View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by AIS Electronic Library (AISeL)

Proceedings of the 52nd Hawaii International Conference on System Sciences | 2019

# No Risk, More Fun! Automating Breach of Confidentiality Risk Assessment for Android Mobile Health Applications

Thomas Brüggemann
blueworld GmbH
brueggemann@blueworld-gmbh.de

Tobias Dehling
Karlsruhe Institute of Technology
dehling@kit.edu

Ali Sunyaev
Karlsruhe Institute of Technology
sunyaev@kit.edu

## Abstract

*With the rapidly rising number of mobile health (mHealth) applications (apps), it is unfeasible to manually review mHealth apps for information privacy risks. One salient information privacy risk of mHealth apps are confidentiality breaches. We explore whether and how static code analysis is a feasible technology for app review automation. Evaluation of our research prototype shows that, on average, our prototype detected one breach of confidentiality risk more than human reviewers. Contributions are the demonstration that static code analysis is a feasible technology for detection of confidentiality breaches in mHealth apps, the derivation of eight generic design patterns for confidentiality breach risk assessments, and the identification of architectural challenges that need to be resolved for wide-spread dissemination of breach of confidentiality risk assessment tools. In terms of effectiveness, humans still outperform computers. However, we build a foundation for leveraging computation power to scale up breach of confidentiality risk assessments.*

## 1. Introduction

The market for mobile phone and tablet applications (apps) has grown extensively [1]. It is increasingly easier for companies and single developers to create unique apps that reach millions of users around the planet via digital app stores. This market growth also affects mobile health (mHealth) apps. mHealth apps support users in resolving health-related issues and try to remedy health-related information deficiencies [2]. However, mHealth apps also require users to reveal personal, health-related information to receive a tailored app experience. Users are concerned about their privacy when using smartphones apps since it often remains unclear how and where user information is sent, processed, and stored [3, 4]. Consequently, use of mHealth apps poses information privacy risks to users; in particular, use of mHealth apps poses breach of confidentiality risks [5, 6]. Prior to app use, it is challenging for users to assess what kind of private information apps collect. A more expedient approach would be to assess breach of confidentiality risks of apps on app store level to provide users with the desired information right where they need it [7]. But the high volume of available apps makes it laborious to review all of them by hand [1]. An automated solution is needed. Static code analysis is used to analyze large amounts of application source code and to detect faults or vulnerabilities [8]. Extant research yields no insights whether employing static code analysis for breach of confidentiality risk assessment is feasible and how it should be implemented to cover the detection of a holistic range of breach of confidentiality risks. Extant research applied static code analysis to limited subsets of breach of confidentiality scenarios and often in manual and non-automated ways [9, 10, 11, 12]. The objective of this study is to develop a prototypical breach of confidentiality risk detection tool for Android apps and to evaluate its performance in comparison to human source code reviewers.

While the proposed tool can work on an arbitrary set of Android apps, we use the context of mHealth apps since mHealth apps are by nature prone to breach of confidentiality risks. Breach of confidentiality takes place whenever private user information is unwillingly disclosed [13]. It violates the trust of users and impedes a fruitful vendor-to-customer relationship, which is the foundation for successful mHealth apps [14]. Users demand a 'trustworthiness measure' to support their app installation decisions at app store level [15]. Therefore, automating detection of breach of confidentiality risks is of high importance. Due to the high level of diversity among different breach of confidentiality risks to detect, implementation of such a tool is challenging. We propose an implementation-structure that enables separation of breach of confidentiality risk detection algorithms into individual strategy-components and answer the research question: To what degree can breach of confidentiality risk assessment for Android apps be automated? Our work contributes to the software engineer-

URI: https://hdl.handle.net/10125/59864
ISBN: 978-0-9981331-2-6
(CC BY-NC-ND 4.0)

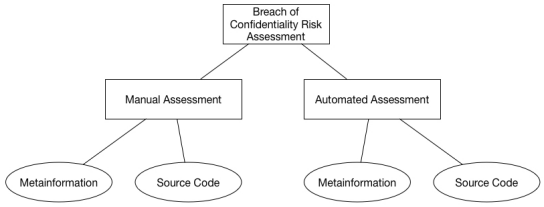



**Figure 1. Overview of extant research streams regarding breach of confidentiality risk assessment of apps.**

ing literature in three ways: First, we extract a holistic set of breach of confidentiality risks from extant literature. Second, we present an automated assessment tool that identifies the breach of confidentiality risks within the source code of Android apps. Third, we identify design patterns and challenges for building automated breach of confidentiality risk assessment tools for Android apps.

The paper is organized as follows: First, we outline extant research streams in the field of breach of confidentiality risk assessment. Second, the research approach of the study is outlined. We then present the implementation of our automated breach of confidentiality risk assessment tool and present eight generic design patterns that were identified through source code review during implementation. In the next section, we evaluate the performance of the automated breach of confidentiality risk assessment tool in comparison to human reviewers. We conclude this study with an outline of threats to validity and future research ideas.

