PADERBORN UNIVERSITY
The University for the Information Society

Faculty for *Computer Science*, Electrical Engineering and Mathematics

Cooperative Android App Analysis

Felix Pauck

Dissertation

submitted in partial fulfillment
of the requirements for the degree of
Doktor der Naturwissenschaften (Dr. rer. nat.)

Advisors

Prof. Dr. Heike Wehrheim
Prof. Dr. Eric Bodden

January 6, 2023

Abstract

Program analyses allow us to assess whether a program fulfills certain properties. For instance, they can be used to check whether private data on our smartphones is secure – not visible to the public or manipulable by potential adversaries. To rely on analyses, they must be as effective and efficient as possible. However, the construction of a perfect analysis has partially been proven to be undecidable. Therefore, the construction of effective and efficient program analyses has actively been researched for decades.

In this thesis we present an approach to combine existing analyses such that their complementary effects can be brought together beneficially. To realize this approach we developed the AQL, an interface language that can be used to interact with arbitrary analyses in the same generic way. The language follows the principles of component-based software engineering; every analysis is treated as a component available for composition. Hence, it allows us to build analysis compositions called *cooperative analyses*.

By tracking an analysis's resource usage (e.g. analysis time), we usually evaluate the efficiency of an analysis. Analogously, by comparing actual analysis results with expected results, we typically evaluate an analysis's effectiveness. In the past, such evaluations were often error-prone and irreproducible since they were performed manually and often by multiple people with different knowledge and resources leading to different and differently-interpreted results.

To cope with these issues, we further present an approach that allows us to evaluate analyses (cooperative or not) automatically. This approach makes use of the AQL to interact with arbitrary analyses, and to guarantee that actual and expected analysis results are described in an unambiguous way. Thereby, arbitrary analyses can be evaluated under the same circumstances and errors due to misinterpretation can be avoided. We call experiments conducted via this approach *automatic and reproducible benchmarks*.

To evaluate the effectiveness of the cooperative analysis approach, we widely applied it in the area of Android taint analysis. Overall, we instantiated six cooperative analyses following 18 different strategies while incorporating 32 tools and compared them against 13 standalone taint analysis tools. To perform this comparison, various micro and real-world benchmark suites have been employed as automatic and reproducible benchmarks. In the end, we found that cooperative analyses most of the time outperform their standalone counterparts.

Finally, the thesis reveals that bringing together various solution approaches for different analysis challenges in cooperative analyses may lead to improvements over state-of-the-art standalone approaches. Furthermore, it is shown that these improvements can be measured automatically and in a reproducible fashion.

Zusammenfassung

Programmanalysen ermöglichen es uns festzustellen, ob ein Programm bestimmte Eigenschaften erfüllt. Zum Beispiel können sie genutzt werden, um zu prüfen, ob private Daten auf unseren Smartphones sicher bzw. nicht öffentlich einsehbar oder manipulierbar durch potenzielle Angreifer sind. Um sich auf Analysen verlassen zu können, müssen diese so effektiv und effizient wie möglich sein. Jedoch wurde teilweise schon bewiesen, dass die Konstruktion einer perfekten Analyse unentscheidbar ist. Daher wird die Konstruktion von effektiven und effizienten Programmanalysen seit Jahrzehnten stetig weiter erforscht.

In dieser Arbeit präsentieren wir einen Ansatz, mittels dessen existierende Analysen kombiniert werden können, um ihre komplementären Eigenschaften vorteilhaft zusammenzubringen. Zur Realisierung dieses Ansatzes wurde die AQL entwickelt, eine Schnittstellensprache, die es uns erlaubt mit beliebigen Analysen auf dieselbe Art zu interagieren. Die Sprache wurde im Einklang mit Konzepten des komponentenbasierten Softwareentwurfs entwickelt. Dementsprechend wird jede Analyse als eine Komponente betrachtet, die zur Komposition bereitsteht. Folglich ermöglicht uns die Sprache die Erstellung von Analysekombinationen, die wir *kooperative Analysen* nennen.

Durch das Messen der Ressourcennutzung einer Analyse (z.B. benötigte Zeit), können wir die Effizienz dieser Analyse bestimmen. Ebenso können wir die Effektivität durch den Vergleich tatsächlicher und erwarteter Ergebnisse ermitteln. In der Vergangenheit waren derartige Auswertungen oft fehleranfällig und nicht reproduzierbar, da sie händisch und oftmals von mehreren Personen mit unterschiedlichen Vorkenntnissen und Ressourcen durchgeführt wurden. Häufig kamen dadurch unterschiedliche bzw. unterschiedlich interpretierte Ergebnisse zustande.

Um diese Probleme zu lösen, präsentieren wir einen weiteren Ansatz, der es uns erlaubt (kooperative) Analysen automatisch zu evaluieren. Der Ansatz nutzt die AQL, um mit unterschiedlichen Analysen zu interagieren und um zu garantieren, dass tatsächliche und erwartete Ergebnisse eindeutig beschrieben sind. Somit können beliebige Analysen unter denselben Gegebenheiten durchgeführt und Fehler aufgrund von Fehlinterpretationen vermieden werden. Wir nennen Experimente *automatische und reproduzierbare Benchmarks*, wenn sie mittels dieses Ansatzes durchgeführt werden.

Zur Feststellung der Effektivität des kooperativen Analyseansatzes setzen wir ihn vielfältig im Bereich der Android Taint-Analyse ein. Insgesamt haben wir sechs kooperative Analysen konstruiert, die 18 unterschiedliche Strategien verfolgen und 32 Analysewerkzeuge nutzen. Diese kooperativen Analysen haben wir mit 13 nicht-kooperativen Taint-Analysewerkzeugen anhand von unterschiedlichen Mikro-Benchmarks und Realen-Benchmarks verglichen. Zur Durchführung dieser Vergleiche kamen die von uns entwickelten, automatischen und reproduzierbaren Benchmarks zum Einsatz. Abschließend konnten wir feststellen, dass kooperative Analysen in den meisten Fällen besser als ihre nicht-kooperativen Gegenstücke funktionieren.

Letztendlich konnte aufgezeigt werden, dass das Zusammenbringen unterschiedlicher Lösungsansätze für diverse Analyseherausforderungen in kooperativen Analysen zu Verbesserungen gegenüber modernsten Programmanalyseansätzen führen kann. Zudem konnten diese Verbesserungen automatisch und reproduzierbar gemessen werden.

Acknowledgment

Without the people who supported me, this thesis would never have seen the light of day. First, I want to thank Heike Wehrheim for her support, guidance, and especially for the academic freedom she gave me. Right from the start of my doctorate, I was able to pursue and evolve existing and new ideas of my own. If nothing else, this circumstance has made the work of the last few years a phase of my life that I will always remember happily.

I also want to thank Eric Bodden. Without him, his lectures and the capture-the-flag event he brought to Paderborn, I would have never discovered my passion for research tailored to Android app analysis. Furthermore, thanks to Marie-Christine Jakobs and Ben Hermann, not only for joining the examination committee and their work on joint publications, but also for the many interesting discussions and chats we had together in Paderborn and at conferences.

Thanks to all coauthors and collaborators of the research papers we published together. In particular to Linghui Lou for the extensive collaboration in context of our united research on real-world benchmarks and their evaluation, and to Shikun Zhang and Stefan Schott for their work on two distinct master theses that I was given the chance to supervise and which we could together forge into research papers.

Many thanks to Elisabeth Schlatt for her help with respect to administrative issues and for always listening. I also want to thank the CRC-901 (On-The-Fly Computing), in particular Ulf-Peter Schroeder, for helping me with organizational issues and giving me the chance to travel to scientific conferences across the world. I also want to thank all my colleagues from my advisors' research groups and from the CRC. You made this time fun, interesting and unforgettable in many ways.

Last but not least, I want to thank my friends, my family and especially Melanie, who always encouraged me and accepted my busy schedule while working on this thesis without any complaints.

Table of Contents

1 Introduction

Android is the most-used – not only mobile – operating system [169]. It is employed on a broad range of devices, ranging from smartphones and tablets over watches, TVs and desktop computers to cars. Unquestionably, the most iconic device is the smartphone. The smartphone in particular deals with huge amounts of sensitive data recorded by sensors (e.g. GPS location), provided willingly by the user (e.g. contact data), or required technically and legally (e.g. IMEI number, a unique device identifier).

Because of Android's huge user base and the amount of sensitive data dealt with, Android has become a compelling attack surface [104] – not only, but especially for data thefts [164, 166, 173]. Consequently, there is an arms race taking place between attackers, trying to steal data, and defenders, attempting to prevent such attacks. Attackers always try to find new methods to access data in unauthorized and/or undetectable ways. Defenders steadily develop new instruments to detect (and avoid) attacks. One such instrument is *program analysis* or, in the context of Android, *app analysis*.

Many competitive defenders take part in this race by contributing novel concepts for analyses which frequently get implemented in new, improved or updated tools. However, most defenders work independently – everyone is working on the same task (detecting and preventing attacks) but focuses on different subtasks. For example, subtasks may concentrate on certain analysis challenges caused by the Android framework (e.g. communication between apps) or an app's programming language (e.g. reflection). Thus, the Android app analysis tasks are already divided. To divide *and conquer*, the different defenders and their techniques must be brought together. In this thesis we propose *cooperative analysis* as an approach to combine different program analyses so that defenders can work together without the need to become experts at all tasks.

1.1 Approach

Program analysis is applied in various research areas, traditionally in compiler construction and more recently in security and software engineering [1, 4, 10]. It serves different purposes ranging from simple (code-)style checks over code optimization [5, 11] to bug finding [16] and vulnerability detection [18, 19]. For the latter, program analysis can have different objectives, for example, detecting privacy leaks [33], data races [24] or API misuses [71]. Generally, a program analysis can be of two types: a *static* analysis inspects the code of a program without executing it, a *dynamic* analysis executes a program and monitors its behavior [10]. Consequently, static analyses may analyze whole programs while approximating unknown runtime information, whereas dynamic analyses can only argue about code that was executed and monitored without the need to approximate information. Both types of analysis are frequently applied in research and industry. For example, Google's annual security report for 2021 states:

> "Google has developed an automated application risk analyzer that performs static and dynamic analysis [...] to detect potentially harmful app behavior." [143, 144]

Different program analysis implementations may be required in order to deal with different programming languages, execution environments or operating systems. In this thesis we propose *cooperative analysis*, an approach to combine program analyses that serve different purposes, focus on different objectives or come with different requirements to build a more comprehensive analysis that unifies the capabilities and benefits of the analyses it combines.

Especially in the area of Android app analysis, in which we face manifold analysis tasks, for instance in regard to an app's programming language or the Android framework, we want to apply cooperative analysis so that one analysis can deal with a single task at a time while multiple analyses together chase a bigger objective. In the context of this thesis, this bigger objective is embodied in the detection of privacy leaks. One particular analysis type allowing us to do so is *taint analysis*.

Taint analysis may be implemented statically [33, 37] or dynamically [26, 56] to track the flow of sensitive information through a program. It starts tracking at so-called *sources*, i.e. statements extracting sensitive information. Whenever tainted or tracked information reaches a *sink*, a statement that reveals information to the outside, a privacy leak in form of a *taint flow* is reported. Taint flows describe connections of sources and sinks.

Benign taint flows may intentionally be programmed into an app, for example, a smartphone user may want the ability to knowingly share contact data via a messaging app. However, taint flows may also be programmed into an app by accident such that they can be exploited to unknowingly exfiltrate sensitive information. In case of malware apps, taint flows are added to an app on malicious purpose. Additionally, malware developers typically make extensive use of Android framework or programming language features to hide malicious taint flows from analyses. Ultimately, we often find a whole *jungle of features* that must be analyzed to uncover taint flows. For example, app components may communicate with each other such that an analysis is required to surveil this communication. Often we find dynamically-loaded or native code in an app, both of which particularly challenge static approaches that need to consider any possibly loaded code or other programming languages used to implement the native code portion. Dynamic analyses are often challenged by taint flows which can only be triggered under very special circumstances, e.g. only at a certain time of day. To cooperatively analyze this jungle of features we employ and develop specialized analysis tools for different tasks and bring together their results.

