis Java project is an executable Java archive file called *AIPRAT.jar* that can be executed in the command line terminal. In order for *AIPRAT.jar* to perform the static code analysis on APK files, two preparation steps are required.

The first preparation step is to run an Android data flow analysis tool over the APK files that extract potential data flows. The data flow analysis is achieved with an open

---

[36] https://web.archive.org/web/20160422015538/http://astyle.sourceforge.net/, visited 05/25/16.

source tool called *FlowDroid*, introduced by Arzt et al. (2014).[37] *FlowDroid* extends the Java optimization framework *Soot*, which was already used by Enck et al. (2011) for post-decompilation optimization tasks.[38] The data flow is analysed by scanning an intermediate byte code format provided by *Soot* for so called 'sources' and 'sinks'.[39] A source is the origin of a data flow, for instance the user input of data via a textfield and a sink is the destination that data flows. An example for a sink is a HTTP internet connection or a local log file. *FlowDroid* is also able to emulate Android lifecycle entry points. While a regular Java program has a single entry point to start the application from, the *main()* function, Android apps provide multiple entry points. The entry points of an Android app are determined by the states an app can be in. It can for instance return from being in the background, do a fresh start and return from being offline. All these entry points are being emulated by *FlowDroid* into a single *main()* function call. The output of the data flow analysis is one XML file per analysed APK file that contains a list of sinks and the coherent sources of data flows to that sink. The XML file will be parsed by the main static code analysis tool later on and the sink and source methods will be interpreted in the context of information privacy risks.

The machine learning text classifiers will be trained within the second preparation step. During the static code analysis phase of this study, we will be making great use of the naive Bayes classifier. A machine learning text-classifier classifies text segments into distinct categories. The categories are predefined in the training phase of the classifier, since every trained text segment is assigned with a training category. These training categories are the categories the classifier can assign to new, previously unseen, text segments. The incisive feature of a Bayes classifier is the fact, that it chooses to classify a category to a new segment of text by picking the most probable category.[40] A naive Bayes classifier furthermore assumes that all categories are distinct and independent of each other. Even though this might not always be the case in a real life usage scenario, the naive Bayes classifier still performs well enough for a wide range of use cases.

---

37 See Arzt et al. (2014), p. 259-269.

38 See Enck et al. (2011), p. 5.

39 See Arzt et al. (2014), p. 264.

40 For this and the following two sentences see Rish (2001), p. 41.

In the case of the static code analysis in this study, we will be using the naive Bayes classifier to classify URLs into categories. The categories that URLs can belong to, in the context of this study, are: advertisement, delivery services, government, instant-messaging, (data-) aggregation services, search engines and social networks. While the set of categories might not complete in terms of all possible and available categories, it is sufficient for the classification of URLs within this static code analysis to classify into the mentioned category-set. In oder for the naive Bayes classifier to classify text into categories we trained a naive Bayes classifier implementation with meta-information about URLs from the previously mentioned categories. First, it was necessary to collect URLs for the categories to train the naive Bayes classifier and we used a collection of URLs from *URLBlacklist.com*[41]. *URLBlacklist.com* provides URL lists for the categories advertisement, government, instant-messaging, search engines and social networks. *programmableweb.com* catalogues API descriptions including the service providers' URL. We developed a program to automatically download and store the API directory for the two remaining categories, from *programmableweb.com*.

Next, to acquire meta-information for all the URLs, we implement a downloader for the HTML source-code of all URLs and store the 'description' HTML-meta tag content in a file.[42] The 'description' meta-tag contains a small amount of text, provided by the website owner, that describes the content or function of the website topic. We use this 'description' meta-information to train the classifier with the associated categories.

As soon as the preparation steps are finished, the main static code analysis tool is ready to perform the information privacy risks analysis. We call the main static code analysis tool '*AIPRAT*', short for automated information privacy risk assessment tool from here on. The fundamental concept of *AIPRAT* is to iterate over all available apps and apply a set of analysis operations, called 'strategies', to the source code of these apps. A strategy tries to identify information privacy risk factors by applying algorithms, for instance feature extraction or text search, to parts of the app source code. There are two types of strategies

---

41 https://web.archive.org/web/20160617050003/http://urlblacklist.com/?sec=download, visited 06/17/16.

42 For this and the next sentence see https://web.archive.org/web/20160404111603/http://www.w3.org/wiki/HTML/Elements/meta, visited 07/13/16.

in *AIPRAT* , generic and specific strategies.

