Cooperative static analysis of Android applications

by

Felix Pauck

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Fakultät für Elektrotechnik, Informatik und Mathematik
Heinz Nixdorf Institut und Institut für Informatik
Fachgebiet Softwaretechnik
Warburger Straße 100
33098 Paderborn

Cooperative static analysis of Android applications

Master's Thesis

Submitted to the Software Engineering Research Group
in Partial Fulfillment of the Requirements for the
Degree of
Master of Science

by
Felix Pauck
Warburger Str. 52
33098 Paderborn

Thesis Supervisor:
Prof. Dr. Heike Wehrheim
and
Prof. Dr. Eric Bodden

Paderborn, May 2017

Declaration

(Translation from German)

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Original Declaration Text in German:

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

City, Date

Signature

Contents

1 Introduction **1**
- 1.1 Approach . . . 2
- 1.2 Thesis' Contents . . . 2

2 Fundamentals **5**
- 2.1 Android . . . 6
  - 2.1.1 Components . . . 6
  - 2.1.2 Permissions . . . 7
  - 2.1.3 Inter-Component Communication . . . 8
  - 2.1.4 Manifest . . . 9
  - 2.1.5 Running Example (1/3): The Scenario . . . 9
- 2.2 Analyses . . . 13
  - 2.2.1 Information Flow Analyses . . . 13
  - 2.2.2 Challenges & Solutions . . . 14
  - 2.2.3 Tools . . . 15
  - 2.2.4 Running Example (2/3): Cooperative Analysis . . . 17

3 Conceptual Design **23**
- 3.1 Analysis Query Language (AQL) . . . 23
  - 3.1.1 AQL-Questions . . . 24
  - 3.1.2 AQL-Answers . . . 26
  - 3.1.3 Attributes . . . 28
  - 3.1.4 AQL-Operators . . . 29
  - 3.1.5 AQL-Queries . . . 30
- 3.2 AQL-System . . . 32
  - 3.2.1 Configuration . . . 32
  - 3.2.2 Workflow . . . 34
- 3.3 AQL Syntax . . . 37
  - 3.3.1 Syntax of AQL-Queries . . . 37
  - 3.3.2 Syntax of AQL-Answers . . . 42
  - 3.3.3 Running Example (3/3): AQL in Practice . . . 47

4 Implementation **55**
- 4.1 Overview . . . 55
  - 4.1.1 Configuration . . . 55
  - 4.1.2 Implementation Details . . . 58

4.2 Structure 60
4.3 Manual 63

5 Evaluation **67**
5.1 RQ1a: Can analysis tools combined through the AQL be more precise than immature tools? 67
5.2 RQ1b: Can analysis tools combined through the AQL be more precise than mature tools? 70
5.2.1 Annotations 76
5.3 RQ2: Can the power of a single analysis tool be increased by means of the AQL? 77
5.4 RQ3: Is the AQL-System capable of analyzing sets of one or more real world apps efficiently? 79
5.5 RQ4: Can the AQL improve the performance of an analysis? . . . 81
5.6 Summary 83

6 Conclusion **85**
6.1 Summary 85
6.2 Future Work 86

Appendix

A XML Schema Definitions (XSDs) **89**
A.1 AQL-Answer XSD 89
A.2 Configuration XSD 93

B Digital Appendix **95**

Bibliography **97**

List of Figures

2.1 Android Activity Lifecycle 7
2.2 Running Example Overview 10

3.1 AQL-System Overview 32
3.2 AQL-System Workflow 34
3.3 AQL-Answer as Graph 53

4.1 AQL-System: UML Class Diagram 61
4.2 AQL-System: UML Sequence Diagram 62
4.3 Screenshot of AQL-System's GUI 65

5.1 Screenshot of PAndA^2's result (SimpleApp) 69
5.2 Workflow of the FD+IC3-System 71
5.3 Workflow of the IccTA-System 71
5.4 Workflow of AQL-Comparer 72
5.5 Bar-chart (Absolute successful and failed cases) 75
5.6 Bar-chart (Precision, Recall, F-Measure, Successrate) 76
5.7 Workflow sketch of the AQL-Minimizer 82

List of Tables

3.1 Grammar of AQL Questions 38

4.1 Parameters 64

5.1 DroidBench Evaluation Results 74
5.2 Evaluation Results for the set of Real World Apps 80

1 Introduction

Nowadays not only computers are used for electronic data processing. Quite the contrary is the case: We live in the age of the Internet of Things. In this term "Things" refers to a plethora of devices that are meant to make life easier. To do so, they collect, process and exchange data. Excellent examples are smartphones and tablets, which have already become indispensable. Smartwatches, smart-TVs and intelligent board computers in cars are becoming more and more a part of everyday life.

All these "Things" deal with all sorts of data. The user knowingly enters data such as contact information, messages and mails as well as passwords and banking accounts. Additionally, the devices collect data on their own with sensors that are always active. For example, many devices track their location, scan fingerprints or automatically record what the user hears and sees. Since this data should be considered private and security-sensitive, it has to be protected.

Thus, there are rules and mechanisms that offer protection. For example, it is prohibited by law to access data without authorization. Infringement would result in penalties. Furthermore, any piece of software should be designed with aspects of security in mind to avoid unintentional data leaks. However, since it is difficult and thereby time consuming and expensive to develop secure software, security often cannot be guaranteed. Nevertheless, tools that are able to automatically find security issues, can be used to convince a user that a piece of software is trustworthy and secure.

This thesis focuses on such tools in the context of mobile devices, more precisely, on the analysis of mobile applications (apps). Apps are usually downloaded and installed without checking their functionality. Some apps, for example, might do a lot more than the user expects them to. Along with that, data might be leaked by accident or be revealed on malicious purpose. To develop an analysis that detects such security issues is challenging for various reasons. First, the field of app analysis is rather new in comparison to the analysis of other software artifacts. Second, each mobile operating system has its unique features which makes it tough or impossible to apply existing mechanisms. Third, most mobile operating systems are updated quite frequently. As a consequence analyses may have to be updated as well in order to stay up-to-date. Fourth, the ongoing race between attackers who hide malicious code and program analysts who try to detect this hidden code, makes it hard to keep the precision of an analysis at its best. Luckily, mature tools are available to precisely answer certain analysis questions. However, one tool capable of accurately answering question A might be imprecise when it comes to question B.

To avoid imprecise answers we want to bring expert tools together to cooperatively answer analysis questions as precisely as possible. This can be realized by following the divide and conquer approach: A complex analysis task is divided into parts and each part is answered by the most precise tool. To this end, a common language is required that allows us to generally formulate analysis tasks and questions as well as solutions and answers. Consequently, the conceptual design of such a language that permits the cooperation of analysis tools represents the main goal of this thesis.

1.1 Approach

The approach used to achieve the main goal of this thesis, developing a language that allows the formulation of analysis questions and answers, is summarized in the following.

First, the content of such a language or, more precisely, the content of such analysis questions and answers is specified. To do so, a formal definition of all possibly occurring elements in questions and answers is provided. Second, it is defined how to process questions and answers in order to combine analyses. Therefor, it is described how to split analysis questions into smaller parts and how to combine the answers associated with these smaller parts. The third part explains how to acquire these associated answers: The concept of a system is presented that can be used to interact with analysis tools.

These three parts lead to the syntactic definition of one language, that can be used to represent the formally defined questions and answers. It describes how to split and combine questions and answers respectively, and explains when to ask analysis tools for their precise results. This definition represents the main goal of this thesis.

Furthermore, to test this approach in practice and to evaluate its use and power, a tool that uses this language is described and implemented.

1.2 Thesis' Contents

In the first part, all required fundamentals are introduced. For that purpose, the operating system Android, which represents the field of application, along with its unique features, properties and challenges is described (see Section 2.1). Next, a general description of program analyses is provided and followed by a detailed presentation of specific program analyses considering apps built for Android (see Section 2.2). As a part of the fundamentals chapter, a running example is introduced and continued in the following chapters.

The conceptual design (see Chapter 3), presented thereafter, explains the previously described approach in detail. Thereby the terms analysis question and answer are introduced and explained. Of central importance in this chapter is

the formal and syntactic definition of a language that can be used to precisely describe such questions and answers as well as a system that uses this language.

In Chapter 4 the structure of an implementation of such a system is described. Furthermore, it is highlighted how this implementation is linked to the concept. The next chapter deals with the evaluation, which shows that this implemented system works as expected. Additionally, the results of several experiments considering different scenarios are presented.

This thesis ends with a summary of what has been achieved. In addition an outlook is given that explains why and how to continue the project started with this thesis.

2 Fundamentals

In the field of information technology, security generally refers to the protection of sensitive and private data. More precisely, it corresponds to the "protection of information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction in order to provide confidentiality, integrity, and availability" [oNSS17]. In this context *confidentiality* tells us that information should only be accessible to authorized and trusted individuals and software artifacts. The property *integrity* stands for the completeness and accuracy of data. Last, *availability* refers to the need, that information has to be accessible whenever it is required. When it comes to mobile devices such as smartphones, for example, the device' location or the stored contact data demand protection. On the one hand, nobody should be able to access or modify that information without being permitted to do so. On the other hand, once the permission is granted, the access to such data should be immediate and accurate.

Typical security flaws in the area of mobile devices have its origins in the apps which are installed. Apps can be downloaded from various markets and are developed by thousands of different developers. Although this does not seem to be a problem at first glance, it becomes a problem once we realize that we cannot trust some developers, markets or apps. Apps might be developed and shipped in order to steal, corrupt or manipulate data. This is why techniques are needed to prevent or detect security issues.

Constructive techniques can be used to restrict the access to certain resources or to avoid unwanted communication and thereby prevent security problems. For example, apps built for the operating system called Windows Phone 7 "can only send messages to a small number of trusted system applications (e.g. the browser)" [CFGW11]. The operating system Android protects resources such as the camera of a mobile device with permissions which have to be requested and granted, otherwise an app cannot use the resource. All these constructive techniques have to be implemented into the system itself and cannot be changed often or quickly since all apps built for the system would have to be changed as well. Hence, it is very important to find a compromise between restriction and openness, for example, an app should be able to communicate with other apps but there should be rules to do so. Whether an app follows such rules can be shown with analytic techniques. Analyses are capable of showing certain properties and detecting security problems that cannot be prevented. For instance, an analysis could be built to detect whether a resource such as the camera of a mobile device is accessed by an app or not.

In this chapter we present some analyses along with tools implementing them.

Furthermore, we show how these analyses could be brought together to cooperate. The analyses and tools, which stand in the focus of this thesis, are built for a specific operating system, namely Android. This is necessary, because every operating system has its own unique features, as does Android. These features and the arising analysis challenges are explained in the following section.

2.1 Android

Android is an open-source operating system for mobile devices such as smartphones, tablets, notebooks and even cars. Google took over the development of Android in 2005. Since then the operating system has become more and more popular. Nowadays, Android is the leading operating system with more than 86% [Gar17, tF17] of all mobile devices utilizing it. Primarily, this operating system supports the execution of smaller software products, mostly called *applications* or *apps* in the context of mobile devices. Apps provide a broad variety of different functionalities, ranging from simple text editing or text messaging apps to banking or 3D-gaming apps. The programming language used to develop Android apps is based on Java[1]. Thus, it can also be imagined as Java extended by Android libraries. Nevertheless, the structure of Android apps differs from the structure of Java programs. In the following, these differences, along with the general structure of Android apps, are described. Furthermore, some security features of Android and available app-to-app communication mechanisms are explained.

2.1.1 Components

Each Android app consists of a set of components. There are four types of components available in Android, which are represented by Java classes [Goo17c]:

1. *Activity* components serve as the user interface of an app. Hence, any interaction with the user is handled by an Activity. Each Activity represents only one view of the complete user interface. This is why most apps consist of multiple Activities associated with different functionalities, for example, to take photos, view maps, send emails etc.

2. *Service* components are mostly used to execute long-running operations in the background such as polling data from the Internet in order to display notifications on certain events. In contrast to Activities, Services have no user interface.

3. A *Content provider* manages the access to any sort of structured data comparable to a database. For instance, the contact data is stored and handled by a Content provider.

[1] Java is a well known, object-oriented programming language.

4. *Broadcast receiver* components can receive messages from other components or directly from the operating system. Thereby, an app can react on system wide events. For example, the operating system broadcasts a message once the battery of a mobile device runs low.

A single app can contain an arbitrary amount of components of any kind.

In contrast to ordinary Java programs there exists no single starting point to launch an app. The execution of a component is started by a call to a predefined callback method. For instance, the Android Activity lifecycle describes the set of callback methods for Activities. Figure 2.1 shows a simplified version of this lifecycle which consists of a subset of all available callback methods and connections between them. This simplified version is sufficient to describe how it works: Each node in the figure stands for a callback method or the state of the Activity whereas the edges represent the execution order. Once an Activity component is launched the `onCreate()` method is executed followed by the `onResume()` method. Before an Activity is shut down the two methods `onPause()` and `onDestroy()` are executed. It is called a cycle, because the first two methods (`onCreate()` and `onResume()`) can be executed again after the `onPause()` method has been executed. If the user simply returns to the running Activity only the `onResume()` method is executed again. However, the system might kill the Activity process to spare resources for example. In this case and if the user navigates to the Activity again, the `onCreate()` and `onResume()` methods are executed again.

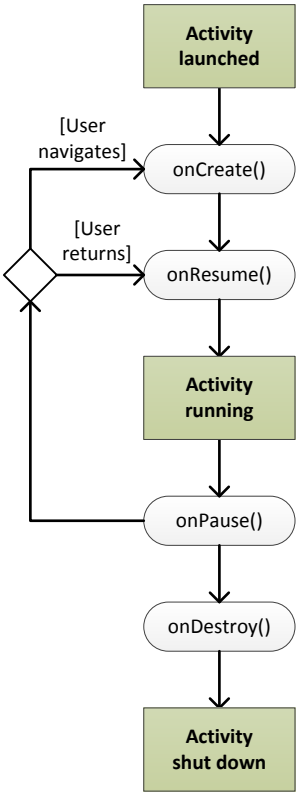

Figure 2.1: Android Activity Lifecycle

Once an app is started at least one component has to be launched. Which one is launched and which other ways are available to make use of other components is defined in an app's manifest file which is described at the end of this section (see Section 2.1.4).

2.1.2 Permissions

Android has several security mechanisms. For instance, any app is executed in its own runtime environment, represented by a virtual machine. One emerging consequence is that one app cannot directly access the data of another one. Other security mechanisms can be found when it comes to cost generating actions such as sending SMS to well-known premium receivers. In such a case, the Android operating system double-checks if the user really wants to perform this

action [Goo17a].

Furthermore, Android possesses a permission system. Through this system certain resources, such as the camera of a mobile device, are protected against being used without knowledge of the user. Accessing a permission-protected resource is mostly implemented through an Android API call. As soon as the system receives an API call, it checks whether the involved permission has been granted to the app. Which permissions are required by an app has to be specified in its manifest (see Section 2.1.4). Before Android version 6.0 the user had to grant all required permissions before the installation of the app. In the current version, the user has to grant each permission individually, once the app requests the usage [Goo17a]. An attempt to access a resource protected by a permission without declaring the use of this permission in the manifest or without being granted to use it, results in an app-crash triggered by an exception.

However, only a subset of all available permissions has to be granted by the user, others are granted by default. The permission attribute *protection-level* tells us, which permissions belong to this subset. The protection-level can be set to one of the following values:

- *dangerous* - These permissions have to be granted by the user, because the resources protected by these are considered as security-sensitive.
- *normal* - These permissions are granted by default.

  > "Normal" permissions should imply minor risk and serve only as a "heads-up" for the user that the application is requesting access to such functionality.[AYU+09]

- *signatureOrSystem* - A permission of this type is granted automatically, if the app requiring the permission belongs to the operating system or if it is signed with the same signature as the app declaring the permission.
- *signature* - The app requiring a permission of this type has to be signed with the same signature as the app declaring it.

Developers can also use available permissions or define their own custom permissions in order to protect components of their own. Custom permission have to be declared in the Android manifest (see Section 2.1.4). How to launch or interact with a component of another app is explained in the following.

2.1.3 Inter-Component Communication

Android components communicate with each other via so-called *intents*. From an abstract perspective intents can be described as envelopes. These envelopes might be filled with data and sent from a sender component to a receiver component. If the sender knows the receiver component, the intent can be send explicitly by naming the class of the receiver component. In case the receiver is unknown, an

intent can be send implicitly by defining properties of a possible receiver. Whether a component can receive this intent or not depends on its intent-filters. Only if the properties of at least one intent-filter match the properties of the implicit intent, it is a valid receiver. Three different properties can be used:

- *action* - This property refers to a string which mostly hints at the functionality of the receiving component. For example, if the value of this string, namely the action-string, is "SEND" the component can be used to send data.
- *category* - Setting the category to "READABLE", for instance, indicates that the receiver should be a text viewing or editing component.
- *data* - This last property consists of different attributes which describe the content of any data sent to the receiver which can be URLs, phone numbers etc.

Whether these properties match is decided by the Android system and has been described in a paper by Damien Octeau et al. [OJD+16]. Both, explicit and implicit intents can target components of the same app or components of other apps. In the latter case an additional flag (`exported`) has to be set to true. In the following the expression *inter-app* is used to state that more than one app is involved. The opposite is the case if the expression *intra-app* is used.

An intent-filter can be defined dynamically in an app's source code. However, the most common way is to define it statically by placing its definition in the Android manifest which is described in the subsequent section.

2.1.4 Manifest

Every Android app comes with exactly one manifest file in XML[2] format. This manifest is a static, predefined collection of various information. First it holds general information like the app's name, version or the package it belongs to. It also contains information mentioned before such as components and their intent-filters as well as information about used and custom permissions. Two examples of such a manifest are shown in the following example.

2.1.5 Running Example (1/3): The Scenario

In this section, a scenario is introduced, which is used as running example from here on. The scenario consists of two apps which are interacting with each other in order to overcome their limitations and thereby leak security sensitive data. The two apps are presented now and along with that parts of their source code and their Android manifests are highlighted. These parts play an important role in the analyses used to detect the security issue.

[2]The Extensible Markup Language (XML) is used to manage structured data.

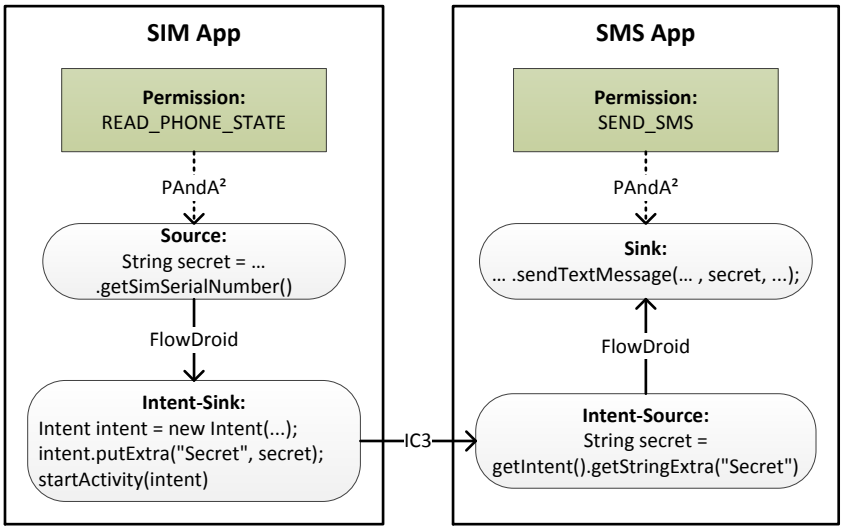

Figure 2.2: Running Example Overview

Figure 2.2 depicts the complete scenario: On the left-hand-side the `SIMApp` can be found. An extract from the source code of this app is shown in Listing 2.1. This app uses the `READ_PHONE_STATE` permission by calling the `getSimSerialNumber()` method (see Line 6 of Listing 2.1) in order to get the serial number of the SIM-card. Then this app sends the serial number via an implicit intent to another component (see Lines 9-11 of Listing 2.1). If the `SMSApp` (right-hand-side in Figure 2.2) receives the intent, it proceeds by extracting the serial number from this intent. This happens by calling the `getStringExtra(..)` method (see Line 5 of Listing 2.2). Afterwards the extracted serial number is send via SMS to a potentially untrustworthy recipient. For this purpose, the `sendTextMessage(..)` method is called (see Lines 8 of Listing 2.2). This call requires the `SEND_SMS` permission.

```
1  public class SIMAppMainActivity extends Activity {
2      ...
3      private void source() {
4          // Source
5          TelephonyManager manager = (TelephonyManager)
               getSystemService(Context.TELEPHONY_SERVICE);
6          String secret = manager.getSimSerialNumber();
7
8          // Intent Sink
9          Intent intent = new Intent("de.upb.fpauck.CALLSINK");
10         intent.putExtra("Secret", secret);
11         startActivity(intent);
12     }
13 }
```

Listing 2.1: SIMApp (Source Code)

```
1  public class SMSAppMainActivity extends Activity {
2      ...
3      private void sink() {
4              // Intent Source
5              String secret = getIntent().getStringExtra("Secret");
6
7              // Sink
8              SmsManager.getDefault().sendTextMessage("+49111111111",
                   null, secret, null, null);
9          }
10 }
```

Listing 2.2: SMSApp (Source Code)

Listings 2.3 and 2.4 show the manifests of the `SIMApp` and the `SMSApp` respectively. In the following, the content, which is relevant for this example, is explained. First of all, the `uses-permission` tags in each manifest show the permissions required by each app (see Line 4 of Listing 2.3 and Listing 2.4). These tags are followed by the `application` tag. Each `application` tag contains a definition of all components and their intent-filters. In case of the `SIMApp`, one activity component can be found, namely the `SIMAppMainActivity`. For this component only one intent-filter is defined (see Lines 8-11 of Listing 2.3). The action-string and category of this intent-filter tell the system that this component should be launched once the app is explicitly started, for example, through the user interface. The same intent-filter is defined for the `SMSAppMainActivity` Activity which is the only component of the `SMSApp`. However, another intent-filter is defined for this component (see Lines 12-15 of Listing 2.4). This tells us that another way to communicate with this component exists. Furthermore, since the `exported` flag is not set explicitly for this intent-filter, the default value (true) is supposed. Thereby, communication between this component and components of other apps is not forbidden.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.upb.fpauck.simapp">

    <uses-permission android:name="android.permission.
        READ_PHONE_STATE" />

    <application [...] >
        <activity android:name=".SIMAppMainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.
                    LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Listing 2.3: SIMApp (Manifest)

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.upb.fpauck.smsapp">

    <uses-permission android:name="android.permission.SEND_SMS" />

    <application [...] >
        <activity android:name=".SMSAppMainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.
                    LAUNCHER" />
            </intent-filter>
            <intent-filter>
                <action android:name="de.upb.fpauck.CALLSINK" />
                <category android:name="android.intent.category.
                    DEFAULT"/>
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Listing 2.4: SMSApp (Manifest)

In summary, the `SIMApp` requires the `READ_PHONE_STATE` permission and tries to transfer security sensitive data to the `SMSApp` which leaks this information. To do so, it also requires a permission, namely the `SEND_SMS` permission. An analysis, for example a so-called taint analysis which is described in the next section, should automatically find such a leakage of information.

2.2 Analyses

Analyses can focus on different aspects. On the one hand, there exist analyses that measure and rate the performance of a piece of hard- and/or software. On the other hand, there exist analyses, called *program* or *software analyses*, which focus on certain properties of software. For example, a software analysis could be used to determine which values a variable could hold. A general introduction into software analyses as well as information on how to build an analysis and well-known analyses examples can be found in the book Principles of Program Analysis [NNH99].

Software analyses can be of differing accuracy. There exist flow-insensitive analyses that do not consider any kind of flow. Such analyses assume that any statement might be executed after any other statement. This is why information flow analyses, which are described in more detail in this section, are always flow-sensitive. However, analyses can still be of differing precision, depending on its context-, field- or object-sensitivity [ARF+14]. Additionally, an analysis for Android apps could be intra- or inter-procedural as well as it could only consider intra- or inter-component/app flows.

Furthermore, Android with its unique structure and features leads to some challenges in the field of software analyses. These challenges and how to deal with them is also addressed in this section. Next in this section, some state-of-the-art tools performing analyses on Android apps are introduced. Lastly, the running example is continued.