Fortunately, there are already many static, dynamic and hybrid Android app analysis tools that focus on different features. For example, there are tools that resolve dynamically loaded code [54], reveal taint flow parts in native code [75, 93], or deal with communication between app components [32, 46, 97]. Although these tools rarely work together, there exist a few examples for Android app analysis tools that cooperate. For instance, there are taint analysis tools [35, 45] that attempt to find taint flows across component boundaries, i.e. taint flows that start in one component and end in another. These tools typically combine two implementations: one that detects flows inside components and another finding connections between components. Up to now, these combinations have not followed principles like *low coupling, high cohesion* [2], hence, if one implementation is outdated because it was updated by their authors, the whole combination possibly needs to be updated too. Consequently, we experience that many of these tools, which originally delivered promising results, are not effective, maintained or up-to-date anymore.

To solve this issue, cooperative analysis follows the fundamentals of component-based software engineering [9, 12]. In this context Lagaisse and Joosen define a component as follows:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-parties." [17]

With respect to this definition, program analyses represent the independent components that we, as a third-party, want to combine. To realize our approach, we attach “contractually specified interfaces” by defining a query language that allows us to interact with arbitrary program analyses. Thereby, cooperative analyses can easily be composed by formulating a query, and components (program analyses) can be swapped out by simply adapting a query. The design, specification and implementation of this query language represents the first of three major contributions of this thesis. All three contributions are described in more detail in the following section.

1.2 Contributions and Structure

In this thesis we elaborate three major contributions. The first describes the concept and realization of the cooperative analysis approach. The second focuses on benchmarking standalone and cooperative taint analyses. The third and last contribution deals with six different instances of cooperative analyses and their evaluation.

To realize the cooperative analysis approach, we derive requirements from related work, i.e. we examine analysis tools, query languages, result formats and existing cooperative concepts to identify best practices and to avoid the opposite. On this basis, we will design and specify a new query language, namely the *Android App Analysis Query Language (AQL)*. In addition, *strategies* are proposed to simplify and automate the interaction with cooperative analyses composed via the AQL. As last part of this first contribution, the AQL-System, a framework implemented for the AQL, is introduced. All details related to this first contribution can be found in Chapter 3.

Program analyses and especially taint analyses are usually evaluated on the basis of benchmarks [21, 52]. With respect to Android a benchmark consists of a set of apps and a description of the expected analysis results – mostly referred to as *ground truth* [67, 91]. We designed and developed BREW, a framework based on the AQL-System, that allows to create and refine benchmark suites and their ground truths and to automatically execute and evaluate tools on benchmarks. Benchmarks executed via BREW are reproducible with respect to the ground truth specified for a benchmark – no manual interpretation needed. With BREW at hand, we first evaluate 13 standalone Android taint analysis tools. While doing so, we evaluate whether these 13 tools keep their promises as denoted in the respective proposing papers. In particular, we check which features are actually supported, if promised accuracy scores can be reached and if the tools are ready to be used in practice facing real-world scenarios. The results, which show that most promises are kept, embody *the baseline* against which we compare cooperative analyses as a part of the third and last contribution. The Chapter entitled “Automatic and Reproducible Benchmarks” (see Chapter 4) is dedicated to this second contribution.

The last contribution uses the first contribution (the AQL) to compose cooperative analyses and strategies, and the second contribution (automatic and reproducible benchmarks) to evaluate these. Therefore, six cooperative analyses tailored to different analysis challenges are proposed and evaluated. For each cooperative analysis the involved tools, strategies and results are documented. In the end, a comparison against the baseline shows that cooperative analyses outperform standalone analyses most of the time. This comparison and all results in regard to these six cooperative analyses can be found in the evaluation chapter (see Chapter 5).

Before these three contributions are tackled, all the required background knowledge is introduced (see Chapter 2). This includes important details about Android as an operating system and framework, information about Android app analyses, their benchmarks and existing frameworks and tools. The conclusion given at the end wraps up the cooperative analysis topic and presents a vision for its future (Cooperative Analysis for Everyone).

Furthermore, all artifacts related to this thesis can be found in the (digital) Appendix A.6. It includes the raw result data of all experiments and all frameworks, tools and benchmark suites that were developed as a part of this thesis.

1.3 Publication Details

The AQL was first presented in a master thesis [63] six years ago and has steadily been extended and improved since then. It also represents the basis for multiple studies published in accordance with this thesis.

In the ReproDroid study [67], which was awarded with an *ACM Distinguished Paper Award* at ESEC/FSE 2018, we checked whether standalone Android taint analyses keep their promises. An updated and extended answer to this question can be found in the baseline section (see Section 4.2). One of the extensions originates from the TaintBench study [91], which we published in the EMSE Journal (2022). As a part of this study we developed the TaintBench benchmark, a benchmark suite that consists of real-world malware apps for which a ground truth was manually determined. Updated and extended results with respect to this study can also be found in the baseline section.

The CoDiDroid study [75], published 2019 also at ESEC/FSE, presents cooperative analyses composed via the AQL. The cooperative analyses proposed are part of those presented in Chapter 5. Our studies on merging [76], slicing [87] and simplifying [92] Android apps propose three additional building blocks for cooperative analyses presented in the same chapter.

Another related publication dealing with the novel concept of *benchmark fuzzing* [94] for Android taint analyses was recently awarded with an *IEEE Best Research Paper Award* at SCAM (2022). However, using this concept in the context of cooperative analysis is currently considered future work.

2 Background

The required background information and terminologies used throughout this thesis are introduced and explained in this chapter. First, some features and specialties of the Android framework and its apps as well as the Java programming language are explained. This is required to understand the challenges that arise from these for program analyses targeting Android apps. Details about these challenges and general information about program analysis, and in particular Android app analysis with a focus on taint analysis, are given next. Third, it is explained how to benchmark Android app analyses. The important but often mixed up terminologies in this context are clearly defined for the remainder of this thesis. Lastly, information about the tools that implement Android app analyses, the frameworks they are based on and the benchmark suites used to evaluate them are provided. All sections are accompanied by one running example which is split into four parts (plus a fifth part in another chapter). It can be found at the end of each section (see 2.1.5, 2.2.4, 2.3.4 and 2.4.4 as well as 3.2.1).

2.1 Android

Android is an open source, linux based operating system originally (since 2007) designed for the deployment on smartphones. Nowadays, it is deployed on various other devices such as tablets, TVs or even desktop computers and cars. Primarily, Android allows the execution of various programs, called applications (apps). Apps can be downloaded from app markets (e.g. Google's Play Store [142]) or open source repositories (e.g. F-Droid [132]). There is a plethora of apps that are marketed this way. The apps fulfill a wide variety of purposes from simple text editing or messaging apps to banking or even 3D-gaming apps. The programming language used to develop these apps originally was Java; today Kotlin[1] is primarily used. Furthermore, the inclusion of native libraries allows to integrate C/C++ code fragments. However, once compiled into an Android application package (`.apk` file) it makes no difference if the app was written in Java or Kotlin since these packages only hold compiled bytecode in form of one or more `.dex`[2] files which in turn contain the compiled Java bytecode. If there are native C/C++ libraries attached, these will be part of the `.apk` file in form of compiled shared object files. Additionally, each Android application package holds one Android manifest file that provides general information about the app (see Subsection 2.1.3) and many resource files (e.g. user interface layout files, style sheets or images).

In comparison to ordinary Java programs, Android apps are slightly different. For example, the Android library is always available – accordingly it is often colloquially said that an app is written in Android meaning that Java/Kotlin plus Android libraries were used. The Android software development kit (SDK) includes API-specific development versions of this library as well as the build tools required to build an Android app. Android apps also do not have a main method like the one which is called at the start of a Java program. Instead, the Android system manages when and which parts of an app are executed. Especially the latter difference makes it particularly challenging to develop program analyses targeting Android apps.

[1]"Android mobile development has been Kotlin-first since Google I/O in 2019." [162]

[2]Although the Dalvik virtual machine has been replaced by ART (Android Runtime – first appearance 2013 with Android 4.4 "KitKat" and released 2014 with Android 5.0 "Lollipop" [163]) the Dalvik executable file format (`.dex`) is still in use.

In the following, we will take a closer look at the structure of apps and features of the Android framework.

2.1.1 Components

Every Android app consists of at least one component. Any component is implemented by extending component classes provided by the Android framework (library). Each component class comes with a set of methods, the so-called *life-cycle methods*.

> **Life-cycle methods** are called when certain events are triggered that change the state of a component (e.g. app started).

In general, they belong to the bigger set of *callback methods*.

> **Callback methods** are specified to be triggered once a certain event happens (e.g. button clicked or device's battery running low).

The four main component types are introduced in the following list:

- **Activities:** Probably the most commonly used component type is the one responsible for user interfaces, the so-called *activities* [103]. An activity is implemented by extending the class `android.app.Activity` and by defining its layout. For the latter, a layout (`.xml`) file is usually used. Anything visible to the user is principally defined in layouts. For example, a button and which callback method is invoked once it is clicked, can be defined in an activity's layout file. A simplified version of an activity's life-cycle is visible in Figure 1. Once an activity is launched, the

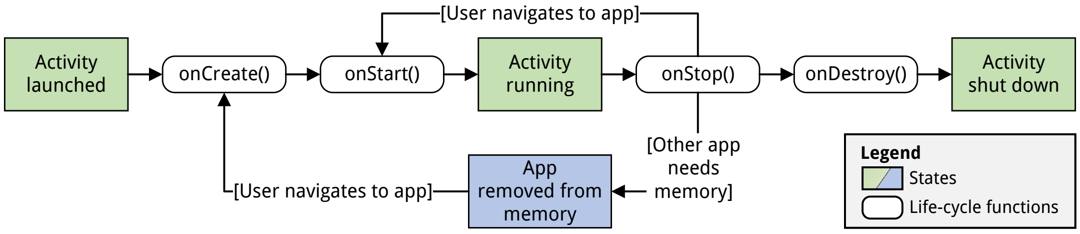


Figure 1: An Activity Components Life-Cycle

  methods `onCreate` and `onStart` are executed in sequence. Once a running activity is stopped, `onStop` is executed. Thereafter the activity might be shut down or be restarted depending on what caused the activity to be stopped.

- **Services:** To execute tasks in the background, *service* [103] components are employed by extending `android.app.Service`. The tasks typically executed by a service may take longer to finish (e.g. synchronizing information). Services come with their own life-cycle and may continue running even though the app has been closed.

- **Broadcast Receivers:** Whenever an app should react to (system-wide) events, it needs to implement a *broadcast receiver* [103] (`android.content.BroadcastReceiver`). The event that triggers the broadcast receiver's implementation is defined at the moment it is registered. For example, a weather forecast app may request updates every 5 minutes, however, when the device's battery runs low a broadcast receiver could be triggered that lowers the update frequency to reduce the app's energy consumption. The events to which broadcast receivers can react are defined in the Android framework, however, developers can also trigger or receive custom events.

- **Content Providers:** The fourth component type, namely *content providers* [103], are intuitively comparable to databases. They hold structured information that can be accessed by different apps. For instance, all information about a user's contacts are typically stored in a database that is accessible via a content provider. Content providers are implemented by extending the `android.content.ContentProvider` class and accessed via queries similar to REST API requests.

2.1.2 Inter-Component/-App Communication (ICC/IAC)

Components of any type may interact with each other, for example, one activity may call another one to switch from one user interface to the next. To realize such interactions the components must communicate with each other. The communication between components that belong to the same app is called *inter-component communication (ICC)*. Whenever the components belong to different apps, it is called *inter-app communication (IAC)*. The ICC/IAC interface given by the Android framework works via so-called *intents* [153].

> **Intents** are sent from one component to another – implicitly triggering the receiving target component. An intent can also be broadcasted to various (broadcast) receivers and may transport data.