## 2. Related Work

Extant research focuses on either manual or automated assessment of breach of confidentiality risks within apps, as depicted in Figure 1. Manual assessment can be categorized into two fields of interest. First, evaluation of metainformation regarding information privacy practices, such as privacy notices or user interface characteristics [16, 17]. Second, manual assessment conducted on technical aspects of apps, such as analyzing the source code or monitoring data connections [18, 19, 1, 9, 20, 21, 22, 23]. Research on automated breach of confidentiality risk assessment is sparse. Extant research on automated breach of confidentiality risk assessment can be classified into two categories. First, previous research focused on automatically detecting potential breach of confidentiality risks within metainformation provided by the app, for example, by applying text-classification and machine learning tools to privacy notices [24, 25, 26]. Automated assessment of metainformation is challenging because privacy notices are written in complex legal language and are difficult to process [27]. A second focus of automated assessment of breach of confidentiality risks has been put on source code analysis and data communication analysis of apps [25]. Researchers already use static code analysis and dynamic code analysis to perform automated source code assessments to identify breach of confidentiality risks. However, previous research focused only on single breach of confidentiality risks or special use cases. In this work, we identify a holistic set of breach of confidentiality risks of Android apps and do not narrow our scope to specific risks. Knowing to what degree automated breach of confidentiality risk assessment is possible opens up new research opportunities towards more transparent communication of breach of confidentiality risks.

## 3. Research Approach

Our research approach is structured in three steps: First, we identify relevant breach of confidentiality risks that can be detected in Android app source code from previous research. Second, we implement a breach of confidentiality risk assessment tool for Android apps to explore the boundaries of automated detection. Third, we evaluate the implementation by comparing the detection rate of breach of confidentiality risks with those of two human source code reviewers. We derived the breach of confidentiality risks, that are to be detected, from a catalog of information privacy practices that was developed based on a review of privacy notices of mHealth apps [28]. The content of a privacy notice informs users about the information privacy practices the associated app implements. Therefore, the catalog of information privacy practices forms the basis for our breach of confidentiality risk assessment. We reviewed the catalog of information privacy practices and excluded information privacy practices based on two criteria: (1) The information privacy practice does not represent a potential breach of confidentiality risk of an app (e.g., the information retention policy of an app provider). (2) The information privacy practice is infeasible to detect via a static code analysis of app source code (e.g., data handling and exchange on the app providers' servers). The review was conducted independently by two of the authors. Disputes were resolved in group discussions of all authors. The resulting breach of confidentiality risks are listed in Table 1.

We evaluated the performance of the breach of confidentiality risk assessment tool by comparing its output to the results of a manual static code analysis by two human reviewers. To ensure high-quality results for the

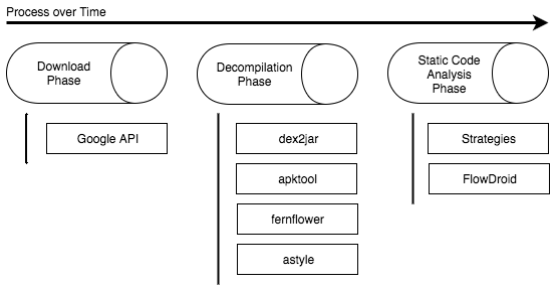



**Figure 2. Diagram of the three phases of the automated breach of confidentiality risk assessment tool over time.**

human code reviews, we chose two reviewers that were trained on the breach of confidentiality risk catalog and experienced in professional software development. The task for each reviewer was to identify as many breach of confidentiality risks as possible by analyzing the source code files of six sample apps.

# 4. AUTOMATED BREACH OF CONFIDENTIALITY RISK ASSESSMENT TOOL

We developed the automated confidentiality breach risk assessment tool (*ACCuRATe*). *ACCuRATe* operates in three phases, as depicted in Figure 2.

## 4.1. Download Phase

In the first phase, Android APK files are retrieved. We used an mHealth app repository that lists mHealth apps from the *Google PlayStore* as our main app data source [29]. We excluded apps that are not available for free. Free apps are more prone to breach of confidentiality risks due to the increased use of business models based on advertising [12, 30]. Nevertheless, *ACCuRATe* will be able to perform its assessment on either free or non-free apps. To download the APK files, we used an open source implementation of an undocumented *Google PlayStore* API [31]. The undocumented part of the Google API allows authenticated users to download APK files via HTTP requests, as if the files were downloaded by the *Google PlayStore* app on an Android device.

## 4.2. Decompilation Phase

The decompilation script uses a chain of tools to gain access to the source code from an APK file. First, we used the tool *dex2jar* [32] to extract the Java Archive (JAR) file from the APK file. *dex2jar* can read and write Dalvik EXcecutionable (DEX) and Java class files. The next step extracts resource files, such as the Android Manifest file from the APK file. The Android Manifest contains metainformation about the app in a structured Extensible Markup Language (XML) format [11, 12]. The metainformation includes the package name of the app and the requested permissions, such as camera or geolocation usage. To extract the Android Manifest file from the APK file, *ACCuRATe* uses the *apktool* [33]. Another important outcome of the extraction of resource files is retrieving the layout and localization files. These files include information on user interface components used within the app and content of labels and text fields. This information is used within *ACCuRATe* to trace information collection data flows.

An obstacle in decompiling Java source code is obfuscation. Obfuscation is a security feature that tries to hide the logic of Java code by renaming classes, variables, and method names by disassembling the code into pieces that are difficult to interpret for a human reader [12]. The core of the decompilation process is a tool called *fernflower* [1, 34], which is equipped with methods to cope with obfuscated source code to a certain degree. The result of the decompilation phase is a directory labeled 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.

## 4.3. Static Code Analysis Phase

First, an Android data flow analysis tool extracts potential data flows from the APK files. The data flow analysis is performed using the open source tool *FlowDroid* [35]. A source is the origin of a data flow (e.g., user input via a text field). A sink is the destination of a data flow. An example for a sink is an HTTP internet connection or a local log file.