2.2.1 Information Flow Analyses

Information flow analyses can be used to find security issues. Such analyses consider information flows in a piece of software. An information flow describes the transfer of information from one point to another. If, for example, a variable x is assigned to a variable y, there exists a direct information flow from x to y. In contrast, an information flow can also be indirect. For instance, it could be checked whether the value stored in one variable matches the value held by another one. In such a case we gain the information that both variables are equal or not which allows inferences about the values stored in both variables.

Type-based information flow analysis [MS13] assigns security levels to each variable. The most basic version considers only two levels: H meaning high security or private/sensitive data and L representing low security or public data. In addition, one of the simplest security policies says that flows from variables in H to variables in L are forbidden. For instance, let us assume variable x holds private data, then $x \in H$ holds. If $y \in L$ holds as well and an analysis detects a flow from x to y, then a security leak is detected since data from a variable in H flows to a variable in L.

Another type of information flow analyses considers graphs, mostly program dependence graphs (PDGs) [FOW87]. Because of that, analyses of this type are

called *PDG-based information flow analyses* [MS13]. A PDG, for example, represents all information flows by edges. If there exists a flow from x to y, then there also exists an edge from any node representing a definition of x to a node representing y. Assuming that such a flow is forbidden, an analysis can search for a path between these two nodes in order to detect a security leak.

A third type, so-called *taint analyses*, can be imagined as a combination of type- and PDG-based analyses. Comparable to a PDG, a graph is used to find paths between predefined sources (H) and sinks (L). In this context, a source refers to a statement which loads or generates security sensitive data. An Android example for such a statement could be a statement accessing the serial number of a mobile device. Other statements which leak information to the outside are considered sinks. In Android this, for instance, could be a statement used to send an SMS or a simple logging statement. A taint analysis reports any path that leads from a source to a sink. As a part of this thesis, an inter-app taint analysis is build by combining different analyses.

2.2.2 Challenges & Solutions

When it comes to Android app analyses, some challenges have to be overcome. In the following, two of those are described in more detail.

As mentioned before Android apps have no single starting point. That means there is no `main(..)` method as in Java programs. Thereby, a typical flow-sensitive analysis cannot be executed since it misses an initial starting point in order to construct the information flow paths from there on. One way to overcome this issue is to create such a single starting point. This is mostly done by automatically generating a dummy main method based on all callback methods which belong to the analyzed app [ARF+14]. These can be methods which are called once a button has been pressed or methods of the Activity lifecycle for instance.

Since Android components mostly communicate via intents, these intents have to be analyzed as well in order to find out if an intent can be received by a certain component. In other words, to determine inter-component or inter-app flows which lead from one component or app to another. To do so can be challenging as there are dozens of ways to generate intents and intent-filters. In addition, there are even more ways to define, for example, the action-string of an intent. To address this challenge different approaches can be applied. One state-of-the-art approach is to dynamically execute the app partially and log which components are called in all possible execution orders.

Luckily, there are some mature tools available which are already able to overcome such challenges. In the next section, two tools which handle the two challenges described above are introduced among other tools.

2.2.3 Tools

There exist a lot of tools for static Android app analyses. Most of them perform information flow analyses based on different approaches and with distinct focuses [ARF+14, LBB+15a, KFB+14, GKP+15, RCT+14]. Other tools like COVERT [BSGM15] or PAndA^2 [JTP16] concentrate on analyzing Android permissions. Again other tools focus on the analysis of inter-component and inter-app communication [OLD+15, OMJ+13]. In the following, three mature tools, which represent implementations of static analyses, are described in more detail. They have been chosen because they can be combined into a highly precise inter-app taint flow analyses that uses permission-protected statements as sources and sinks. Furthermore, in contrast to many other tools, these tools are up-to-date and freely available.

- **FlowDroid** [ARF+14] is one of the most precise taint analysis tools available for Android apps, because it is context-, flow-, field- and object-sensitive. Furthermore, it generates a dummy main method as a single starting point for the analysis. However, its potential is limited since it only considers intra-component flows.

  It takes an Android Application Package file (.apk file) as input and generates a textual result which represents the constructed, tainted flows. Among other options, a predefined list of sources and sinks as well as several dependencies in the classpath can be used to configure FlowDroid.

- **IC3** [OLD+15] can mainly be used to analyze inter-component and inter-app communication. Therefor, it extracts the three properties action, category and data from intents and intent-filters of an entire app. The analysis used to extract these values is an inter-procedural analysis which is context- and flow-sensitive.

  IC3 uses Java classes as input. However, it can be applied on Android apps since the developers of IC3 provide another tool[3] which can reproduce these classes from an .apk file. The output generated by IC3 can directly be written into a database or output in form of a protocol buffer [Goo17d].

- **PAndA^2** [JTP16] is a framework for analyses of Android apps. It comes with different built-in analyses. One of those is an intra-app taint analysis. Another one focuses on analyzing the permissions used by an app. To this end, it detects whether a permission is declared as being used in the manifest and checks whether there exists a statement whose execution requires this permission.

  It takes an .apk file as input and produces filterable textual and graphical results at different levels of detail. The format of the textual result can

[3]The tool Dare reproduces Java bytecode from .apk files [OJM12].

be plain text or an interactively viewable HTML document. The graphical result comes as a scalable vector graphic embedded in an HTML document.

FlowDroid and PAndA^2 are based on the soot framework [VRCG+10]. Furthermore, these two and also IC3 use Jimple [VRCG+10] as intermediate language to reference statements, methods and classes. Jimple is a simplified version of Java source code, developed as a part of the soot framework. One specialty is the use of at most three components per statement. Thereby, for example, the number of different types of statements is reduced to less than a tenth compared to instructions available in Java bytecode. That is one reason why it is perfectly suited for analyses.

Apart from that, the results produced by all three tools have nothing in common and without further work their results cannot be combined. Nevertheless, each tool is powerful and precise in their analysis field.

Because of these commonalities it seems likely to combine the three tools. In fact there have been attempts to do so before, for example, the two tools called IccTA [LBB+15a] and DidFail [KFB+14] combine the results of FlowDroid with results of Epicc [OMJ+13]. Epicc is the predecessor of IC3. Just like IC3 it is used to analyze inter-component and inter-app communication. However, it is less precise. In contrast to IccTA, DidFail was not designed to replace Epicc with another tool or updated to use IC3 instead of Epicc. This is one reason, why IccTA is still one of the most powerful implementations of an inter-app taint analysis. Other tools such as DroidSafe [GKP+15] or FUSE [RCT+14] are also performing inter-app taint analyses and neither use FlowDroid nor Epicc or IC3. However, these are outdated and no further updates are planed.

Next to these tools performing static analyses there exist a few tools that run dynamic analyses. TaintDroid, for example, is able to execute a dynamic taint analysis which is described in the associated paper by William Enck et al. [EGC+10]. In contrast to this outdated dynamic analysis tool, Harvester [RAMB16] is a state-of-the-art one. It was designed to deobfuscate malicious apps, for instance, malware that tries to hide its malicious code through extensive use of reflection. Therefore, Harvester precisely determines all possible values that can be used in the context of a certain statement. For example, harvester can find out which method is called by a reflective call of Java's `getMethod` function. In the context of Android, Harvester can also be used to determine intent and intent-filter attributes (`action, category, data`) which cannot be determined by static analyses.

In the next section the three tools (FlowDroid, IC3, PAndA^2) presented above are applied on the two apps introduced with the running example in Section 2.2.4. The results of all three tools are combined in order to detect a security leak that would remain undetected otherwise. In the chapter hereafter, a concept is presented that allows arbitrary combinations of analysis tools.

2.2.4 Running Example (2/3): Cooperative Analysis

The previously started example considering the `SIMApp` and the `SMSApp` is continued here. A look at the results produced by the tools FlowDroid, IC3 and PAndA[2] enables us to understand how these results have been produced and how to combine them.

As stated above, FlowDroid detects intra-app taint flows. Therefor, it firstly determines all sources and sinks that can be found in a single app. Sources and sinks are represented by specific statements in the source code. A source always reveals security sensitive data, for example the SIM card's serial number. A sink leaks information to the outside by sending an SMS for instance. FlowDroid finds these sources and sinks through a comparison of all statements with a predefined, configurable list of sources and sinks. Once the tool has found all sources and sinks, it determines all paths between these sources and sinks. The result produced for the `SIMApp` looks as follows:

```
...
Found a flow to sink virtualinvoke $r0.
<de.upb.fpauck.simapp.SIMAppMainActivity:  void startActivity(
android.content.Intent)>($r1),
from the following sources:
  • $r4 = virtualinvoke $r3.
    <android.telephony.TelephonyManager:  java.lang.String
    getSimSerialNumber()>() (in <de.upb.fpauck.simapp.SIMAppMainActivity:
    void source()>)
...
```

FlowDroid found a flow from the only source (`getSimSerialNumber()`) to the only sink (`startActivity(..)`). A similar flow can be found in the `SMSApp`:

```
...
Found a flow to sink virtualinvoke $r3.
<android.telephony.SmsManager:  void sendTextMessage(
java.lang.String,...String,...String,android.app.PendingIntent,
...PendingIntent)>("+49111111111", null, $r2, null, null),
from the following sources:
  • $r2 = virtualinvoke $r1.<android.content.Intent:
    java.lang.String
    getStringExtra(java.lang.String)>("Secret")
    (in <de.upb.fpauck.smsapp.SMSAppMainActivity:  void sink()>)
...
```

This time the source is represented by the `getStringExtra(..)` statement that is connected to a sink, which in turn is represented by the `sendTextMessage(..)` statement. Figure 2.2 on page 10 depicts these two flows through the two edges annotated with FlowDroid. A third edge connects the only sink of the `SIMApp` with the only source of the `SMSApp`. In order to detect the flow represented by this edge, IC3 comes into play. In contrast to FlowDroid, IC3 does not directly output flow paths, but it indirectly collects sources and sinks. More precisely, it

collects *intent-sources* and *intent-sinks*. These types of sources and sinks refer to statements, that are involved once a component communicates with another one. The results produced for the `SIMApp` and the `SMSApp` are presented in Listing 2.5 and Listing 2.6, respectively.

```
name: "de.upb.fpauck.simapp"
version: 1
used_permissions: "android.permission.READ_PHONE_STATE"
components {
  name: "de.upb.fpauck.simapp.SIMAppMainActivity"
  kind: ACTIVITY
  exported: true
  intent_filters {
    ...
  }
  exit_points {
    instruction {
      statement: "virtualinvoke r0.<de.upb.fpauck.simapp.
          SIMAppMainActivity: void startActivity(android.content.
          Intent)>(r1)"
      class_name: "de.upb.fpauck.simapp.SIMAppMainActivity"
      method: "<de.upb.fpauck.simapp.SIMAppMainActivity: void source
          ()>"
      id: 10
    }
    kind: ACTIVITY
    intents {
      attributes {
        kind: ACTION
        value: "de.upb.fpauck.CALLSINK"
      }
      attributes {
        kind: EXTRA
        value: "Secret"
      }
    }
  }
}
analysis_start: 1479897285
analysis_end: 1479897290
```

Listing 2.5: IC3 result for the `SIMApp`

```
1  name: "de.upb.fpauck.smsapp"
2  version: 1
3  used_permissions: "android.permission.SEND_SMS"
4  components {
5    name: "de.upb.fpauck.smsapp.SMSAppMainActivity"
6    kind: ACTIVITY
7    exported: true
8    extras {
9      extra: "Secret"
10     instruction {
11       statement: "r1 = virtualinvoke r2.<android.content.Intent: java
             .lang.String getStringExtra(java.lang.String)>(\"Secret\")"
12       class_name: "de.upb.fpauck.smsapp.SMSAppMainActivity"
13       method: "<de.upb.fpauck.smsapp.SMSAppMainActivity: void sink()>
             "
14       id: 2
15     }
16   }
17   intent_filters {
18     attributes {
19       kind: ACTION
20       value: "de.upb.fpauck.CALLSINK"
21     }
22     attributes {
23       kind: CATEGORY
24       value: "android.intent.category.DEFAULT"
25     }
26   }
27   intent_filters {
28     ...
29   }
30 }
31 analysis_start: 1479897295
32 analysis_end: 1479897300
```

Listing 2.6: IC3 result for the `SMSApp`

The intent-sink is represented by the Lines 11-30 of Listing 2.5. On the one hand, the referenced statement (`startActivity(..)`) can be found there. On the other hand, the intent attributes belonging to the intent, which is used to start another activity, are listed. For example the action-string "de.upb.fpauck.CALLSINK". This action-string can be found in the result produced for the `SMSApp`, too (see Line 20 in Listing 2.6). In this case, the origin of the intent-source is represented by the Lines 11-13 of Listing 2.6. In particular, the statement `getStringExtra(..)` is referenced. Thereby, the complete intent-source is described with its origin and its intent-filter attributes. Since the two action-strings match, it can be assumed that there exists a flow from the detected intent-sink to the detected intent-source.

In the following an extract from the analysis result output by PAndA[2] for the `SIMApp` is shown:

```
...
-----$r4 = virtualinvoke $r3.<android.telephony.TelephonyManager:
java.lang.String getSimSerialNumber()>() in method
<de.upb.fpauck.simapp.SIMAppMainActivity:  void source()>-----
```

- android.permission.READ_PHONE_STATE (REQUIRED)

```
...
-----specialinvoke $r1.<android.content.Intent:
void <init>(java.lang.String)>("de.upb.fpauck.CALLSINK") in method
<de.upb.fpauck.simapp.SIMAppMainActivity:  void source()>-----
```

- android.permission.READ_PHONE_STATE (MAYBE REQUIRED)

```
...
```

It shows the `READ_PHONE_STATE` permission and all statements that require this permission. In this case, the statement calling the `getSimSerialNumber()` method requires this permission to directly access resources protected by it. This is symbolized through the permission group which is displayed in brackets behind the permission's name. The call to the constructor of an `Intent` object might also require that permission, because the intent could possibly address a receiver who requires it. PAndA^2 informs about that possibility by assigning the `MAYBE REQUIRED` permission group.

Similarly, the `SEND-` and `WRITE_SMS` permission can be found in the result for the `SMSApp`:

```
...
-----virtualinvoke $r3.<android.telephony.SmsManager:
void sendTextMessage(java.lang.String,java.lang.String,
java.lang.String,android.app.PendingIntent,android.app.PendingIntent)>
("+49111111111", null, $r2, null, null) in method
<de.upb.fpauck.smsapp.SMSAppMainActivity:  void sink()>-----
```

- android.permission.SEND_SMS (REQUIRED)
- android.permission.WRITE_SMS (MISSING)

```
...
```

The call of the `sendTextMessage(..)` method requires two permissions, namely the `SEND_SMS` and the `WRITE_SMS` permission. From the manifest of the `SMSApp` (see Listing 2.4 on page 12) we know that the `SEND_SMS` has been marked as used. The `WRITE_SMS` permission instead is not marked as used. However, this is no problem since the protection level of this permission is set to normal. Thereby, it is guaranteed, that the app will work as expected even if that permission is not declared for use.

The results of all three tools look completely different. Nevertheless, there are some parts, which appear in all of them. For example the statement calling the `getSimSerialNumber()` method is referenced in one result of FlowDroid as well as in one result of IC3 and PAndA^2. Based on these common elements the

results can be merged in order to analyze the complete scenario (see Figure 2.2 on page 10). Thereby we can detect an inter-app taint flow from a source to a sink, both protected by at least one permission. This flow starts with the statement calling the `getSimSerialNumber()` method. PAndA^2 tells us that this statement requires the `READ_PHONE_STATE` permission and FlowDroid has detected a flow from this statement to an intent-sink of the `SIMApp`. IC3 connects this intent-sink of the `SIMApp` with an intent-source of the `SMSApp`. FlowDroid in turn tells us that this intent-source is connected to the `sendTextMessage(..)` statement. According to PAndA^2 this statement requires two permissions. In summary, a flow from the `getSimSerialNumber()` statement, requiring the `READ_PHONE_STATE` permission, to the `sendTextMessage(..)` statement, requiring the `SEND_SMS` and `WRITE_SMS` permissions, is found. This flow represents a possible security issue since the `SIMApp` was neither meant to send an sms nor to leak the sim card's serial number.

The concept which is developed in the next chapter precisely describes a way how to automatically combine such results. Thus, this example is continued at the end of the next chapter.

3 Conceptual Design

The running example at the end of the previous chapter indicates that it is reasonable to combine different analysis results computed by mature tools. This chapter focuses on designing a concept that can be used to achieve this. It brings analysis tools together and makes them cooperate, which can be advantageous in various situations:

- Whenever a program analyst is confronted with analysis results that are too imprecise and do not match their expectations, they can try to apply other tools or combinations of other tools in order to achieve more precise results.
- Analysis developers could build new analyses on the basis of other analyses while having in mind that the results are combined in the end. Thereby they do not have to rebuild the basis. More importantly, they need a way to combine these analyses with their new one.
- During the execution of one analysis, different questions may arise. Some of them might be answered precisely by the analysis itself. Other answers to such questions might be approximated or guessed. To avoid such approximations or guesses, specialized tools could be asked on demand once such a question arises.

In all these situations, a language is required, that allows the formulation of analysis questions and answers. In this chapter the concept of a query language for Android app analysis tools, namely the Analysis Query Language (AQL), is proposed. Therefor, two structural parts are defined formally and syntactically. First, the structure of *questions* which can be asked in order to execute analysis tools and, second, the structure of *answers* that represent the results produced by these tools. As a third part of this chapter, the conceptual structure of a system using the AQL is explained. At the end of this chapter the running example is continued again. In doing so, it is demonstrated how the AQL and an associated system works.

3.1 Analysis Query Language (AQL)

Query languages are typically used to manage structured information. Two famous examples are the Structured Query Language (SQL), that is used to access data held in a relational database, or the XML Path Language (XPath), that can

be used to navigate to and process certain nodes of XML documents. However, there exist slightly different query languages such as the Program Query Language (PQL) [MLL05] which is used to initialize static analyses in order to detect and resolve issues in a program. For instance, the PQL allows to formulate queries that can be issued to eliminate security concerns like points of attack for SQL-Injection. In this case the program represents the structured information that is managed.

All query languages have in common, that they allow different actions to read, filter, combine or, speaking generally, process some sort of information. In case of the AQL, this functionality is needed as well. Analysis results, which represent the information of interest, should be readable in order to reuse them. Different analysis tools should be cooperating by combining their results. Lastly, any results, combined or not, should be filterable to reduce its content to the essential parts.

In the next sections, the structure of the AQL is defined. Among other parts, questions and answers that belong to the AQL are formally described. This definition and formal description is used as a basis for the syntax of the AQL defined in the sections thereafter.

3.1.1 AQL-Questions

In the context of Android app analyses, different questions can be asked. The analyst Bob for example might want to know whether there exists an information flow from one program location in an app to a second location in the same or another app. If Alice wants to answer these questions, she could ask for information about intent-sinks and -sources in order to find inter-component flows. To assist her, Bob could have already asked for a list of intents and intent-filters. A third person, Charlie, could be interested in a completely different field, for example he might want to know whether there are any permissions required to execute a certain method. To sum it up, a collection of questions that can be asked is shown below:

- Does there exist a flow from x to y?
- Which explicit/implicit intents are launched by x?
- Which intent-filters are defined for x?
- Which intent-sinks or -sources can be found in x?
- Which permissions are required by x?

Hereby, x and y reference statements, methods or classes of an app or an entire app. These questions are precisely formulated here.

First, it has to be specified what we are asking for. For this purpose, the questioner must choose a *subject of interest* from the following set of subjects ($\mathcal{S}$):

$$\mathcal{S} = \{\text{Flows, Intents, IntentFilters, IntentSinks, IntentSources, Permissions}\}$$

Second, the analysis *target* has to be referenced. Since an analysis can target parts such as program locations of one or more apps as well as one or more entire apps, a way to describe all these possible targets is required. Therefore, let us assume, the abstract set *App* describes the set of all existing apps, then $Statements_X$, $Methods_X$ and $Classes_X$ represent sets of all statements, methods and classes of apps in $X \subseteq App$. Putting these sets together adds up to a set of all *references* (R) which can be used in a question:

$$R = \{(s_X, m_X, c_X, X) \mid s_X \in Statements_X \cup \{\varepsilon\} \wedge m_X \in Methods_X \cup \{\varepsilon\} \wedge c_X \in Classes_X \cup \{\varepsilon\} \wedge X \subseteq App\}$$

In this context, ε stands for null values. Accordingly, s_X, m_X and/or c_X might be null in a reference. X is not allowed to be empty, because it has to be set in order to know which apps should be analyzed. Furthermore, let R^C describe a subset of R that only contains references to components. For these *component references* $s_X = \varepsilon$ and $m_X = \varepsilon$ always holds. Considering the collection of questions given above, x and y are elements of the set R.

With these two sets a question (q) can be described in a tuple consisting of a subject of interest (s) and a subset of the available references (R_q): $q = (s, R_q)$. Thus, the set of all questions ($\mathcal{Q}$) is:

$$\mathcal{Q} = \{(s, R_q) \mid s \in \mathcal{S} \wedge R_q \subset R \wedge 1 \leq |R_q| \leq 2\}$$

The cardinality of R_q is bounded because we only ask for properties in a reference or for flows from one reference to another. To do so, at least one and not more than two references are required.

Someone could, for example, ask for a list of all permissions that are required by the `sendTextMessage(..)` statement in the `SMSApp` from the running example. This question (q') can be formulated as follows:

$$s' = \text{Permissions}$$
$$R' = \{(\text{sendTextMessage(..)}, \varepsilon, \varepsilon, \{\text{SMSApp}\})\}$$
$$q' = (s', R')$$

Asking for flows from the `SIMApp` to the `SMSApp`, for instance, would require two references (cf. q''):

$$s'' = \text{Flows}$$
$$R'' = \{(\varepsilon, \varepsilon, \varepsilon, \{\text{SIMApp}\}), (\varepsilon, \varepsilon, \varepsilon, \{\text{SMSApp}\})\}$$
$$q'' = (s'', R'')$$

From now on the term *AQL-Questions* refers to all questions in $\mathcal{Q}$. The number of these can be increased by extending the set of subjects of interest, but in the scope of this thesis only the subjects of interest in $\mathcal{S}$ are considered. The counterpart, namely the *AQL-Answers*, is specified next.

3.1.2 AQL-Answers

AQL-Answers should be able to present information for each subject of interest in $\mathcal{S}$. A general definition of all available answers ($\mathcal{A}$) is:

$$\mathcal{A} = \bigcup_{s \in \mathcal{S}} \mathcal{A}_s$$

A_s refers to the information relevant for each subject of interest s. What information belongs to which subject of interest is specified in the following:

- [s = Flows]
  A flow symbolizes the transfer of information from one program location to another. To describe such a flow, these two locations have to be specified. The set of references R can be used to do so. Thereby, the set of all flows can be described by:

  $$\mathcal{A}_{\text{Flows}} = \{(r_{\text{start}}, r_{\text{end}}) \mid r_{\text{start}}, r_{\text{end}} \in R\}$$

- [s = Intents]
  Intents are used for inter-component communication. One component sends an intent to another component. To describe the sending component the set of references (R) can be used again. In case of an explicit intent the receiver can be identified by a component reference ($r \in R^C$). But in case of an implicit intent the receiver needs to be recognized through the *information triple* action, category and data. Let us assume

  $$I = \{i \mid i = (\textit{action, category, data})\}$$

  is the set of all combinations of these three properties. Then the set of all intents can be described as:

  $$\mathcal{A}_{\text{Intents}} = \{(r_{\text{origin}}, t) \mid r_{\text{origin}} \in R \wedge t \in R^C \cup I\}$$

- [s = IntentFilters]
  Intent-filters can be predefined in the Android manifest or assigned at runtime, but anyway intent-filters are described by the three properties summarized in I. These properties tell us which intents can be received by the app that specifies the filter. The set of intent-filters can be defined as follows:

  $$\mathcal{A}_{\text{IntentFilters}} = \{(r_{\text{origin}}, i) \mid r_{\text{origin}} \in R^C \wedge i \in I\}$$

  Since any intent-filter belongs to a component, r_{origin} refers to this component.