Intents are divided into two types: explicit and implicit. *Explicit intents* directly name the component that should receive the intent by naming the respective Java class. Therefore, the intent sender must know the receiver which is why explicit intents are designed to be used in ICC context only. *Implicit intents* are only describing a potential receiver without knowing if there is a component that suits the description. To receive an implicit intent a component must specify an *intent filter* [153] – the counterpart of an (implicit) intent.

> **Intent filters** specify which intents are eligible to be received by a component. An arbitrary number of intent filters can be specified for a single component.

Intent and intent filters follow the lock and key principle: both define an *intent triple* consisting of the three elements action, category and data. The *action* element describes the action the receiving component should perform. The *category* describes to which kind the receiving component must belong. Optionally, the *data* element can be specified to e.g. influence which type of data a component may send or receive. Only if the triple of an intent matches the triple of an intent filter the associated component may receive the intent. The Android system decides if two triples match and delegates the intent accordingly.

> An intent **matches** an intent filter if the associated intent triples match.

2.1.3 Manifest

Each Android app comes with one Android manifest: an `.xml` file that summarizes the app's properties and contents. First of all, it contains general information, for example, the name of the app, a unique app identifier[3] (package name) or the targeted and minimally-supported Android version.

> The package name often also refers to the **main package**, a package that potentially contains most or all of an app's code.

[3]On each Android device only one app can be installed per package name. Furthermore, the package name is also a unique identifier in markets such as Google's Play Store.

However, this is only a convention, i.e. there are also cases where only little or no classes can be found in the main package or any of its sub-packages [92].

Second, it lists the hardware features used and the *permissions* required by the app.

> **Permissions** must be granted by the user before certain Android framework methods can be accessed. Any permission possibly required by an app must be declared in its manifest.

For instance, to use a device's camera the associated hardware feature (`android.hardware.camera`) and permission (`android.permission.CAMERA`) must be declared in the manifest [121].

Lastly, the manifest contains a list of all components comprised in the app. The intent filters of the respective components are also denoted in this list. Typically, the *launcher activity* and its associated intent filter is included in this list.

> The **launcher activity** is started once the app is launched by the user. Hence, there should be only one activity for which an intent filter with action `android.intent.action.MAIN` and category `android.intent.category.LAUNCHER` is specified.

Note, intent filters can be declared at runtime, thus, the intent filters included in the manifest must not form a final set.

2.1.4 Features and Specialties

Java and especially Android come with more features and specialties. For brevity, we will only briefly introduce five more. The first two are basic concepts:

- **Fields and static fields** are Java variables that can be used across method and class boundaries, respectively. Intuitively, static fields can be interpreted as global variables.
- **Threads** can be used in Java to execute various program parts concurrently. Android extends the concept of threads by introducing `AsyncTasks` (until Android API 30 [119]). However, from a programming/analysis perspective an `AsyncTask` is just a wrapper for a `Thread`.

Fields in the context of concurrently executed threads may lead to data races – the same variables may be written or read in an unforeseeable order. Consequently, variables may hold different values depending on the order of execution. Hence, these basic and commonly used concepts often complicate analysis approaches.

Two different concepts, that are usually used less frequently but often extensively to hide malicious behavior, are reflection and native code:

- **Reflection** [188] is a concept provided by Java to load classes and methods which can be used as objects in the program. For example, instead of directly calling a method it can be loaded as an object and then be invoked via that object.
- **Native code** [155] can be loaded via the Java Native Interface (JNI). Native code in this context typically is C or C++ code that can be executed as a part of a Java program or Android app in form of a so-called native method.

Both concepts allow to load code parts dynamically which again makes it more difficult to decide which parts are executed when.

In order to enable older Android devices to execute up-to-date apps, support libraries are attached to apps to guarantee *backward compatibility*.

> **Compatibility** or **support libraries** can usually be found in the package `andro-id.support` (or `com.android.support`) and in the package `androidx` since the introduction of Android Jetpack [158].

Due to the frequent release of new Android versions, these libraries have grown steadily. With respect to code size (e.g. lines of code), such libraries are often bigger than the app itself. In consequence, analyses must scale with respect to such libraries in order to be applicable.

2.1.5 Running Example 1: Introduction (Part 1/5)

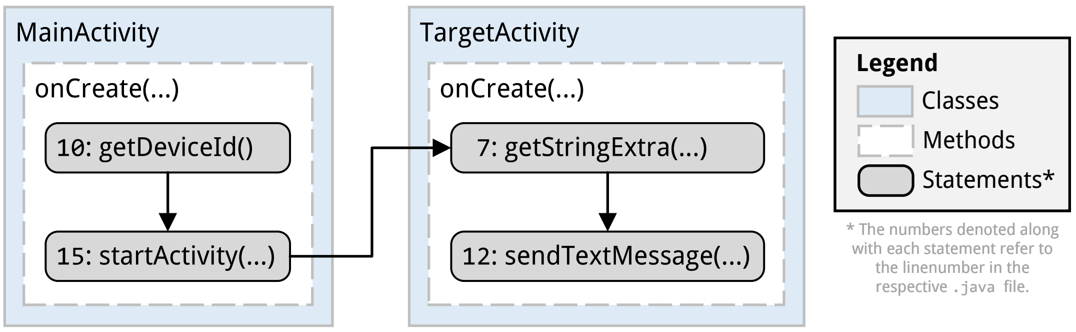


Figure 2: Overview (Example 1 – Part 1)

A running example is introduced in this subsection. It is used for explanation purposes throughout the whole background chapter. It exemplifies most of the concepts explained above. An overview of the example can be found in Figure 2. The example consists of one app with two activity components (`MainActivity` and `TargetActivity`) that communicate with each other (ICC). Both activities override the life-cycle method `onCreate`. If the user launches the app, the `onCreate` method of `MainActivity` is called first, since this activity is declared as the launcher activity in the app's manifest (see Lines 10–13 in Listing 1). Line 5 and 6 of the manifest show two permission declarations. The `READ_PHONE_STATE` permission is required to retrieve information such as the sim card's serial number or the device's identification number (IMEI[4]). The second permission `SEND_SMS` is required to – as the name suggests – send short messages.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="de.foellix.aql.thesis.examples.example1">
4
5     <uses-permission android:name="android.permission.READ_PHONE_STATE" />
6     <uses-permission android:name="android.permission.SEND_SMS" />
7
8     <application android:label="Example1" >
9         <activity android:name=".MainActivity">
10            <intent-filter>
11                <action android:name="android.intent.action.MAIN" />
12                <category android:name="android.intent.category.LAUNCHER" />
13            </intent-filter>
14        </activity>
15        <activity android:name=".TargetActivity">
16            <intent-filter>
```

[4]IMEI: International Mobile Equipment Identity

```
                <action android:name="de.foellix.aql.thesis.examples.TARGET" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Listing 1: Manifest of the App (Example 1)

Listing 2 shows the source code of the `MainActivity`. In Line 7 the layout of the activity's user interface is loaded. Line 9 and 10 are used to extract the IMEI and store it inside the variable `imei`. This variable is attached to an intent in Line 14. Via `startActivity` (Line 16) this intent is *launched*. If there is no other app that declares the action defined in Line 13, the Android system directly delegates the intent to the `TargetActivity` component – otherwise it will ask the user which component should be used.

```
 1  public class MainActivity extends Activity {
 2      public static final String DATA = "data";
 3
 4      @Override
 5      protected void onCreate(Bundle savedInstanceState) {
 6          super.onCreate(savedInstanceState);
 7          setContentView(R.layout.activity_main);
 8
 9          TelephonyManager tm = ((TelephonyManager)
                → getSystemService(Context.TELEPHONY_SERVICE));
10          String imei = tm.getDeviceId(); // Source
11
12          Intent intent = new Intent();
13          intent.setAction("de.foellix.aql.thesis.examples.TARGET");
14          intent.putExtra(DATA, imei);
15
16          startActivity(intent); // (Intent-)Sink
17      }
18  }
```

Listing 2: Source Code of the `MainActivity` (Example 1)

The source code of `TargetActivity` can be found in Listing 3. The IMEI is extracted from the received intent in Line 7. In Line 12 the IMEI is finally sent to an arbitrary receiver via a short message. By calling `finish` (Line 14) the second activity is directly shut down after the execution of its `onCreate` method.

```
 1  public class TargetActivity extends Activity {
 2      @Override
 3      protected void onCreate(Bundle savedInstanceState) {
 4          super.onCreate(savedInstanceState);
 5          setContentView(R.layout.activity_target);
 6
 7          String imei = getIntent().getStringExtra(MainActivity.DATA); //
                → (Intent-)Source
 8
 9          Log.d("DEBUG INFO","Sending SMS"); // No Sink
10
11          SmsManager sms = SmsManager.getDefault();
12          sms.sendTextMessage("+49 1337", null, "Leak: " + imei, null, null); //
                → Sink
```

```

        finish();
    }
}
```

Listing 3: Source Code of the `TargetActivity` (Example 1)

The presented example targets Android API 19 (Android 4.4 "KitKat", 2013). An up-to-date version of the same example would require permission requests at runtime (API $\geq$ 23), the usage of `getImei` instead of the deprecated method `getDeviceId` (API $\geq$ 26) and the additional permission `READ_PRIVILEGED_PHONE_STATE` which can only be requested by and granted to system apps (API $\geq$ 29). These changes already prohibit some of the security flaws that will be highlighted when the example is continued (see Subsection 2.2.4). However, all these changes hinder readability which is why we use its simplest version for explanatory purposes.

In Subsection 2.2.4 the example will be continued by introducing the properties an analysis should consider. The example is described in Subsection 2.3.4 as a benchmark case. In Subsection 2.4.4 the example is concluded by showing how it could successfully be analyzed, if existing analysis tools cooperate. Lastly, in Subsection 3.2.1 it is used to intuitively introduce the AQL.

2.2 Android App Analysis

Software or program analyses that target Android apps are called *(Android) app analyses*. Most of the analysis properties and characteristics described in the following are valid for or applicable to Android app analysis as well as program analysis in general. For readability and brevity, we will only refer to apps under analysis even though most statements are valid for other (Java) programs as well. As any other program analysis, Android app analysis can roughly be separated into two different kinds: static and dynamic.

A *dynamic* analysis [26, 56, 83] executes the app under analysis and *monitors* certain properties. To do so, the app's code is usually adapted. Typically, instructions are added that output, for example, when and in which order certain program locations are reached. When such instructions are added into the compiled version of an app, we call this process *instrumentation*.

In contrast, a *static* analysis [25, 33, 34, 35, 37, 40, 42, 45, 58] only deals with an app's (source) code without actually executing it. An adaptation of an app's code is (normally) not required, however, usually the code is transformed into an *intermediate representation (IR)* that better suites analyzability, e.g. unique names for unique variables.

Hence, a static analysis is usually able to analyze an entire app whereas a dynamic analysis can only analyze those parts of an app that are executed. In that respect the given inputs define which parts of an app are executed.

> Dynamic analyses naturally **under-approximate** – if a property cannot be shown for certain inputs, it is not guaranteed that the property cannot possibly be shown for other inputs.

> Static analyses often **over-approximate** – information required for an accurate analysis of an app are summarized (or approximated) since they are hardly (or not at all) computable, e.g. user inputs.

For example, considering a loop index `i`, a dynamic analysis may know its concrete value under certain inputs. However, other inputs may lead to a different valuation for `i`. A static analysis could for example use predicates to approximate the value of `i`, e.g. no matter how often the loop is executed, `i` is and will always be greater than `0` ($i \geq 0$). In the end, it is a tradeoff: both types have their own advantages and disadvantages which is why hybrid analyses and combinations of different analyses have recently become more and more popular [57, 75, 84].

The subsequent subsections provide more information about properties and characteristics of static and dynamic analyses.