In the second preprocessing step, a machine learning text classifier was trained. We used a Naïve Bayes classifier to segment text into distinct categories [36]. The categories are predefined in the training phase and later assigned new, to previously unseen text segments by the classifier. We used the Naïve Bayes classifier to classify Uniform Resource Locators (URLs) into categories of data recipients to identify where data is sent to. The URL categories were identified based on a literature review [28]. We acquired URLs in these categories from multiple online URL lists [37, 38]. To acquire metainformation for all these URLs, we implemented a download tool for the HTML source code of the URLs and stored the content of the description HTML metatag in a file [39]. We used this metainformation to train the Naïve Bayes classifier with the as-

sociated URL categories. After these preparatory steps, the main static code analysis performs the breach of confidentiality analysis by iterating over all available apps and applying a set of analysis operations, that is, strategies, to the source code of the apps.

### 4.4. Specific & Generic Strategies

Within *ACCuRATe*: two types of strategies are used—generic and specific strategies. Specific strategies assess a single breach of confidentiality risk, as listed in in Table 1. During the development process, generic patterns emerged that we consolidated into generic strategies. Each specific strategy either uses or inherits functionality from at least one generic strategy to perform its risk-detection functionality.

We describe each generic strategy along the following five components for commmunicating software patterns: Identification, Context, Problem, Solution, and Consequences [40]. The component Identification expresses the name of the design pattern. The purpose of the Context component is to demonstrate the situation that poses a problem. The Problem component explains the forces [40] that the design pattern tackles. The solution reveals how these forces can be overcome Finally, the Consequences component describes the state after the application of the design pattern.

**Table 1. Generic strategies of ACCuRATe and the corresponding specific strategies**

| Generic Strategies | Specific Strategies |
|---|---|
| Existence | LocalStorage, Cookies, Microphone, NetworkConnectionSensor, WiFiSensor, Surveys, FingerprintScanner, Camera, NearFieldCommunication, GPSSensor, OtherUserDeviceStorage, CloudStorage, SharingWithPublic, TextInformation, SecurityDuringProcessing, Purchases, SecurityDuringTransfer, SharingWithOtherUsers, SharingWithUnrelated, SharingWithAnalyst, SharingWithAdvertiser |
| Permissions | BluetoothSensor, LocalStorage, OtherUserDeviceStorage, Camera, NearFieldCommunication, Fingerprint-Scanner |
| Input | TextInformation |
| TraceBack | GPSSensor, ProviderStorage, Location, VideoInformation, ImageInformation, OnlineContacts, SecurityDuringStorage |
| ProviderURL | ProviderStorage |
| URLCategory | ThirdPartyStorage, SharingWithGovernment, SharingWithAggregator, SharingWithDelivery |
| Input-Information-Collection | Health, Demographics, GovernmentIdentifier, FinancialIdentifier, Name, OnlineContact, PhysicalContact, Preferences, Ideology, OwnUniqueIdentifier, UserDevice |

**DataFlowStrategy** Context: The DataFlowStrategy can be used, whenever all data flows within the source code of an app need to be identified. This helps in detecting leaks of private information. A data flow analysis identifies data flow connections between sources and sinks via a constructed call-graph [35]. A source is the origin of a data flow. A sink is the destination of a data flow. Problem: Data flows from sensitive input sources (e.g., user input fields) may be leaked. To uncover these malicious data flows, a traversable call graph needs to be generated. Solution: The DataFlowStrategy parses the pre-extracted data flow XML from the *FlowDroid* preparation step and allows iteration over all identified data flow sources and sinks. The DataFlowStrategy filters the sources and sinks for certain search words and provides feedback if the search words were found within sources and sinks. The filtering of the sources of sinks for given search words can be seen in lines 6 and 15 of Listing 1. Consequences: A data flow analysis via the computed call graph is possible and potential breaches of confidentiality can be detected.

**ExistenceStrategy** Context: Some breach of confidentiality risks can be detected by the pure existence of certain application programming interfaces (APIs) or software libraries within the source code of an app. Problem: Potentially risky source code fragments can be identified by checking for existing interface or library usage (e.g., the existence of an analytics library within the source code of an app). No search methodology for code fragments in source code is initially available when implementing a static code analysis. Solution: The ExistenceStrategy is shown in listing 2. This strategy scans the full source code of an app and collects source code lines that match a given search pattern. Consequences: Fast risk detection is possible by searching for potential breach of confidentiality risks within the source code.

```
MultiMap<SerializedSinkInfo, SerializedSourceInfo> results =
      this.app.dataflow.getResults();

for (SerializedSinkInfo sink : results.keySet()) {
 if (this.params.containsKey("sinkIncludes")) {
  for (String searchTermSink : (LinkedList<String>)
        this.params.get("sinkIncludes")) {
   if (sink.toString().contains(searchTermSink)) return new
         StrategyResult(StrategyResultCertainty.HIGH, true) }}

 for (SerializedSourceInfo source : results.get(sink)) {
  if (this.params.containsKey("sourceIncludes")) {
   for (String searchTermSource : (LinkedList<String>)
         this.params.get("sourceIncludes")) {
    if (source.toString().contains(searchTermSource)) return
          new StrategyResult(StrategyResultCertainty.HIGH,
          true); }}}}
```