- [s = IntentSinks]
  Intent-sinks are special intents. That is why they can be described in the same way intents can be described:

  $$\mathcal{A}_{\text{IntentSinks}} \subseteq \mathcal{A}_{\text{Intents}}$$

  However, intent-sinks implicitly carry more information than intents, since they represent ends of information flow paths. This means that there is some information, for example, tainted data in a taint analysis, that reaches the origin referenced by an intent-sink.

- [s = IntentSources]
  Intent-sources represent the counterpart of intent-sinks. An intent-source might be the starting location of an information flow path. They can be described just like intent-sinks and intents with one small but important difference: The reference (r_{origin}) has to refer to a statement that, for instance, extracts information from an intent. Nevertheless, the definition is completely equal:

  $$\mathcal{A}_{\text{IntentSources}} = \{(r_{\text{origin}}, t) \mid r_{\text{origin}} \in R \wedge t \in R^C \cup I\}$$

  Note that t refers to a component reference in case of an explicit intent and to an information triple in case of an implicit intent. Furthermore, $\mathcal{A}_{\text{IntentSources}} \cap \mathcal{A}_{\text{Intents}} = \emptyset$ and $\mathcal{A}_{\text{IntentSources}} \cap \mathcal{A}_{\text{IntentFilters}} = \emptyset$ holds, because intent-sources never have an origin (r_{origin}) in common with an intent or an intent-filter. Additionally, intent-filters only reference implicit intents.

- [s = Permissions]
  In the context of AQL-Answers, we might want to show which permissions are used by a reference. Assuming that P describes the set of permissions that are available in the Android system, $A_{\text{Permissions}}$ can be defined as follows:

  $$\mathcal{A}_{\text{Permissions}} = \{(r_{\text{origin}}, p) \mid r_{\text{origin}} \in R \wedge p \in P\}$$

Using this description of answers and considering question q' which asked for permissions required by the `sendTextMessage(..)` statement the answer should be A':

$$r_1 = (\text{sendTextMessage(..)}, \text{sink()}, \text{SMSAppMainActivity}, \{\text{SMSApp}\}) \in R$$
$$A' = \{(r_1, \text{SEND\_SMS}), (r_1, \text{WRITE\_SMS})\} \subset \mathcal{A}_{\text{Permissions}} \subset \mathcal{A}$$

This answer contains both permissions (`SEND_SMS` and `WRITE_SMS`) that are required by the precisely linked statement (r_1).

Question q'' which asks for flows from the `SIM-` to the `SMSApp` could be answered as follows:

$$r_2 = (\text{startActivity(..)}, \text{source()}, \text{SIMAppMainActivity}, \{\text{SIMApp}\}) \in R$$
$$t_2 = (\text{de.upb.fpauck.CALLSINK}, \varepsilon, \varepsilon) \in I$$
$$A_2 = \{(r_2, t_2)\} \subset \mathcal{A}_{\text{IntentSinks}}$$

$$r_3 = (\text{getStringExtra(..)}, \text{sink()}, \text{SMSAppMainActivity}, \{\text{SMSApp}\}) \in R$$
$$t_{3,1} = (\varepsilon, \varepsilon, \text{SMSAppMainActivity}, \{\text{SMSApp}\}) \in R^C$$
$$t_{3,2} = (\text{android.intent.action.MAIN}, \text{android.intent.category.LAUNCHER}, \varepsilon) \in I$$
$$t_{3,3} = (\text{de.upb.fpauck.CALLSINK}, \varepsilon, \varepsilon) \in I$$
$$A_3 = \{(r_3, t_{3,1}), (r_3, t_{3,2}), (r_3, t_{3,3})\} \subset \mathcal{A}_{\text{IntentSources}}$$

$$A_4 = \{(r_2, r_3)\} \subset \mathcal{A}_{\text{Flows}}$$

$$A'' = A_2 \cup A_3 \cup A_4 \subset \mathcal{A}$$

A'' tells us that there exists a flow from the `startActivity(..)` statement in the `SIMApp` to the `getStringExtra(..)` statement in the `SMSApp`. It consists of three parts. On the one hand, A_2 contains information about the intent-sinks of the `SIMApp`. On the other hand, A_3 contains descriptions of the intent-sources of the `SMSApp`. We can conclude that there exists a flow between both apps, because one intent-sink shares an information triple with one intent-source. More precisely, t_2 is equal to $t_{3,3}$. Lastly, the set A_4 shows the flow between both apps and thereby answers question q''.

3.1.3 Attributes

Some information generated by analysis tools is very specific, for example, the permission groups assigned by PAndA[2]. This information should not get lost once the tool's result is transformed into an AQL-Answer. This is why the *attribute*-function is introduced consequently. Let the set $\mathscr{P}$ contain all name-value-pairs, that represent the analysis specific information, and $\mathscr{P}^*$ all available combinations of answer elements and the information

$$\mathscr{P} = \{p \mid p = (name, value)\}$$
$$\mathscr{P}^* = \{(a, p) \mid a \in \mathcal{A} \wedge p \in \mathscr{P}\}$$

then the function can be defined as:

$$attribute\colon \mathcal{A} \to 2^{\mathcal{P}}(\mathscr{P})$$

with

$$attribute(a) = \{p \mid (a, p) \in \mathscr{P}^*\}$$

Considering the answer A', for example, the permission groups generated by PAndA[2] can be restored:

$$\text{attribute}((r_1, \text{SEND\_SMS})) = \{(\text{permission group}, \text{REQUIRED})\}$$
$$\text{attribute}((r_1, \text{WRITE\_SMS})) = \{(\text{permission group}, \text{MISSING})\}$$

For now, we assume that the elements in $\mathscr{P}$ are known. Later on, it is described where this information is located and how it can be determined (cf. Section 3.3.2).

3.1.4 AQL-Operators

In this section three *AQL-Operators*, which can be used to combine or filter AQL-Answers, are introduced.

The *unify operator* unifies AQL-Answers. This means that it collects all information from two different AQL-Answers and puts it into one. The function *unify* defines the operator precisely:

$$\begin{array}{rlcl} \textit{unify}: & \mathcal{A} \times \mathcal{A} & \rightarrow & \mathcal{A} \\ & (A_1, A_2) & \mapsto & A_1 \cup A_2 \end{array}$$

The second operator is called *connect operator*. To define it, we first introduce a new set operator: $\sqcup$. It can be used to create a set of flows by connecting intent-sinks ($A_1 \subseteq \mathcal{A}_{\text{IntentSinks}}$) with intent-sources ($A_2 \subseteq \mathcal{A}_{\text{IntentSources}}$):

$$\begin{gathered} A_1 \sqcup A_2 \\ = \{a \mid \exists a_1 = (r_1, t_1) \in A_1 \ \exists a_2 = (r_2, t_2) \in A_2 \text{ with } t_1 = t_2 : a = (r_1, r_2)\} \subseteq \mathcal{A}_{\text{Flows}} \end{gathered}$$

The resulting set contains one flow from reference r_1 to r_2 for each intent-sink and intent-source that holds the same information triple or component reference as target ($t_1 = t_2$). With this operator the connect operator can be defined through the following *connect* function:

$$\textit{connect}: \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$$

with

$$\begin{aligned} \textit{connect}(A_1, A_2) = \quad & A_1 \cup A_2 \cup \\ & \Big(((A_1 \cup A_2) \cap \mathcal{A}_{\text{Flows}}) \cup \\ & \quad \big(((A_1 \cup A_2) \cap \mathcal{A}_{\text{IntentSinks}}) \sqcup \\ & \quad\ ((A_1 \cup A_2) \cap \mathcal{A}_{\text{IntentSources}}) \big) \\ & \Big)^* \end{aligned}$$

Equally to the unify operator it constructs the set-union of both inputs. Furthermore, all flows are added that are part of the transitive closure of all flows in both input sets as well as the flows constructed by connecting all intent-sinks and intent-sources of both input sets. For example, if the flows (r_x, r_y) and (r_y, r_z) exist, a flow from r_x to r_z is added. If in addition an intent-sink (r_v, t) and an intent-source (r_w, t) exists, the flow (r_v, r_w) is added as well. Let us assume there also exists a flow from r_z to r_v, then, for instance, the flow (r_x, r_w) is added as well since the whole transitive closure is considered.

The third operator takes only one input set. It is called *filter operator* and the function *filter1* specifies what it does:

$$\textit{filter1}\colon \mathcal{A} \to \mathcal{A}$$

with

$$S = \{\text{Permissions, IntentSinks, IntentSources}\} \subset \mathcal{S}$$

and

$$\begin{aligned}\textit{filter1}(A) = \quad & (A \cap \mathcal{A}_{\text{Intents}}) \cup (A \cap \mathcal{A}_{\text{IntentFilters}}) \cup \\ & \Bigg(\bigcup_{s \in S} \{a \mid \exists a_1 = (r, i) \in A \cap \mathcal{A}_s \;\; \exists a_2 = (r_x, r_y) \in A \cap \mathcal{A}_{\text{Flows}} \\ & \qquad \text{with } (r = r_x) \vee (r = r_y) : a = (r, i)\} \Bigg)\end{aligned}$$

It outputs the input set, but beforehand it removes all permissions, intent-sinks and -sources (a_1) whose reference does not appear in any flow paths (a_2). Intents and intent-filters from the input set are kept in the output set.

The filter operator can also be used together with a subject of interest in order to filter out all elements of the selected subject of interest. Function *filter2* describes how it works.

$$\textit{filter2}\colon \mathcal{A} \times \mathcal{S} \to \mathcal{A}$$

with

$$\textit{filter2}(A, s) = \bigcup_{s' \in \mathcal{S} \setminus \{s\}} A \cap A_{s'}$$

It is also possible to provide a name-value-pair along with the subject of interest.

$$\textit{filter3}\colon \mathcal{A} \times \mathscr{P} \times \mathcal{S} \to \mathcal{A}$$

with

$$p = (\text{name, value}) \in \mathscr{P}$$

and

$$\textit{filter3}(A, p, s) = \begin{cases} \{a \mid a \in A \text{ with } \textit{attribute}(a) = (\text{name, value})\} & \text{if } s = \varepsilon \\ \{a \mid a \in A \cap A_s \text{ with } \textit{attribute}(a) = (\text{name, value})\} \cup \textit{filter2}(A, s) & \textit{else} \end{cases}$$

In this case only those elements are kept in the output set, that have the chosen name-value-pair assigned or belong to a different subject of interest. If no subject of interest is specified ($s = \varepsilon$) only elements with the chosen name-value-pair assigned to it are kept in the output set.

3.1.5 AQL-Queries

In the previous section three AQL-Operators were introduced which are able to combine AQL-Answers. When to apply which operator is defined by AQL-Queries. The precise syntax of AQL-Queries is defined in form of a grammar in

Section 3.3.1. However, in the following two functions are introduced that allow us to describe these queries in an abstract way. First, the function *ask* is defined. It symbolically connects questions with their associated answers without providing information how to get or compute these answers:

$$\begin{array}{rccc} ask\colon & \mathcal{Q} & \rightarrow & \mathcal{A} \\ & q & \mapsto & A \end{array}$$

In this definition A refers to the answer associated with question q. More precisely, if question q is asked and there exists an analysis tool capable of answering this question, it replies a result that represents the answer A. In the next section, it is explained how an AQL-Answer is computed based on the input of an AQL-Question. But for now we assume that the answer to any question is known. With this function any query can be formulated through the function *query* which should rather be interpreted as an abstract sequence of instructions than a mathematical function:

$$query\colon \mathcal{Q} \cup \mathcal{A} \rightarrow \mathcal{A}$$

with

$$query(x) = \begin{cases} x & \text{if } x \in \mathcal{A} \\ ask(x) & \text{if } x \in \mathcal{Q} \end{cases}$$

In the following two examples are presented that explain how to interpret such queries. For example, query u' is used to ask question q':

$$\begin{array}{rl} u' = & query(q') \\ \rightarrow & ask(q') \\ \rightarrow & A' \end{array}$$

Each line started with an arrow symbol illustrates one instruction of the query. At the end of the sequence, only the answer to query u' is left. Another query (u''), that uses the unify operator in order to combine the answers to questions q' and q'', looks like:

$$u'' = query(unify(ask(q'), ask(q'')))$$

Interpreted as sequence of instructions, this query tells us that we first have to ask for the answers to q' and q'' before the answers to these questions can be combined with the unify operator which in turn can be used to produce the final answer. Thus, the following three steps have to be done in order to respond to the query:

$$\begin{array}{rl} u'' = & query(unify(ask(q'), ask(q''))) \\ \rightarrow & query(unify(A', A'')) \\ \rightarrow & query(A' \cup A'') \\ \rightarrow & A' \cup A'' \end{array}$$

Let these two queries (u' and u'') as well as any other constructible query belong to the set $\mathcal{U}$, the set of all AQL-Queries. Since the output of any query always is a single AQL-Answer, $\mathcal{U} \subseteq \mathcal{A}$ holds. Nevertheless, queries are more than answers once they are interpreted as sequences of instructions as described above.

Furthermore, let the function *parts* refer to all questions inside a query $u \in \mathcal{U}$:

$$\begin{array}{rccl} parts\colon & \mathcal{U} & \rightarrow & 2^{\mathcal{P}}(\mathcal{Q}) \\ & u & \mapsto & \{q_0, ..., q_n\} \text{ with } n \in \mathbb{N} \end{array}$$

This function abstractly describes the process of looking up all questions that are mentioned in a query u. For instance, $parts(u')$ is equal to $\{q'\}$ whereas $parts(u'')$ is equal to $\{q', q''\}$.

Queries such as u' and u'' are used as input by the system introduced in the next section.

3.2 AQL-System

In this section we describe the structure of a system using the AQL, namely an *AQL-System*. A very brief overview of such a system is presented in Figure 3.1. An AQL-System works like a black-box that takes an AQL-Query as input and computes a fitting AQL-Answer as output. Inside the black-box, the system is asking one or more analysis tools for their precise results. To do so, it requires a configuration that is visible to and editable from the outside to link these tools with the system. How to do so is described precisely in the next section. Afterwards, the processes inside the black-box are explained in detail.

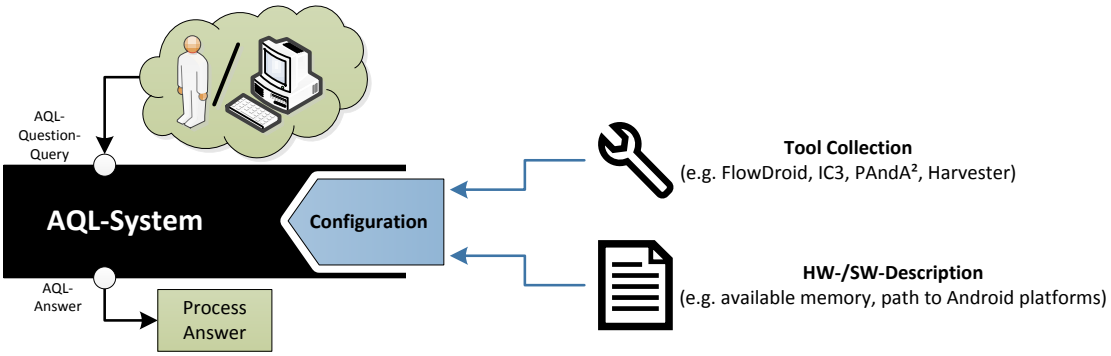

Figure 3.1: AQL-System Overview

3.2.1 Configuration

The configuration plays a central role in the conceptual design of an AQL-System, because it describes the links between the system and the tools that are executed while a question is being answered. It significantly decides how powerful and

extensible a system is. First of all, a configuration should contain all the relevant information about the hard- and software of the host, for example, how much memory is available or where the Android libraries can be found. Second, the configuration has to define a list of tools that can be asked for their precise results. For each tool in the list, general information such as the name and version of the tool has to be stored along with information about how and in which cases a certain tool should be executed and which type of information the result holds.

To answer the question which tool is executed in which case, we first want to differentiate between two types of tools, namely analysis tools and preprocessors. Whereas analysis tools like FlowDroid, IC3 or PAndA^2 take an app as input and produce a result of their own format as output, preprocessors also take an app or a description of an app as input and produce a preprocessed app version as output. For instance, Harvester in the role of a preprocessor can produce a deobfuscated version of an app. In the workflow section it is explained, that both types of tools come into play during different stages (see Section 3.2.2).

To determine which tool should be asked for its precise result or to preprocess an app, let us assume the set T_c is used to model analysis tools and preprocessors defined in a configuration c. Then, each piece of software $t \in T_c$ is described by its name:

$$T_c = \{t_0, ..., t_n\}, n \in \mathbb{N}$$
$$\text{For example: } T_c = \{\text{FlowDroid, PAndA}^2\text{, IC3, Harvester}\}$$

Furthermore, for each $t \in T_c$ at least one keyword has to be stored in configuration c. These keywords hint at the functionality of the tool. The function *keywords* returns these keywords for a certain analysis tool or preprocessor:

$$\textit{keywords}\colon T_c \to K_{\text{tools}} \cup K_{\text{preprocessors}}$$

with

$$K_{\text{tools}} = \{\text{IntraAppFlows, InterAppFlows}\} \cup (\mathcal{S} \setminus \{\text{Flows}\}) =$$
$$\{\text{IntraAppFlows, InterAppFlows, Intents, IntentFilters, IntentSinks, IntentSources, Permissions}\}$$
$$K_{\text{preprocessors}} = \{k_0, ..., k_n\}$$

and

$$\textit{keywords}(t) = \begin{cases} K_t \subseteq K_{\text{tools}} & \text{if } t \text{ is an analysis tool} \\ K_t \subseteq K_{\text{preprocessors}} & \text{if } t \text{ is a preprocessor} \end{cases}$$

Notice that the set of keywords to describe analysis tools (K_{tools}) is related to the set of subjects of interest ($\mathcal{S}$) introduced along with the AQL-Questions. While this set is predefined, the elements of $K_{\text{preprocessors}}$ are defined directly in the configuration. The *keywords* function helps to determine which tool or preprocessor should be used (see next section).

3.2.2 Workflow

The internal workflow of an AQL-System is depicted in Figure 3.2. As the black-box in the Figure 3.1, the AQL-System takes a query asked by a user or a program and outputs a fitting answer that can be processed further.

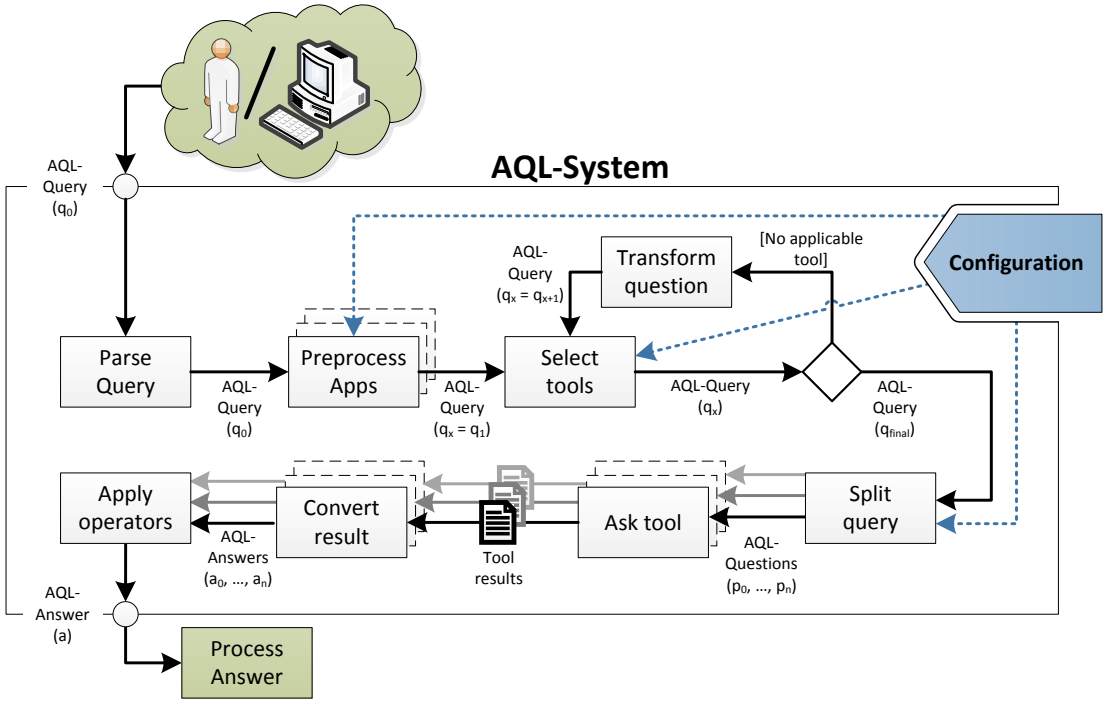

Figure 3.2: AQL-System Workflow

At first, an AQL-System parses the query (q_0). Then the apps that are involved in the query are preprocessed. On this stage various things can happen, for example, an app could be replaced with its up-to-date version, downloaded from a market, or a deobfuscated version, which is generated on-the-fly by Harvester. Which preprocessor should be executed exactly has to be declared in a question itself. Since the set *Apps* holds all available apps it also holds the preprocessed ones. Function *preprocess* defines how to address such a preprocessed version inside a question:

$$preprocess\colon\ Apps \times K_{\text{preprocessors}} \rightarrow Apps$$

with

$$k \in keywords(s)$$

and

$$preprocess(a, k) = a \text{ preprocessed by preprocessor } s = a'$$

In this context a' correlates to a version of app a preprocessed by a preprocessor t. Thus, if a configuration holds the keyword k_1 for preprocessor s_1, this preprocessor

can be applied in context of a question like q''', that asks for information about flows in the deobfuscated version of the `SIMApp`:

$$k_1 = \text{DEOBFUSCATE}$$
$$q''' = (\text{Flows}, \{(\varepsilon, \varepsilon, \varepsilon, \{\mathit{preprocess}(\text{SIMApp}, k_1)\})\})$$

After inserting the preprocessed apps, and thereby transforming q_0 into q_1, the system has to determine which tools should be executed (see "Select tools" in Figure 3.2). This step is executed in a loop. During each iteration of the loop, the system attempts to find analysis tools capable of answering all questions inside query q_x. The first iteration of the loop is started with $q_x = q_1$ as input. On the one hand, if all parts ($\mathit{parts}(q_x) = \{p_0, ..., p_n\}$) of query q_x can be answered with analysis tools declared in configuration c, the system exits the loop. On the other hand, if at least one part cannot be answered, because there is no suitable tool available, the system tries to transform the question. Thereby, q_x becomes q_{x+1} and the next iteration of the loop is started with $q_x = q_{x+1}$ as input. If the query cannot be transformed, the system cannot answer the query and has to abort[1].

The function *selectTool* comes into play, once we want to determine whether all questions inside a query can be answered with the available tools. It is used to match the subject of interest of each question $p \in \mathit{parts}(q_x)$ with keywords declared for a tool in configuration c. $\mathit{selectTool}(p) = s$ then refers to the tool that is capable of answering p. The function itself is defined as follows:

$$\mathit{selectTool}\colon Q \to T_c$$

with

$$\mathit{selectTool}(p = (s, R)) =$$

$$\begin{cases} t_{\text{soi}} & \text{if } \exists t_{\text{soi}} \in T_c : s \in \mathit{keywords}(t_{\text{soi}}) \\ t_{\text{inter}} & \text{if } \exists t_{\text{inter}} \in T_c : \text{InterAppFlows} \in \mathit{keywords}(t_{\text{inter}}) \wedge s = \text{Flows} \wedge [\ |R| = 2\ \wedge \\ & r_{x,1} = (s_1, m_1, c_1, a_1), r_{x,2} = (s_2, m_2, c_2, a_2) \in R \text{ with } a_1 \neq a_2\ ] \\ t_{\text{intra}} & \text{if } \exists t_{\text{intra}} \in T_c : \text{IntraAppFlows} \in \mathit{keywords}(t_{\text{intra}}) \wedge s = \text{Flows} \wedge [\ |R| = 2\ \wedge \\ & r_{x,1} = (s_1, m_1, c_1, a_1), r_{x,2} = (s_2, m_2, c_2, a_2) \in R \text{ with } a_1 = a_2\ ] \vee \\ & [\ |R| = 1\ ] \\ \varepsilon & \text{else} \end{cases}$$

If the subject of interest of the question considers information flows, depending on the references, t_{inter} or t_{intra} is selected. If it is only one reference or two references that refer to the same app, t_{intra} is selected. If in contrast two references refer to different apps, t_{inter} is selected. Otherwise if a different subject of interest than information flows is chosen, t_{soi} is picked. If there is no tool available in c that fits, a null value is output by the function.