2.2.1 Analysis Representations

Commonly, *intermediate representations (IRs)* of the app under analysis are used as a basis to carry out a static analysis or to perform an instrumentation for a dynamic analysis. In short, IRs are designed to simplify the implementation of analyses. Intuitively, IRs are less complex than source code but better readable than bytecode.[5] Different optimizations are possible with and without loss of information. Often the number of different operations is reduced or the identification of methods and variables is simplified.

Java bytecode supports more than 200 different operations [22, 152] and an analysis would probably have to react differently to all of these, hence, when it comes to IRs, operations of the same type are usually aggregated. Consequently, it is easier to build an analysis that reacts in the same way when facing the same type of operation. For example, in Java bytecode a method call and a static method call are two different operations, but in an IR there typically exists just one method call operation and most analyses react equally whether a static or non-static method is called. To avoid loss of information (and to be able to go back to the original representation), the IR must make sure that the information about static method calls is preserved (e.g. by a flag attached to any method call).

Furthermore, IRs often are *three-address-code* [4, 10] languages, i.e. any operation uses at most three operands. Thus, operations such as $a = x+y+z$ are split into two operations ($t = x + y$ and $a = t + z$) which again can be handled the same way by an analysis.

In Java variables with the same name can refer to different variables in different scopes. Hence, many IRs enforce to use unique variable names so that an analysis can unambiguously identify variables by name. IRs enforcing a *single static assignment (SSA)* [4, 10] form go one step further, each variable may only be assigned once.

Usually, not only IRs are used as analysis basis. Most analyses use graph representations in addition. Three different *graph representations* are frequently employed. A brief description of these three can be found in the list below:

- **Control flow graphs (CFGs) [1, 10]:** The control flow of an app describes which statements may be executed after another. Hence, the nodes of a CFG represent basic blocks.

  > **Basic blocks** [1] are statements (e.g. assignments or method calls) or structural elements (e.g. conditions of branches or loops).

  The directed edges of a CFG denote which blocks may be executed after another.

- **Dependence Graphs (DGs) [4]:** As in a CFG, the nodes of a DG represent basic blocks. The directed edges of a DG may describe various dependencies between certain basic blocks. For example, an edge may describe a data dependency, i.e. the data used in a block is dependent on the block(s) where the data is defined.

- **Call graphs (CGs) [4, 10]:** Which method is called by a certain invoke statement can be looked up in a CG. Each node in a CG represents one method. An edge

[5]In Appendix A.1 we compare three code formats: Java source code, Jimple code (IR) and bytecode.

describes that the method (node) where the edge ends can be called from the method where the edge starts.

Which type of IR or graph comes into play depends on the analysis to be executed.

While static analyses mostly use these representations to carry out their analysis, dynamic analyses are often used to enrich these representations. In consequence, hybrid analyses are composed that use dynamic analysis to enrich representation which are then used to run a more precise static analysis. For example, call graphs could be enriched by dynamic analysis: instead of approximating which method(s) may be called by a certain method call, it is monitored which methods are actually called during execution.[6]

2.2.2 Analysis Types

Static analyses can further be divided into *forward* and *backward* analyses. Dynamic analyses are naturally forward ones, since they follow the program flow during execution. Static forward analyses try to determine properties about the past. For example, the *reaching definition (RD)* [10] analysis attempts to determine which variable definitions reach which variable uses. Thus, information about the past – which definitions have been seen – is gathered. One example for a backward analysis, would be the *live variables (LV)* [10] analysis commonly used to eliminate dead code. It provides information about the future by detecting if a variable is used before it is assigned again.

The analysis this thesis focuses on is a security analysis, namely *taint analysis*. It can be implemented as a forward or backward analysis. Taint analyses attempt to find data flows between statements of two types: sources and sinks. *Source* statements extract sensitive information, for instance, when login credentials are provided by the user or sensor data (e.g. GPS location) is accessed. Statements that extract device identifiers (e.g. phone number or IMEI) are also considered to be sources since they can be misused to track individuals. *Sinks* are statements that forward information, for example, by sending short messages or uploading data to the Internet. In Android, sources and sinks are often protected by permissions, thus, there are techniques exploiting this fact to identify sources and sinks on the basis of the Android API [30]. Other techniques use machine learning to do so [36]. A taint analysis either tracks tainted data from sources to sinks or the other way around from sinks to sources. Let us consider the first case. Once the analysis reaches a source, it taints (marks) the variable that carries the sensitive data. If the tainted data reaches a sink, the possible data leak is reported in form of a taint flow. A *taint flow* is usually described by naming the source and sink it connects. Seldom intermediate steps are also reported.

Taint flows are security-critical or malicious whenever the information is leaked to unknown receivers or extracted without informing the user. On the one hand, security-critical taint flows are not always maliciously integrated into an app, often implementation mistakes or unknown vulnerabilities are responsible and get exploited to access and leak data. On the other hand, malicious software like malware, in particular spyware, explicitly attempts to hide its behavior by hiding taint flows through obfuscation, for example, by unnecessarily but extensively nesting objects in one another. Benign taint flows may also be detected by taint analysis, for example, the possibility to willingly and knowingly share contact data via a short message with a friend may be detected as a taint flow. Due to the various possibilities to constitute taint flows it becomes challenging to build an effective and efficient taint analysis. In the following, we will define what makes an analysis effective or efficient and how to evaluate an analysis's performance.

[6] For example, Azim et al. [72] run an instrumented app to get so-called trace files.

2.2.3 Analysis Effectiveness and Efficiency

The efficiency of an analysis is typically evaluated by measuring the runtime of a tool implementing an analysis and comparing it with other tools implementing the same type of analysis. Runtime is also called *execution* or *analysis time* in this context. Furthermore, resource usage is taken into account, e.g. how much memory is required to execute the analysis. Sometimes crashes are counted and also taken into account, since they are not necessarily a bug in the analysis tool but occur, for example, if a tool requires more memory than available while analyzing certain apps.

The effectiveness of any Android app analysis depends on (1) its scope, (2) its awareness of features and (3) its sensitivity.

1) In general an analysis can be an intra- or inter-procedural analysis. While *intra-procedural* analyses only consider one method at a time, *inter-procedural* analyses also analyze the interplay of multiple methods. Analogously, we differentiate Android app analyses as being *intra-* or *inter-component/app* analyses.

> An **intra-component/app** analysis considers one component/app at a time;
> an **inter-component/app** analysis more than one at a time.

2) Analyses targeting Android apps can moreover be aware of callbacks, life-cycles, ICC/IAC, native code, reflection and many other features provided by the programming language Java or the Android execution environment, however, we focus on these five *features*:

- **Callback:** Which methods are callback methods as well as when and where they can be called is known by the analysis.
- **Life-Cycle:** The analysis takes any possible sequence of life-cycle method invokes into account.
- **ICC/IAC:** Intents and intent filters as well as the inter-component connections they represent are recognized by the analysis.
- **JNI:** Native methods called from within the app's code are no black boxes to the analysis. It is able to *uncover* properties of the native parts that are accessed.
- **Reflection:** Whenever a class or method is accessed via reflection the analysis is able to determine which class or method is accessed. Thereby reflection is *resolved*.

3) When quantifying an analysis's effectiveness (in terms of accuracy) the following *sensitivities* are frequently named [29].

- The control flow of a program specifies the execution order of statements. Loops, branches, exceptions and method calls influence or alter this order. An analysis taking an app's control flow into account is **flow-sensitive**.
- The same method can be called from different program locations and with different parameter values. A context describes from where and with which values a method is called. Tracking from which context a method is called makes an analysis **context-sensitive**.
- Analyses may differentiate elements in e.g. arrays, lists or maps. A **field-sensitive** analysis must be able to differentiate fields which belong to the same object.

- In Java, a class can be instantiated arbitrarily often and different variables may refer to the same object. In the tiny example below (see Listing 4), `a` and `b` refer to different objects whereas `b` and `c` represent the same. Thereby, `b.f` is increased in Line 4 although only `c.f++` is denoted. Analyses that differentiate objects of the same type and take aliasing into account are called **object-sensitive**.

```
1 A a = new A();
2 A b = new A();
3 A c = b;
4 c.f++;
```

Listing 4: Aliasing

- Concurrent programs may write and read values to and from the same variables. An analysis taking the inference of writes in one thread and reads in another thread into account is **thread-sensitive**.

- Flows are often split up depending on the fulfillment of conditions. For example, if-statements typically split up a flow into two branches (if and else). An analysis needs to be able to evaluate arbitrary conditions to become **path-sensitive**.

In general, it holds that a more sensitive analysis is more effective. The actual effectiveness in terms of accuracy is most commonly evaluated via benchmarks and measured by computing precision, recall and F-measure – details are provided in the next section about benchmarks (see Section 2.3).

The major goal of analysis developers is to design and build analyses that are efficient *and* effective. However, in most cases a tradeoff between these two properties exists, hence, not seldom the most effective analysis is also the most inefficient.

2.2.4 Running Example 1: Analyzable Properties (Part 2/5)

In this part of the example we want to answer the question: what type of taint analysis is required to successfully analyze the running example?

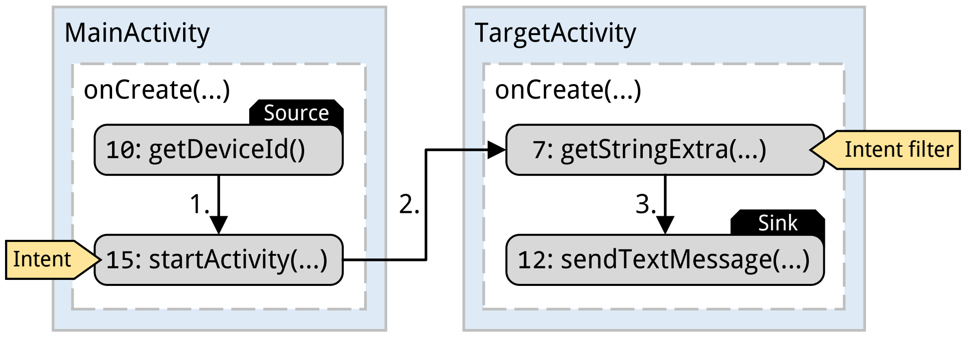


Figure 3: Overview (Example 1 – Part 2)

The taint flow, that should be found, starts in `MainActivity` at the source `getDeviceId()` and ends in `TargetActivity` at the sink `sendTextMessage(...)` as depicted in Figure 3, i.e. it starts and ends in two different components. Hence, the analysis must be able to analyze ICC. Since source and sink can be found in the `onCreate(...)` life-cycle method of the respective components, the analysis must further be life-cycle aware. Lastly, the analysis must be flow-sensitive since it is important in which order the statements are executed - in fact, this holds for any taint analysis scenario.

The flow-sensitive, inter-component, life-cycle aware analysis must then identify (1) sources and sinks, (2) statements that realize the ICC and (3) intents and intent filters. Lastly, all this information must be brought together to find the three flows that realize the taint flow:

1. Intra-component flow from the source to `startActivity(...)`
2. *Inter*-component flow from `startActivity(...)` to `getStringExtra(...)`
3. Intra-component flow from `getStringExtra(...)` to the sink.

To determine the inter-component flow part (2), an intent must be compared to an intent filter to find out whether their associated intent triples match. In this example, only the matching action strings must be compared (see Line 17 in Listing 1 on Page 21 and Line 13 Listing 2 on Page 22). In real-world scenarios, such tasks are usually more complex because the intent triples are harder to compare, e.g. multiple action strings and categories are involved. Similarly, detecting flow part 1 and 3 may become more difficult, once features like reflection, callbacks or threads come into play. Even a simple if branch with a condition that involves unpredictable user inputs may force a dynamic analysis to under- or a static analysis to over-approximate.

2.3 Benchmarks

Benchmark is a manifold term that is defined and interpreted differently in distinct focus areas. To clarify which definition and interpretation is accurate throughout this thesis, the term benchmark as well as various related terms are introduced in the following. Note that all terms are defined with respect to Android taint analysis only. Figure 4 provides an overview of all terms.