**Listing 1. The generic strategy DataFlowStrategy.java parses pre-extracted data flow XML files.**

```
for (String file : files) {
 FileScanner scanner = new FileScanner(file);
 try {
  LinkedList<Snippet> snippets = new LinkedList<Snippet>();
  for (String searchTerm : (LinkedList<String>)
        this.params.get("searchFor")) {
   snippets.addAll(scanner.scan(searchTerm)); }

  if (snippets.size() > 0) return new
        StrategyResult(StrategyResultCertainty.HIGH, true,
        snippets);
 } catch (FileNotFoundException e) {} }
```

**Listing 2. The generic strategy ExistanceStrategy.java to find the existance of search words within source code.**

**InputStrategy** Context: Information regarding user input fields of an app is of interest for detecting breach of confidentiality risks. Users can input sensitive information into input fields while it remains unclear where the information is sent, stored, or processed. Problem: Initially, there is no overview of all user input fields and their metainformation (e.g., hint texts) within an app when analyzing the source code. Solution: The InputStrategy iterates over all XML layout configuration files of an app. The InputStrategy attempts to identify user input fields by scanning the layout files for the search terms 'EditText', 'AutoCompleteTextView', 'CheckBox', 'RadioButton', and 'RadioGroup'. The input fields metainformation contain the user interface control 'id', a 'hint' field, and a 'text' field. Consequences: A clear overview over all user input fields that are usually spread across the source code exists. This overview allows for further analysis of user input field data, especially, since the InputStrategy extracts metainformation for each input field.

**TraceBackStrategy** Context: For the breach of confidentiality risk assessment, it is of interest to trace the data flow from a specific code fragment to any information sink. Such data flows could potentially leak private information without user consent. Problem: It is challenging to trace code fragments to certain sinks when

```
for (String file : files) {
 FileScanner scanner = new FileScanner(file);
 try {
  for (String inputField : INPUT_FIELDS)
        snippets.addAll(scanner.scan(inputField));
  if (snippets.size() > 0)
        metaInfos.addAll(this.extractInputMeta(file));
 } catch (FileNotFoundException e) {} }

if (metaInfos.size() > 0) {
 StrategyResult result = new
       StrategyResult(StrategyResultCertainty.HIGH, true,
       snippets);
 result.extra.put("meta", metaInfos);
 return result; }
```

**Listing 3. The InputStrategy.java identifies user input fields and extracts their meta information.**

```
1  Iterator<Edge> edges = this.app.callgraph.iterator();
2  while (edges.hasNext()) {
3   Edge edge = (Edge) edges.next();
4   boolean isStartEdge = false;
5
6   for (String startSink : startSinks) {
7    if (edge.toString().contains(startSink)) {
8     isStartEdge = true; break; }}
9
10  for (String startSinkInverted : startSinksInverted) {
11   if (!edge.toString().contains(startSinkInverted)) {
12    isStartEdge = true; break; }}
13
14  if (isStartEdge) {
15   sinksFound++; boolean foundInCallstack = false;
16   SootMethod tgt = edge.tgt();
17   LinkedList<SootMethod> callers = this.traceBack(tgt);
18         for (SootMethod method : callers) {
19    String mth = method.toString().toLowerCase();
20              for (String test : (LinkedList<String>)
                     this.params.get("searchFor")) {
21     if (mth == null || test == null) continue;
22     if (mth.contains(test.toLowerCase())) {
23      if (foundInCallstack == false) foundInCallstack = true;
24      snippets.add(new
              Snippet(method.getDeclaringClass().getName() +
              ".java", method.toString(),
              method.getJavaSourceStartLineNumber())); }}}}}
```

**Listing 4. Excerpt of the generic strategy TraceBackStrategy.java to trace defined information sources to information sinks via data flow analysis.**

conducting a static code analysis. Solution: 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, as seen in line 22 of listing 4. *ACCuRATe* mainly uses the TraceBackStrategy to find data flows that result in information collection. We define information collection as a data flow that results in storing the information either locally on the device or a data flow that results in sending the information to a remote server. Consequences: After applying the TraceBackStrategy it is possible to search for all information collection data flows of a given code fragment and identify data flows that end in information sharing subroutines. An example usage of this strategy is to detect if any data flows from the Android microphone API ends up in information collection sinks.

```
1  InputStrategy is = new InputStrategy(); is.app = this.app;
2  StrategyResult isResult = is.execute();
3  LinkedList<EditTextMeta> metaTexts =
       InputStrategy.searchMetaFor(isResult, identifiers);
4  LinkedList<String> metaIdTargets = new LinkedList<String>();
5
6  for (EditTextMeta metaText : metaTexts) {
7   metaIdTargets.add(metaText.Id); }
8
9  TraceBackStrategy tbsMeta = new TraceBackStrategy();
10 tbsMeta.app = this.app;
11 tbsMeta.params.put("startSink",
       TraceBackStrategy.INFORMATION_COLLECTION_SINKS);
12 tbsMeta.params.put("searchFor", metaIdTargets);
13 return tbsMeta.execute();
```

**Listing 5. The InputInformationCollectionStrategy takes the user input fields analysis and tries to identify information collection data flows.**

**InputInformationCollectionStrategy** Context: The InputInformationCollectionStrategy can be used when the identification of information collection data flows that contain values from the user input fields of the app is desired. Problem: The InputInformationCollectionStrategy counteracts the forces that sensitive user information might be leaked via an information collection sink and that these user input data flows are not detectable without an appropriate strategy. Solution: 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 and traces back the call graph in an attempt to identify the user input field 'ids' within the call graph path (see lines 11-13 of listing 5). Consequences: If a user input field is found within the call graph that leads to an information collection sink, the InputInformationCollectionStrategy identified potentially malicious information sharing of user input information.