For example, the query could consider flows between two different apps. In such a case the system looks for a tool that is capable of answering inter-app information

[1]For the sake of clarity this abort is not depicted in Figure 3.2.

flow questions. If such a tool is specified in the system's configuration, it can be executed in the next step. Otherwise the system can try to transform the query in order to produce a query that can be answered with the tools available. For instance, instead of asking for inter-app flows, we could ask for intent-sinks and -sources and connect them with the connect operator. Which transformations are feasible is up to the system itself. In the context of this thesis, we only consider one transformation described by the following rule:

If a query of the following format is detected and shall be transformed

$$\begin{aligned}&\mathit{query}(\\&\quad (\mathit{Flows}, \{r_1 = (s_1, m_1, c_1, A_1), r_2 = (s_2, m_2, c_2, A_2)\})\\&) \text{ with } A_1 \neq A_2\end{aligned}$$

it is transformed into the following one:

$$\begin{aligned}&\mathit{query}(\mathit{filter}(\\&\quad \mathit{connect}(\mathit{ask}((\mathit{Flows}, \{r_1\})),\\&\qquad \mathit{connect}(\mathit{ask}((\mathit{Flows}, \{r_2\})),\\&\qquad\quad \mathit{connect}(\mathit{ask}((\mathit{IntentSinks}, \{r_1\})),\\&\qquad\qquad\quad \mathit{ask}((\mathit{IntentSources}, \{r_2\}))\\&\qquad\quad)\\&\qquad)\\&\quad)\\&))\end{aligned}$$

Thereby an inter-app analysis can be composed of an intra-app analysis and an analysis capable detecting intent-sinks and -sources as presented in the running example. This rule is applied in the continuation of the running example at the end of this chapter.

The next step is titled "Split query". While this step is executed the query is split into its questions inside the query. Then finally the tools capable of answering the individual questions are asked for their precise results. In particular, for each question $p \in \mathit{parts}(q_{\text{final}})$ the tool $\mathit{selectTool}(p)$ is executed. How to execute these tools has to be specified in the configuration. Continuing the example mentioned before, one question could consider the intent-sinks and another one the intent-sources. This step as well as the preprocessing step can be done in parallel. Multiple instances of the same tool or different tools can be executed at the same time as long as the system requirements are not exceeded, for example, the tools executed at the same time require more memory than the host can offer.

Once a tool has finished its analysis, the produced result has to be converted into an AQL-Answer. Therefor, a tool specific converter is required that parses and interprets the produced result in order to output an AQL-Answer. If finally all results have been converted into AQL-Answers $(a_0, ..., a_n)$ the AQL-Operators are applied according to the order they were specified in the query. After this step

the AQL-System outputs a single AQL-Answer. This answer may be processed further afterwards, for instance, it could be evaluated, viewed or used in a bigger analysis.

3.3 AQL Syntax

In this section the syntax of the AQL that was introduced in the previous section is specified. Thereto, a grammar and an XML schema definition (XSD) is presented. During this presentation it is explained, why these two elements are sufficient to describe the syntax of AQL-Queries and AQL-Answers. Additionally, new and technical but content-independent elements of AQL-Queries and AQL-Answers are introduced along with an explanation why they are needed. At the end of this section, the presented syntax is used in a continuation of the running example.

3.3.1 Syntax of AQL-Queries

Grammar G in EBNF[2] format defines language $L(G)$ that can be used to formulate AQL-Queries (see Grammar 3.1). Such queries have to define which AQL-Questions should be answered and which AQL-Operators should be applied on the produced answers. Derivation examples using this grammar can be found in the running example at the end of this chapter.

The terminals (T) of G consist of 6 sets. The first two sets represent the subjects of interest of AQL-Questions: While T_{FromTo} stands for all subjects of interest that can be used together with two references in order to, for example, describe flows from one point to another, T_{In} stands for all subjects of interest that can only be used together with one reference. The next two sets, namely T_{op1} and T_{op2}, describe all three previously defined AQL-Operators. All operators that combine one or more answers are defined in T_{op1}. In contrast, T_{op2} holds the filter operator which can only be applied on a single answer. The next set consists of a collection of symbols that are required to structure the language $L(G)$. The last set adds arbitrary strings to the set of all terminals T.

The set N holds the non-terminals of grammar G. These are explained in the context of the set of production rules (P). In the following we walk through this set and thereby describe how to derive AQL-Queries.

AQL-Queries

The non-terminal $\langle query \rangle$ is defined as start symbol of G and used on the left-hand side of the first production rule ($p1$). This production rule tells us, that each $\langle query \rangle$ can be represented by one or more $\langle element \rangle$s. An $\langle element \rangle$ can be derived by applying $p2$:

[2]Extended Backus–Naur Form

$$
\begin{aligned}
G &= (T, N, P, S) \\
\\
T_{\text{FromTo}} &= \{\text{"Flows"}\} \\
T_{\text{In}} &= \{\text{"Permissions"}, \text{"IntentSources"}, \text{"IntentSinks"}, \text{"IntentFilters"}, \text{"Intents"}\} \\
T_{\text{op1}} &= \{\text{"UNIFY"}, \text{"CONNECT"}\} \\
T_{\text{op2}} &= \{\text{"FILTER"}\} \\
T &= T_{\text{FromTo}} \cup T_{\text{In}} \cup T_{\text{op1}} \cup T_{\text{op2}} \cup \\
&\quad \{\texttt{"?"}, \texttt{"!"}, \texttt{"'"}, \texttt{"FROM"}, \texttt{"TO"}, \texttt{"IN"}, \texttt{"->"}, \texttt{"("}, \texttt{")"}, \texttt{"["}, \texttt{"]"}, \texttt{"|"}, \\
&\quad \texttt{"="}, \texttt{"Statement"}, \texttt{"Method"}, \texttt{"Class"}, \texttt{"App"}\} \cup \\
&\quad \{w \mid w = (\text{any symbol except } \texttt{"'"})^+\} \\
N &= \{\langle answer \rangle, \langle element \rangle, \langle fromTo \rangle, \langle in \rangle, \langle operator \rangle, \langle operators1 \rangle, \\
&\quad \langle operators2 \rangle, \langle query \rangle, \langle question \rangle, \langle reference \rangle, \langle string \rangle, \langle soi \rangle, \\
&\quad \langle soiFromTo \rangle, \langle soiIn \rangle\} \\
S &= \langle query \rangle \\
P &= \{ \\
&\quad p1 : \langle query \rangle ::= \langle element \rangle^+, \\
&\quad p2 : \langle element \rangle ::= \langle question \rangle\ \texttt{"?"} \mid \langle answer \rangle\ \texttt{"!"} \mid \langle operator \rangle, \\
&\quad p3 : \langle question \rangle ::= \\
&\qquad\qquad (\langle soiFromTo \rangle\ (\langle fromTo \rangle \mid \langle in \rangle)) \\
&\qquad \mid\ (\langle soiIn \rangle\ \langle in \rangle), \\
&\quad p4 : \langle answer \rangle ::= \langle string \rangle, \\
&\quad p5 : \langle operator \rangle ::= \\
&\qquad\qquad (\langle operators1 \rangle\ \texttt{"["}\ \langle element \rangle\ (\texttt{","}\langle element \rangle)^*\ \texttt{"]"}) \\
&\qquad \mid\ (\langle operators2 \rangle\ \texttt{"["}\ \langle element \rangle \\
&\qquad\qquad (\texttt{"|"}\ \langle string \rangle\ \texttt{"="}\ \langle string \rangle)^? \\
&\qquad\qquad (\texttt{"|"}\ \langle soi \rangle)^?\ \texttt{"]"}), \\
&\quad p6 : \langle fromTo \rangle ::= \texttt{"FROM"}\ \langle reference \rangle\ \texttt{"TO"}\ \langle reference \rangle, \\
&\quad p7 : \langle in \rangle ::= \texttt{"IN"}\ \langle reference \rangle, \\
&\quad p8 : \langle reference \rangle ::= \\
&\qquad\qquad (\texttt{"Statement"}\ \texttt{"("}\ \langle string \rangle\ \texttt{")"}\ \texttt{"->"})^? \\
&\qquad\qquad (\texttt{"Method"}\ \texttt{"("}\ \langle string \rangle\ \texttt{")"}\ \texttt{"->"})^? \\
&\qquad\qquad (\texttt{"Class"}\ \texttt{"("}\ \langle string \rangle\ \texttt{")"}\ \texttt{"->"})^? \\
&\qquad\qquad \texttt{"App"}\ \texttt{"("}\ \langle string \rangle\ (\texttt{"|"}\ \langle string \rangle)^?\ \texttt{")"}, \\
&\quad p9 : \langle operators1 \rangle ::= \{w \mid w \in T_{\text{op1}}\}, \\
&\quad p10 : \langle operators2 \rangle ::= \{w \mid w \in T_{\text{op2}}\}, \\
&\quad p11 : \langle soi \rangle ::= \langle soiFromTo \rangle \mid \langle soiIn \rangle, \\
&\quad p12 : \langle soiFromTo \rangle ::= \{w \mid w \in T_{\text{FromTo}}\}, \\
&\quad p13 : \langle soiIn \rangle ::= \{w \mid w \in T_{\text{In}}\}, \\
&\quad p14 : \langle string \rangle ::= \texttt{"'"}\ (\text{any symbol except } \texttt{"'"})^+\ \texttt{"'"} \\
&\quad \}
\end{aligned}
$$

Grammar 3.1: Grammar of AQL Questions

- On the one hand, if an ⟨*element*⟩ is derived to ⟨*question*⟩ `"?"`, it indirectly refers to an answer. More precisely, to the answer of a question represented by ⟨*question*⟩. The question mark at the end highlights the end of such a question.

- On the other hand, if derived to ⟨*answer*⟩ `"!"`, it directly refers to a previously computed answer. The exclamation mark at the end distinguishes directly addressed answers from indirectly addressed answers. ⟨*answer*⟩ symbols can be derived to ⟨*string*⟩ symbols by rule $p4$. In this context ⟨*string*⟩s represent locations of .xml files which are used to store answers.

- If derived to ⟨*operator*⟩, the first thing expected is an AQL-Operator which in turn combines one or more ⟨*element*⟩s (cf. $p5$). This part is accurately described below in the paragraph about AQL-Operators.

Note that each AQL-Query might consist of arbitrary many AQL-Questions that can be combined by AQL-Operators or not.

AQL-Questions

Rule $p3$ is used to derive AQL-Questions. On the left-hand side the non-terminal ⟨*question*⟩ can be found. On the right-hand side two options can be chosen.

$$\begin{array}{lrcl} p3: & \langle question \rangle & ::= & (\langle soiFromTo \rangle\ (\langle fromTo \rangle\ |\ \langle in \rangle)) \\ & & | & (\langle soiIn \rangle\ \langle in \rangle) \end{array}$$

Option one is used to derive an AQL-Question that possibly considers more than one reference, for example, a question that asks for flows from a specific location to another. Option two is used to derive questions that always consider only one reference. Option one and two respectively use the non-terminals ⟨*soiFromTo*⟩ and ⟨*soiIn*⟩ to derive subjects of interest of the sets T_{FromTo} and T_{In} (cf. $p12$ and $p13$). Furthermore, the non-terminals ⟨$fromTo$⟩ and ⟨*in*⟩ are employed to structure the involved references and thereby produce easily readable questions.

References

Any reference can consist of at most four parts as shown before: $r = (s, m, c, A) \in R$ (see Section 3.1.1). In context of $p8$ each of these four parts is introduced with a keyword: `"Statement"`, `"Method"`, `"Class"` or `"App"`. While the first three parts are optional the last part is not. Each part is separated from another by an arrow symbol (`"->"`).

To this end, a statement should be defined by its Jimple representation. Hence, the ⟨*string*⟩ symbol is, for instance, derived to a string such as:

```
$r4 = virtualinvoke $r3<android.telephony.TelephonyManager: java.lang.String
                        getSimSerialNumber()>()
```

It depends on the system how these strings are processed. For example, while comparing the statement above to other statements it might be reasonable to ignore variables and only provide and use the generic version of the statement:

```
<android.telephony.TelephonyManager: java.lang.String getSimSerialNumber()>
```

Similarly, methods and classes should be defined by their Jimple representation. However, when it comes to apps the first ⟨*string*⟩ symbol somehow represents the app, for example, by its file path whereas the second, optional ⟨*string*⟩ symbol after the optional pipe (`"|"`) represents a preprocessor. One valid derivation to reference an app can, for example, look like the following one:

```
App('/path/to/apkfile.apk')
```

The .apk file is directly referred in order to precisely reference the app. It depends on the configuration of an AQL-System which preprocessors are available, but the following two derivations might be valid, too:

```
App('/path/to/apkfile.apk', 'DEOBFUSCATE')
App('pkg.class.app', 'PLAYSTORE')
```

The first one again references an app by its .apk file, but this time a preprocessor is defined. The keyword `'DEOBFUSCATE'`, for instance, could call the tool Harvester in order to deobfuscate the referenced app. The deobfuscated version of this app is used in order to answer the associated question then.
The second one references an app by its package- and classname. These are unique in Google's PlayStore[3]. Thus, a preprocessor addressed by the keyword `'PLAYSTORE'` could, for example, download the current version of a certain app from the PlayStore.

AQL-Operators

AQL-Operators are derived by rule $p5$. On the left-hand side of this rule we find the nonterminal ⟨*operator*⟩. On the right-hand side one of two alternatives can be chosen.

$$
\begin{array}{lrl}
p5: & \langle operator \rangle & ::= \quad (\langle operators1 \rangle \ \texttt{"["} \ \langle element \rangle \ (\texttt{","} \langle element \rangle)^* \ \texttt{"]"}) \\
 & & \ \ | \quad (\langle operators2 \rangle \ \texttt{"["} \ \langle element \rangle \\
 & & \qquad (\texttt{"|"} \ \langle string \rangle \ \texttt{"="} \ \langle string \rangle)^? \\
 & & \qquad (\texttt{"|"} \ \langle soi \rangle)^? \ \texttt{"]"})
\end{array}
$$

The first alternative considers AQL-Operators such as the unify or connect operator. These operators are referenced by the nonterminal ⟨*operators1*⟩ and can be further derived by rule $p9$. Furthermore, these operators come along with one or more ⟨*element*⟩s. These ⟨*element*⟩s refer to answers, as described before

[3] `https://play.google.com/store` - Accessed on 04/03/2017

(see paragraph about AQL-Queries), that should be combined with the assigned operator.
Note that arbitrary many ⟨*element*⟩s might be combined here. To do so, the functions representing the two operators have to be interleaved. Let us assume $a_1, a_2, ..., a_n \in A$ with $n \in \mathbb{N}$ refer to n answers, then the unify operator, for example, can be applied as follows:

$$\texttt{UNIFY [}a_1, a_2, ..., a_n\texttt{]}$$

Accordingly, the *unify* function has to be applied $n - 1$ times to compute the unified answer.

$$unify(a_1,\ unify(a_2,\ ...\ unify(a_{n-1}, a_n)...))$$

The second alternative considers the filter operator, which is derivable by rule $p10$. It always considers only one ⟨*element*⟩. Since the filter operator represents overall three functions, the optional parts of this alternative are used to distinguish which function should be utilized. If the shortest form without any optional parts is given, function $filter1$ is applied. The first optional part ((⟨*string*⟩ "=" ⟨*string*⟩)$^?$) can be used to define a name-value-pair. Thereby, the first occurrence of ⟨*string*⟩ represents the name. The second one represents the value. Once a name-value-pair is assigned, function $filter2$ is used. With the second optional part (("|" ⟨*soi*⟩)$^?$) a specific subject of interest that should be filtered may be added. If only a subject of interest and no name-value-pair is provided function $filter3$ is used, once the operator is applied. The selected subject of interest is filtered out in this case.

Annotations regarding Grammar G

Grammar G can easily be extended by adding subjects of interest or operators. For that purpose, we only have to change the respective sets of terminals. For example, in order to add a new operator, namely $unify2$, we just have to add "UNIFY2" to T_{op1}. Additionally, the possibilities to filter certain answers could quickly be extended. Instead of simple name-value-pairs, boolean terms could be supported. However, in the context of this thesis the included filter mechanisms are sufficient.

Next we discuss why the definition of grammar G (see Grammar 3.1) is imprecise to some extend. For example, the keywords to address certain preprocessors are not explicitly included in the list of terminals. Anyhow, they have to be defined in the configuration of a AQL-System.
Defining any part of references by arbitrary strings is imprecise as well. However, on the one hand, we can benefit from this imprecision, since we are not forced to use Jimple as intermediate representation. On the other hand, this can become a disadvantage once we try to combine answers that use different intermediate representations. This is why we suggest to use Jimple only.

3.3.2 Syntax of AQL-Answers

The XSD (see Listing A.1 in the Appendix) defines how AQL-Answers can be represented in a single XML document. Since most of the XSD's code is repetitive and neither self-explanatory nor easily readable, the XSD is not described in detail in the following. Instead we walk through examples of valid instances and thereby describe the schema defined by the XSD.

An XML document that represents a valid instance of this XSD can contain arbitrary many representations of flows, intents, intent-filters, intent-sinks, intent-sources and permissions. Each of those contains at least the information formally described in Section 3.1.2, even if this information is represented in a different way.

```
1  <reference>
2      <statement>
3          <statementfull>r1 = virtualinvoke r2.&lt;android.content.
               Intent: java.lang.String getStringExtra(java.lang.String)&
               gt;("Secret")</statementfull>
4          <statementgeneric>android.content.Intent: java.lang.String
               getStringExtra(java.lang.String)</statementgeneric>
5          <parameters>
6              <parameter>
7                  <type>java.lang.String</type>
8                  <value>"Secret"</value>
9              </parameter>
10         </parameters>
11     </statement>
12     <method>&lt;de.foellix.sinkapp.SinkMainActivity: void sink()&gt;<
           /method>
13     <classname>de.foellix.sinkapp.SinkMainActivity</classname>
14     <app>
15         <file>SMSApp.apk</file>
16         <hashes>
17             <hash type="MD5">505197d8859303afc60ffee8ff298f39</hash>
18             <hash type="SHA-1">9
                   c2ef3ad3b0376f6c023bfca15f402f0eb00c976</hash>
19         </hashes>
20     </app>
21 </reference>
```

Listing 3.1: References

Most elements refer to parts of an app by using references. References ($r = (s, m, c, A) \in R$) are represented as depicted in Listing 3.1. Each reference still consists of descriptions of a statement, a method, a class and an app. However, in this case each of these elements is described in more detail. A statement is defined by its original Jimple representation that was found by the analysis, a generic description of this statement and a detailed list of the involved parameters (cf. Lines 2-15). Methods and classes are only specified through their Jimple representation (cf. Lines 16-18 and 19-21). An app is described in two different

ways: On the one hand, the .apk file helps to directly recover an app. On the other hand, the declaration of hashes helps to recognize an app if the path, filename or environment has been changed. It can be checked whether an app equals the app used during the analysis by checking if their hashes match. Note that the declaration of a statement, method and classname is optional whereas the declaration of an app is required by the XSD.

Intents and intent-filters as well as intent-sinks and -sources require a way to describe the triple action, category and data (i = (action, category, data) $\in I$). Listing 3.2 shows how this triple can be described in context of the XSD.

```
<action>android.intent.action.VIEW</action>
<category>android.intent.category.BROWSABLE</category>
<data>
    <scheme>http</scheme>
    <host>www.website.com</host>
    <path>/path/on/site</path>
</data>
```

Listing 3.2: action, category, data

The elements action and category refer to the strings that, for example, can be found in the Android manifest. The data element is represented by several sub-elements that match the attributes available to describe data in Android. There are even more attributes accessible in Android then shown in Listing 3.2. However, the XSD offers possibilities to describe all of them. The triple that is described in Listing 3.2 could be assigned to an intent-filter, for example. Such an intent-filter would only accept intents that want to browse a website that can be found on the web under the following address: http://www.website.com/path/on/site.

The *attribute* function is implemented by arbitrary sets of attributes consisting of name and value tags. These describe the name-value-pairs that can be used to filter answers. Listing 3.3 shows an example.

```
<attributes>
    <attribute>
        <name>PermissionGroup</name>
        <value>REQUIRED</value>
    </attribute>
    <attribute>
        <name>FoundBy</name>
        <value>PAndA2</value>
    </attribute>
</attributes>
```

Listing 3.3: Attributes

From here on references, the action-category-data triple and attributes are shortened for the sake of clarity. In the following, examples for all subjects of interest are shown.

We can describe flows as demonstrated in Listing 3.4.

```
<flows>
```

```
2       <flow>
3           <reference type="from">...</reference>
4           <reference type="to">...</reference>
5           <attributes>...</attributes>
6       </flow>
7   </flows>
```

Listing 3.4: Flows

There can be multiple flow-tags included in one flows-tag. Each flow is described by two references that describe the start- (type="from") and endpoint (type="to") of the flow. Additionally, a set of attributes can be added to each flow.

Intents can be displayed differently depending on whether they are explicit or implicit (cf. Listing 3.5). Lines 2-10 show an implicit intent. The reference refers to the `startActivity(..)` statement that uses this intent. In this case, the target is described by the action-category-data triple. In case of an explicit intent (see Lines 11-16), the target is described by a reference to the targeted class. These references should neither have a statement nor a method assigned. To any of these intents attributes can be attached, even if the example does not show attributes for the explicit intent.

```
1   <intents>
2       <intent>
3           <reference>...</reference>
4           <target>
5               <action>...</action>
6               <category>...</category>
7               <data>...</data>
8           </target>
9           <attributes>...</attributes>
10      </intent>
11      <intent>
12          <reference>...</reference>
13          <target>
14              <reference>...</reference>
15          </target>
16      </intent>
17  </intents>
```

Listing 3.5: Intents

Intent-filters are defined by a reference, a target and optionally by attributes. The references of intent-filters refer to the component they are assigned to. Which intents are received by this component is described in the target-tag. See Listing 3.6 for an example.

```
1   <intentfilters>
2       <intentfilter>
3           <reference>...</reference>
4           <target>
5               <action>...</action>
6               <category>...</category>
```

```
7               <data>...</data>
8           </target>
9           <attributes>...</attributes>
10      </intentfilter>
11  </intentfilters>
```

Listing 3.6: IntentFilters

Intent-sinks and -sources are described by the exact same elements as visible in Listing 3.7. Each sink or source is respectively described by a reference that either points to a location where an intent is launched or a statement that somehow extracts information from an intent. Furthermore, the target of each intent-sink or -source is described through an action-category-data triple or a component reference, that allows us to match intent-sinks and -sources. Attributes can be added to both descriptions.

```
1  <intentsinks>
2      <intentsink>
3          <target>
4              <action>...</action>
5              <category>...</category>
6              <data>...</data>
7          </target>
8          <reference>...</reference>
9          <attributes>...</attributes>
10     </intentsink>
11     <intentsink>
12         <target>
13             <reference>...</reference>
14         </target>
15         <reference>...</reference>
16     </intentsink>
17 </intentsinks>
18
19 <intentsources>
20     <intentsource>
21         <target>
22             <action>...</action>
23             <category>...</category>
24             <data>...</data>
25         </target>
26         <reference>...</reference>
27         <attributes>...</attributes>
28     </intentsource>
29     <intentsource>
30         <target>
31             <reference>...</reference>
32         </target>
33         <reference>...</reference>
34     </intentsource>
35 </intentsources>
```

Listing 3.7: IntentSinks

Lastly, permissions are represented by their name, a reference and attributes (see Listing 3.8).

```
<permissions>
    <permission>
        <name>android.permission.SEND_SMS</name>
        <reference>...</reference>
        <attributes>...</attributes>
    </permission>
</permissions>
```

Listing 3.8: Permissions

The name-tag directly refers to the name of the permission which is predefined in the Android system or defined as a custom permission. The reference points at the location where the permissions is required. That might, for example, be a location where an intent is launched or otherwise a statement that attempts to access a resource like the camera of a mobile device.