To run a benchmark we need the tool(s) to be benchmarked and a benchmark suite. The *benchmark suite* represents the corpus of a benchmark and consists of multiple benchmark cases. Each *benchmark case* comprises one app, the *benchmark app*, and a description of one taint flow, the *expected (analysis) result*. This taint flow can either be an expected or a not-expected taint flow.

> While an **expected** taint flow should be found by an analysis, a **not-expected** taint flow should explicitly *not* be found by an analysis.

The same benchmark app may be used for multiple benchmark cases. In case of an inter-app taint flow multiple benchmark apps may belong to a single benchmark case. Only a collection of apps – without a description of expected analysis results – does not form a benchmark suite.

To benchmark an analysis tool, each of the following steps has to be performed per benchmark case: The *actual (analysis) result* must be computed by executing the analysis tool for the associated benchmark app. Then the expected result must be compared against the actual one, i.e. it must be checked whether the expected or not-expected taint flow is contained in the list of actually detected taint flows.

Note, a benchmark suite always holds benchmark cases without actual results. A *benchmark* is only complete if it holds both, the suite and the actual results, so that the benchmark's outcome can be determined. Nonetheless, the term benchmark is also used as an abbreviation for *specific* benchmark suites, for example, we will use micro benchmark and DroidBench benchmark as an abbreviation for micro benchmark suite

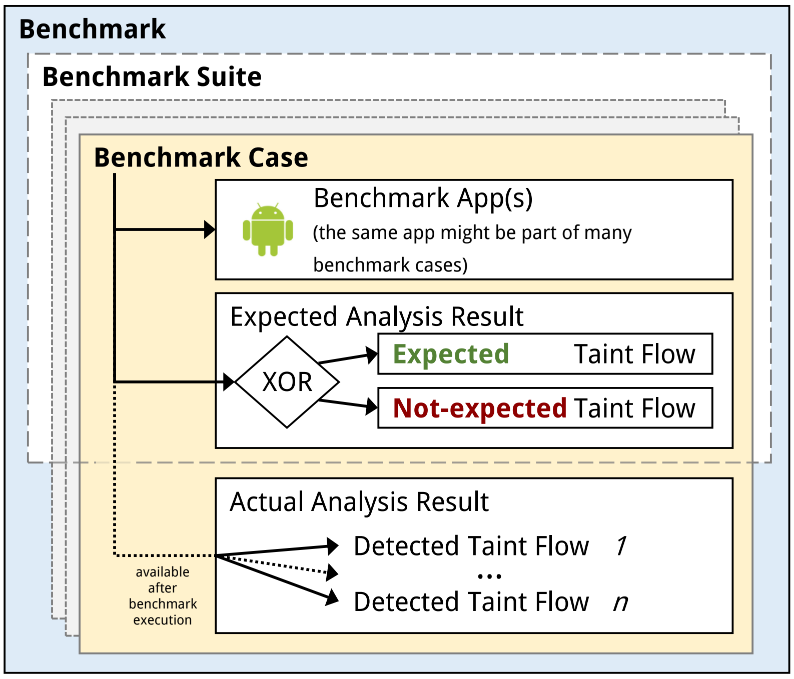


Figure 4: Overview of Benchmark Terminologies

and DroidBench benchmark suite, respectively. How to quantify a benchmark's outcome is explained in the next subsection. Subsection 2.3.2 defines what a benchmark's ground truth is.

2.3.1 Evaluation Metrics

The metrics that are typically used to describe the outcome of a benchmark are the empirical experimentation metrics: precision, recall and F-measure. To compute these, we first have to categorize the results per benchmark case as *True Positive (TP)*, *False Positive (FP)*, *False Negative (FN)* or *True Negative (TN)*. For example, if the actual result – computed for a benchmark case with an expected taint flow – holds a detected taint flow that matches the expected one, this benchmark case is counted as a true positive. Table 1 generalizes what has been explained for the example. The columns show what

Table 1: Comparing an Expected Analysis Result (Columns) with an Actual One (Rows)

		Expected Analysis Result	
		expected taint flow...	not-expected taint flow...
Actual Analysis Result	...does *match a* detected taint flow	**True Positive (TP)**	**False Positive (FP)**
	...does ***not*** *match any* detected taint flow	**False Negative (FN)**	**True Negative (TN)**

kind of taint flow the expected result holds (expected or not-expected). The rows define if a matching detected taint flow is available in the actual result or not. The cells then show the categorization (TP/FP/FN/TN). Once the categorization of all benchmark cases is completed, the accuracy metrics can be computed to summarize the outcome of the respective benchmark. Therefore, we count how many benchmark cases belong to each of these four categories ($|TP|, |FP|, |FN|, |TN|$) and calculate precision, recall and F-measure as follows:

Precision (P):

$$P = \frac{|TP|}{|TP| + |FP|}$$

Recall (R):

$$R = \frac{|TP|}{|FN| + |TP|}$$

F-measure (F):

$$F = 2 \cdot \frac{P \cdot R}{P + R}$$

Precision reflects how often a detected flow matched an expected one in relation to how often it matched a not-expected one. Recall relates to how often expected results were matched or not. Intuitively, precision describes how many of the detected flows were expected and recall describes how many expected flows were detected. The F-measure represents the harmonic mean of precision and recall, hence, can be interpreted as the *summarizing* accuracy metric. The perfect desirable value for all three metrics is 1.0 (100%): All expected flows but no not-expected flow were detected.

There are many other metrics to describe the outcome of a benchmark, for example, the *success rate* (often referred to as accuracy itself):

Success Rate (S):

$$S = \frac{|TP| + |TN|}{|TP| + |TN| + |FP| + |FN|} = \frac{|\text{Successful cases}|}{|\text{All benchmark cases}|}$$

In the context of this thesis, accuracy will always be measured with precision, recall and F-measure. This is the de facto standard which offers the best comparability to values reported in previous works [33, 35, 37, 42, 45, 58]. The success rate will only be used if precision, recall and F-measure are incomputable. This may be the case if the ground truth (see next subsection) of a benchmark suite is not available.

2.3.2 Ground Truths

The most crucial part of a benchmark suite is normally its ground truth.

> A *complete* list of *all* expected taint flows forms the **ground truth** of a benchmark suite.

If a ground truth is available, not-expected taint flows must not be defined explicitly in benchmark cases. The implicit definition then is: any taint flow that is not part of the ground truth is a not-expected taint flow.

Unfortunately, a ground truth is often missing. It is sometimes claimed that a benchmark comes with a ground truth but actually, the provided ground truth is incomplete.

> A list of expected (and not-expected) taint flows is called an ***incomplete* ground truth** unless it is proven that one part of it (expected or not-expected taint flows) is complete – then it would be a (complete) ground truth.

The expected taint flows contained in an incomplete ground truth are often manually or experimentally determined, thus, there are no guarantees that no taint flows, which should be expected, were missed (under-approximation). It may also be the case that taint flows are falsely declared as expected ones (over-approximation). When using an incomplete ground truth, taint flows that are not categorizable may be detected. By explicitly defining not-expected taint flows in benchmark cases, we can categorize true negatives and false positives although our list of expected taint flows might not be complete. Figure 5 depicts how to compute precision and recall when using an (in-)complete ground truth. Whenever an undefined (see "?" in Figure 5) taint flow is detected, it should be added to the incomplete ground truth, if it can clearly be identified as expected or not-expected taint flow. Ultimately and most importantly:

> Benchmark results are and will remain **comparable** as long as they are computed for the *same* (incomplete) ground truth.

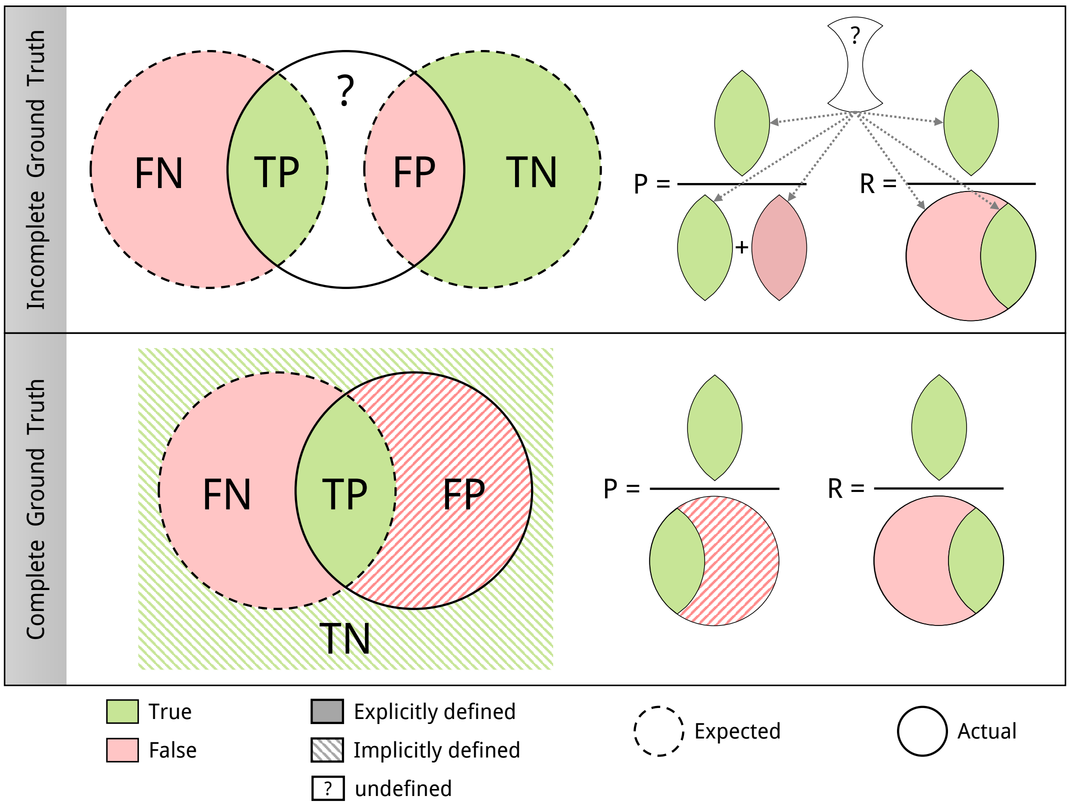


Figure 5: Precision and Recall for Complete and Incomplete Ground Truths

Table 2: Overview of Ground Truth Formats

Format	**Granularity**	**Machine readable**	**Tool in-dependent**	**Description**
Numeric	coarse	✔	✔	The ground truth only denotes how many taint flows are expected (and not-expected) to be found.
Textual	intermediate	✘	✔	The taint flows expected (and not-expected) are described in natural language.
Categorical	intermediate	✔	✔	The taint flows are only described by categories, for example, `IMEI to SMS`.
Intermediate Representation	precise	✔	✘	The IR is used to describe the taint flow – the connected source and sink statements can unambiguously be identified through this description.

To guarantee that a benchmark is reproducible the ground truth must furthermore be unambiguous. Hence, its format and granularity is as decisive as its availability. Table 2 shows frequently used formats. When a ground truth is too coarse, the categorization of results may be hindered, for example, the information that one taint flow is expected (numeric format) does not allow to decide if the intended taint flow was detected or another one. If a ground truth is not machine-readable, the automatic evaluation of the associated benchmark often renders impractical. For instance, converting natural language into a machine-readable format usually requires additional inputs to avoid discrepancies. In addition, natural language descriptions can be interpreted differently by different individuals which is another source of uncertainty. If a ground truth is only compatible with a certain set of tools – using the same IR for instance – it is hardly applicable for tool comparison. In the end:

> **Reproducible** benchmarks, that can be executed and evaluated automatically, require a precise, machine-readable and tool independent ground truth.

Hence, to achieve reproducibility, combinations of different ground truth formats are regularly used.

2.3.3 Benchmark Types

There are different kinds of benchmarks for Android taint analysis. In this subsection, three types are briefly introduced (micro, generated/injected and real-world benchmarks). The way they are created is what makes them different.