**PermissionStrategy** Context: It is required for Android apps to declare permissions to use certain features, such as GPS location or internet access, within the AndroidManifest file [41]. Problem: The PermissionStrategy counteracts the force that there is no overview of declared Android app permissions when implementing a static code analysis for breach of confidentiality risk assessment. Solution: 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.[1] The PermissionStrategy enables a search through these permissions by a given search pattern (see line 7 of listing 6). Consequences: An overview of the declared permissions within the app is available and searchable. Feature usage such as the camera, microphone, or geolocation can easily be detected via this strategy.

[1]See https://web.archive.org/web/20160425141027/https://developer.android.com/training/permissions/declaring.html, visited 06/14/18

```
1  String manifest = this.app.getManifestFile();
2  FileScanner scanner = new FileScanner(manifest);
3  LinkedList<Snippet> snippets = new LinkedList<Snippet>();
4
5  for (String searchTerm : (LinkedList<String>)
       this.params.get("searchFor")) {
6   try { snippets.addAll(scanner.scan(searchTerm));}
7     catch (FileNotFoundException e) {} }
8
9  if (snippets.size() > 0) return new
       StrategyResult(StrategyResultCertainty.HIGH, true,
       snippets);
```

**Listing 6. The generic strategy PermissionStrategy.java to find the existence of Android permissions declarations.**

**ProviderURLStrategy** Context: It is generally questionable if the app provider collects personal user information just to tailor app experience to the user needs or if the app provider shares the sensitive information with third parties. To detect data connections to servers of the app provider, the ProviderURLStrategy can be used. Problem: Each data connection from an app either targets a hostname or an IP address. In case a hostname is used, it is unclear if the hostname belongs to a server of the app provider or not. Solution: The ProviderURLStrategy iterates over all extracted URLs found within the source code and checks the similarity between the URL host in comparison to the app package name (see line 8 of listing 7). We observed that 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 (see line 12 of listing 7. Consequences: After applying the ProviderURLStrategy, we gain knowledge about potential communication of an app with its app provider via a data connection.

**URLCategoryStrategy** Context: Information sharing of apps is often indicated by data flows to Uniform Resource Locators (URL). This is, however, too detailed information for automated breach of confidentiality risk assessment because risk scores would have to be identified for all possible web resources. The URLCategoryStrategy tackles this problem by categorizing URLs into data recipient categories. Problem: When categorizing URLs into data recipient categories, one must consider that URLs do not yield information on the roles of data recipients. Solution: Making use of URLs to automatically identify data recipients with whom apps share information. Newly encountered URLs must be

```
StrategyResult result = new StrategyResult();
String[] packageNameParts =
      this.app.getPackageName().split("\\.");

for (AppUrl url : this.app.urls) {
 for (String packageNamePart : packageNameParts) {
     if (packageNamePart.length() > 3) {
   for (String urlPart : url.url.split("/")) {
    int similarity = StringAnalyzer.isSimilar(urlPart,
          packageNamePart);
    double similarityScore = 1.0 - (double) similarity /
          (double) Math.max(url.url.length(),
          packageNamePart.length());
                if (similarityScore > 0.9) {
     result.extra.put("url", url.url);
     result.found = true;
     result.certainty =
           StrategyResultCertainty.fromDouble(similarityScore);
     result.snippets.add(url.snippet);
     return result; }}}}}
```

**Listing 7. The generic strategy ProviderURLStrategy.java tries to identify URL connections to the app provider domain.**

```
double probs = 0.0; int probCount = 0;
LinkedList<Snippet> snippets = new LinkedList<Snippet>();

if (!this.params.containsKey("searchFor")) return new
      StrategyResult(StrategyResultCertainty.HIGH, false);

for (AppUrl appUrl : this.app.categorizedUrls) {
 if (appUrl.category == (String)
       this.params.get("searchFor")) {
  probs += appUrl.certainty;
  probCount++;
  snippets.add(appUrl.snippet); }}

if (probCount > 0) return new
      StrategyResult(StrategyResultCertainty.fromDouble(probs
      / (double) probCount), true, snippets);
```

**Listing 8. The URLCategoryStrategy.java queries for the existance of URLs of a specific category within the app source code.**

categorized into data recipient categories. *ACCuRATe* uses the categories advertiser, delivery services, government, instant messaging, (data) aggregation services, search engines, and social networks [28]. URLs are categorized based on information stored in the description HTML metatag for the respective URL. This information is used as input for a Naïve Bayes classifier which assigns URLs to the recipient categories based on training data. During app assessment, the preprocessing allows for easy retrieval of data recipients by querying what recipient categories were assigned to a certain URL. Consequences: Applying the URLCategoryStrategy requires some effort for preprocessing URLs and categorizing them into data recipient categories. *ACCuRATe* employs the URLCategoryStrategy to establish an overview of all data recipients of an app and calculate associated breach of confidentiality risk scores.

# 5. Evaluation

## 5.1. Download Phase

The original dataset from the repository of app store listings contains 5,379 app entries from the Google PlayStore in the category 'Medical' and 'Health and Fitness' [29]. From this 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 server-error or were no longer available on the Google PlayStore.