There can be arbitrary many elements of each type and attributes can be assigned to any of these. All flows, intents, intent-filters, intent-sinks, intent-sources and permissions are collected as subelements of the answer tag. Following the rules of this indirectly described schema results in well-formed XML documents that represent complete AQL-Answers. In the next section, the running example is continued. The AQL-Answers presented there are valid instances of the introduced XSD.

3.3.3 Running Example (3/3): AQL in Practice

The running example associated with the `SIMApp` and the `SMSApp` is continued now. In order to get to know whether there exists a flow from a permission-protected source of the `SIMApp` to a permission-protected sink of the `SMSApp`, we formulate an AQL-Query. Considering this query and assuming that we got an AQL-System which is setup to use the tools FlowDroid, IC3 and PAndA^2, we explain the AQL-Answer expected as output of such a system.

The following derivation sequence depicts the derivation of the initial query:

$$
\begin{aligned}
& \langle \textbf{query} \rangle \\
\xrightarrow{p1} \quad & \langle \textbf{element} \rangle \\
\xrightarrow{p2} \quad & \langle \textbf{question} \rangle \texttt{ ?} \\
\xrightarrow{p3} \quad & \langle \textbf{soiFromTo} \rangle \ \langle fromTo \rangle \texttt{ ?} \\
\xrightarrow{p12} \quad & \text{Flows } \langle \textbf{fromTo} \rangle \texttt{ ?} \\
\xrightarrow{p6} \quad & \text{Flows FROM } \langle \textbf{reference} \rangle \texttt{ TO } \langle reference \rangle \texttt{ ?} \\
\xrightarrow{p8} \quad & \text{Flows FROM App(} \langle \textbf{string} \rangle \texttt{) TO } \langle reference \rangle \texttt{ ?} \\
\xrightarrow{p14} \quad & \text{Flows FROM App('SIMApp.apk') TO } \langle \textbf{reference} \rangle \texttt{ ?} \\
\xrightarrow{p8,p14} \quad & \text{Flows FROM App('SIMApp.apk') TO App('SMSApp.apk') ?}
\end{aligned}
$$

Each line symbolizes the appliance of one or more production rules of grammar G. Which rules are applied is denoted on top of the arrows on the left-hand side. The last line shows the final derivation of the initial query (q_0). Since there are no preprocessors assigned, the system can directly try to find applicable tools to answer all questions inside query q_0. In this case, it is only one question that considers inter-app flows. However, the configuration of the assumed AQL-System does not offer any applicable tool, because FlowDroid is only capable of answering intra-app flow questions and IC3 as well as PAndA^2 cannot answer any flow questions at all. According to this the function *selectTool* returns a null value for query q_0. As a consequence the system attempts to transform the query. Thereto, the transformation rule described in Section 3.2.2 is used. The transformed query q_1 looks as follows:

```
FILTER [
    CONNECT [
        Flows IN App('SIMApp.apk') ?,
        Flows IN App('SMSApp.apk') ?,
        IntentSinks IN App('SIMApp.apk') ?,
        IntentSources IN App('SMSApp.apk') ?
    ]
]
```

Query q_1 is similar to the query q_0 even if it looks completely different. The two flow questions contained in q_1 consider the intra-app flows from source to intent-sink and from intent-source to sink. Asking for intent-sinks of the `SIMApp` and intent-sources of the `SMSApp` and combining the associated answers with the connect operator allows us to construct the inter-app flow part of the complete flow visualized in Figure 2.2 on Page 10. Furthermore, connecting this flow with the two intra-app flows gives us the complete flow. Applying the filter operator on top removes irrelevant intent-sinks and -sources as well as flows that are not part of the complete flow.

Now, the function *selectTool* can find a tool for each of the four questions included in query q_1. FlowDroid can handle the intra-app flow questions whereas IC3 is able to answer the intent-sink and -source questions. Once the results produced by these tools are computed and converted into AQL-Answers, these answers can be combined by applying the connect operator. Thereafter, the filter operator can be applied and the final answer can be output. According to our expectations the answer to query q_1 should look as denoted in Listing 3.9. The denoted answer is shortened to only show the essential parts for the sake of clarity.

```
1  <?xml version="1.0" encoding="utf-8" standalone="yes"?>
2  <answer>
3      <intentsources>
4          ...
5          <intentsource>
6              <target>
7                  <action>de.upb.fpauck.CALLSINK</action>
8                  ...
9              </target>
10             <reference>
11                 <statement>
12                     ...
13                     <statementgeneric>android.content.Intent: java.
                           lang.String getStringExtra(java.lang.String)</
                           statementgeneric>
14                     ...
15                 </statement>
16                 ...
17                 <app>
18                     <file>SMSApp.apk</file>
19                     ...
20                 </app>
21             </reference>
22         </intentsource>
23     </intentsources>
24
25     <intentsinks>
26         <intentsink>
27             <target>
28                 <action>de.upb.fpauck.CALLSINK</action>
29             </target>
30             <reference>
```

```
31                  <statement>
32                      ...
33                      <statementgeneric>de.upb.fpauck.simapp.
                            SIMAppMainActivity: void startActivity(android
                            .content.Intent)</statementgeneric>
34                      ...
35                  </statement>
36                  ...
37                  <app>
38                      <file>SIMApp.apk</file>
39                      ...
40                  </app>
41              </reference>
42          </intentsink>
43      </intentsinks>
44
45      <flows>
46          ...
47          <flow>
48              <reference type="from">
49                  <statement>
50                      ...
51                      <statementgeneric>android.telephony.
                            TelephonyManager: java.lang.String
                            getSimSerialNumber()</statementgeneric>
52                  </statement>
53                  ...
54                  <app>
55                      <file>SIMApp.apk</file>
56                      ...
57                  </app>
58              </reference>
59              <reference type="to">
60                  <statement>
61                      ...
62                      <statementgeneric>android.telephony.SmsManager:
                            void sendTextMessage(java.lang.String,java.
                            lang.String,java.lang.String,android.app.
                            PendingIntent,android.app.PendingIntent)</
                            statementgeneric>
63                      ...
64                  </statement>
65                  ...
66                  <app>
67                      <file>SMSApp.apk</file>
68                      ...
69                  </app>
70              </reference>
71              <attributes>
72                  <attribute>
73                      <name>complete</name>
74                      <value>true</value>
```

```
75                    </attribute>
76                </attributes>
77            </flow>
78        </flows>
79    </answer>
```

Listing 3.9: Expected AQL-Answer

The answer consists of three parts: Intent-sources, intent-sinks and flows. The intent-source part shows the defined action for this source, namely `de.upb.fpauck.CALLSINK`. The intent-sink part shows this action string, too. Thereby, it can be assumed that these apps can communicate with each other. More precisely, there exists a flow from the `startActivity(..)` statement of the `SIMApp` to the `getStringExtra(..)` statement of the `SMSApp`. This flow as well as the two expected intra-app flows are part of the flows contained in this answer. However, to display this answer in a compact way only the constructed complete flow is denoted. It leads from `getSimSerialNumber()` to `sendTextMessage(..)` and is recognizable as complete flow by the assigned attribute.

This AQL-Answer almost completely answers our initial question, except the fact that our source and sink should require a permission. Thus, we want to check if the detected source and sink require a permission. In particular, which permissions are required by the `getSimSerialNumber()` and the `sendTextMessage(..)` statement. Let us assume that the answer presented above is stored in the file `answer.xml`, then we can formulate the following query in order to attach the permissions:

```
UNIFY [
    'answer.xml' !
    Permissions IN
        Statement('android.telephony.TelephonyManager: java.lang.String
            getSimSerialNumber()')
        -> Method('de.upb.fpauck.simapp.SIMAppMainActivity: void source()')
        -> Class('de.upb.fpauck.simapp.SIMAppMainActivity')
        -> App('SIMApp.apk')
    ?,
    Permissions IN
        Statement('android.telephony.SmsManager: void sendTextMessage(..)')
        -> Method('de.upb.fpauck.smsapp.SMSAppMainActivity: void sink()')
        -> Class('de.upb.fpauck.smsapp.SMSAppMainActivity')
        -> App('SMSApp.apk')
    ?
]
```

PAndA[2] is used to answer the two new questions. After that the unify operator is applied in order to unify the final result. This adds the lines denoted in List-

ing 3.10 to the previously shown answer.

```
...
    <permissions>
        <permission>
            <name>android.permission.READ_PHONE_STATE</name>
            <reference>
                <statement>
                    ...
                    <statementgeneric>android.telephony.
                        TelephonyManager: java.lang.String
                        getSimSerialNumber()</statementgeneric>
                </statement>
                ...
                <app>
                    <file>SIMApp.apk</file>
                    ...
                </app>
            </reference>
            <attributes>
                <attribute>
                    <name>PermissionGroup</name>
                    <value>REQUIRED</value>
                </attribute>
            </attributes>
        </permission>
        ...
        <permission>
            <name>android.permission.SEND_SMS</name>
            <reference>
                <statement>
                    ...
                    <statementgeneric>android.telephony.SmsManager:
                        void sendTextMessage(java.lang.String,java.
                        lang.String,java.lang.String,android.app.
                        PendingIntent,android.app.PendingIntent)</
                        statementgeneric>
                    ...
                </statement>
                ...
                <app>
                    <file>SMSApp.apk</file>
                    ...
                </app>
            </reference>
            <attributes>
                <attribute>
                    <name>PermissionGroup</name>
                    <value>REQUIRED</value>
                </attribute>
            </attributes>
        </permission>
```

```
45            ...
46        </permissions>
47    ...
```

Listing 3.10: Expected AQL-Answer

These lines tell us, that the `getSimSerialNumber()` statement is requiring the `READ_PHONE_STATE` permission whereas the `sendTextMessage(..)` statement requires the `SEND_SMS` and `WRITE_SMS` permissions. Furthermore, the permission-groups assigned by PAndA[2] are visible as attributes.

The complete AQL-Answer is pictured in Figure 3.3. All references that are part of this answer are shown at the bottom in blue. The blue arrows under these references symbolize the flows. All bold arrows stand for flows that are not part of another flow. These are marked by the attribute `complete` with the value `true` (see Lines 73-74 in Listing 3.9). Intent-sources can be found on the right-hand side in red color whereas intent-sinks are placed on the left-hand side in green color. Lastly, the permissions can be found on the top of the picture in purple. The red, green and purple arrows connect the associated elements with the references they refer to. This picture reveals, for example, the connection between the `READ_PHONE_STATE` permission and the `SEND_SMS` permission. We can simply follow the edges from one permission to another.

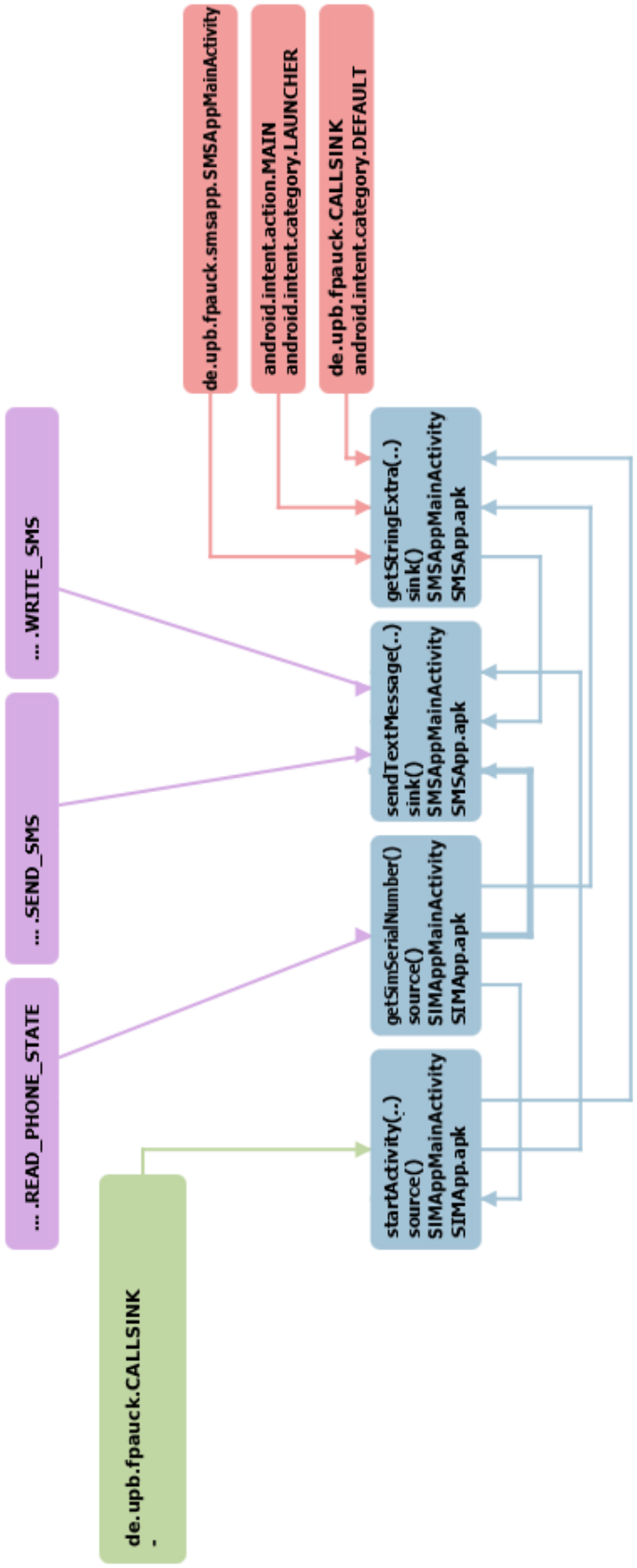

Figure 3.3: AQL-Answer as Graph

4 Implementation

In the previous chapter the AQL along with the concept of a system using the AQL was introduced. Now the Java implementation of a tool that follows this concept is described. From hereon the term *AQL-System* refers to this tool. At first in this chapter a short overview is given that highlights how this implementation is linked to the concept and which technologies are used to construct it. The structure of this AQL-System is briefly presented thereafter. Furthermore, it is explained how to configure, install and use the tool.

4.1 Overview

The previous chapter contained a detailed description of the different steps a system has to perform in order to fully use the AQL (cf. Figure 3.2 on page 34). These steps are summarized in the following four phases:

1. *Parsing* – During this phase, the input AQL-Query is parsed.
2. *Preprocessing* – While this phase is executed, all preprocessors mentioned in the query are executed.
3. *Analyzing* – During the third phase, the analysis tools, specified in the configuration and selected by the system in order to answer the query, are executed.
4. *Converting* – In this last phase, the results of all executed tools are converted into AQL-Answers and combined with the associated AQL-Operators.

Before the implementation details of these four phases are described, the configuration used by the AQL-System is described.

4.1.1 Configuration

Any tool can typically be configured through parameters or one or more configuration files. In case of the AQL-System, configuration files in XML format are used. The XSD in Appendix A.2 describes the structure of such files and is involved in the generation of Java classes that represent a configuration. For this purpose JaxB[1], a native part of Java, is utilized. JaxB allows us to generate the

[1] Java Architecture for XML Binding: `https://jaxb.java.net/` - 04/19/2017

required classes and comes with a parser which is able to translate an XML configuration file into object instances of these generated classes. As described in the last chapter (cf. Section 3.2.1), each configuration file must contain the required information about the AQL-System's environment and the preprocessors and analysis tools used by the system. Listing 4.1 shows an exemplary configuration file.

```
1  <config>
2      <androidPlatforms>/path/to/android-sdks/platforms</
           androidPlatforms>
3      <maxMemory>8</maxMemory>
4      <tools>
5          <tool name="Analysis tool 1" version="1">
6              <questions>IntraAppFlows, InterAppFlows</questions>
7              <priority>1</priority>
8              <instances>0</instances>
9              <memoryPerInstance>4</memoryPerInstance>
10
11             <path>/execution/environment/of/this/tool</path>
12             <run>/execution/environment/of/this/tool/run.sh %MEMORY%
                   %APP_APK%</run>
13             <result>/execution/environment/of/this/tool/%
                   APP_APK_FILENAME%_result</result>
14         </tool>
15     </tools>
16     <preprocessors>
17         <tool name="Preprocessor 1" version="1">
18             <questions>PREPROCESS</questions>
19             ...
20             <result>/execution/environment/of/this/preprocessor/%
                   APP_APK_FILENAME%_preprocessed.apk</result>
21         </tool>
22     </preprocessors>
23 </config>
```

Listing 4.1: Example of a Configuration file

The provided information about the system's environment is shown in the Lines 2 and 3. On the one hand, the path to Android's platform files is referenced. Each of these files represents the API of a certain version of the Android operating system. Most analysis tools require these files in order to execute their analyses. On the other hand, the maximum of memory that can be occupied by the system must be specified. The two elements `tools` and `preprocessors` can hold arbitrary many `tool`-elements. Each `tool`-element describes one analysis tool or preprocessor respectively. A tool is identified by the values assigned to the two attributes `name` and `version` (see Line 5 and 17). The following subelements of each `tool`-element describe in which case a tool may be executed:

- `<questions>`: The content of this tag tells us which AQL-Questions can be answered with the associated tool. With "Analysis tool 1", for example, intra- and inter-app flow questions can be answered (see Line 6 in Listing 4.1). In the context of preprocessors the keyword associated with a

certain preprocessor is defined here (see "PREPROCESS" in Line 18 in Listing 4.1).

- `<priority>`: If there are two or more tools available which are capable of answering the same AQL-Questions the priority decides which tool is executed. The tool with the highest priority is always preferred. However, if the execution of this tool fails, the system tries another one with the next lower priority. It is also possible to specify the same tool twice with different settings. If, for example, the execution of a tool fails, because it ran out of memory, another tool or the same tool with different settings, aiming to optimize its memory consumption, may not fail.

- `<instances>`: This subelement tells us how many instances of the associated tool can be executed concurrently. A value of 0 states that arbitrary many instances may be executed at the same time.

- `<memoryPerInstance>`: This defines how much memory is provided to each instance of the associated tool.

The AQL-System executes different tools in parallel to answer different parts of an AQL-Query, if and only if the sum of all values stored in the different `<memoryPerInstance>`-tags does not exceed the available memory (see `<maxMemory>`). Furthermore, if the same tool should be executed twice, it is not allowed to exceed the number of instances defined by the `<instances>` tag.

How to execute a preprocessor or an analysis tool is specified through the following subelements:

- `<path>`: This describes a path to the directory where the tool shall be executed. In the example shown in Listing 4.1 the path is set to "/execution/environment/of/this/tool" (see Line 11). Thereby it is possible to, for instance, refer to .apk files stored in this location directly without providing the complete path again.

- `<run>`: The run tag describes how to call a certain tool by defining the location of a bash script, for example.

- `<result>`: This describes where the result can be found, once a tool finishes successfully. (A *-symbol can be used inside this tag to reference an arbitrary substring.)

In the last three subelements the following variables can be used:

%APP_APK%:	The .apk file referenced in an AQL-Question
%APP_APK_FILENAME%:	The filename of the .apk file without path and ending
%APP_NAME%:	The app's name specified in its manifest
%APP_PACKAGE%:	The app's package specified in its manifest
%ANDROID_PLATFORMS%:	The Android platforms folder (Specified through `<androidPlatforms>`)
%MEMORY%:	The memory available to an instance of a tool (Specified through `<memoryPerInstance>`)

These are replaced at runtime with the corresponding values.

4.1.2 Implementation Details

During the first phase, namely the Parsing phase, the AQL-System uses a generated parser to parse the input query. To generate this parser JavaCC[2] is utilized. JavaCC is able to generate a parser on the basis of a grammar. In this case we used the grammar described in the last chapter (see Grammar 3.1 on Page 38). By generating this parser, we can assume a lower probability of errors in its code and ensure that the language we specified through the grammar is understood exactly. Furthermore, if the AQL is adapted in future this allows us to easily update the parser.

During the Preprocessing phase, preprocessors, specified in the configuration and referred to in the input query, are executed. They can be run concurrently if enough memory is available. The same preprocessor might be executed multiple times as long as the number of allowed concurrent instances is not exceeded. Furthermore, the AQL-System executes a preprocessor only once per app and stores the result. Whenever the same combination of app and preprocessor is used in the same or in a different query it simply loads the result again. This provides the advantage that we can speed up or completely skip the preprocessing phase if we are able to reuse previously computed results.

The Analyzing phase follows upon the Preprocessing phase. During the Analyzing phase, the AQL-System first checks whether it is able to answer the query in its initial form. If not, it tries to transform the query. The implementation described in this chapter only implements one transformation rule which was also used in the third part of the running example (see Section 3.3.3). Assuming that the references x, y refer to two different apps, this transformation rule allows the system to replace a question of the format:

Flows `FROM` x `TO` y `?`

with the following one:

[2] Java Compiler Compiler: `https://javacc.org` - 04/19/2017

```
FILTER [
    CONNECT [
        Flows IN x ?,
        Flows IN y ?,
        IntentSinks IN x ?,
        IntentSources IN y ?
    ]
]
```

If there is no tool available to answer a certain question and the transformation rule cannot be applied, the AQL-System tries to load a previously computed answer. For example, imagine two tools that can only be executed on different operating systems: In such a case we can run one tool first and store the answer. Later on we can run the other tool and use the previously computed answer to respond to the complete query. The AQL-System also loads answers if it has executed the exact same tool in order to answer the exact same question before. This avoids the redundant execution of tools and thereby speeds up the analysis process. If the question slightly differs the answer can either be loaded or computed depending on the scope of the underlying analysis tool. For example, the tool FlowDroid has to be run only once per app. Thus, if we reference different methods of the same app in two distinct questions, these questions are different, but the tool is executed in the exact same way. Because of that, there is no reason to compute the answer again. It should be preferred to load the answer since this speeds up the analysis process. Just like the preprocessors, analysis tools are executed in parallel during this phase if and only if it does not violate memory or number-of-instances constraints.

In the last phase, namely the Converting phase, the results computed by analysis tools are converted into AQL-Answers. Therefore, tool-specific converters are required. They have to read the result of an analysis tool and generate an AQL-Answer that contains the same information. To this end, a result is parsed and interpreted. Consequently objects that hold the thereby collected information are instantiated. In the context of the implementation described in this chapter, these objects are instances of generated datastructure classes. These classes are used to represent AQL-Answers. For this purpose, the XSD used to define the syntax of AQL-Answers (see Appendix A.1) is converted into Java classes by JaxB. This guarantees that the generated answers are valid AQL-Answers. As a last step in this phase the AQL-Operators are applied.

4.2 Structure

The structure of the AQL-System is depicted in Figure 4.1. It shows a simplified UML[3] class diagram. For the sake of clarity cardinalities as well as most attributes and methods have been omitted. The `System` class marked with green color plays the most important, central role in this diagram. It has access to all other parts and is made accessible to the user through the user interface, which is symbolized by the `ui`-package in the top left corner. Note that the `CLI` class of the `ui` package contains Java's `main(..)` method. The classes marked with blue color have been generated with JaxB or JavaCC. This highlights that numerous classes are generated. Thereby many possible programming errors are avoided. The red converter classes are the only classes that are tool-specific. Once a new tool is introduced to this system, a new converter has to be implemented.

Based on the execution example shown in the UML sequence diagram in Figure 4.2, it is explained how the different classes interact with each other. On the left hand side the different phases are marked. Next to these an actor that represents the user of the system is symbolized. By calling the `main(..)` method of the `CLI` class he starts the Parsing phase. First, the arguments of the question are read. Second, the query which has to be one of the arguments is forwarded to the `System` class by calling the `query(..)` method. At this point the query is represented by a single string. Because of that the system calls the `Parser` (see 3. in Figure 4.2). The `Parser` creates a `Question` object that represents the query (`q0`). As depicted in the class diagram one question can consist of several other questions. In this regard the query `q0` consists of several parts which are represented by `Question` objects as well (`p0, ..., pn`). Once the system receives `q0`, it requests all references used inside the query. For each reference that should be preprocessed, the system selects a suitable preprocessor from the configuration and, consequently, generates a `PreprocessorTask` object. In the sequence diagram only one is generated for `preprocessor0`. This task is used to start the preprocessor itself in a separate process. Thereby several preprocessors could be executed at the same time. Once the preprocessor finishes his computation the associated `PreprocessorTask` asynchronously calls the `preprocessingFinished(..)` method (see 8. in Figure 4.2). Once all `PreprocessorTask` have called this method, the Preprocessing phase ends. The Analyzing phase works similar to the Preprocessing phase. For each part of `q0` a tool is selected which is executed in its own `ToolTask`. Once an analysis tool finishes its computation, the `answerAvailable(..)` method is called. In the following Converting phase all result files computed by the analysis tools are converted into `Answer` objects. As a last step before the system returns the final answer, it has to apply the AQL-Operators assigned in query `q0` (see 14. in Figure 4.2).