Most commonly used for the evaluation of novel approaches are *micro benchmarks*. Small and manually created apps, that have only been created for evaluation purposes, form their corpus. Usually, a single micro benchmark app comprises only one property (a taint flow that exploits one specific feature). Because of this, it is easy to create the ground truth for such benchmark suites.

Generated benchmarks or *injected benchmarks* partially share this advantage. The property that is generated/injected into the benchmark app, is known and can easily be included in the benchmark's ground truth. In order to create such a benchmark suite, the benchmark apps are either automatically generated from scratch, whereby known taint flows are incorporated, or they are created by injecting a property (a taint flow with known features) into an existing app. Especially in the latter case, it is likely that the ground truth of the benchmark suite is incomplete, since apart from the injected taint flows, no information is given about other taint flows.

Real-world benchmarks represent the most-valuable type of benchmark suites. They consist of labeled real-world apps. The benchmark apps are taken from e.g. app markets such as Google's Play Store and the ground truth describes taint flows inside these apps. However, creating the ground truth for a complex real-world app is a challenging task. Finding all taint flows inside one app and proving that there are no others is often impossible. Summarizing, a tradeoff between representativeness and ground truth completeness exists. Intuitively, the more a benchmark suite reflects the real world, the harder it gets to create its ground truth.

2.3.4 Running Example 1: The Benchmark Case (Part 3/5)

Let us consider the running example again. It holds one taint flow from `getDeviceId()` to `sendTextMessage(...)` that should be found by an analysis. The statement calling the logging function `d` (see Line 9 in Listing 3 on Page 22) can also be interpreted as a sink.

Hence, a taint flow that could wrongly be detected could also start at `getDeviceId()` but end in `Log.d(...)`.

Table 3: Complete Ground Truth of a Benchmark Suite only Consisting of the Running Example

	Expected Analysis Results (Taint Flows)		**Categorization**
Benchmark App	**Source**	**Sink**	(expected or not-expected)
`example1.apk`	Statement: `getDeviceId()` Method: `onCreate(...)` Class: `MainActivity` App: `Example1`	Statement: `sendTextMessage()` Method: `onCreate(...)` Class: `TargetActivity` App: `Example1`	expected ✔

To form a benchmark suite with a complete ground truth we only need to specify one benchmark case – see Table 3. The not-expected taint flow from `getDeviceId()` to `Log.d(...)` is implicitly defined since it is a complete ground truth and any taint flow that is detected but not declared in the expected analysis result is categorized as not-expected. Issues that may arise, manifest in partial flows. For example, the intra-component flow from `getDeviceId()` to `startActivity(...)` may be detected and reported in addition to the complete flow. Consequently, it would get classified as false positive since it is not declared in the expected analysis result. The correctness of this interpretation is questionable.

Table 4: Incomplete Ground Truth of a Benchmark Suite only Consisting of the Running Example

	Expected Analysis Results (Taint Flows)		**Categorization**
Benchmark App	**Source**	**Sink**	(expected or not-expected)
`example1.apk`	Statement: `getDeviceId()` ...	Statement: `sendTextMessage()` ...	expected ✔
`example1.apk`	Statement: `getDeviceId()` Method: `onCreate(...)` Class: `MainActivity` App: `Example1`	Statement: `Log.d()` Method: `onCreate(...)` Class: `TargetActivity` App: `Example1`	not-expected ✘

An incomplete ground truth would require to explicitly define the not-expected taint flow, too – see Table 4. Now the partial flow is neither implicitly nor explicitly declared and would possibly be ignored during evaluation.

In either case (complete or incomplete ground truth), the partial flow must be handled separately and with respect to the terms of the individual benchmark.

2.4 Analysis Frameworks, Tools and Benchmark Suites

In the last part of the background chapter, existing frameworks, tools and benchmark suites which regularly appear in the context of Android taint analysis are presented. The frameworks, tools and benchmark suites that have been developed for a part of this thesis are summarized and listed in Appendix A.2.

2.4.1 Frameworks

The optimization and analysis framework **Soot** [11, 179] as well as the T. J. Watson libraries for analysis (**WALA**) [189] are frequently used to build Java or Android analyses. While WALA is *not* specifically tailored to Android, Soot comes with built-in Android support. Consequently, most Android taint analysis tools are based on Soot. Soot uses its own intermediate representation called *Jimple* – “like Java but simple”. It is a three-address-code language and as the name suggests it is an equivalent to Java. Thus, all program constructs that can be implemented in Java can be reproduced in Jimple. To do so it only needs 15 different types of statements [22]. For comparison: Java bytecode uses more than 200 distinct instructions [152].[7] Accordingly, in order to implement an analysis at most these 15 statement types must be taken into account. Furthermore, method calls are never nested in Jimple and Jimple always references variables explicitly by using local variables only. When a (static) field is accessed, for example, it is assigned to a local variable first. These are some of the reasons why Soot and Jimple are used as basis for most Android taint analysis tools, which are presented next. Of course, frameworks targeting other programming languages and objectives while implementing different techniques exist as well [28, 77].

2.4.2 Tools

When it comes to taint analysis tools for Android, two tools can be identified as state-of-the-art [67, 68]: **Amandroid** [37, 98] and **FlowDroid** [33, 133]. Both tools are steadily improved and new versions have been released recently. Another frequently mentioned taint analysis tool is **DroidSafe** [42, 131], but in contrast its development has come to an end in 2016. While FlowDroid and DroidSafe are based on Soot, Amandroid is based on its own analysis framework. Depended on their basis, the tools use different IRs but all three employ similar graph representations for their analysis. Nowadays, all three claim to support the analysis of ICC.[8] Furthermore, it is claimed that Amandroid and FlowDroid are context-, flow-, field- and object-sensitive and that they support callbacks, especially life-cycles. Another similarity of these two is the usage of a configurable *list of sources and sinks*. Such a list is used to identify instances of sources and sinks inside an app. Various tools have been proposed to create such lists, since a perfect static list does not seem to exist. One often named tool is the machine learning tool **SuSi** [36]. When referring to the *SuSi list*, the sources and sinks list created by SuSi is meant.

While FlowDroid is a standalone analysis tool, it is also known as `soot-infoflow-android` library, which is the extension of Soot that makes Soot usable in Android context. As a result, any Soot-based Android analysis tool is also an extension of FlowDroid. One prominent extension is the inter-component analysis tool **IccTA** [45] which has recently been integrated into FlowDroid. Although it is integrated, we will continue using the name IccTA whenever this extension is meant. IccTA takes an ICC model as additional input and forwards it to FlowDroid so that FlowDroid is able to analyze ICC. For the computation of the ICC model **Epicc** [32] was used originally. However, Epicc was quickly replaced after the release of IccTA by its successor **IC3** [46, 147]. Effectively, both tools (Epicc and IC3) gather information about intents and intent filters. Recently, another tool (**ICCBot** [97]) tackling the same task has been proposed. ICCBot seems to be a promising candidate as it allows to gather more accurate and more comprehensive information about ICC. It partially employs dynamic analysis to determine intent triple elements more accurately. It provides information about intent sinks (and intents), intent sources (and intent filters) and also inter-component flows between those. However, since a different ICC model (output format) is used, ICCBot cannot be used directly together with IccTA. **DidFail** [35, 127] is another taint analysis tool with ICC

[7] In Appendix A.1 we compare three code formats: Java source code, Jimple code (IR) and bytecode.
[8] Since version 2.8 (2020) FlowDroid supports ICC, however, for that it requires one additional input.

capabilities. It also combines Epicc with FlowDroid, however, it does so internally and therefore does not require an ICC model as additional input.

The Precise Intent Matcher (**PIM** [75, 171]) is a dynamic tool which can also be used to find inter-component flows. On an Android (virtual) device, e.g. an emulator, it runs an app that functions as a server. The analysis tool PIM functions as the client. Once this client faces the task to decide if an intent matches an intent filter, it asks the server to compare the associated intent triples. The server instantiates both intent triples and asks the Android system it is deployed in, whether the given intent triples match. If so, it outputs a positive answer that states that a match has been found. When exact statements (intent sink and intent source) are provided as input, it replies an inter-component flow by connecting the respective statements. For example, if an intent sink is embodied by a `startActivity` statement and an intent source by a `getStringExtra` statement, these two are connected if the associated intent triples match.

For instance, IccTA's analysis can further be lifted up, from inter-component to inter-app level, by employing **ApkCombiner** [44, 109]. ApkCombiner, as the name suggests, combines various Android apps into a single one, thus, any tool able to analyze inter-component scenarios can analyze inter-app scenarios by analyzing such combined apps. The Android Merge Tool (**AMT** [76, 102]) was also developed to combine or merge apps. Since it has been published more recently (2019), it may be seen as the successor of ApkCombiner which has not been updated since 2014. **DIALDroid** [58, 126] represents another extension of FlowDroid that can be employed to analyze inter-app scenarios without requiring ApkCombiner or AMT.

Almost all tools mentioned above, especially all the taint analysis tools, will be employed in the experiments conducted for this thesis (see Section 4.2 and Chapter 5). These taint analysis tools are considered to be close contenders as they all implement a flow- or context-sensitive static taint analysis. Nonetheless, there are many other related taint analysis tools that have not been discussed yet. A general overview of analysis tools for Android apps can be found in three surveys [55, 60, 64]. These surveys collect and summarize tools and their functionality as outlined in research papers.

In the following, a few static taint analysis tools are mentioned along with an argumentation that explains why they were not considered to be used in experiments. **Apposcopy** [34], **WeChecker** [40], **Separ** [49] and the recently proposed tool **SparseDroid** [74] should fit perfectly, however, they are not publicly available. **SCanDroid** [25] is publicly available and fits into our scope as well, nonetheless, it is largely outdated and cannot produce results for any (micro) benchmark suite that we use for our experiments. **DroidInfer** [43] employs an interesting type-based approach. However, the tool seemed not ready for competitive comparison since its execution fails for most (micro) benchmark cases. **HornDroid** [51, 146], as the name suggests, uses Horn clauses to decide which sinks are able to leak sensitive data. Due to the use of logic, it is able to distinguish definite from only potential leaks. Unfortunately, it does not provide information about sources that are connected to truly leaking sinks, hence, it cannot be used to detect taint flows directly.

In addition to these static tools, dynamic tools have also been developed for years. Enck et al. became the most successful pioneers in this area by developing **TaintDroid** [26] in 2010. However, due to the introduction of ART (Android Runtime – replacement of the Dalvik VM released in 2014 [163]) many dynamic tools, including TaintDroid, cannot be used anymore. A more recent approach that allows the deployment of a dynamic taint analysis for ART is called **TaintMan** [83]. As in the field of static tools, there are dynamic tools tailored to certain challenges, for example, **IacDroid** [56] embodies a dynamic tool that focuses on IAC scenarios. Other dynamic tools implement atypical taint analysis techniques. **MutaFlow** [62], for instance, is a mutation-based taint analysis tool. It mutates the sensitive information extracted at sources and checks whether the mutated value reaches out to any sinks. Since any dynamic approach requires guidance (e.g. execution traces or test cases) to run its analysis, we do not consider them as being contenders in the same league as static tools. Hence, dynamic taint analysis tools are not included in our experiments.

StubDroid [47], another extension of Soot (also known as `soot-infoflow-summaries`), can be used to create summaries of methods. Such summaries can be used in an analysis to avoid unnecessary analysis repetitions for the same method. StubDroid gathers information about methods such as which input parameters actually influence e.g. the method's output. Using StubDroid can therefore potentially improve the efficiency and scalability of an analysis, in particular, if used to summarize methods of the Android library.

Another tool dealing with libraries, especially compatibility or support libraries, is **APK-Simplifier** [92, 108]. It takes an app as input and outputs a list of *trusted* support libraries. These are trusted, since they are equal to libraries that can be downloaded from an official Android repository. To decide whether libraries are equal, APK-Simplifier employs a clone detection tool [14]: For each class of an app, for which a counterpart (a class of an official library) exists, it is checked whether they are clones. If classes are clones with a certainty of a configurable threshold they are added to the output list. Thereby, APK-Simplifier will never trust a manipulated library class. Please note that there are many alternatives to APK-Simplifier that also deal with the well-explored field of library detection in Android context [48, 59, 61, 69, 88]. While most of these allow to detect support libraries, a few can be used to detect third-party libraries as well. Nonetheless, APK-Simplifier is the first designed to be used in cooperative analysis context.