Generic strategies are strategies that contain algorithms that are configurable and usable by other specific strategies. An example for a generic strategy is the Java-class *analyze.src.strategies.ExistenceStrategy*. The ExistenceStrategy is able to scan through all source code files of an app and search for a set of words. If one or more of the source code files contains one or more of the search words, the *ExistenceStrategy* returns a set of source code snippets that contain the lines of code containing the search words. In total *AIPRAT* contains eight generic strategies in the Java-package *analyze.src.strategies*: DataFlowStrategy, ExistenceStrategy, InputStrategy, TraceBackStrategy, InformationCollectionStrategy, PermissionStrategy, ProviderUrlStrategy, and UrlCategoryStrategy. The DataFlowStrategy parses the pre-extracted dataflow XML from the *FlowDroid* preparation phase and allows iterating over all identified dataflow sources and sinks. Thereby, the DataFlowStrategy allows to pass parameters along that filter the sources and sinks for certain search words and provide feedback if the search words were found within sources and sinks. To find strings within the source code of an app, one can make use of the ExistenceStrategy. This generic strategy scans the full source code of an app and collects source code lines that match a given search pattern. The InputStrategy iterates over all XML layout configuration files of an app. User interface controls that are displayed within an app are declared in these XML layout configuration files. The InputStrategy tries to identify all user input fields and therefore scans the layout files for the search terms 'EditText', 'AutoCompleteTextView', 'CheckBox' and 'RadioButton'. As soon as a user input field has been found, the InputStrategy extracts all meta information about this input field possible. The input fields meta information generally contain the user interface control 'id', a 'hint' field and a 'text' field. The meta information are collected and stored together with the input field information.

An important feature in static code analysis, especially for assessing information privacy risks, is to the ability to trace data flows from a source to a sink. With the help of the call graph construction feature of FlowDroid, the TraceBackStrategy starts at a given set of start-sinks and traverses the call graph back until either a source is found or a given search pattern is matched. This allows consequent strategies to define a search pattern for data flows to specific sinks. In the case of this thesis, we will mainly use the TraceBackStrategy to find data flows that end in a information collection scenario. We

define information collection as a data flow that results in storing the information either locally on the device the app is run, or that results in sending the information to a remote server. With making use of the InputStrategy and the TracebackStrategy, the InputInformationCollectionStrategy takes the user input fields analysis one step further and allows for information collection analysis. First, all user input fields are detected and stored. In a second step, the InputInformationCollectionStrategy executes a TraceBackStrategy that starts at all available information collection sinks (local file storage and remote server connections) and traces back the call graph in an attempt to identify the user input field 'ids' within the call graph path. If a user input field 'id' is found, a data flow towards an information collection sink is identified and a potential information privacy risks revealed. A less sophisticated approach is being used by the PermissionStrategy. It is required for Android apps to declare permissions to use certain features, such as the GPS location or internet access, within the 'manifest' file. The PermissionStrategy enables a search through these permissions by a given search pattern.

The last two generic strategies concern the URLs that are listed within the app source code and that are potentially target to information transfers. The ProviderUrlStrategy iterates over all extracted URLs from the source code and checks how similar the URL host is in comparison to the app package name. The package name is often similar to the hostname of the app provider or contains similar name parts. The ProviderUrlStrategy takes these potential sub-parts of the package name into account and returns a probability that a URL connection to the app provider is established. Finally, the URLCategoryStrategy enables a search for a given category of URLs within the app source code. All URLs are classified into distinct categories upon loading the app into the static code analysis tool via a machine learning text classification technique. In order to check if a URL of a certain category exists, the iterates over all classified URLs and matches their categories to the search category.

A specific strategy, on the other hand, targets the exploration of an information privacy practices directly and contains the information privacy practices hierarchy identifier, introduced by Dehling, Sunyaev (2016), in its Java-classname.[43] The Java-class *analyze.src.strategies.CI213_Strategy* contains a search pattern for the information pri-

[43] See Dehling, Sunyaev (2016), p. 6.

ExistenceStrategy	CH23, CH310, CH312, CH33, CH35, CH38, CH43, CH44, CI212, CI213, CI214, CI221, CI222, CI223, CI231, CI242, CI243
InputInformationCollection-Strategy	CI321, CI322, CI323, CI324, CI325, CI326, CI341, CI343, CI344, CI345, CI346
InputStrategy	CI314
PermissionsStrategy	CH43, CH44, CI211, CI212, CI214, CI231
ProviderUrlStrategy	CH45
TraceBackStrategy	CH21, CH22, CH42, CH45, CI221, CI311, CI312, CI315, CI333, CI335
UrlCategoryStrategy	CH34, CH36, CH37, CH46

Tab. 3-1: Specific strategies in relation to the generic strategies that they make use of.

vacy practices with the hierarchy identifier CI213, which refers to the information privacy practices Content (C) → InformationCollectionContent (I) → InformationCollectionSensorContent (2) → EnvironmentSensorContent (1) → MicrophoneContent (3). Therefore a specific strategy may contain a combination of one or many generic strategies to try to identify the risk of the associated information privacy practices is posing through static code analysis. In the example of the specific strategy *analyze.src.strategies.CI213_Strategy*, the class extends the generic strategy class *analyze.src.strategies.ExistenceStrategy* and sets the search parameters of the *ExistenceStrategy* to 'MediaRecorder.setAudioSource(' and 'MediaRecorder.AudioSource.MIC'. The *CI213_Strategy* class scans via the parent-class *ExistenceStrategy* all of the source code files off the app for source code that uses the Android microphone API.

In the following section we will introduce the specific strategies implemented in the automated information privacy risk assessment tool in greater detail. Table 3-1 displays and overview of the generic strategies in the first column and the specific strategies that make use of the coherent generic strategies in the second column. The specific strategy classes in this chapter and in table 3-1 are abbreviated with the information privacy practices hierarchy identifier of the matching strategy. *CH21* for example stands for *CH21_Strategy.java* in this chapter.

ExistenceStrategy. The generic ExistenceStrategy ist the most widely used strategy within the analysis tool. Strategy *CH23* uses the ExistenceStrategy twice to check if data transfers are secure. First strategy *CH23* checks for the existence count of HTTP connections and second, it check for the existence count of HTTPS connections. The result of

CH23 is the ratio of HTTP to HTTPS connections. The strategies *CH310*, *CH312* and *CH38* try to identify a user initiated sharing of content with other users or the general public. The strategies accomplish this by searching the source code of an app for triggering the Android sharing dialog window. Additionally we search for methods that open sharing dialogs to three large social media networks, *facebook*, *twitter*, and *Google+*. Strategy *CH33* uses the generic ExistenceStrategy extensively to identify the usage of advertisement libraries within the app. All search words are names of Android advertisement library identifiers and are derived from the analysis of Android ad libraries by Book, Pridgen, Wallach (2013).[44] In order to scan the source code of an app for the usage of analytics services, as stated in the hierarchy item *CH35*, the strategy implementation for *CH35* uses the ExistenceStrategy to check for analytics services and libraries search words. The search words are derived from the open-source listing of Android libraries on https://android-arsenal.com in the category 'Analytics'.[45] *CH43* and *CH44* search for function names that indicate a writing of data to the internal or external storage of an Android device. Internal and external storage can be a local database that every Android app instance can write to, the filesystem or a 'shared preferences' key value store, provided by the Android runtime. Internal storage refers to storage of data on the devices' flash memory itself and external storage refers to storage of data on inserted storage mediums, such as storage card. Usage of the Android devices' internal camera is accomplished via a standardised API. In order to detect the usage of the internal camera, the *CI212* strategy searches for the code snippets that are designated to launch the camera view inside of an Android app. A similar approach is used by the *CI213* strategy, that aims at identifying the usage of the microphone recording by utilizing the ExistenceStrategy to find microphone calling function names. *CI214* searches the source code for the usage of the Android near-field communication (NFC) API, via the ExistenceStrategy. The Android API has one particular method call to initiate a NFC connection, so detecting this feature via the ExistenceStrategy is effortless. The strategies for *CI221*, *CI222* and *CI223* utilize the ExistenceStrategy to detect wether the user's location is queried. This can either happen by calling the Global Positioning System (GPS) location, or by using the approx-

---

44 See Book, Pridgen, Wallach (2013), p. 9.

45 http://web.archive.org/web/20160514040053/http://android-arsenal.com/tag/5, visited 07/03/16.

imate location via the internet network connections of the device. A fairly new technique to smart-devices is the usage of a fingerprint sensor. This functionality is exposed via a single API entry-point in Android and can be detected by the *CI231* strategy using the ExistenceStrategy. Also exposed via a single API entry-point is the ability to store and retrieve cookies[46]. Therefore the *CI242* strategy can effortlessly detect the usage of such cookies within an app. More difficult is the detection of potential user surveys within an app. *CI243* uses the ExistenceStrategy to search the app source code for search words such as 'survey', 'audit' and 'syllabus'.