[3]Unified Modeling Language: `http://www.omg.org/spec/UML/` - 05/22/2017

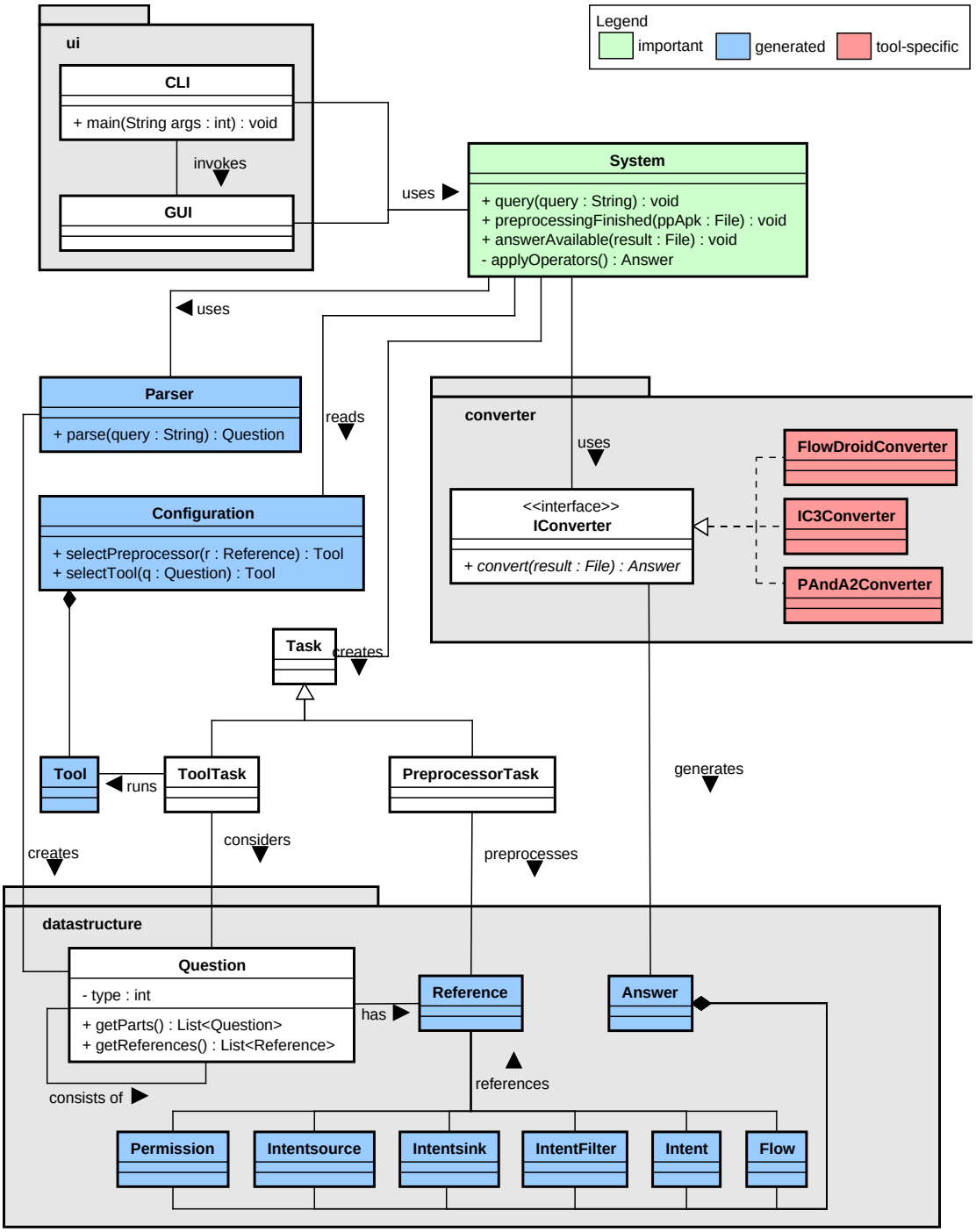

Figure 4.1: AQL-System: UML Class Diagram

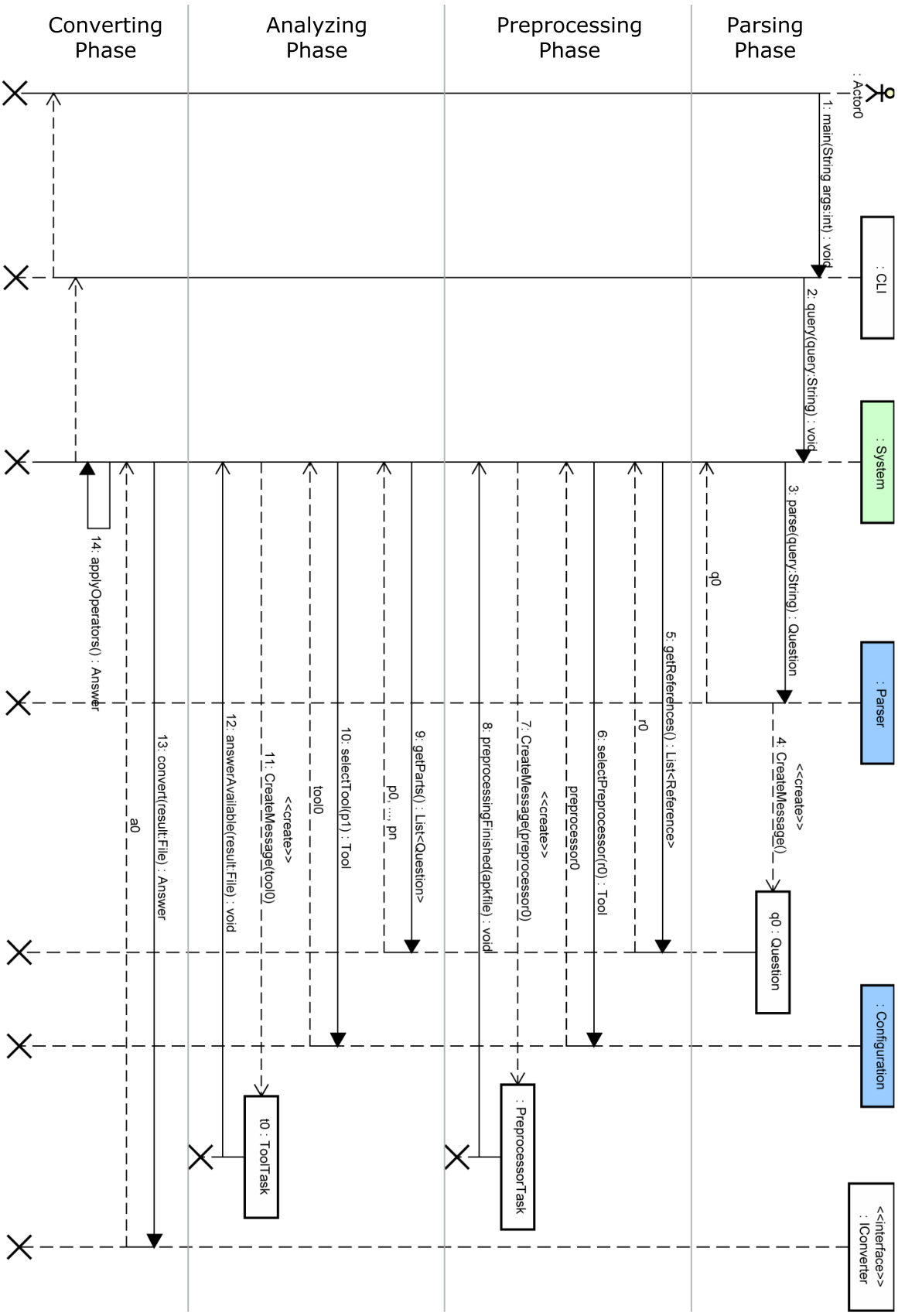

Figure 4.2: AQL-System: UML Sequence Diagram

4.3 Manual

Installation The CD-Rom attached to this thesis contains the AQL-System as an executable .jar file as well as the Eclipse[4] project used to built it. Note that Java 1.8 or newer is required in order to run and build the AQL-System. However, no preprocessor and only one analysis tool, namely PAndA^2, is shipped with the tool. While PAndA^2 can be found in the "Tools" directory, FlowDroid[5] and IC3[6] can be obtained online. In the "Tools" directory several bash and batch scripts can be found next to PAndA^2. These can be used to execute FlowDroid, IC3 and PAndA^2 on Linux or FlowDroid and PAndA^2 Windows. Furthermore, these scripts are included in a exemplary configurations (see "Examples" directory). Once all tools that should be used are installed, the configuration file ("config.xml") has to be adapted. How to adapt the configuration is explained in Section 4.1.1. Thereafter the AQL-System is ready to be used.

Usage The AQL-System can be accessed from the command-line through the following command:

java -jar /path/to/AQLSystem.jar

Furthermore, the parameters shown in Table 4.1 may be attached. The following command, for example, queries the system in order to ask for all permissions used by the `SIMApp`:

java -jar /path/to/AQLSystem.jar -query "Permissions IN App('SIMApp.apk') ?" -o "result.xml"

The answer to this query is stored in the file "result.xml". It is also possible to use the AQL-System inside another Java project. Therefore, add the AQLSystem.jar as library and use the following three instructions to import the `System` class and to invoke a query:

```
import de.foellix.aql.system.System;

final System system = new System();
system.query("Permissions IN App('SIMApp.apk') ?");
```

Furthermore, the method `getAnswerReceivers()` of a `System` class object can be used to obtain and edit a list of objects[7]. Each object in the list receives all answers computed by the associated system.

A screenshot of the AQL-System's graphical user interface (GUI) is shown in Figure 4.3. The GUI consists of two parts, namely the *Editor* to formulate AQL-

[4] Eclipse IDE: `https://www.eclipse.org` - 04/20/2017

[5] `https://github.com/secure-software-engineering/soot-infoflow-android/wiki` - 04/20/2017

[6] `http://siis.cse.psu.edu/ic3/` - 04/20/2017

[7] These objects have to implement the interface `IAnswerAvailable`

Parameter	Meaning
-help, -h, -?, -man, -manpage	Outputs a very brief manual, which contains a list of all available parameters.
-query "X", -q "X"	This parameter is used to assign the AQL-Query. X refers to this query.
-config "X", -cfg "X", -c "X"	By default the config.xml file in the tool's directory is used as configuration. With this parameter a different configuration file can be chosen. X has to reference the path to and the configuration file itself.
-output "X", -out "X", -o "X"	The answer to a query is automatically saved in the "answers" directory. This parameter can be used to store it in a second file. X has to define this file by path and filename.
-preferexecute, -pe	This parameter is used to force the execution of analysis tools even if a similar question has been asked before.
-timeout "X"s/m/h, -t "X"s/m/h	With this parameter the maximum execution time of each tool can be set. If it expires the tool's execution is aborted. X refers to this time in seconds (e.g. 10s), minutes or hours.
-debug "X", -d "X"	The output generated during the execution of this tool can be set to different levels. X may be set to: "error", "warning", "normal", "debug", "detailed" (ascending precision from left to right). By default it is set to "normal".
-gui	If this or no parameters at all are provided the graphical user interface is started. It allows to formulate queries and display answers in a handy way.

Table 4.1: Parameters

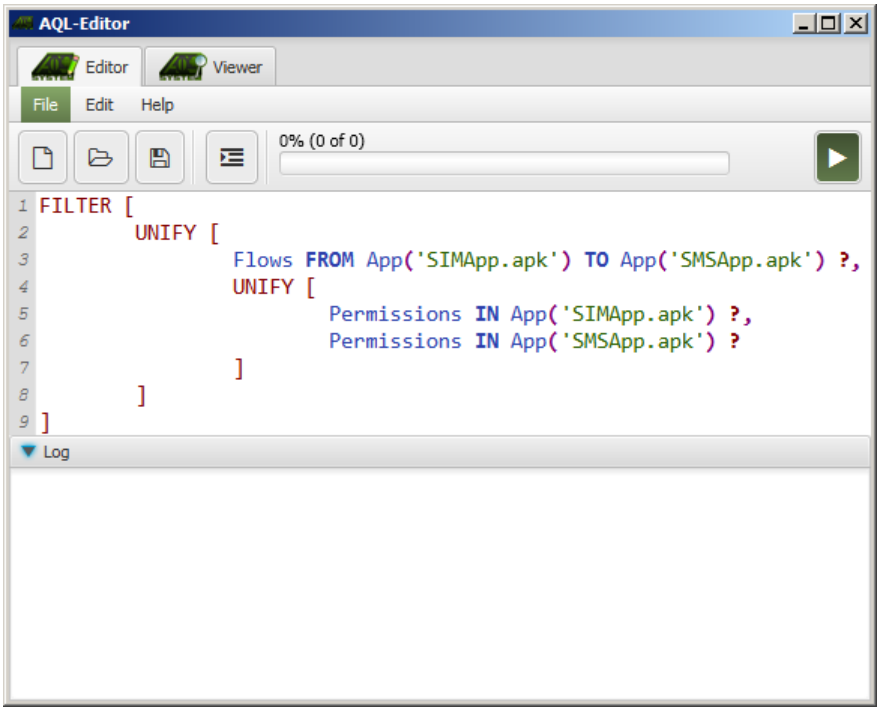

Figure 4.3: Screenshot of AQL-System's GUI

Queries and the *Viewer* to view AQL-Answers. The Editor is capable of highlighting and auto-formatting AQL-Queries as well as asking them by clicking the play-button at the top right corner. Once a query is asked, the bottom area titled with "log" is used to display status messages including errors and warnings. The progressbar at the top shows how many preprocessors and analysis tools have to be executed until the query can be answered. The viewer is not visible in the screenshot. It offers two possibilities to display an AQL-Answer: On the one hand, a textual view with syntax highlighting can be used to directly view and edit an .xml file representing an AQL-Answer. On the other hand, a graphical view shows the flows, permissions, intent-sources and -sinks in a single graph. Thereby the graphical view provides a better overview. An example of such a graph is drawn in Figure 3.3 on Page 53 and was explained at the end of Section 3.3.3.

5 Evaluation

In the previous chapters the AQL, including a concept and an implementation of a system using it, was described. Now this implementation is used to highlight the advantages of asking different expert tools for their precise results. Additionally, other benefits achievable through the AQL are pointed out. In doing so, the following *research questions* (RQ) are answered:

RQ1 Can analysis tools combined through the AQL be more precise than...

a) immature tools?

b) mature tools?

In order to answer these questions we evaluate whether an analysis composed of different analysis tools can be more precise than a single tool which performs the same analysis. During this evaluation custom apps and benchmark apps are analyzed by the AQL-System.

RQ2 Can the power of a single analysis tool be increased by means of the AQL? For example, preprocessors can be used along with the AQL. These may positively influence the output of an analysis tool by adapting its input.

RQ3 Is the AQL-System capable of analyzing sets of one or more real world apps efficiently?
To this end, it is evaluated if new knowledge about a set of apps can be gained by reusing and combining stored AQL-Answers computed for single apps only.

RQ4 Can the AQL improve the performance of an analysis?
Hence, it is evaluated if the execution time and/or memory consumption of an analysis can be decreased. For instance, by reusing results of one analysis in order to enhance another.

The next sections address each research question in detail. At the end of this chapter the answers to all question are summarized.

5.1 RQ1a: Can analysis tools combined through the AQL be more precise than immature tools?

To answer RQ1a we assume that immature tools are as precise as combinations of different expert tools and show a contradiction to this assumption. Consequently,

analysis results computed by an immature tool are directly compared with AQL-Answers to AQL-Queries that simulate the same analysis. More precisely, a brief comparison of PAndA^2's intra-app taint analysis and the AQL-System, configured with the expert tools FlowDroid and PAndA^2 itself but only using its intra-app permission usage analysis, is evaluated to show that it is highly beneficial to use expert tools. Furthermore, two different apps are analyzed. The so-called `SimpleApp` and `StaticApp`. An extract of the source code of both apps is shown in Listing 5.1 and 5.2. Both apps are identical apart from one simple but effective difference. The tainted variable secret is defined as a local variable (see Line 6 in Listing 5.1) of the `sourceSink()` method of the `SimpleApp`. However, in context of the `StaticApp` it is defined as a static class variable (see Line 2 in Listing 5.2).

The result of PAndA^2's intra-app taint analysis executed for the `SimpleApp` can be seen in Figure 5.1. The result contains two flows from one permission (`READ_PHONE_STATE`) to two other permissions (`WRITE_SMS`, `SEND_SMS`). PAndA^2 always only considers permissions as sources and sinks. To do so, it searches for permission-protected statements which are defined as potential sources for tainted data or sinks that may leak information to the outside.

```
1  public class SimpleAppMainActivity extends Activity {
2      ...
3      private void sourceSink() {
4          // Source
5          TelephonyManager manager = (TelephonyManager)
               getSystemService(Context.TELEPHONY_SERVICE);
6          String secret = manager.getSimSerialNumber();
7
8          // Sink
9          SmsManager.getDefault().sendTextMessage("+49111111111", null,
               secret, null, null);
10     }
11 }
```

Listing 5.1: SimpleApp (Source Code)

```
1  public class StaticAppMainActivity extends Activity {
2      static String secret;
3      ...
4      private void sourceSink() {
5          // Source
6          TelephonyManager manager = (TelephonyManager)
               getSystemService(Context.TELEPHONY_SERVICE);
7          StaticAppMainActivity.secret = manager.getSimSerialNumber();
8
9          // Sink
10         SmsManager.getDefault().sendTextMessage("+49111111111", null,
               StaticAppMainActivity.secret, null, null);
11     }
12 }
```

Listing 5.2: SimpleApp (Source Code)

Analysis Mode: Summary
Detail Level: RES_TO_RES

App's name: simpleapp	
Source(s)	**Paths (to sinks)**
android.permission.READ_PHONE_STATE	android.permission.WRITE_SMS
android.permission.READ_PHONE_STATE	android.permission.SEND_SMS

Figure 5.1: Screenshot of PAndA^2's result (SimpleApp)

We can formulate the following AQL-Query to perform the same analysis:

```
FILTER [
    UNIFY [
        Flows IN App('SimpleApp.apk') ?,
        Permissions IN App('SimpleApp.apk') ?
    ]
]
```

Using an AQL-System with a configuration that contains FlowDroid and PAndA^2 (intra-app permission usage analysis) to respectively respond to Flows-Questions and Permissions-Questions, the answer to this query holds the same two flows with more details. It includes a flow from the `getSimSerialNumber()` to the `sendTextMessage(..)` statement and informs the user which permissions are required by these two statements.

According to this, the two results are equal. Both tools detected the security issue, which is the leakage of the SIM card's serial number. However, when it comes to the `StaticApp` the two results differ. While the AQL-Answer still holds the same two flows, PAndA^2's result does not contain any because PAndA^2 does not consider static variables at all. Thereby PAndA^2 does not report the security issue and misinforms the user by telling him that the app is trustworthy. This shows that the precision of the immature tool PAndA^2 is lower than the precision of the AQL-System, which is a contradiction to our initial assumption. Hence, the answer to RQ1a is: Yes, analysis tools combined through the AQL can be more precise than immature tools.

In summary, this small and simple example shows that immature tools suffer from their limitations. Thus, it definitely is beneficial to combine results computed by expert tools. Note that this comparison would not be possible without the AQL. The AQL allows us to perform the same analysis with two different tools even if the tools set up in the AQL-Systems configuration perform different analyses, that, for example, use different definitions of sources and sinks. In the next section the second part of RQ1 is answered.

5.2 RQ1b: Can analysis tools combined through the AQL be more precise than mature tools?

The goal of RQ1b is to find out if combinations of different expert tools can be more precise than a single mature tool. To answer this question two differently configured AQL-Systems are set up: One uses a combination of different expert tools whereas the other one is only uses one mature tool. Based on the AQL-Answers, which are computed by these two systems for a certain set of apps, different measures that reflect the precision of each system are calculated. The values computed for these measures are compared and interpreted in order to answer RQ1b.

The set of apps used in this part of the evaluation belongs to the Android analysis benchmark DroidBench[1]. It is one of the most commonly used benchmarks for Android app analyses. Almost any paper considering such analyses refers to DroidBench in their evaluation chapter. DroidBench itself comprises different sets of various apps that exploit certain Android features in common and uncommon ways, just like real world apps and especially malicious apps might do it to hide their behavior. Each of these apps is fully functional on an Android device. All apps are shipped as .apk files as well as eclipse projects, that allow everyone to look up the source code of these apps. Comments in the source code describe what feature of Android is exploited by an app and most importantly how many and which issues an analysis should be able to find.

In this thesis four sets of apps from the DroidBench benchmark have been chosen, namely the complete Inter-App and Inter-Component Communication set as well as the complete Lifecycle and Reflection set. On the one hand, each app of any of these sets, except the Inter-App Communication set, stands for one benchmark or test case. On the other hand, the Inter-App Communication set consists of 3 apps, which can be brought together to built two different cases. The `SendSMS` app together with the `Echoer` app represent one case whereas the second case is represented by the `StartActivityForResult1` app and the same `Echoer` app.

In the following, we are using the AQL-System described in the previous chapter to analyze these apps. To do so, two different configurations are used. One configuration sets up the AQL-System to use FlowDroid and IC3. The other one makes the AQL-System use just one tool, namely IccTA. With both setups Inter- and Intra-App flow questions are answerable. From here on we refer to these differently configured AQL-Systems as *FD+IC3-System* and *IccTA-System*.

We expect the results output by both systems to be equal, since IccTA internally uses the exact same tools as the FD+IC3-System, namely FlowDroid and IC3. However, the way multiple apps are analyzed differs. Figure 5.2 depicts, how the FD+IC3-System works. For each app, represented by an .apk file, FlowDroid and IC3 are executed. Then all the results generated by these two tools are combined

[1] `https://blogs.uni-paderborn.de/sse/tools/droidbench/` - 05/12/2017

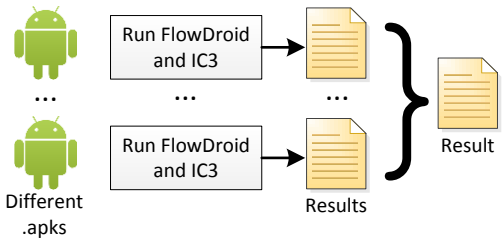

Figure 5.2: Workflow of the FD+IC3-System

by applying the connect operator of the AQL. In contrast, the IccTA-System does not rely on the connect operator. It works as visualized in Figure 5.3. In a

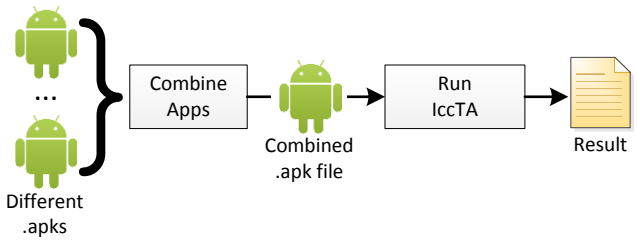

Figure 5.3: Workflow of the IccTA-System

first step, a tool called ApkCombiner [LBB+15b] is used to merge multiple apps represented by .apk files into a single .apk file representing all these apps. This combined .apk file is then used to compute the result by executing IccTA.

Since the Inter-App Communication cases of DroidBench consider at most two apps, we can formulate the queries used to run these benchmark cases as follows.

- FD+IC3-System:

$$\text{Flows } \texttt{FROM App('}x\texttt{') TO App('}y\texttt{') ?}$$

- IccTA-System:

$$\text{Flows } \texttt{IN App('}x\ y\texttt{' | 'COMBINE') ?}$$

x and y symbolize the .apk files of the associated apps. In context of the FD+IC3-System we simply ask for inter-app flows and let the system handle everything else. As shown in the running example (cf. Section 3.3.3), the system transforms such a query, tries to answer its individual parts and then applies the connect operator to combine the answers. Thereby FlowDroid is used to determine intra-app flows and IC3 to detect intent-sinks of x and intent-sources of y. Contrariwise, in context of the IccTA-System, we ask for intra-app flows of a combined app. The keyword "COMBINE" refers to the ApkCombiner set up as a preprocessor. It takes all

.apk files separated by "␣" and puts them together into a single .apk file, which is then used for the analysis.