## 5.2. Decompilation Phase

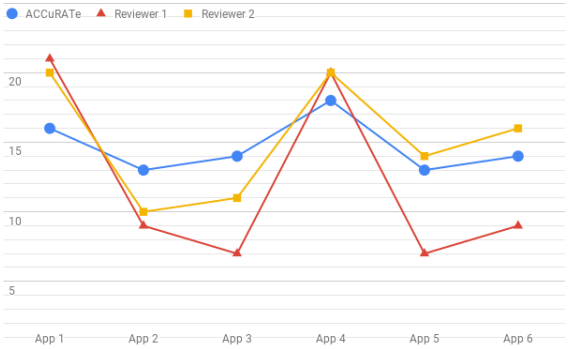
We decompiled 355 of the 2,250 downloaded APK files. Those 355 APK files were selected in order of their file size. Decompilation failed in 24 cases. The decompilation time varied from 16 minutes to 54 minutes and is dependent on the amount and complexity of the source files of the app. Reasons for decompilation failure were heavily obfuscated source code and memory exceptions. The decompilation was tested on a laptop computer with 16 gigabytes of main memory as well as an Amazon Cloud virtual machine instance with 64 gigabytes of main memory. Given that each of the assessed APK files were under 50 megabytes in size, the memory consumption during decompilation was unreasonably high in some cases and demands further investigation before this prototype can be deployed at larger scale.

## 5.3. Static Code Analysis Phase

The static code analysis phase uses a part of the *FlowDroid* library that only computes the call graphs of apps with a limited subset of the analysis features that *FlowDroid* offers. This configuration was chosen by suggestion of the *FlowDroid* authors and has proven to be sufficient for the analysis purposes of this study since it uses less hardware resources during computation.

The static code analysis tool completed the analysis on 317 of the 355 decompiled apps. The reason for the roughly 10% loss from the decompilation to the static code analysis phase is due to errors in the generation of the call graph via *FlowDroid*. The results of *ACCuRATe* reveal that 14 of the 44 implemented strategies did not find any confidentiality breaches within the analysis phase, 30 did. On average, 4.86 potential confidentiality breaches were found per app via the implemented strategies.

Figure 3 shows the results of the two human reviewers that inspected the source code of six test-apps for confidentiality breaches as well as the results of *ACCu-*




*App 1 = com.siyami.apps.patientregister v9.0.0.0,*
*App 2 = com.medicaljoyworks. prognosis.questions v2.1,*
*App 3 = com.szyk.myheart v2.4.24,*
*App 4 = cz.ulikeit.lifesquare v1.3.2,*
*App 5 = com.medicaljoyworks.prognosis.questions v5.0.0,*
*App 6 = wikem.chris v5.4.2*

**Figure 3. Number of detected breach of confidentiality risks within ACCuRATe and the average of the two human reviewers.**

*RATe*. On average, *ACCuRATe* detected one breach of confidentiality risk more than the human reviewers.

**5.3.1. Benefits & Drawbacks** *ACCuRATe* yields at least the following five benefits in comparison to app assessment by human reviewers: Speed: *ACCuRATe* performs the breach of confidentiality risk assessment faster than human reviewers. While it takes a human hours to complete a thorough code analysis and risk assessment, *ACCuRATe* performs the same task in a matter of seconds. Consistency: As seen in Figure 3, the results of the human reviewers vary among the two humans and are not consistent. *ACCuRATe* on the other hand will always return a consistent result since it is not influenced by situational factors such as skills, mood, or age of human reviewers. Availability: *ACCuRATe* can perform its breach of confidentiality risk assessment task 24 hours a day and seven days a week, while a human reviewer is tied to working hours or other time constraints. Costs: In comparison to employing human reviewers for manual risk assessment of apps, *ACCuRATe* can be provided at favorable costs. Computation power is less expensive than human labor and prices for computing power continuously diminish [42]. Scalability: Risk assessment can be scaled easily with *ACCuRATe*. In comparison, scaling human app assessment is cumbersome. Humans need to be trained on static code analysis and the catalog of breach of confidentiality risks to detect.

Human reviewers have at least one advantage over *ACCuRATe*—understanding or anticipating the context of source code based on their experience. For example, *ACCuRATe* did not identify the collection of health information in the test app 'com.siyami.apps.patientregister' because the matching source code was heavily obfuscated and therefore metainformation on context was scarce. The human reviewers could identify the collection of health information because they were able to interpret the context in which the obfuscated code was used based on their experience. Another example where the human reviewers were superior to *ACCuRATe* was the assignment of unique identifiers to users. A unique user identification can be a randomly generated string, the email address of the user or any other unique user characteristic. A human reviewer has the advantage to interpret variable names and the context that variables are used in.

**5.3.2. Design Challenges** We identified three design challenges during the implementation of *ACCuRATe* that need to be resolved to facilitate automated breach of confidentiality risk assessment at app store scale. First, the acquisition of app binaries or source code is cumbersome. App binaries were downloaded from the *Google PlayStore* via an undocumented API. The download was unreliable and failed in some cases. To be able to increase the assessment scale of *ACCuRATe*, app stores need to provide a more reliable way to access app source code. Second, some app source code is heavily obfuscated. As a countermeasure, app stores could reject app submissions if their source code is heavily obfuscated and, thereby, support decompilation for risk assessment purposes. Third, the capability of *FlowDroid* to generating call graphs is dependent on its configuration and the examined APK file size. To run *ACCuRATe* at app store scale, it is necessary to either improve *FlowDroid* to lower main memory consumption, to run *ACCuRATe* on machines with extensive hardware resources, or to limit the maximum file size of APK files submitted to app stores. Minimizing the APK file size should also be of interest to the app vendors since downloading apps on smartphones via cellular networks is slow and consumes the available data volume quickly.