The name of the next Soot-based tool, **PermissionFinder** [170], directly describes what it can be used for. By iterating over all Jimple statements that appear in an app's IR, it identifies those statements that require a permission in order to be executed successfully. To do so, it checks whether a statement is denoted as a key in a mapping between generic Jimple statements and permissions [106]. This mapping has been created by mining the official Android framework documentation [95].

One kind of tools that heavily rely on analyses are slicers [3, 6]. *Slicers slice* an app from or to a certain *slicing criterion*. Whenever they slice from one criterion to another, the process is called *chopping* [7]. Various approaches [8, 13, 20] exist and most are tailored to a specific programming language and individual features. In the context of Android apps, only a few approaches exist that function nowadays: the dynamic slicers **AndroidSlicer** [72] and **Mandoline** [84] and the static slicer **Jicer** [87, 159]. Similar to analysis tools, dynamic slicers execute the app to find the slice, whereas static slicers analyze its source code (mostly in form of dependence graphs) to slice the app. All three tools partially use Jimple, e.g. as an output format, but differ in terms of precision and feature support.

In the evaluation section (see Section 5.1) we will only use **Jicer**, thus, we describe it slightly more detailed here. Jicer is a static, inter-procedural slicer for Jimple that is flow-, context-, field-, object- and thread-sensitive and also takes callbacks and life-cycles into account. Static slicers typically operate on *program* or *system dependence graphs (PDGs/SDGs)* depending on whether the slicer is an intra- or inter-procedural slicer. Jicer operates on so-called *app dependence graphs (ADGs)* [87], i.e. on an extended version of an SDG. For example, an ADG holds nodes and edges that model the life-cycle of Android activity components. A unique feature of JICER is that it allows the slice to be output as an `.apk` file. The slice is then either cut out of the app or the only part remaining in it. Consequently, any analysis tool able to analyze `.apk` files is able to analyze slices created by JICER. From an conceptual perspective JICER can be seen as a reducer that removes program parts which are irrelevant for an analysis [66]. To properly scale, JICER uses STUBDROID, i.e. instead of slicing through library methods it attempts to use summaries created before by STUBDROID.

The tool called **DROIDRA** [54, 130] also uses `.apk` files as input and output, however it fulfills a completely different purpose, i.e. it can be used to resolve reflection by instrumentation. More precisely, it searches for statements that use reflection and attempts to resolve the reflection by replacing these statements with equivalent, non-reflective statements. For example, if a method `foo` is invoked reflectively (`getClass().getMethod("foo", ...).invoke(...)`), this invoke is replaced by a direct method call (`this.foo()`).

A tool that deals with native code, another analysis challenge given by the Java programming language, is **NOAH** [75, 168]. NOAH allows detecting sources and sinks in native code. To do so, it checks whether source or sink statements declared in the SUSI list appear in an app's native code portion. If so, it identifies the native call as source or sink respectively and reports a connection between the native call and this particular source or sink.[9] DROIDRA and NOAH are also based on SOOT.

Next, we will have a look at benchmark suites, but beforehand note that there also are benchmark generators like **LAVA** [53] or **GENBENCHDROID** [94]. LAVA produces benchmark cases by bug-injection and hence knows exactly when to expect which bug to occur. Accordingly, the ground truth LAVA creates for its benchmark apps is potentially incomplete. GENBENCHDROID implements the concept of benchmark fuzzing and generates benchmark cases for Android taint analysis from scratch, hence, it may create complete ground truths (see Section 2.3.2).

2.4.3 Benchmark Suites

There exist various benchmark strategies and suites available for many kinds of software and hardware. In the following, benchmark suites designed for taint analysis (mostly Android taint analysis) are presented.

DROIDBENCH [33, 128] is the most used benchmark suite for the evaluation of Android taint analysis tools. It is a micro benchmark that comprises overall 190 benchmark apps associated with 18 different categories and comes with a ground truth which is complete for most of the benchmark cases comprised. The ground truth is documented in form of textual source code comments which are only human-readable apart from the number documenting how many flows are detectable in the respective benchmark app. Along with the introduction of reproducible and automatic benchmarks (see Chapter 4), we defined its complete ground truth in a machine-readable format. Additionally, we added two new categories to DROIDBENCH: FEATURE-CHECKING and INTENT-MATCHING. The first aims at checking the support of certain features. For example, it holds an iconic benchmark case to test if an analysis is capable of dealing with ICC. More details are presented in Section 4.2. The second consists of three apps only. Still, these apps exploit most of the possible ways to define implicit intents and intent filters. Thus, it can be used to evaluate how accurately an analysis detects which intents match which intent filters (see Section 5.1).

[9]For a similar purpose, two alternative tools (**JUCIFY** [93] and **NATIVEDROID** [70]) have been proposed recently. A comparison of all three approaches has not been conducted yet.

ICC-Bench [37, 148] is another well-known Android taint analysis micro benchmark. It uses the same style as DroidBench with respect to its ground truth and consists of 24 benchmark apps that focus on inter-component communication.

Securibench [178] is another often-cited micro benchmark, however, it is not tailored to Android – it holds Java classes instead of Android apps. Still, it is very similar to DroidBench – many benchmark cases show commonalities, i.e. exploit the same features in the same way.

DIALDroid-Bench [58, 125] is a real-world app set that comes without any ground truth. However, the set of 30 benchmark apps is fixed. This allows us to evaluate different tools on DIALDroid-Bench and to compare the results. Without further information this is not the case for e.g. the 50 best rated apps on the Google Play Store, as this set can be different (different apps or versions of the same) depending on when and where it is downloaded.

TaintBench [91, 186] is another real-world (malware) benchmark that comes with an incomplete ground truth which finally allows to more accurately evaluate Android taint analysis tools on 39 real-world apps. TaintBench is one of the benchmark suites employed in our experiments (see Section 4.2 and Chapter 5).

Another real-world benchmark similar to TaintBench was recently proposed. It consists of 18 benchmark apps that have been taken from the **FossDroid** collection [141], hence, we refer to this suite as FossDroid benchmark [86]. Its incomplete ground truth holds 756 manually classified taint flows (693 expected and 63 not-expected benchmark cases).

The **SV-Comp** [185] benchmark represents a large collection of C and Java programs for which a ground truth is available. Taint flows are not encoded in its ground truth, instead mostly safety-realted properties are encoded (e.g. whether a certain (error-)location is reachable). The benchmark is annually extended and used to run the *Software Verification Competition (SV-Comp)* [31]. In this competition various software verifiers compete against each other in order to find out which verifier is the most effective and efficient one. More precisely, who is able to successfully verify how many programs of the SV-Comp benchmark, and how much resources (e.g. time and memory) does it require to do so. We want to emphasize that competitions like SV-Comp often boost the progress in a particular area and allow that progress to be measured. However, such competitions do neither exist in the area of Android app analysis nor in the area of taint analysis.

2.4.4 Running Example 1: Tool Results (Part 4/5)

As described in the second and third part of the running example, one taint flow should be found when analyzing it. In this last part we exemplify how it can be found by employing two of the tools described above.

First, we are running FlowDroid. The result determined by it is shown in Listing 5[10] (see Page 40). It comes in XML format and each taint flow is encoded in one `<Result>` tag. To represent a taint flow, the `<source>` and `<sink>` it connects is identified by naming the statement, its line number and the method where the statement was found. Statement and method are named in Jimple format. When using Jimple, the class to which a method belongs is implicitly named as well – the method's description starts with the respective fully-qualified class name. To summarize, the result holds two taint flows, although we are just expecting one:

- from `getDeviceId()` (Line 23) to `startActivity(...)` (Line 15), and

[10]The results have been shortened to foster readability.

- from `getStringExtra(...)` (Line 9) to `sendTextMessage(...)` (Line 5).

These two taint flows exactly match the two intra-component partial flows as described in Part 2 (see Subsection 2.2.4). To complete the finding, the *inter*-component flow that connects the two *intra*-component flows must be detected.

Hence, as a second tool we run IC3. Its result can be found in Listing 6[10] (see Page 41). In this case, the Protocol Buffer [174] format is used. The two components of the running example are described separately (see `components` tags in Line 1 and Line 25). The description of `MainActivity` contains one exit point (see `exit_points` tag in Line 30). This exit point refers to a statement that probably provides information to another component. Again, Jimple is used to name the statement, method and class that embody the exit point. For the `TargetActivity` component one extra is described inside the `extras` tag (see Line 5). An extra represents data attached to an intent. The extra is also linked to a Jimple statement (see Lines 8–10). Both, the description of the exit point and the extra, additionally hold information about the intent or intent filter associated with them. To sum up, IC3 does not output taint flows, it only identifies statements that realize ICC and provides information about the intent triples involved. With respect to the current example, comparing the action described for the exit point and for the extra (see Lines 40–41 and Lines 16–17) is sufficient to conclude that the information leaving one component at the exit point may reach the second component. Note that the category `android.intent.category.DEFAULT` (Lines 20–21) is not given for the exit point, nonetheless, the described intent and intent filter match due to Android's matching algorithm. We can also see that the extra uses the information attached to the intent of the exit point by comparing the name of the extra (`data`) that is *sent* and *received* (see Lines 44–45 and Line 6). Finally, we can conclude that IC3's result implicitly represents the missing inter-component flow:

- from `startActivity(...)` to `getStringExtra(...)`.

We have two options to get the complete inter-component taint flow: the result of IC3 can be given to FlowDroid as an ICC model to trigger its IccTA extension, or the two results can be combined by concatenating the partial flows. Option two would require a third tool that can interpret and combine both results. Either way the taint flow from `getDeviceId()` to `sendTextMessage(...)` can be found.

```
<?xml version="1.0" encoding="UTF-8"?>
<DataFlowResults FileFormatVersion="102" TerminationState="Success">
    <Results>
        <Result>
            <Sink Statement="virtualinvoke
                → $r4.&lt;android.telephony.SmsManager: void
                → sendTextMessage(...)&gt;(...)" LineNumber="21"
                → Method="&lt;de.foellix.aql.thesis.examples.example1.
                → TargetActivity: void onCreate(...)&gt;">
                <AccessPath Value="$r3" Type="java.lang.String"
                    → TaintSubFields="true"/>
            </Sink>
            <Sources>
                <Source Statement="$r3 = virtualinvoke
                    → $r2.&lt;android.content.Intent: java.lang.String
                    → getStringExtra(...)&gt;(...)" LineNumber="16"
                    → Method="&lt;de.foellix.aql.thesis.examples.example1.
                    → TargetActivity: void onCreate(...)&gt;">
                    <AccessPath Value="$r3" Type="java.lang.String"
                        → TaintSubFields="true"/>
                </Source>
            </Sources>
        </Result>
        <Result>
            <Sink Statement="virtualinvoke
                → r0.&lt;de.foellix.aql.thesis.examples.example1.MainActivity:
                → void startActivity(...)&gt;(...)" LineNumber="25"
                → Method="&lt;de.foellix.aql.thesis.examples.example1.
                → MainActivity: void onCreate(...)&gt;">
                <AccessPath Value="r2" Type="android.content.Intent"
                    → TaintSubFields="true">
                    <Fields>
                        <Field Value="&lt;android.content.Intent:
                            → java.lang.Object[] extraValues&gt;"
                            → Type="java.lang.Object[]"/>
                    </Fields>
                </AccessPath>
            </Sink>
            <Sources>
                <Source Statement="$r5 = virtualinvoke
                    → r4.&lt;android.telephony.TelephonyManager:
                    → java.lang.String getDeviceId()&gt;()" LineNumber="20"
                    → Method="&lt;de.foellix.aql.thesis.examples.example1.
                    → MainActivity: void onCreate(...)&gt;">
                    <AccessPath Value="$r5" Type="java.lang.String"
                        → TaintSubFields="true"/>
                </Source>
            </Sources>
        </Result>
    </Results>
    <PerformanceData>...</PerformanceData>
</DataFlowResults>
```