Similarly, the queries for the Inter-Component Communication set differ as well:

- FD+IC3-System:

```
FILTER [
    CONNECT [
        Flows IN App('x') ?,
        IntentSinks IN App('x') ?,
        IntentSources IN App('x') ?
    ]
]
```

- IccTA-System:

```
Flows IN App('x') ?
```

While the IccTA-System's query is self-explanatory, the FD+IC3-System's query is not. We have to ask for intent-sinks and -sources, because FlowDroid, used to answer the intra-app question part, does not compute flows between components. Thus these flows are computed by connecting intent-sinks and -sources.

For any other set of DroidBench the queries are equal:

```
Flows IN App('x') ?
```

Formulating all DroidBench cases as AQL-Queries allows us to get one AQL-Answer for each case. This in turn enables us to automatically compare analysis results of different systems and tools to each other and to expected results. To do so, we first formulate the expected results as AQL-Answers. Then, a tiny tool, called AQL-Comparator, which has been developed for this part of the evaluation, can be used to perform a fully automated comparison of these expected results and the actual ones. The AQL-Comparator's workflow is depicted in Figure 5.4. As

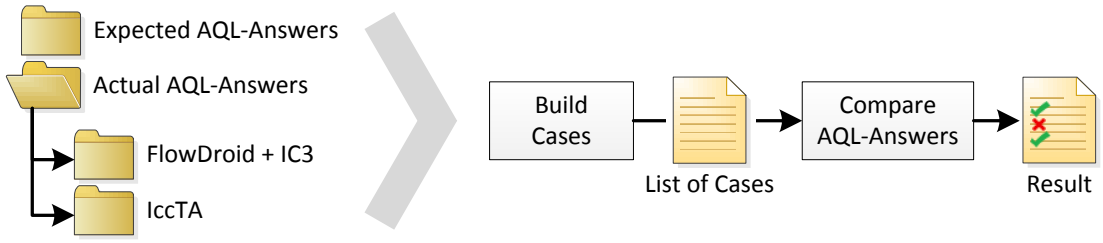

Figure 5.4: Workflow of AQL-Comparer

input it takes multiple directories that contain expected AQL-Answers on the one hand and actual AQL-Answers on the other hand. At first, the AQL-Comparator

maps actual answers to expected ones by their filenames and thereby generates a list of cases. This list can contain two types of cases: *Positive cases* if the expected answer ($E \in \mathcal{A}$) should be a subset of the actual answer ($A \in \mathcal{A}$) or *negative cases* if the actual answer should not contain any element of the expected answer. Then the AQL-Comparator computes a single boolean value (r) for each case which tells us whether the actual answer *matches* the expected one. The formula in order to calculate r can be described as follows:

$$r = \begin{cases} \texttt{true} & \text{if a positive case is considered and} \\ & E \cap A_{\text{Flows}} \subseteq A \text{ holds.} \\ \texttt{true} & \text{if a negative case is considered and} \\ & \text{for all } f \in E \cap A_{\text{Flows}} \text{ there does not exist an } f' \in A \text{ with } f = f' \\ \texttt{false} & \text{else} \end{cases}$$

The results computed by the AQL-Comparator are interpreted and listed in Table 5.1. It presents the considered case in the first column. In the second and third column the individual results for the FD+IC3-System and the IccTA-System are shown. On the one hand, the symbol ⊛ tells us that the output considering a positive case was `true` whereas the ⋆-symbol tell us that the output of a negative case was `true`. On the other hand, the symbols ⋆ and ◯ provide the information that the AQL-Comparator has output `false` considering a negative and a positive case respectively. If two of these symbols appear in the same cell of the table, the associated case consists of a positive and a negative part. In other words, in these cases one flow, representing a security leak, should be found and another one should not be found, since, for example, the program location representing the source cannot be reached. Note that in the following the symbols $|⊛|$, $|⋆|$, $|⋆|$, $|◯|$ and sums of these (e.g. $|⊛ + ⋆|$) refer to the associated number of occurrences. To interpret these numbers, one could say the higher $|⊛|$ and $|⋆|$ are the better. Accordingly, the lower $|⋆|$ and $|◯|$ are the better.

Table 5.1 shows that contrary to our expectations the results for the FD+IC3- and the IccTA-System differ, but only when it comes to the inter-component cases. This is related to the different approaches the two systems follow (cf. Figure 5.2 and 5.3). More precisely, the connect operator of the AQL constructs inter-component flows as described in the conceptual design chapter (see Section 3.1.4), while IccTA internally seems to do even more.

The different cases show, that both approaches have their advantages and disadvantages. For example, the security issue in the case referenced by "ActivityCommunication2" cannot be detected with the FD+IC3-System but with the IccTA-System. However, the IccTA-System additionally outputs a false-positive. Thus it reports more security issues than there actually are. The case listed thereafter ("ActivityCommunication3") shows that the FD+IC3-System is able to find issues, which remain undetected by the IccTA-System.

⊛: True-Positive, ⋆: True-Negative, ⋆: False-Positive, ◯: False-Negative
†: Empty AQL-Answer, ×: Manually adapted

Case	FD+IC3	IccTA
Inter-App Communication		
SendSMS + Echoer	⊛×	⊛×
StartActivityForResult1 + Echoer	⊛×	⊛×
Inter-Component Communication		
ActivityCommunication1	⊛	⊛
ActivityCommunication2	◯ ⋆	⊛ ⋆
ActivityCommunication3	⊛ ⋆	◯ ⋆
ActivityCommunication4	⊛ ⋆	⊛ ⋆
ActivityCommunication5	⊛ ⋆	⊛ ⋆
ActivityCommunication6	◯ ⋆	◯ ⋆
ActivityCommunication7	◯ ⋆	◯ ⋆
ActivityCommunication8	◯ ⋆	⊛ ⋆
BroadcastTaintAndLeak1	◯	⊛
ComponentNotInManifest1	⋆	⋆
EventOrdering1	◯	⊛
IntentSink1	◯	◯†
IntentSink2	◯	⊛
IntentSource1	⊛×	⊛×
ServiceCommunication1	Broken Manifest ⇒ Omitted	
SharedPreferences1	◯	⊛
Singletons1	◯†	◯†
UnresolvableIntent1	◯	◯
Lifecycle		
ActivityLifecycle1	⊛	⊛
ActivityLifecycle2	⊛	⊛
ActivityLifecycle3	⊛	⊛
ActivityLifecycle4	◯	◯
ActivitySavedState1	⊛	⊛
ApplicationLifecycle1	⊛	⊛
ApplicationLifecycle2	⊛	⊛
ApplicationLifecycle3	⊛	⊛
AsynchronousEventOrdering1	⊛	⊛
BroadcastReceiverLifecycle1	⊛	⊛
BroadcastReceiverLifecycle2	◯†	◯†
EventOrdering1	⊛	⊛
FragmentLifecycle1	◯†	◯†
FragmentLifecycle2	◯†	◯†
ServiceLifecycle1	⊛	⊛
ServiceLifecycle2	⊛	⊛
SharedPreferenceChanged1	◯	◯
Reflection		
Reflection1	⊛	⊛
Reflection2	◯†	◯†
Reflection3	◯†	◯†
Reflection4	◯†	◯†

Table 5.1: DroidBench Evaluation Results

Firgure 5.5 summarizes the total number of successful ($|⊛ + \star|$) and failed ($|\star + ◯|$) cases. Of a total of 47 cases the IccTA-System was able to successfully detect 1 more issue. This number is raised to 4, if we ignore the 3 false-positive results (see $\star$ in Table 5.1) by not considering the 8 negative cases. Based on these values we can compute the successrate, which represents the ratio of successful cases to all cases:

$$\text{Successrate: } \frac{|⊛ + \star|}{|⊛ + \star + ◯ + \star|}$$

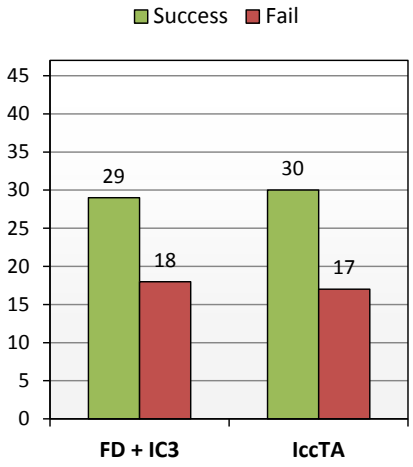

Figure 5.5: Bar-chart (Absolute successful and failed cases)

Additionally, the common measures (Precision, Recall, F-Measure) to express the outcome of detection-related experiments, such as the execution of different analyses, are defined as follows [Faw06]:

$$\text{Precision: } p = \frac{|⊛|}{|⊛ + \star|}$$

$$\text{Recall: } r = \frac{|⊛|}{|⊛ + ◯|}$$

$$\text{F-Measure: } 2 * \frac{p * r}{p + r}$$

These measures can be found in several papers and can be used to compare evaluation results of different papers [ARF+14, LBB+15a, RAMB16]. The measures precision and recall illustrate how trustworthy positive and negative results are. For example, a precision of 100% tells us that all issues that have been found are correctly identified as issues. In contrast, a recall of 100% tells us that all issues that could be found have been found. The F-Measure describes the harmonic mean of precision and recall. It can be interpreted as a rating for analyses. The exact values of all four measures are presented in Figure 5.6. Since the FD+IC3-System did not output any false-positive its precision is at its best (100%). The IccTA-System only has a precision of almost 90%, but in favor of recall it is better than the FD+IC3-System (83% to 72%). F-Measure and successrate of both systems are almost equal: They only differ by 2%. This shows us, that both systems are similarly powerful. Considering RQ1b it can be concluded that the answer should be: No, analysis tools combined through the AQL are not always more precise than mature tools, but they are almost as precise.

In summary, it is safe to say that the competitive power of both systems is almost equal. Additionally, we demonstrated that the AQL allows us to evaluate experiments considering a benchmark such as DroidBench automatically.

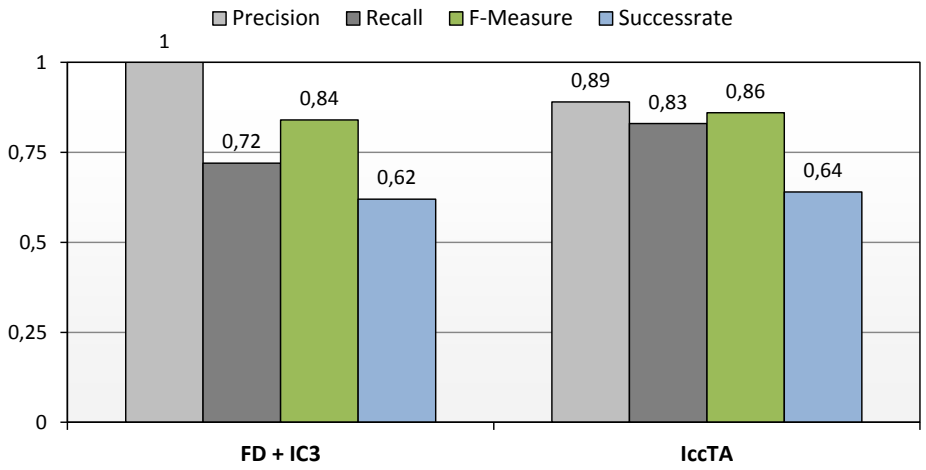

Figure 5.6: Bar-chart (Precision, Recall, F-Measure, Successrate)

5.2.1 Annotations

In three cases both systems required a little help to deliver a successful result. These cases are marked with a $^{\times}$-symbol. Both systems successfully identified the tainted source, the sink as well as intent-sink and -source. However, no system was able to compute the complete tainted flow from source to sink without slightly adapting the result. In all three cases the same problem occurred. A `startActivityForResult(..)` intent-sink and an intent-source with its origin in method `onActivityResult(..)` were detected in the same component. The connection between these two is implicitly always available because the result, computed by whatever component is targeted by the `startActivityForResult(..)` statement, is returned to the `onActivityResult(..)` method. In all three cases both systems also detected the tainted flow part inside the targeted component. Nevertheless, none of the systems reported any flow between the intent-sink (`startActivityForResult(..)`) and the intent-source inside the `onActivityResult(..)` method. This flow has been added manually by combining the given answers with manually created ones that hold the missing flow part. To do so, the connect operator of the AQL was used. This manually performed adaption could easily be automated with help of the AQL. One could develop a tool that connects all `startActivityForResult(..)` statements with all intent-sources in the `onActivityResult(..)` method of the same component, if and only if a taint flow was found in the targeted component.

One case ("ServiceCommunication1"[2]) could not be analyzed by any system, because the manifest file of that app could not be parsed since it contains a syntactical error in its XML code.

[2] `https://github.com/secure-software-engineering/DroidBench/blob/master/eclipse-project/InterComponentCommunication/ServiceCommunication1/AndroidManifest.xml` (Line 26) - 04/26/2017

Some of the AQL-Answers computed by both systems were empty. The associated cases are marked with a †-symbol in Table 5.1. This tells us, that the issue has not been detected and additionally it states that the system was unable to even find parts of the flow that represents the security issue. The majority of answers considering the Reflection set of DroidBench was empty, since the reflection could not be resolved statically. The only reflective call that could be resolved belongs to the app and case referenced by "Reflection1". In the following section it is evaluated how to tackle the other unresolved cases.

Lastly, there is one technical annotation: The same converter can be used to convert results output by FlowDroid or IccTA, because IccTA internally uses FlowDroid and thereby produces results of the same format. Thus, no additional converter is needed to use the IccTA-System.

5.3 RQ2: Can the power of a single analysis tool be increased by means of the AQL?

To answer RQ2, which asks for options to increase the power of an analysis, we use a preprocessor that should allow us to enhance an analysis.

In the previous section among others the Reflection set of DroidBench is evaluated. Most of the cases could not be analyzed successfully since it can be hard and, in some cases, impossible to resolve the reflection due to the limited possibilities of static analyses. This is one reason why the earlier introduced tool Harvester combines static and dynamic analyses. In context of the AQL-System, Harvester can be set up as a preprocessor, that is able to deobfuscate an app. More precisely, to resolve the reflection calls and automatically build a so-called enriched app without reflection.

Correspondingly, the queries to evaluate the three DroidBench apps "Reflection2", "Reflection3" and "Reflection4" can be changed as follows:

Flows `IN App('`x`') ?`
↓ *changed to* ↓
Flows `IN App('`x`' | 'DEOBFUSCATE') ?`

Now, Harvester is executed in order to produce a preprocessed enriched version of the three apps represented by x in the queries. Although Harvester successfully detects and resolves all reflective classes and method calls inside all three apps, it does not replace all occurrences in the preprocessed app versions. After having consulted the developers of Harvester, it is safe to say that a successful analysis of these three apps is currently not possible but will be in future. Harvester is a state-of-the-art tool, which is still in development and in its current state it is able to resolve reflection but it only replaces reflective method calls with non-reflective ones in the enriched apps it outputs. Reflective classes, for example, are not replaced, yet.

An extract of an app, namely `ReflectiveApp`, can be found in Listing 5.3.

```
1  public class ReflectionAppMainActivity extends Activity {
2      ...
3      private void sourceSink() {
4          // Source
5          TelephonyManager manager = (TelephonyManager)
               getSystemService(Context.TELEPHONY_SERVICE);
6          String secret = "";
7          try {
8              Method method = manager.getClass().getMethod("
                   getSimSerialNumber");
9              secret = (String) method.invoke(manager);
10         } catch ... { ... }
11
12         // Sink
13         SmsManager.getDefault().sendTextMessage("+49111111111", null,
               secret, null, null);
14     }
15 }
```

Listing 5.3: SimpleApp (Source Code)

This app only contains one reflective method call. While the query

Flows `IN App('`ReflectiveApp.apk`') ?`

leads to an empty answer, the answer to the following query

Flows `IN App('`ReflectiveApp.apk`' | '`DEOBFUSCATE`') ?`

reveals the security issue. A flow from the `getSimSerialNumber()` statement (Lines 8 and 9 in Listing 5.3) to the `sendTextMessage(..)` statement (Line 13 in Listing 5.3) is found. Thus, by using Harvester as a preprocessor we enhanced the taint analysis performed by the AQL-System and thereby increased the power of FlowDroid. Considering RQ2 this shows us that the AQL can be used to increase the power of an analysis tool.

On the one hand, we can conclude that the accuracy of an AQL-System depends on the tools and preprocessors configured in its configuration. On the other hand, tools and preprocessors are not influenced by the AQL-System. Furthermore, we have seen that it is possible to combine static and dynamic analyses through the AQL. To this effect the AQL is future-proof, since any kind of analysis (static or dynamic) can be used along with an AQL-System and since the accuracy of an AQL-System will always grow with the underlying analyses.

5.4 RQ3: Is the AQL-System capable of analyzing sets of one or more real world apps efficiently?

If the AQL-System is able to analyze sets of real world apps, is asked by RQ3. In order to provide an answer, a set of real world apps has to be chosen. Then an attempt can be made to analyze this set and, depending on the outcome, RQ3 can be answered.

As real world apps, we denote all apps that can be downloaded publicly from markets such as Google's PlayStore[3]. The evaluation considering a subset of these apps deals with two different objectives. On the one hand, we want to find out how many intra- and inter-app taint flows can be found. More precisely, how many permission-protected sources are connected to permission-protected sinks in the same or another app. On the other hand, we want to evaluate the performance of the AQL-System. Therefor, we measure the execution time and show how to reduce it by means of the AQL.

We choose a subset of 34 real world apps downloaded from Google's PlayStore. One of them is the most downloaded and top-rated app WhatsApp. Many more apps that are considered during this evaluation are part of the 100 most downloaded apps (in Germany), too. A full list of all evaluated apps along with the associated results can be found in Table 5.2.

To evaluate these apps we run an AQL-System configured to use FlowDroid, IC3 and PAndA[2] on a Debian[4] virtual machine with 32 gigabytes of available memory. We ask for intra-app taint flows, intents, intent-filters, intent-sinks and -sources as well as for permissions. The inter-app taint flows are computed based on the answers to these questions by combining them.

The previously mentioned Table 5.2 shows the number of findings for each subject of interest as well as the time spent to compute these. The columns are labeled with the subject of interest or "Time" in the first row. "Time" refers to the execution time measured in seconds that is required in order to answer the query considering the associated subject of interest and the app mentioned in the first column of the table. In case of the column labeled with "Permissions" the first row shows two annotations: The column annotated with "occurrences" reflects the number of statements that require a permission in order to be executed. In the "individual" column it is summarized how many different permissions are required by these statements.

Some cells of the table are empty, because the associated query could not be answered in less than 15 minutes or the tool responsible for the analysis ran out of memory. By granting more time and by increasing the maximum of available memory it should be possible to fill these empty cells.

The last 4 rows in the table show calculated values that summarize the findings. In particular, the last row demonstrates the sum of all findings and approximates

[3] `https://play.google.com/store`
[4] `https://www.debian.org/` - 05/22/2017

†: Exception, ×: Abort (> 900s)

#	App	Permissions		Time	Flows	Time	Intent-				Time	Flows	Time
		occurrences	*individual*	*(s)*		*(s)*	*s*	*filters*	*sinks*	*sources*	*(s)*	*adapted*	*(s)*
1	com.amazon.mShop.android.shopping	3179	26	206	3746	150	–	–	–	–	–×	2	105
2	com.droid27.transparentclockweather	1864	14	238	86	60	–	–	–	–	–×	0	47
3	com.espn.fantasy.lm.football	2615	20	417	–	–×	–	–	–	–	–×	–	–
4	com.estrongs.android.pop	7020	31	857	–	–×	–	–	–	–	–×	0	68
5	com.google.android.apps.photos	5137	32	508	3	31	–	–	–	–	–×	0	26
6	com.google.android.calculator	57	4	21	115	19	3	0	2	4	23	0	6
7	com.google.android.contacts	3687	26	135	210	20	–	–	–	–	–×	0	17
8	com.google.android.deskclock	842	14	62	–	–×	–	–	–	–	–×	0	29
9	com.google.android.dialer	5614	41	205	–	–×	–	–	–	–	–×	0	49
10	com.google.android.gm	8954	26	883	–	–×	–	–	–	–	–×	0	174
11	com.google.android.GoogleCamera	691	21	159	0	34	6	0	3	4	72	0	29
12	com.google.android.keep	1713	18	202	–	–×	–	–	–	–	–×	0	44
13	com.google.android.videos	1647	16	271	–	–×	–	–	–	–	–×	0	62
14	com.google.android.youtube	3596	28	625	–	–×	–	–	–	–	–×	0	172
15	com.google.zxing.client.android	420	12	29	0	8	24	117	7	15	25	0	9
16	com.mxtech.videoplayer.ad	2309	21	420	557	36	47	36	5	14	859	0	30
17	com.overlook.android.fing	627	12	78	93	20	–	–	–	–	–×	0	19
18	com.reddit.frontpage	2444	19	348	–	–×	–	–	–	–	–×	0	40
19	com.tellm.android.app	921	13	155	–	–×	–	–	–	–	–×	0	33
20	com.tendadigital.chwaziApp	5	4	5	0	3	2	0	1	0	7	0	3
21	com.thescore.esports	1527	20	346	92	24	1	3	1	7	77	0	23
22	com.valvesoftware.android.steam.community	772	11	163	4	17	412	134	9	28	462	0	14
23	com.whatsapp	7929	34	474	–†	6541	–	–	–	–	–×	0	86
24	com.yamaha.npcontroller	147	9	23	3	6	29	0	3	3	72	0	5
25	com.zattoo.player	1693	16	495	109	50	–	–	–	–	–×	0	41
26	com-asus-flashlight	322	9	40	21	20	65	13	12	11	165	0	13
27	com-fileexplorer-app	707	16	85	13	13	89	10	11	6	38	0	11
28	de.burgerking.kingfinder	599	9	126	88	24	318	224	8	26	244	0	25
29	de.gmx.mobile.android.mail	4470	25	350	43	35	–	–	–	–	–×	0	24
30	de.hafas.android.db	1834	17	327	–	–×	–	–	–	–	–×	–†	6762
31	de.mcdonalds.mcdonaldsinfoapp	934	15	276	10	37	378	218	11	25	878	0	29
32	de.radio.android	1436	19	225	14	24	–	–	–	–	–×	0	23
33	de.upb.unipin	480	15	17	9	10	29	0	14	3	26	0	7
34	tv.twitch.android.app	824	11	247	88	54	–	–	–	–	–×	0	36
	Arithmetic mean	**2265**	**18**	**685**	**241**	**315**	**108**	**58**	**7**	**11**	**227**	**2**	**8061**
	Median	**1587**	**17**	**216**	**32**	**24**	**29**	**10**	**7**	**7**	**72**	**2**	**8090**
	Σ	**77016**	**624**	**23293**	**5304**	**695**	**1403**	**755**	**87**	**146**	**2948**	**0**	**244**
	Σ (approximated with median)	**77016**	**624**	**23293**	**5688**	**7500**	**2012**	**965**	**234**	**293**	**4460**	**0**	**29**

Table 5.2: Evaluation Results for the set of Real World Apps

the cases where no result could be computed with the median value. Considering absolute numbers, more than 5000 flows, about 1400 intents, 750 intent-filters, 100 intent-sinks and 150 intent-sources as well as more than 77,000 permission uses are analyzed.

Since all these findings are stored as AQL-Answers these can be used to gain more information about the whole set of real world apps. First of all, 657 inter-app flows, that represent possible flows between two different apps of the considered set, are found by combining all intent-sinks and -sources through the connect operator of the AQL. 54 ends of these flows are protected by permissions. Connecting these flows with the intra-app taint flows results in overall 875 inter-app flows. However, none of these flows in turn connects a permission-protected source with a permission-protected sink. To this end, we can say, that this set of real world apps seems to be trustworthy since we could not discover a malicious cooperation. Without the AQL neither this nor any other conclusion considering inter-app analyses could be made based on the information gathered in Table 5.2.