## 6. Threats to validity

First, the 'detection rate', in our example, is heavily influenced by the knowledge and skill-set of the two human reviewers that conducted the manual code analysis, however, they were both experts in coding of mHealth apps. Future research should increase the amount of human reviewers. A more uniform distribution of human

reviewer skills and training is desirable to enhance the findings of this research. Second, we chose to only assess free apps and excluded paid apps. The reason for this approach is the high costs for downloading large numbers of paid apps to perform breach of confidentiality assessments. Moreover, free apps are more prone to breach of confidentiality risks due to the increased use of business models based on advertising [12, 30]. Additionally, over 95% of the installed apps on Android smartphones are free apps [15]. Third, the current implementation of *ACCuRATe* automatically excludes apps that could not be decompiled due to heavy obfuscation. Future research should investigate whether these apps should be flagged with high scores for breach of confidentiality risks since they are prone to hide functionality; thus, they may bear a greater risk for misuse of user information.

## 7. Conclusion & Future Research

This study answers the question to what degree the breach of confidentiality risk assessment of Android mHealth apps can be automated by identifying generic assessment strategies and evaluating their effectiveness. In the growing market of apps, especially mHealth apps, information privacy risks are becoming more prevalent. Due to the large number of mHealth apps in the app stores and its continuous growth, manual review is infeasible. We introduced a tool that requires little human configuration and is able to download, decompile, and analyze Android apps with respect to their breach of confidentiality risks. The *ACCuRATe* source code is publicly available.[2] The results of this study, the design patterns, and the insights on the limitations and benefits of the static code analysis for automated breach of confidentiality assessment serve as foundation for future research on privacy-enhancing technologies in the mHealth domain. A major factor for improvement are the hardware resources required for decompilation and source code analysis. Future research could use machines with more extensive hardware resources to run the decompilation and analysis phases on more apps. Future research could also enhance the current implementation in a way so that it identifies and highlights risky parts of the source code to be additionally reviewed by a human reviewer. We encourage future research on the integration of automated breach of confidentiality risk assessment into app stores. Future research could analyze the effects of integrating an *ACCuRATe* that performs a static code analysis right after a developer submits an app and that transparently communicates the breach of confidentiality risks within the source code to the users. Breaches of confidentiality only become relevant to users if they perceive them; but they suffer from them all to often. *ACCuRATe* enables users to avoid privacy violations and to embrace there privacy preferences.

[2] https://github.com/thomasbrueggemann/ACCuRATe, visited 09/02/17

## References


[1] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," in *Proceedings of the USENIX Conference on Security*, p. 21, 2011.

[2] S. Kumar, W. Nilsen, M. Pavel, and M. Srivastava, "Mobile Health: Revolutionizing Healthcare Through Transdisciplinary Research," *IEEE Computer*, vol. 46, no. 1, pp. 28–35, 2013.

[3] D. He, M. Naveed, C. A. Gunter, and K. Nahrstedt, "Security Concerns in Android mHealth Apps," in *Proceedings of the AMIA Annual Symposium*, pp. 645–654, 2014.

[4] C. Chen, D. Haddad, J. Selsky, J. E. Hoffman, R. L. Kravitz, D. E. Estrin, and I. Sim, "Making Sense of Mobile Health Data: An Open Architecture to Improve Individual- and Population-Level Health," *Journal of Medical Internet Research*, vol. 14, no. 4, p. e112, 2012.

[5] K. Degirmenci, N. Guhr, and M. H. Breitner, "Mobile Applications and Access to Personal Information: A Discussion of Users' Privacy Concerns," in *Proceedings of the International Conference on Information Systems*, 2013.

[6] T. Dehling and A. Sunyaev, "Secure Provision of Patient-Centered Health Information Technology Services in Public Networks—Leveraging Security and Privacy Features Provided by the German Nationwide Health Information Technology Infrastructure," *Electronic Markets*, vol. 24, no. 2, pp. 89–99, 2014.

[7] L. Cranor, S. Egelman, J. Tsai, and A. Acquisti, "The Effect of Online Privacy Information on Purchasing Behavior: An Experimental Study," in *Proceedings of the International Conference on Information Systems*, pp. 254–268, 2007.

[8] D. Baca, B. Carlsson, and L. Lundberg, "Evaluating the Cost Reduction of Static Code Analysis for Software Security," in *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pp. 79–88, 2008.

[9] M. Haris, H. Haddadi, and P. Hui, "Privacy Leakage in Mobile Computing: Tools, Methods, and Characteristics," *arXiv PrePrint*, 2014. eprint 1410.4978.

[10] J. Kim, Y. Yoon, K. Yi, and J. Shin, "Scandal: Static Analyzer for Detecting Privacy Leaks in Android Applications," in *Proceedings of the IEEE Workshop on Mobile Security Technologies*, 2012.

[11] A. Shabtai, Y. Fledel, and Y. Elovici, "Automated Static Code Analysis for Classifying Android Applications Using Machine Learning," in *Proceedings of the International Conference on Computational Intelligence and Security*, pp. 329–333, 2010.

[12] L. Xu, *Techniques and Tools for Analyzing and Understanding Android Applications*. PhD thesis, University of California, Davis, CA, USA, 2013.

[13] D. J. Solove, "A Taxonomy of Privacy," *University of Pennsylvania Law Review*, vol. 154, no. 3, pp. 477–564, 2006.

[14] T. Dehling, F. Gao, S. Schneider, and A. Sunyaev, "Exploring the Far Side of Mobile Health: Information Security and Privacy of Mobile Health Apps on iOS and Android," *JMIR mHealth and uHealth*, vol. 3, no. 1, p. e8, 2015.