Listing 5: Result of FlowDroid (Example 1)

```
components {
  name: "de.foellix.aql.thesis.examples.example1.TargetActivity"
  kind: ACTIVITY
  exported: true
  extras {
    extra: "data"
    instruction {
      statement: "$r3 = virtualinvoke $r2.<android.content.Intent:
          → java.lang.String getStringExtra(java.lang.String)>(\"data\")"
      class_name: "de.foellix.aql.thesis.examples.example1.TargetActivity"
      method: "<de.foellix.aql.thesis.examples.example1.TargetActivity: void
          → onCreate(android.os.Bundle)>"
      id: 5
    }
  }
  intent_filters {
    attributes {
      kind: ACTION
      value: "de.foellix.aql.thesis.examples.TARGET"
    }
    attributes {
      kind: CATEGORY
      value: "android.intent.category.DEFAULT"
    }
  }
}
components {
  name: "de.foellix.aql.thesis.examples.example1.MainActivity"
  kind: ACTIVITY
  exported: true
  intent_filters {... }
  exit_points {
    instruction {
      statement: "virtualinvoke
          → $r0.<de.foellix.aql.thesis.examples.example1.MainActivity: void
          → startActivity(android.content.Intent)>($r2)"
      class_name: "de.foellix.aql.thesis.examples.example1.MainActivity"
      method: "<de.foellix.aql.thesis.examples.example1.MainActivity: void
          → onCreate(android.os.Bundle)>"
      id: 10
    }
    kind: ACTIVITY
    intents {
      attributes {
        kind: ACTION
        value: "de.foellix.aql.thesis.examples.TARGET"
      }
      attributes {
        kind: EXTRA
        value: "data"
      }
    }
  }
}
```

Listing 6: Result of IC3 (Example 1)

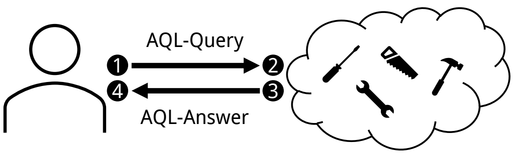
3 Cooperative Analysis

A cooperative analysis makes use of different analyses that have different objectives and different strengths or weaknesses. The principle is always the same: final (or intermediate) results of various analyses are combined or used to enhance another analysis. To design a cooperative analysis approach that follows the principles of component-based software engineering, we must find an interface that allows the user to interact with arbitrary analysis tools and to combine analysis tool results. So to say, users and tools have to speak the same language. Hence, as a first step we must find this one language that every participant understands.

In the area of databases and information systems *query languages* are used in a similar context. By asking a query, information is retrieved from a database. Databases typically hold information in table-like data structures. Thus, to retrieve information we have to determine which table cells we want to retrieve. Usually, the columns represent categories of data and the rows stand for entries. After identifying the columns and rows of interest, the data can be retrieved or, for example, be sorted, filtered, or combined with other data. In case of the language we are looking for, we are not issuing queries to a database but to a set of analysis tools. Different analysis tools support to retrieve different information – similar to the different columns of a table in a database. Every result of an analysis tool represents an entry (a row in the table). As the analogies suggest the data retrieved should also be, for instance, filterable and combinable. Consequently, we are looking for a query language to embody our approach.

The requirements this language must and should fulfill are described in Section 3.1. To derive these requirements, we are taking a closer look at related work in terms of analysis tools, query languages, result formats and existing cooperative approaches. In particular, the inspection of query languages will show that there is no language available that suits our purpose. In consequence, we present a novel query language that forms the centerpiece of our cooperative analysis approach. This language is called Android App Analysis Query Language (AQL) and consists of two parts – AQL-Queries and AQL-Answers:

❶ users compose queries,
❷ analysis tools accept queries and
❸ produce answers
❹ that are returned to the users.



In this context, we first present the syntax and semantics of AQL-Queries and the structure of AQL-Answers (see Section 3.2). In Section 3.3 we present (cooperative analysis) strategies and show that they can be realized by means of the AQL. The semantics and strategies are implemented in the corresponding AQL-System (see Section 3.4) which is accessible from anywhere via AQL-WebServices (see Subsection 3.4.5).

3.1 Hard & Soft Requirements (Related Work)

The requirements presented in this section must be fulfilled or implemented by the AQL and its implementation. All requirements will be derived from related work (analysis tools, query languages, result formats and existing cooperative approaches). On the one hand, *hard requirements* (▶) must be fulfilled, because they are technically required and the approach's functionality is bound to them. On the other hand, the *soft requirements*

(▷) must not necessarily be fulfilled, but related approaches and own experiences indicate their usefulness. We start by deriving hard and soft requirements from the analysis tools described as a part of the previous chapter (see Section 2.4). At the end of this section, all derived requirements are summarized (see Subsection 3.1.4).

3.1.1 Analysis Tools

Every analysis tool is accessible via command line. Only a few tools have an optional GUI in addition. The commands to execute different tools are as different as the tools themselves. Every tool comes with various command line parameters and other configuration options that allow an expert to fine-tune the tool for its designated purpose. To foster readability we are not examining the commands and options themselves; instead, a summary of these can be found in Table 5 on Page 45. Each row in the table represents one tool. The first two columns identify each tool by its name and version.

The column entitled with "Java" refers to the Java version the respective tool requires – typically older tools require Java 8 while more recently released tools are ready to be used with Java 17.[11] Not all tools require only Java to be executed. For example, DidFail partially uses Python scripts to carry out its analysis. DroidSafe and IccTA (old version from 2016) use bash scripts and make files which are designed to be executable only on Unix operating systems. Dynamic tools such as PIM require running Android emulators and consequently come with certain hardware requirements. Accordingly, we infer our first requirement:

▶ **Req. 1:** Tools must be accessible in different execution environments.

In the "Inputs" column all inputs are listed that are necessary to run the respective tool. For example, almost every tool in this set requires the app to be provided as an `.apk` file (see "APK" in the table). Column "Extra" lists information that is additionally required to e.g. automatically identify the result of an analysis. For instance, IC3 computes an ICC model and the file representing it is named by the package name provided in the manifest of the given app (`.apk` file). The "Output Format" column describes the structure of the results produced by each tool. The following "Jimple" column contains a check mark, if the results use the Jimple IR to reference statements, methods or classes. Which information are given in a tool's output is coarsely described by keywords in the column entitled with "Outputs". Finally, the last column indicates the availability of each tool and provides a commit id whenever an available and publicly accessible repository is given as reference.

Not summarized in the table are configuration options given by the execution environment. For each tool, we can control how much memory it shall use at maximum by using built-in tool mechanics or by setting up the Java virtual machine that is executing them. Thus, for each tool we get an additional input, in the following described as "Memory". Some tools furthermore delete or overwrite previous results when they are executed again, i.e. concurrently executing multiple instances of such tools will lead to data races. The related additional input is named "Instances".

[11] In the context of this thesis Java$\leq$8 and Java>8 are used as synonyms for Java 8 and 17.

Table 5: Tool Properties

Name	Version / Date	Java	Inputs	Extra	Output Format	Uses Jimple	Output	Availability (Commit)
Amandroid*	3.1.2	>8	APK	APK-Filename	Textual (Custom)	✘	Taint Flows, Intent Filters, Intents[4]	✔ (65aec77) [99]
Amandroid	3.2.0	>8	APK	APK-Filename	Textual (Custom)	✘	Taint Flows	✔ (415ad9f) [100]
Amandroid	3.2.1	>8	APK	APK-Filename	Textual (Custom)	✘	Taint Flows	✔ (06596c6) [101]
FlowDroid*	April 2017 (Nightly)	≤8	APK, APD[1]	-	Textual (Custom)	✔	Taint Flows	✘ [137]
FlowDroid	2.7.1	>8	APK, Sources&Sinks, APD[1]	-	Structured (XML)	✔	Taint Flows	✔ (72734bd) [135]
FlowDroid	2.9.0	>8	APK, Sources&Sinks, APD[1]	-	Structured (XML)	✔	Taint Flows	✔ (e17e615) [136]
FlowDroid	2.10.0	>8	APK, Sources&Sinks, APD[1]	-	Structured (XML)	✔	Taint Flows	✔ (0174ec4) [134]
DIALDroid	September 2017	≤8	APKs, APD[1], Database Information	-	Database Entries (SQL)	✔	Taint Flows	✔ (5df5734) [126]
DidFail	March 2015	≤8	APK(s)	-	Textual (Custom)	✔	Taint Flows	✔ [127]
DroidSafe	June 2016 (Final)	≤8	APK	APK-Filename	Textual (Custom)	✔	Taint Flows	✔ (1eab2fc) [131]
IccTA*	February 2016	≤8	APK, APD[1]	APK-Packagename	Textual (Custom)	✔	Taint Flows	✔ (831afaa) [151]
IccTA	2.9.0	>8	APK, Sources&Sinks, APD[1]	APK-Packagename	Structured (XML)	✔	Taint Flows	✔ (e17e615) [150]
IccTA	2.10.0	>8	APK, Sources&Sinks, APD[1]	APK-Packagename	Structured (XML)	✔	Taint Flows	✔ (0174ec4) [149]
NOAH	2.0.1	>8	APK	APK-Filename	Structured (AQL)	✔	Taint Flows, Sources, Sinks	✔ (8726d19) [168]
PIM [d]	2.0.1	>8	AQL-Answer(s)	-	Structured (AQL)	✔	Taint Flows[3]	✔ (e583317) [171]
HornDroid	0.0.1	>8	APK	APK-Filename	Structured (JSON)	✔	Sinks	✔ (cd52ba4) [146]
IC3	0.2.1	≤8	APK, APD[1]	APK-Packagename	Structured (ProtoBuf)	✔	Intents, Intent Filters[4], Intents[4]	✔ (2c4de4b) [147]
APK-Simplifier	2.0.1	>8	APK	-	Textual (Custom)	✘	Class List	✔ (e409c4b) [108]
Jicer	2.0.0	>8	APK, ABT[2], Slicing Criterion/Criteria	-	Binary (APK) / Jimple	✔	Slice	✔ (a596528) [159]
Jicer-ICC	2.0.0	>8	APK, ABT[2], Slicing Criterion/Criteria, ICC Dependence Graph Edges	-	Binary (APK) / Jimple	✔	Slice	✔ (a596528) [159]
AMT	2.0.1	>8	APKs	APK-Filename	Binary (APK)	✔	Preprocessed App	✔ (9a7d6bd) [102]
ApkCombiner	1.0.1	>8	APK, APD[1]	-	Binary (APK)	✘	Preprocessed App	✔ (05c9568) [109]
DroidRA	April 2017	≤8	APK, APD[1]	APK-Filename	Binary (APK)	✔*	Preprocessed App	✔ (b766a32) [130]

*: oldest tool variant, d: dynamic analysis tool, 1: APD = Android Platforms Directory, 2: ABT = Android Build Tools directory, 3: Taint flows that involve ICC/IAC only,
4: Intent filters or intents with additional information about related statements – as described below (cf. “Intent Sources” and “Intent Sinks”).

With all this information given, we start inferring requirements by taking a closer look at the inputs given. First, we divide the inputs (and extras) given into the following three categories:

Universal:	APD, ABT
Tool-specific:	Database Information, Instances, Memory, Sources&Sinks
Execution-specific:	APK (APK-Filename, APK-Packagename), AQL-Answer(s), ICC Dependence Graph Edges, Slicing Criterion/Criteria, Sources&Sinks

Universal inputs are required by various tools and always hold the same value. Tool-specific inputs hold different values depending on the tool they are supplied to. However, universal and tool-specific inputs are the same whenever a certain tool is executed. In contrast, the execution-specific inputs may be different dependent on the given analysis task and target. