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 III

Table of tables IV

1. Introduction 1

1.1 Problem Statement 1

1.2 Objectives 2

1.3 Structure 3

2. Combining Source Code Analysis with Information Privacy Risk Assessment 4

2.1 Information Privacy Risk Assessment 5

2.2 Static Code Analysis 5

2.3 Relevant Information Privacy Risk Factors 5

3. Implementation and Evaluation of an Automated Information Privacy Risk Assessment Tool 8

3.1 Implementation of an Automated Information Privacy Risk Assessment Tool 8

3.1.1 Download Phase 8

3.1.2 Decompilation Phase 10

3.1.3 Static Code Analysis Phase 11

3.2 Evaluation of an Automated Information Privacy Risk Assessment Tool 16

4. Feasibility of Automated Information Privacy Risk Assessment 17

4.1 The Automated Information Privacy Risk Assessment of Free Android mHealth Apps 17

4.1.1 Download Phase 17

4.1.2 Decompilation Phase 17

4.1.3 Static Code Analysis Phase 17

4.2 Evaluation of the Automated Information Privacy Risk Assessment Tool 18

5. Discussion 19

5.1 Principle Findings 19

5.2 Contributions 19

5.3 Limitations 19

5.4 Future Research 19

5.5 Conclusion 19

References 22

Declaration of Good Scientific Conduct 23

Curriculum Vitae 24

Index of Abbreviations

API	Application Programming Interface
APK	Android Application Package
GPS	Global Positioning System
JAR	Java Archive
mHealth	Mobile Health
P3P	Platform for Privacy Preferences
URL	Uniform Resource Locator
XML	Extensible Markup Language

List of Tables

1. Introduction

1.1 Problem Statement

The market for mobile phone and tablet applications (apps) has grown extensively since recent years.[1] It is 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 an 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.

[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

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 complaint while using mobile apps.[8] This encourages research to address information privacy risks.

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 poses a threat to users information privacy.[9] Technical evaluation of mobile apps even goes further into the topics of decompilation to analyze device identification or geolocation data leaks.[10] Decompilation reveals to be a feasible 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 and provide feedback on coding styles to the users while programming or ”to find defects in programs”[11]. Static code analysis provides a fast way to analyze source code[12], 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 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[13] and try to implement static code analysis strategies to identify those risks au-

---

[8] See Khalid et al. (2015), p. 5.

[9] For the previous two sentences, see He et al. (2014), p. 645-646.

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

[11] Bardas, Others (2010), p. 1.

[12] See Bardas, Others (2010), p. 5.

[13] See Dehling, Sunyaev (2016), p. 8-17.

tomatically. 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.

2.1 Information Privacy Risk Assessment

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: AUCH FÄLLE IDENTIFIZIEREN, DIE MAN VIELLEICHT NOCH NICHT VERSTANDEN HAT!

For this thesis, we will use the set of information privacy practices extracted from literature, the Platform for Privacy Preferences (P3P) guide[14] and app reviews by Dehling, Sunyaev (2016) as a source to derive information privacy risk factors from. [15] 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 e.g. 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 ex-

[14] https://www.w3.org/TR/P3P/, visited 06/06/2016

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

press 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 certain information retention policy or not.[16] 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* 'InformationFormCon-

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

tent' 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 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.

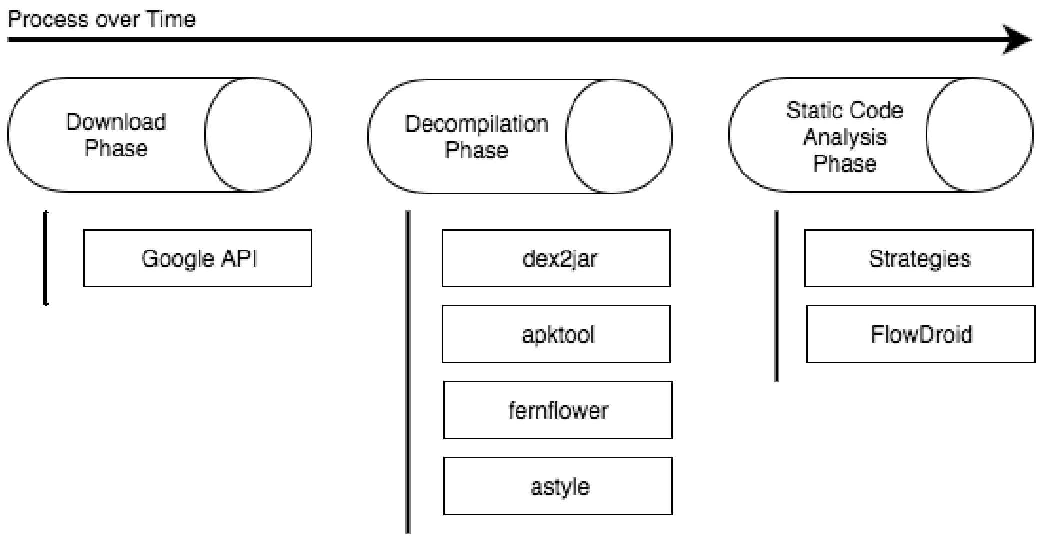
3. Implementation and Evaluation of an Automated Information Privacy Risk Assessment Tool

3.1 Implementation of an Automated Information Privacy Risk Assessment Tool

The 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.

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.

Figure 3-1: Diagram of implementation phases over time.



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.[17] Xu, Liu (2015) list mHealth apps from the Apple AppStore and Android PlayStore and update their repository quarterly by scraping the app stores. The list contains information for example on the app's id, 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 filtered out 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.[18] 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.

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

[18] https://github.com/egirault/googleplay-api, visited 05/12/2016

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[19] 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).[20]

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.[21] The meta information include the package name of the app, the permissions the app requests, e.g. camera usage, internet access or geolocation usage. The *Android Manifest* file is therefore an important indicator for high level activities within the 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*[22]. *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*[23]. *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.[24] An obstacle in decompiling java source code is obfuscation.

19 https://github.com/thomasbrueggemann/AIPRAT/blob/master/decompile/decompile.sh

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

21 This and the next sentence follow Xu (2013), p. 7.

22 http://ibotpeaches.github.io/Apktool/

23 https://github.com/fesh0r/fernflower

24 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*[25] 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.

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 source tool called *FlowDroid*, introduced by Arzt et al. (2014).[26] *FlowDroid* extends the Java optimization framework *Soot*, which was already used by Enck et al. (2011) for post-

[25] http://astyle.sourceforge.net/

[26] See Arzt et al. (2014), p. 259-269.

decompilation optimization tasks.[27] The data flow is analysed by scanning an intermediate byte code format provided by *Soot* for so called 'sources' and 'sinks'.[28] A source is the origin of a data flow, e.g. 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 e.g. 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.[29] 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.

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-

[27] See Enck et al. (2011), p. 5.

[28] See Arzt et al. (2014), p. 264.

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

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*[30]. *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. 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, e.g. feature extraction or text search, to parts of the app source code. There are two types of strategies 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.ExistanceStrategy*. The *ExistanceStrategy* 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

[30] http://www.urlblacklist.com/?sec=download, visited 05/30/2016

code files contains one or more of the search words, the *ExistanceStrategy* 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, ExistanceStrategy, 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 ExistanceStrategy. This generic strategy scans the full source code of an app an 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 smart 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.[31] The Java-class *analyze.src.strategies.CI213_Strategy* contains a search pattern for the information privacy 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 the associated information privacy practices is posing through static code

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

analysis. In the example of the specific strategy *analyze.src.strategies.CI213_Strategy*, the class extends the generic strategy class *analyze.src.strategies.ExistanceStrategy* and sets the search parameters of the *ExistanceStrategy* to 'MediaRecorder.setAudioSource(' and 'MediaRecorder.AudioSource.MIC'. The *CI213_Strategy* class scans via the parent-class *ExistanceStrategy* all of the source code files off the app for source code that uses the Android microphone API.

3.2 Evaluation of an Automated Information Privacy Risk Assessment Tool

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 5379 app entries from the Google PlayStore in the category 'Medical' and 'Health and Fitness'. From this original dataset we extracted the 3180 free apps for further inspection. It was possible to download 2250 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 2250 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[32] or apk-dl.com[33] failed due to those websites effectively blocking automated non-browser traffic. The download of the 2250 APK files took 11 days in total and 18 restart attempts of the download script.

4.1.2 Decompilation Phase

4.1.3 Static Code Analysis Phase

TODO: Live data flow analysis does not work because of computation resources

[32] https://apkpure.com/, visited 06/05/2016

[33] https://apk-dl.com, visited 06/05/2016

4.2 Evaluation of the Automated Information Privacy Risk Assessment Tool

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

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

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

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.

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.

Köln, 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.

Cologne, 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