InputInformationCollectionStrategy. The following strategies try to identify dataflows from text input fields to information collection sinks. *CI321* tries to identify dataflows from financial identification input fields. Financial identifier text input fields will be identified by using the search words: 'creditcard', 'cvc', 'iban', 'bic', 'bankaccount', 'bank', 'mastercard', 'paypal' and 'visa'. If none of the search words are found within the identifiers of text input fields the certainty property of the result object is set to *MEDIUM*. This is because the list of search words might not be complete or the text input field are named in a non expressive way. For example the text input fields might be named 'id=123456' instead of something expressive such as 'id=txtIBAN'. The just explained certainty property value '*MEDIUM*' for non-found input information collection dataflows holds true for the rest of the following strategies that use the InputInformationCollectionStrategy. Strategy *CI322* tries to recognize government identifier dataflows within the app and uses the search words 'socialsecurity', 'insurance', 'tax', 'SSN', 'national', 'government' and 'identification'. *CI323* only needs to identify the information collection of the name of the users, it utilizes the InputInformationCollectionStrategy with the search word 'name'. This automatically includes a search for 'surname' and for example 'middlename', since the InputInformationCollectionStrategy always performs a substring search within the text input field identifiers. The search words to identify an online contact of users are more definite. Online contact information refers to the different ways users can be contacted online, for example via 'email', 'skype', 'facebook', 'twitter', 'chat' and many more. *CI324* uses the previously mentioned search words to identify dataflows of online

---

[46] Cookies are pieces of information stored on the client device, accessible only by the provider that stored the cookie information until they expire. See Laudon, Laudon, Schoder (2010), p. 168.

contact information to information collection sinks. In contrast to the online contact information, the strategy for *CI325* tries to identify information collection of physical contact information. This refers to the address of users or their phone number. To identify users within an app itself, app providers implement their own unique user identifiers into the apps. A unique user identifier could be a username, email address, and assigned Universally Unique Identifier (UUID) or a password. These unique user identifier information collection dataflows are detected by strategy *CI326*. Strategy *CI341* detects information collection of demographic information. The search words for demographic information are derived from the survey recommendations for demographic standards from the German 'Statistisches Bundesamt'.[47] The strategy *CI343* tries to identify users health information collection dataflows. Here, the search words are derived from the manual review of mHealth apps by Brüggemann et al. (2016). The extensive health content search words are 'dosage', 'pill', 'blood', 'heart', 'bloodpressure', 'bloodsugar', 'heartrate', 'disease', 'symptom', 'weight', 'height', 'body', 'bmi', 'temperature', 'medical', 'doctor', 'calories', 'diet', 'sleep', 'carbon', 'hydrate', 'intake', 'haemoglobin', 'anaesthetic', 'urine' and 'hospital'. Ideological content, such as believes or membership of religious or political groups, information collection is detected by the strategy *CI344*. Input fields asking for membership of political parties for example, could be designed as option choice buttons and are captured by the InputInformationCollectionStrategy as well. Preferences of the users are difficult to capture, since preference-capturing input fields usually are connected to some sort of context, that users are asked their preferences about. For the sake of this thesis, we will scan the input fields for expressions of preference, for example: 'like', 'dislike', 'favourite', 'favorite', 'preference' and 'prefered'. We acknowledge that in case we did not find any preference capturing input field information collection, the certainty for this is at a level *'LOW'*. It is much more likely that we were unable to identify the information collection in that case. The last strategy using the generic InputInformationCollectionStrategy is the strategy for *CI346*. Strategy *CI346* tries to identify information collection of user device information, such as the Internet Protocol (IP) address or the operating system name.

---

[47] See https://web.archive.org/web/20151115091328/https://www.destatis.de/DE/Methoden/DemografischeRegionaleStandards/Fragebogen_schriftlich.pdf?__blob=publicationFile, visited 07/05/16.

InputStrategy. The generic InputStrategy is only used by a single specific strategy. The strategy for *CI314* tries to detect and list all of the input fields a user can input information into, within the app. This is done by executing the InputStrategy, explained earlier in this chapter.

PermissionsStrategy. The strategies for *CH43* and *CH44* check, as an additional condition, if a permission to write on external and internal storage is requested. Only if a permission is declared, an app is allowed to use the coherent functionality of the Android API. This is granted by the Android runtime. The strategy for *CI211* relies solely on the generic PermissionsStrategy to identify the usage of Bluetooth functionality within the app. An Android app is not granted access to the Bluetooth interface, if it does not declare the appropriate permission. *CI211* uses the PermissionsStrategy only, because Bluetooth interface APIs are not standardized and not easy to detect individually. To The previously described strategy implementations for *CI212*, *CI214* and *CI231* behave in the same way as *CH43* and *CH44*. They use the PermissionsStrategy as an additional condition, to detect the usage of the camera, near field communication NFC and the fingerprint sensor.

ProviderUrlStrategy. The only specific strategy using the generic ProviderUrlStrategy is the strategy for *CH45*. It uses a list of all potential URLs that belong to the app provider, to identify information collection that sends data to the app providers servers.

TraceBackStrategy. Strategy *CH45* uses, additionally to the ProviderUrlStrategy, the generic TraceBackStrategy. It traces information collection flows to the previously detected app provider URLs. Dataflows that result in file storage on the internal flash drive or an external storage card can also be identified with the generic TraceBackStrategy. The strategy for *CH22* walks up the callgraph of all local storage collection sinks and checks if a function name containing the search words 'cipher' and 'crypt' was found along the way. This would indicate that data stored on the local storage is stored in an encrypted way. The strategy *CH21* uses a special case of the TraceBackStrategy, executed in *CH22*. It uses an inverted sink parameter. This means, that it will trace all dataflows in the callgraph that do not match a start sink search word. It is therefore easy to trace all dataflows that do not end in a local storage sink. This allows strategy *CH21* to identify if encryption is used during processing of data, rather than storage. Strategy *CH42* tries to identify information collection at cloud storage providers. Since it is virtually impossible

to know, list and check for every possible cloud storage provider, the strategy searches for the word 'cloud' within dataflows that result in an information collection sink. We acknowledge the low probability of actually finding a cloud storage dataflow with this technique by setting the certainty level of the result to *'LOW'* by default. A much higher success rate is promised by the standardized Android API to query the users location. Detecting dataflows from these location queries to information collection sinks is done by the *CI221* and *CI333* strategies, mainly by searching for the usage of the 'LocationManager' object in Java. The 'LocationManager' is the single point of entry to query the users location. A similar unambiguity is present for the detection of audio information collection. While the callgraph is being traversed, strategy *CI311* tries to identify function calls that contains the following search words: 'microphone', 'audio', 'recorder', 'mediarecorder', 'music' and 'sound'. The strategy for *CI312* tries to detect the information collection of image information and therefore uses the search words 'picture', 'ACTION_IMAGE_CAPTURE', 'MEDIA_TYPE_IMAGE', 'CAPTURE_IMAGE', 'image', 'setImageDrawable' and 'Bitmap'. Central to image processing in Android is the 'Bitmap' object, as it is the binary representation of an image. Detecting the usage of the 'Bitmap' object in an information collection callgraph path therefore yields a potential image information collection. The same holds true for strategy *CI315*. In comparison to *CI312*, *CI315* tries to detect the information collection of video information. Video processing in Android works immaturely different, than photo processing. Instead of manipulating a 'Bitmap' object, as for the photo processing, videos are stored by the Android API and accessible via an uniform resource identifier (URI).[48] Therefore the *CI315* strategy can only detect video capturing information collection. Strategy *CI335* identifies information collection of the users contacts stored in his address book. Querying the contacts via the Android API is always done via the 'ContactsContract' object. The *CI335* strategy can therefore easily detect information collection dataflows of users contacts by using the TraceBackStrategy.

UrlCategoryStrategy. As explained earlier in this chapter, the generic UrlCategoryStrategy preloads its training data of URL classification into main memory at program start-

---

[48] See http://web.archive.org/web/20160425094249/https://developer.android.com/training/camera/videobasics.html, visited 07/09/16

time. From then on, the UrlCategoryStrategy can be queried if certain URL categories are present within the currently inspected app. Strategy *CH34* searches for URLs that belong to a data aggregation service that potentially aggregates the users data. The *CH36* strategy searches for URL connections to delivery services. Delivery services can be postal service, package shipping or other logistic services. URLs to government institutions are identified by the *CH37* strategy using the generic UrlCategoryStrategy. Finally, the *CH46* strategy tries to detect URL connections to storage providers. Storage providers refers to cloud storage or bulk data storage services that allow external data storage.

3.2 Evaluation of an Automated Information Privacy Risk Assessment Tool

In order to evaluate how well the automated information privacy risk assessment tool is performing in comparison to a human researcher, we analyze a sample of three randomly chosen apps by two human researchers. Each researcher is presented with the source code of the selected apps and the list of relevant information privacy practices to identify. The task for each researcher is to identify as many information privacy practices as possibly by analyzing, searching and reading through the source code files. This will result in an overview comparison on how well the automated information privacy risk assessment tool performs in comparison to a human.

TODO: WACKELIG We acknowledge that a human app reviewer can almost find any information privacy practices risk within the app source code given enough time. In order to find an appropriate time range that the human review should spend analyzing a single app, we looked at common review times of app stores today. The Google PlayStore does not manually review newly uploaded apps, while the Apple AppStore does apply manual review to each and every app update. The average time it takes for an app to pass the Apple AppStore review, which also includes waiting in a queue to be reviewed, is currently two days.[49] In order to keep the review times at a realistic level, a human app reviewer of our test sample apps should therefore not spend more than two days on analyzing a single app.

The reviewers will document their results in a tabular form for each app. For each of the relevant information privacy practices the reviewers will mark if they were able

[49] See http://web.archive.org/web/20160513013009/http://appreviewtimes.com/, visited 05/13/16.

to detect the information privacy risks the information privacy practices poses or not. Additionally they will report the certainty with which they feel they detected a certain information privacy practice. The certainty levels are *LOW*, *MEDIUM*, *HIGH*.

A *HIGH* certainty expresses, that the reviewers are very confident that they found the information privacy practices in question. A *LOW* certainty indicates that the reviewers found a potential indicator for an information privacy practices but is rather uncertain if this is indication expresses the full information privacy practices or if there are not any other indicators for the information privacy practices that were not found yet or are unfindable. To express a neither high nor low certainty about a information privacy practices found, the reviewers can use a *MEDIUM* certainty level.

4. Feasibility of Automated Information Privacy Risk Assessment

Within the following chapter we want to express the results of the implementation and evaluation phase and assess to what degree the automation of information privacy risk assessment is feasible. We will also present the results on how well the automated information privacy risk assessment tool performs in comparison to human researchers.

4.1 The Automated Information Privacy Risk Assessment of Free Android mHealth Apps

4.1.1 Download Phase

The original dataset from the Xu, Liu (2015) repository of app store listings contains 5 379 app entries from the Google PlayStore in the category 'Medical' and 'Health and Fitness'. From this original dataset we extracted the 3 180 free apps for further inspection. It was possible to download 2 250 app APK files via the undocumented Google API. The remaining 930 APK files either returned a Google authentication error or were not available on the Google PlayStore anymore.

Downloading the 2 250 APK files took multiple download-runs over the whole dataset, since the Google API only allows a certain amount of download requests per time unit. The number of allowed download requests varied throughout the download phase and could not be detected exactly. Various tests downloading APK files automatically via websites like apkpure.com[50] or apk-dl.com[51] failed due to those websites effectively blocking automated non-browser traffic. The download of the 2 250 APK files took 11 days in total and 18 restart attempts of the download script.

4.1.2 Decompilation Phase

From the downloaded 2 250 APK files, 355 were decompiled. The reason that only 15% of the 2,250 APK files were decompiled is the time that was available to write this thesis. The decompilation time varied from 4 minutes to 16 minutes, based on the amount and complexity of the source files of the app. The 355 APK files that were decompiles were

---

[50] https://web.archive.org/web/20160528165049/https://apkpure.com/, visited 06/05/2016

[51] https://web.archive.org/web/20160518104842/https://apk-dl.com, visited 06/05/2016

selected in order of their file size. An app with a lower file size was chosen over an app with a larger file size. The reason for this is the restriction of *FlowDroid* to only analyze apps that fit into main memory, explained in section **TODO: SECTION RAUSSUCHEN IN DER ERKLÄRT WIRD, WARUM FLOWDROID LAHM IST.**.

During the decompilation of the APK files from smallest file size to largest, the decompilation failed to finish in 24 cases. Reasons for decompilation failure are heavily obfuscated source code, that forced the decompiler to abort the process and memory exceptions. Memory exceptions appear if the source code files of the currently decompiled app exceed the size of the available main memory of the computer that performs the decompilation. This memory exceeding happens due to the fact that *fernflower* keeps all source code file representations in main memory (including the Java virtual machine overhead) and therefore consumes a lot of memory.

4.1.3 Static Code Analysis Phase

The static code analysis phase went through three revisions. The first revision was based on the idea that the *FlowDroid* API can be used within the *AIPRAT* tool itself as an external library. This attempt failed due to the fact that *FlowDroid* uses significant amounts of main memory. In case *FlowDroid* ran out of memory, for apps with a larger or more complex source code structure, the operating system buffers the main memory to the hard disk which causes the running application to be slow. Further analysis was therefore impossible and the *FlowDroid* analysis would take infeasible amounts of time.

The second revision of the static code analysis phase was to outsource the *FlowDroid* analysis into a separate application that precomputes the dataflows of any app in our dataset and stores the dataflow results in an XML file format. The idea behind this revision was, to be able to run the *FlowDroid* dataflow analysis on high performance virtual machines in the Amazon Cloud. We used a *m4.4xlarge* instance[52], which is equipped with 64 gigabytes of main memory. But even considering the great number of main memory available on the Amazon Cloud virtual machine, *FLowDroid* was reluctant to compute the dataflow for certain apps on the dataset. The second downside of precomputing the

[52] https://web.archive.org/web/20160415154133/http://aws.amazon.com/ec2/instance-types/, visited 07/19/16.

dataflow analysis, is the fact that reading in the pre-computed dataflow analysis from a XML file does not provide the live-context information of using the *FlowDroid* API directly within the code. The *FlowDroid* objects that represent methods and relationships between methods are to complex to be fully stored within XML files. Therefore analysis of the callgraph of methods is very limited with this revisions' approach.

The last and final revision of the static code analysis phase discovered a part of the *FlowDroid* API that only computes the call graphs of apps with a limited subset of all the features that *FlowDroid* yields. The limited subset redeemed to be completely sufficient for the analysis purposes of this thesis. While large and complex apps still take up the main memory quickly, the threshold of APK file size seems to have changed to rather larger files for this approach. Therefore, we decided to conduct the static code analysis on the 15% of the apps with the smallest file size, from the original dataset.

The static code analysis tool was able to finish the analysis on 317 apps from the decompiled 355. The reason for the roughly 10% loss of apps from the decompilation to the static code analysis phase is mainly due to error in the generation of the callgraph by *FlowDroid*. As mentioned before, apps could be to too large in file size for *FlowDroid* to fit into main memory, and the static code analysis would skip to the next app.

Table 4-1 shows an overview of all implemented strategies to identify information privacy risks within an app and the number of found appearances throughout the static code analysis phase. While 15 of the 42 strategies did not find any results within the analysis phase, 27 did. In this case of an information privacy risks analysis, a non-found of a risk by a strategy is not necessarily a negative aspect. Especially the strategies for *CH36* SharingWithDeliveryContent and *CI344* IdeologicalContent do not particularly apply to the context of mHealth apps and their information privacy risks.

4.2 Evaluation of the Automated Information Privacy Risk Assessment Tool

Tab. 4-1: All implemented automatic information privacy risk assessment tool strategies and their found appearance count throughout the static code analysis phase of this thesis.

Information privacy practice hierarchy identifiers	Number of 'Found' Appearances in Results
CH21, CH34, CH36, CH37, CH38, CH43, CH45, CI213, CI222, CI223, CI321, CI322, CI344, CI345, CI346	0
CI343	1
CH42, CI231	2
CI214	3
CI335	4
CI323	5
CI333, CI324, CI325	6
CI243	8
CI211	9
CI341	11
CI326	15
CI212	17
CH310, CH312	21
CI315	22
CI242	30
CH22	31
CI221	32
CH23	34
CI311	44
CH35	54
CH44	95
CH33	105
CI312	159
CI314	206

5. Discussion

5.1 Principle Findings

5.2 Contributions

5.3 Limitations

5.4 Future Research

5.5 Conclusion

References

Arzt et al. (2014)

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, Patrick McDaniel. “Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps”. In: *ACM SIGPLAN Notices*. Vol. 49. 6. ACM. 2014, pp. 259–269.

Baca, Carlsson, Lundberg (2008)

Dejan Baca, Bengt Carlsson, Lars Lundberg: Evaluating the cost reduction of static code analysis for software security. In: Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security - PLAS ’08. 2008, p. 79

Bardas, Others (2010)

Alexandru G Bardas, Others: Static code analysis. In: Journal of Information Systems & Operations Management. No. 2, Vol. 4, 2010, pp. 99–107

Book, Pridgen, Wallach (2013)

Theodore Book, Adam Pridgen, Dan S Wallach: Longitudinal analysis of android ad library permissions. In: arXiv preprint arXiv:1303.0857. 2013,

Brüggemann et al. (2016)

Thomas Brüggemann, Joel Hansen, Tobias Dehling, Ali Sunyaev: “An Information Privacy Risk Index for mHealth Apps”. Manuscript. 2016

Chen et al. (2012)

Connie Chen, David Haddad, Joshua Selsky, Julia E Hoffman, Richard L Kravitz, Deborah E Estrin, Ida Sim: Making sense of mobile health data: an open architecture to improve individual- and population-level health. In: Journal of medical Internet research. No. 4, Vol. 14, 2012, e112

Dehling, Gao, Sunyaev (2014)

Tobias Dehling, Fangjian Gao, Ali Sunyaev: Assessment Instrument for Privacy Policy Content: Design and Evaluation of PPC. In: WISP 2014 Proceedings. 2014,

Dehling, Sunyaev (2016)

Tobias Dehling, Ali Sunyaev: “Designing for Privacy: A Design Theory for Transparency of Information Privacy Practices”. 2016

Enck et al. (2011)

William Enck, Damien Octeau, Patrick McDaniel, Swarat Chaudhuri: A Study of Android Application Security. In: Proceedings of the 20th USENIX Conference on Security. No. August, Vol. SEC’11, 2011, pp. 1–21

Fischer-Hübner (1998)

Simone Fischer-Hübner: Privacy and security at risk in the global information society. In: Information Communication & Society. No. 4, Vol. 1, 1998, pp. 420–441

Haris, Haddadi, Hui (2014)

Muhammad Haris, Hamed Haddadi, Pan Hui: Privacy Leakage in Mobile Computing: Tools, Methods, and Characteristics. In: . 2014,

He et al. (2014)

Dongjing He, Muhammad Naveed, Carl A Gunter, Klara Nahrstedt: Security Concerns in Android mHealth Apps. In: AMIA ... Annual Symposium proceedings / AMIA Symposium. AMIA Symposium. Vol. 2014, 2014, pp. 645–54

Khalid et al. (2015)

Hammad Khalid, Emad Shihab, Meiyappan Nagappan, Ahmed E. Hassan: What Do Mobile App Users Complain About? In: IEEE Software. No. 3, Vol. 32, 2015, pp. 70–77

Kumar et al. (2013)

S. Kumar, W. Nilsen, M. Pavel, M. Srivastava: Mobile Health: Revolutionizing Healthcare Through Transdisciplinary Research. In: Computer. No. 1, Vol. 46, 2013, pp. 28–35

Laudon, Laudon, Schoder (2010)

Kenneth C Laudon, Jane P Laudon, Detlef Schoder: Wirtschaftsinformatik: Eine Einführung 2nd ed., München et al. 2010

Mcclurg (2012)

Jedidiah Mcclurg: Android Privacy Leak Detection via Dynamic Taint Analysis. In: . 2012,

Mitchell et al. (2013)

Stacy Mitchell, Scott Ridley, Christy Tharenos, Upkar Varshney, Ron Vetter, Ulku Yaylacicegi: Investigating privacy and security challenges of mhealth applications. In: 19th Americas Conference on Information Systems, AMCIS 2013 - Hyperconnected World: Anything, Anywhere, Anytime. Vol. 3, 2013, pp. 2166–2174

Pollach (2007)

Irene Pollach: What's Wrong With Online Privacy Policies? In: Communications of the ACM. No. 9, Vol. 50, 2007, pp. 103–108

Rish (2001)

Irina Rish. "An empirical study of the naive Bayes classifier". In: *IJCAI 2001 workshop on empirical methods in artificial intelligence*. Vol. 3. 22. IBM New York. 2001, pp. 41–46.

Shabtai, Fledel, Elovici (2010)

A Shabtai, Y Fledel, Y Elovici. "Automated Static Code Analysis for Classifying Android Applications Using Machine Learning". In: *Computational Intelligence and Security (CIS), 2010 International Conference on*. 2010, pp. 329–333.

Solove (2002)

Daniel J Solove: Conceptualizing privacy. In: California Law Review. 2002, pp. 1087–1155

Xu (2013)

Liang Xu: “Techniques and Tools for Analyzing and Understanding Android Applications”. PhD thesis. 2013

Xu, Liu (2015)

Wenlong Xu, Yin Liu: mHealthApps: A Repository and Database of Mobile Health Apps. In: JMIR mHealth and uHealth. No. 1, Vol. 3, 2015, e28

Declaration of Good Scientific Conduct

Hiermit versichere ich an Eides Statt, dass ich die vorliegende Arbeit selbstständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Viersen, den 01. September 2016

I hereby attest that I completed this work on my own and that I did not employ any tools other than those specified. All texts literally or semantically copied from other works are attributed with proper citations. This work has not been submitted in identical or similar form for any other exam, assessment, or assignment.

Viersen, September 1st, 2016

Curriculum Vitae

Persönliche Angaben

Name:	Thomas Brüggemann
Anschrift:	Hoferkamp 9, 41751 Viersen
Geburtsdatum und -ort:	31.08.1989 in Viersen
Familienstand:	verheiratet

Schulische Ausbildung

1997 - 2001	Katholische Grundschule Boisheim
2001 - 2009	Bischöfliches Albertus-Magnus-Gymnasium in Viersen, Abschluss: Abitur

Grundwehrdienst

07/2009 - 04/2010	Wehrdienstleistender, Luftwaffe - JaboG 31 ”Boelke”, Kraftfahrer vom Dienst, Fliegerhorst Nörvenich

Studium

10/2010 - 03/2014	Universität zu Köln, Wirtschaftsinformatik, Bachelor of Science
10/2014 - 09/2016	Universität zu Köln, Information Systems, Master of Science

Beruflicher Werdegang

05/2010 - 09/2012	Thomas Trefz Consulting, Köln, Softwareentwicklung im Bereich Microsoft .NET
10/2012 - 10/2014	Beister Software GmbH, Aschaffenburg, Softwareentwicklung im Bereich Microsoft .NET
10/2014 - heute	Selbstständiger Softwareentwickler und IT-Berater