[15] E. Chin, A. P. Felt, V. Sekar, and D. Wagner, "Measuring User Confidence in Smartphone Security and Privacy," in *Proceedings of the Symposium on Usable Privacy and Security*, 2012.

[16] R. Adhikari, D. Richards, and K. Scott, "Security and Privacy Issues Related to the Use of Mobile Health Apps," in *Proceedings of the Australasian Conference on Information Systems*, 2014.

[17] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What Do Mobile App Users Complain About?," *IEEE Software*, vol. 32, no. 3, pp. 70–77, 2015.

[18] A. G. Bardas, "Static Code Analysis," *Journal of Information Systems & Operations Management*, vol. 4, no. 2, pp. 99–107, 2010.

[19] C. D'Orazio and K.-K. R. Choo, "A Generic Process to Identify Vulnerabilities and Design Weaknesses in iOS Healthcare Apps," in *Proceedings of the Hawaii International Conference on System Sciences*, pp. 5175–5184, 2015.

[20] K. Huckvale, J. T. Prieto, M. Tilney, P.-J. Benghozi, and J. Car, "Unaddressed Privacy Risks in Accredited Health and Wellness Apps: A Cross-Sectional Systematic Assessment," *BMC Medicine*, vol. 13, no. 1, 2015.

[21] K. Knorr and D. Aspinall, "Security Testing for Android mHealth Apps," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops*, 2015.

[22] J. McClurg, J. Friedman, and W. Ng, "Android Privacy Leak Detection via Dynamic Taint Analysis," tech. rep., Northwestern University, Evanston, IL, USA, 2012.

[23] S. Mitchell, S. Ridley, C. Tharenos, U. Varshney, R. Vetter, and U. Yaylacicegi, "Investigating Privacy and Security Challenges of mHealth Applications," in *Proceedings of the Americas Conference on Information Systems*, pp. 2166–2174, 2013.

[24] L. Yu, T. Zhang, X. Luo, L. Xue, and H. Chang, "Toward Automatically Generating Privacy Policy for Android Apps," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 4, pp. 865–880, 2017.

[25] S. Zimmeck, Z. Wang, L. Zou, R. Iyengar, B. Liu, F. Schaub, S. Wilson, N. Sadeh, S. M. Bellovin, and J. Reidenberg, "Automated Analysis of Privacy Requirements for Mobile Apps," in *Proceedings of the Network and Distributed System Security Symposium*, 2017.

[26] F. Schaub, T. D. Breaux, and N. Sadeh, "Crowdsourcing Privacy Policy Analysis: Potential, Challenges and Best Practices," *it - Information Technology*, vol. 58, no. 5, pp. 229–236, 2016.

[27] A. Sunyaev, T. Dehling, P. L. Taylor, and K. D. Mandl, "Availability and Quality of Mobile Health App Privacy Policies," *Journal of the American Medical Informatics Association*, vol. 22, no. e1, pp. e28–e33, 2015.

[28] T. Dehling, F. Gao, and A. Sunyaev, "Assessment Instrument for Privacy Policy Content: Design and Evaluation of PPC," in *Proceedings of the Pre-ICIS Workshop on Information Security and Privacy*, 2014.

[29] W. Xu and Y. Liu, "mHealthApps: A Repository and Database of Mobile Health Apps," *JMIR mHealth and uHealth*, vol. 3, no. 1, p. e28, 2015.

[30] N. Vallina-Rodriguez, J. Shah, A. Finamore, Y. Grunenberger, K. Papagiannaki, H. Haddadi, and J. Crowcroft, "Breaking for Commercials: Characterizing Mobile Advertising," in *Proceedings of the Internet Measurement Conference*, pp. 343–356, 2012.

[31] E. Girault, "Google Play Unofficial Python API." `https://web.archive.org/web/20140920061441/https://github.com/egirault/googleplay-api`.

[32] B. Pan, "dex2jar." `https://web.archive.org/web/20160614022230/https://github.com/pxb1988/dex2jar`.

[33] R. Wiśniewski and C. Tumbleson, "ApkTool." `https://web.archive.org/web/20170107092036/https://ibotpeaches.github.io/Apktool/`.

[34] R. Shevchenko and I. Ushakov, "Fernflower." `https://web.archive.org/web/20170227204453/https://github.com/fesh0r/fernflower`.

[35] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[36] S. L. Ting, W. H. Ip, and A. H. C. Tsang, "Is Naïve Bayes a Good Classifier for Document Classification?," *International Journal of Software Engineering and Its Applications*, vol. 5, no. 3, pp. 37–46, 2011.

[37] URLBlacklist, "urlblacklist.com." `https://web.archive.org/web/20160617050003/http://urlblacklist.com/?sec=download`.

[38] ProgrammableWeb, "programmableweb.com." `https://web.archive.org/web/20161229042313/https://www.programmableweb.com/`.

[39] T. P. Turner and L. Brackbill, "Rising to the Top: Evaluating the Use of the HTML META Tag to Improve Retrieval of World Wide Web Documents through Internet Search Engines," *Library Resources & Technical Services*, vol. 42, no. 4, pp. 258–271, 1998.

[40] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-Oriented Software Architecture*. Chichester, England: John Wiley & Sons Ltd., 5 ed., 2007.

[41] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 627–637, 2011.

[42] R. J. Gordon, "Does the "New Economy" Measure up to the Great Inventions of the Past?," *Journal of Economic Perspectives*, vol. 14, no. 4, pp. 49–74, 2000.