Since AQL-Answers considering single and multiple real world apps could be computed, we can give a positive response to RQ3: Yes, the AQL-System can analyze single real world apps as efficiently as the underlying tools. However, the AQL also enables us to analyze sets of apps by combining the answers computed per app. The same set of real world apps considered above is used in order to answer RQ4 in the following.

5.5 RQ4: Can the AQL improve the performance of an analysis?

The query used to find intra-app flows cannot be answered in all cases (cf. Table 5.2). For example, in case of WhatsApp the analysis was aborted after more than 1.5 hours, because it ran out of memory. Other analyses were aborted after about 15 minutes due to time constraints. If the AQL could be used to somehow compute results for these cases, a positive feedback considering RQ4 could be provided. Hence, it is explained in the following how results for these cases may be computed.

Let us assume that we are only interested in flows between sources and sinks that are protected by permissions. However, the 5304 intra-app taint flows that are found connect all sorts of sources and sinks.

To remove the unwanted flows that connect sources and sinks which are not protected by permissions, we may filter the results according to the permissions found in these apps. Another option is to only compute flows between permission-protected sources and sinks in the first place. This may also allow us to actually compute results for the 12 cases we could not yet compute results for.

To do so, we can ask for permissions in an app through the AQL. Such questions can be answered in all considered cases (cf. Table 5.2). Then the permissions found can be used to filter the set of sources and sinks which are considered once we ask for flows. Since the AQL-System uses FlowDroid in order to answer flow questions this can be done by adapting the "SourceAndSink.txt" text file. It includes one line per source and sink.

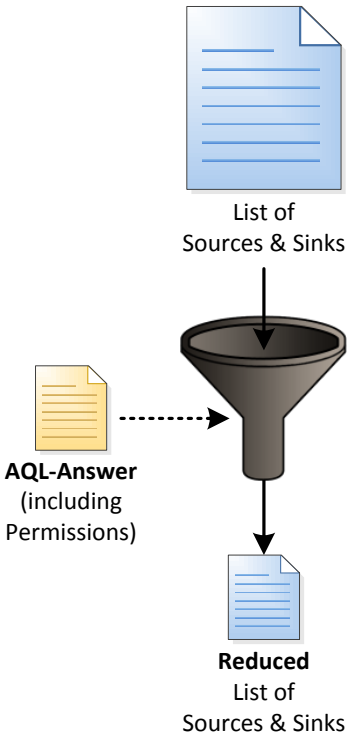

Figure 5.7: Workflow sketch of the AQL-Minimizer

For this purpose, a tiny tool, namely *AQL-Minimizer* has been developed. It could also be used as a preprocessor. Its idea is depicted in Figure 5.7. It takes the original "SourceAndSink.txt" file and an AQL-Answer (A) as input and creates a new file, representing the reduced set of sources and sinks, as output. Let the set X be the set of all original sources and sinks:

$$X = \{(r, b) \mid r \in R \wedge b \in \{\texttt{true}, \texttt{false}\}\}$$

The boolean value b describes if a reference r refers to a source (`true`) or a sink (`false`). Then X^*, the set of all reduced sources and sinks, can be defined as follows:

$$X^* = \{(r, b) \mid (r, b) \in X \wedge \exists (r, p) \in A \cap A_{\text{Permissions}}\}$$

This set is computed by the AQL-Minimizer. On average these sets of reduced sources and sinks are 98.4% smaller than the original set. Instead of 232 sources and sinks, only up to 10 are included in these sets.

Using the sets of reduced sources and sinks gives us the results listed in the last two columns of Table 5.2. The "adapted" flows column shows how many flows are found that connect permission-protected sources with permission-protected sinks. Only in case of Amazon's shopping app such flows can be found. Still, we cannot answer the query considering intra-app taint flows in the "DB Navigator" app (see case: "de.hafas.android.db"). However, the execution time required to answer the queries considering all other cases that could be answered before has been reduced by an average of about 22%. In addition, 11 more cases could be finished in less than 15 minutes, which was not possible before. Note that this is only appropriate, because we used the assumption that every source and sink should be protected by a permission. Without this assumption we would lose a lot of precision by this approach. This shows us, that we can use the AQL to speed up and thereby improve the performance of analyses. Having RQ4 in mind, this

tells us that the AQL can improve the performance of an analysis.

In summary, using the AQL in the context of real world apps provides multiple advantages: It allows us to combine precomputed results. Thereby, for instance, the 875 inter-app flows can be computed as explained in Section 5.4. Furthermore, in certain scenarios it can improve the performance of an analysis by greatly reducing its memory consumption and decreasing its execution time, as shown in this section.

5.6 Summary

The analyses of two custom apps, which have been developed for this evaluation, and 42 apps, which belong to the benchmark DroidBench, have been described in the first two sections of this chapter. Thereby, it has been concluded that combinations of different analysis tools are more precise than immature tools and almost as precise as mature tools. Consequently, both parts of RQ1 have been answered.

During the attempt to answer RQ2 it has been shown that the AQL can be used to increase the power of an analysis tool. In this case FlowDroid was partially enabled to handle apps that contain reflection by using Harvester as a preprocessor.

The successful and efficient analyses of 34 real world apps has shown that the AQL-System can be used in realistic scenarios. Furthermore, the AQL-Answers computed for all 34 apps could be combined to gain even more knowledge about these apps: For example, 875 inter-app flows have been found. Thereby and by minimizing sources and sinks in certain scenarios, a positive response to RQ3 and RQ4 could be given.

Overall, all research questions could be answered and along with that many advantages of the AQL could be highlighted.

6 Conclusion

On the one hand, the content as well as the reached goals and other achievements of this thesis are summarized in this chapter. On the other hand, a greater, yet unreached, goal is described that makes mature analyses in the context of Android available to everyone, especially non-expert users. With this greater goal in mind, it is presented how to continue the project started with this thesis.

6.1 Summary

Developing a language that allows the general formulation of analysis questions and tasks as well as solutions and answers, represents the main goal of this thesis. By developing the Analysis Query Language (AQL) this goal is achieved. The AQL is formally as well as syntactically defined. For the syntactic definition a grammar was introduced that precisely defines which queries can be formulated and which analysis questions can be asked. Equally, an XSD was presented that defines the structure and content of AQL-Answers. Together with the operators, described along with the AQL, these answers can be combined. Thereby, different analysis tools are indirectly brought together in order to create a cooperative analysis.

The benefits of such a cooperative analysis and the advantages of the AQL in general are presented and evaluated. To do so, a tool implementation based on the described concept of an AQL-System is used. We show that such an AQL-System can be more precise than immature tools and with the correct configuration it can at least be as precise as mature tools. Plus, in some cases it can be even more precise. However, the AQL cannot only be used to optimize precision. In certain scenarios it can even lead to performance improvements. In particular, intra-app analyses of real world apps can gain a speed up of more than 22%. Additionally, inter-app analyses, that are tough or impossible to execute without the AQL, can be performed in seconds if each involved app has been analyzed before.

Furthermore, the AQL and systems using it are not restricted to be used in a certain scenario but the contrary is the case: They can be involved in different and varying situations. On the one hand, it can be used, for example, to execute or combine analyses without expert knowledge. On the other hand, experts can build their own analyses on top of AQL-Answers or use the AQL to issue on-demand analyses during their own analyses.

In summary, the approach explained, implemented and evaluated in the context of this thesis can be seen as a powerful solution to improve the security of Android

devices and to protect the user against privacy attacks. More precisely, as a powerful instrument to extract the best of one or more software analyses. This helps to satisfactorily show the trustworthiness of a certain set of apps. Considering the plethora of existing apps and devices and the consequently following flaws, the end-user absolutely requires this help. Above all, the raising number of cyber attacks shows that such processes of verifying security are more important than ever before.

6.2 Future Work

Overall, the main goal of any program analysis is to detect real and existing issues such as security flaws. The one who should profit most from such analyses is the non-expert end-user. However, most analyses and their results can only be executed and interpreted by field experts.

The AQL describes a way to execute analyses much easier by formulating an AQL-Query. Let us assume, that we will get access to different services of a cloud network in the future, in which each service represents one differently configured AQL-System. Then the AQL provides different benefits:

- Results computed by any of these systems can be reused in order to quickly respond to a query.
- To answer more complex queries it might be possible to combine existing answers computed by different systems.
- We can automatically evaluate the accuracy of a system by using a benchmark such as DroidBench (cf. Section 5.2).

$\Rightarrow$ Parts of a query can be answered by the most precise systems in the network.

All this allows us to initialize an analysis and receive results as fast as possible without losing precision or requiring a lot of local computational power.

For instance, imagine there exists an app that performs an inter-app taint analysis, that considers all other apps installed on the device, once we click a button. As a consequence the different expert systems in the cloud are asked for intra-app taint flows, intent-sources and intent-sinks. These partial results might have already been computed per app and can be replied immediately. If not the expert systems in the cloud compute them. By means of the AQL the answers of these systems can be combined in order to perform the inter-app taint analysis. Thereby the end-user can find security issues very easily by pressing one button.

Google permanently performs static and dynamic analyses to improve the trustworthiness of apps in the PlayStore [Goo17a, Goo17b]. Similar to the scenario described in this section, these analyses are performed by different services accessible through a cloud. However, as far as known, Google only performs analyses on single apps. Thus, if they would store their results as AQL-Answers, it might

be possible to combine these in order to perform inter-app analyses and thereby identify sets of apps that cooperate on malicious purpose.

Furthermore, the AQL itself could be improved in future. For example, the matching algorithm of the connect operator (cf. function connect in Section 3.1.4) may be improved in order to become more precise than other mature tools (cf. Section 5.2). Considering the filter operator, the possibilities to filter AQL-Answers may be extended. Even more sophisticated, the whole AQL could be adapted in order to formulate queries that can be used to analyze apps developed for other operating systems than Android.

Appendix A

XML Schema Definitions (XSDs)

A.1 AQL-Answer XSD

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="
    qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="hash">
        <xs:complexType>
            <xs:simpleContent>
                <xs:extension base="xs:string">
                    <xs:attribute type="xs:string" name="type" use="
                        optional"/>
                </xs:extension>
            </xs:simpleContent>
        </xs:complexType>
    </xs:element>
    <xs:element name="file" type="xs:string"/>
    <xs:element name="hashes">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="hash" maxOccurs="unbounded"
                    minOccurs="0"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="method" type="xs:string"/>
    <xs:element name="classname" type="xs:string"/>
    <xs:element name="app">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="file"/>
                <xs:element ref="hashes"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="value" type="xs:string"/>
    <xs:element name="attribute">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="name"/>
```

```
35                  <xs:element ref="value"/>
36              </xs:sequence>
37          </xs:complexType>
38      </xs:element>
39      <xs:element name="reference">
40          <xs:complexType>
41              <xs:sequence>
42                  <xs:element ref="statement" minOccurs="0"/>
43                  <xs:element ref="method" minOccurs="0"/>
44                  <xs:element ref="classname" minOccurs="0"/>
45                  <xs:element ref="app"/>
46              </xs:sequence>
47              <xs:attribute type="xs:string" name="type" use="optional"
                    />
48          </xs:complexType>
49      </xs:element>
50      <xs:element name="attributes">
51          <xs:complexType>
52              <xs:sequence>
53                  <xs:element ref="attribute" maxOccurs="unbounded"
                        minOccurs="0"/>
54              </xs:sequence>
55          </xs:complexType>
56      </xs:element>
57      <xs:element name="permission">
58          <xs:complexType>
59              <xs:sequence>
60                  <xs:element ref="name"/>
61                  <xs:element ref="reference"/>
62                  <xs:element ref="attributes" minOccurs="0"/>
63              </xs:sequence>
64          </xs:complexType>
65      </xs:element>
66      <xs:element name="type" type="xs:string"/>
67      <xs:element name="scheme" type="xs:string"/>
68      <xs:element name="ssp" type="xs:string"/>
69      <xs:element name="host" type="xs:string"/>
70      <xs:element name="port" type="xs:string"/>
71      <xs:element name="path" type="xs:string"/>
72      <xs:element name="action" type="xs:string"/>
73      <xs:element name="category" type="xs:string"/>
74      <xs:element name="data">
75          <xs:complexType>
76              <xs:sequence>
77                  <xs:element ref="type" minOccurs="0"/>
78                  <xs:element ref="scheme" minOccurs="0"/>
79                  <xs:element ref="ssp" minOccurs="0"/>
80                  <xs:element ref="host" minOccurs="0"/>
81                  <xs:element ref="port" minOccurs="0"/>
82                  <xs:element ref="path" minOccurs="0"/>
83              </xs:sequence>
84          </xs:complexType>
```

```
85      </xs:element>
86      <xs:element name="parameter">
87          <xs:complexType>
88              <xs:sequence>
89                  <xs:element ref="type"/>
90                  <xs:element ref="value"/>
91              </xs:sequence>
92          </xs:complexType>
93      </xs:element>
94      <xs:element name="statementfull" type="xs:string"/>
95      <xs:element name="statementgeneric" type="xs:string"/>
96      <xs:element name="parameters">
97          <xs:complexType>
98              <xs:sequence>
99                  <xs:element ref="parameter" maxOccurs="unbounded"
                        minOccurs="0"/>
100             </xs:sequence>
101         </xs:complexType>
102     </xs:element>
103     <xs:element name="statement">
104         <xs:complexType>
105             <xs:sequence>
106                 <xs:element ref="statementfull"/>
107                 <xs:element ref="statementgeneric"/>
108                 <xs:element ref="parameters" minOccurs="0"/>
109             </xs:sequence>
110         </xs:complexType>
111     </xs:element>
112     <xs:element name="target">
113         <xs:complexType>
114             <xs:sequence>
115                 <xs:element ref="action" minOccurs="0"/>
116                 <xs:element ref="category" minOccurs="0"/>
117                 <xs:element ref="data" minOccurs="0"/>
118                 <xs:element ref="reference" minOccurs="0"/>
119             </xs:sequence>
120         </xs:complexType>
121     </xs:element>
122     <xs:element name="intentsource">
123         <xs:complexType>
124             <xs:sequence>
125                 <xs:element ref="target"/>
126                 <xs:element ref="reference"/>
127                 <xs:element ref="attributes" minOccurs="0"/>
128             </xs:sequence>
129         </xs:complexType>
130     </xs:element>
131     <xs:element name="intentsink">
132         <xs:complexType>
133             <xs:sequence>
134                 <xs:element ref="target"/>
135                 <xs:element ref="reference"/>
```

```
136                 <xs:element ref="attributes" minOccurs="0"/>
137             </xs:sequence>
138         </xs:complexType>
139     </xs:element>
140     <xs:element name="intent">
141         <xs:complexType>
142             <xs:sequence>
143                 <xs:element ref="reference"/>
144                 <xs:element ref="target"/>
145                 <xs:element ref="attributes" minOccurs="0"/>
146             </xs:sequence>
147         </xs:complexType>
148     </xs:element>
149     <xs:element name="intentfilter">
150         <xs:complexType>
151             <xs:sequence>
152                 <xs:element ref="reference"/>
153                 <xs:element ref="action"/>
154                 <xs:element ref="category" minOccurs="0"/>
155                 <xs:element ref="data" minOccurs="0"/>
156                 <xs:element ref="attributes" minOccurs="0"/>
157             </xs:sequence>
158         </xs:complexType>
159     </xs:element>
160     <xs:element name="flow">
161         <xs:complexType>
162             <xs:sequence>
163                 <xs:element ref="reference" maxOccurs="unbounded"
                        minOccurs="0"/>
164                 <xs:element ref="attributes" minOccurs="0"/>
165             </xs:sequence>
166         </xs:complexType>
167     </xs:element>
168     <xs:element name="permissions">
169         <xs:complexType>
170             <xs:sequence>
171                 <xs:element ref="permission" maxOccurs="unbounded"
                        minOccurs="0"/>
172             </xs:sequence>
173         </xs:complexType>
174     </xs:element>
175     <xs:element name="intentsources">
176         <xs:complexType>
177             <xs:sequence>
178                 <xs:element ref="intentsource" maxOccurs="unbounded"
                        minOccurs="0"/>
179             </xs:sequence>
180         </xs:complexType>
181     </xs:element>
182     <xs:element name="intentsinks">
183         <xs:complexType>
184             <xs:sequence>
```

```
185                 <xs:element ref="intentsink" maxOccurs="unbounded"
                        minOccurs="0" />
186             </xs:sequence>
187         </xs:complexType>
188     </xs:element>
189     <xs:element name="intents">
190         <xs:complexType>
191             <xs:sequence>
192                 <xs:element ref="intent" maxOccurs="unbounded"
                        minOccurs="0" />
193             </xs:sequence>
194         </xs:complexType>
195     </xs:element>
196     <xs:element name="intentfilters">
197         <xs:complexType>
198             <xs:sequence>
199                 <xs:element ref="intentfilter" maxOccurs="unbounded"
                        minOccurs="0" />
200             </xs:sequence>
201         </xs:complexType>
202     </xs:element>
203     <xs:element name="flows">
204         <xs:complexType>
205             <xs:sequence>
206                 <xs:element ref="flow" maxOccurs="unbounded"
                        minOccurs="0" />
207             </xs:sequence>
208         </xs:complexType>
209     </xs:element>
210     <xs:element name="answer">
211         <xs:complexType>
212             <xs:sequence>
213                 <xs:element ref="permissions" />
214                 <xs:element ref="intentsources" />
215                 <xs:element ref="intentsinks" />
216                 <xs:element ref="intents" />
217                 <xs:element ref="intentfilters" />
218                 <xs:element ref="flows" />
219             </xs:sequence>
220         </xs:complexType>
221     </xs:element>
222 </xs:schema>
```

Listing A.1: XSD for AQL-Answers

A.2 Configuration XSD

```
1 <xs:schema attributeFormDefault="unqualified" elementFormDefault="
      qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
2     <xs:element name="priority" type="xs:int" />
3     <xs:element name="path" type="xs:string" />
4     <xs:element name="run" type="xs:string" />
```

```
5       <xs:element name="result" type="xs:string"/>
6       <xs:element name="questions" type="xs:string"/>
7       <xs:element name="instances" type="xs:int"/>
8       <xs:element name="memoryPerInstance" type="xs:int"/>
9       <xs:element name="tool">
10          <xs:complexType>
11              <xs:sequence>
12                  <xs:element ref="priority"/>
13                  <xs:element ref="path"/>
14                  <xs:element ref="run"/>
15                  <xs:element ref="result"/>
16                  <xs:element ref="questions"/>
17                  <xs:element ref="instances"/>
18                  <xs:element ref="memoryPerInstance"/>
19              </xs:sequence>
20              <xs:attribute type="xs:string" name="name" use="optional"
                    />
21              <xs:attribute type="xs:string" name="version" use="
                    optional"/>
22          </xs:complexType>
23      </xs:element>
24      <xs:element name="androidPlatforms" type="xs:string"/>
25      <xs:element name="maxMemory" type="xs:int"/>
26      <xs:element name="tools">
27          <xs:complexType>
28              <xs:sequence>
29                  <xs:element ref="tool" maxOccurs="unbounded"
                        minOccurs="0"/>
30              </xs:sequence>
31          </xs:complexType>
32      </xs:element>
33      <xs:element name="preprocessors">
34          <xs:complexType>
35              <xs:sequence>
36                  <xs:element ref="tool" maxOccurs="unbounded"
                        minOccurs="0"/>
37              </xs:sequence>
38          </xs:complexType>
39      </xs:element>
40      <xs:element name="config">
41          <xs:complexType>
42              <xs:sequence>
43                  <xs:element ref="androidPlatforms"/>
44                  <xs:element ref="maxMemory"/>
45                  <xs:element ref="tools"/>
46                  <xs:element ref="preprocessors"/>
47              </xs:sequence>
48          </xs:complexType>
49      </xs:element>
50  </xs:schema>
```

Listing A.2: XSD for Configuration Files

Appendix B

Digital Appendix

The directories and their content, which can be found on the attached CD-Rom, are described in the following.

- **AQL** In this directory the AQL-System can be found as executable .jar file and in form of an Eclipse project. Besides, the Eclipse projects of all tools used during the evaluation are located here, namely the AQL-Comparator and the AQL-Minimizer. Lastly, the syntax of the AQL is stored in this directory in human and machine readable formats.
- **Apps** The .apk files of all apps used during the evaluation are included in this directory.
- **Examples** Multiple configuration examples as well as an AQL-Query and an AQL-Answer example can be found in this directory. Furthermore, it contains the .apk files of the `SIMApp` and the `SMSApp`.
- **Tools** This directory contains the tool PAndA^2. Additionally, bash and batch scripts, that can be used to execute other tools, can be found in the associated subdirectories (FlowDroid, IC3).

Bibliography

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[AYU+09] Shabtai Asaf, Fledel Yuval, Kanonov Uri, Elovici Yuval, and Dolev Shlomi. Google android: A state-of-the-art review of security mechanisms. *Computing Research Repository*, 2009.

[BSGM15] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. Covert: Compositional analysis of android inter-app permission leakage. *IEEE Transactions on Software Engineering*, pages 866–886, 2015.

[CFGW11] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[EGC+10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, pages 393–407, Berkeley, CA, USA, 2010. USENIX.

[Faw06] Tom Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, pages 861–874, 2006.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, pages 319–349, 1987.

[Gar17] Gartner. Gartner says worldwide sales of smartphones grew 9 percent in first quarter of 2017. `http://www.gartner.com/newsroom/id/3725117`, 2017, (accessed May 23, 2017).

[GKP+15] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. Information flow analysis of android applications in droidsafe. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, NDSS '15, San Diego, CA, USA, 2015.

[Goo17a] Google. Android security 2015 year in review. `https://source.android.com/security/reports/Google_Android_Security_2015_Report_Final.pdf`, 2016, (accessed February 13, 2017).

[Goo17b] Google. Android security 2016 year in review. `https://source.android.com/security/reports/Google_Android_Security_2016_Report_Final.pdf`, 2017, (accessed May 22, 2017).

[Goo17c] Google. Application fundamentals. `https://developer.android.com/guide/components/fundamentals.html`, (accessed February 14, 2017).

[Goo17d] Google. Protocol buffers. `https://developers.google.com/protocol-buffers`, (accessed February 14, 2017).

[JTP16] Marie-Christine Jakobs, Manuel Töws, and Felix Pauck. Panda2: Analyzing permission use and interplay in android apps (tool paper). In *Proceedings of the Workshop on Formal and Model-Driven Techniques for Developing Trustworthy Systems*. School of Computing Science, University of Newcastle upon Tyne, 2016.

[KFB+14] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, pages 1–6, New York, NY, USA, 2014. ACM.

[LBB+15a] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 280–291, Piscataway, NJ, USA, 2015. IEEE Press.

[LBB+15b] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Apkcombiner: Combining multiple android apps to support inter-app analysis. In *Proceedings of the 30th IFIP International Information Security Conference (SEC)*, IFIP SEC '15, pages 513–527, 2015.

[MLL05] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: A program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 365–383, New York, NY, USA, 2005. ACM.

[MS13] Heiko Mantel and Henning Sudbrock. *Types vs. PDGs in Information Flow Analysis*, pages 106–121. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[OJD+16] Damien Octeau, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 469–484, New York, NY, USA, 2016. ACM.

[OJM12] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting android applications to java bytecode. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 6:1–6:11, New York, NY, USA, 2012. ACM.

[OLD+15] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 77–88, Piscataway, NJ, USA, 2015. IEEE Press.

[OMJ+13] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Security Symposium*, USENIX Security '13, pages 543–558, Washington, D.C., 2013. USENIX.

[oNSS17] Committee on National Security Systems. National information assurance (ia) glossary. `https://www.hsdl.org/?view&did=7447`, 2010, (accessed February 14, 2017).

[RAMB16] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, NDSS '16, San Diego, CA, USA, 2016.

[RCT+14] Tristan Ravitch, E. Rogan Creswick, Aaron Tomb, Adam Foltzer, Trevor Elliott, and Ledah Casburn. Multi-app security analysis with fuse: Statically detecting android app collusion. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, PPREW-4, pages 4:1–4:10, New York, NY, USA, 2014. ACM.

[tF17] IDC Analyze the Future. Smartphone os market share, 2016 q3. `http://www.idc.com/promo/smartphone-market-share/os`, 2016, (accessed February 14, 2017).

[VRCG+10] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *Proceedings of the CASCON First Decade High Impact Papers*, CASCON '10, pages 214–224, Riverton, NJ, USA, 2010. IBM Corp.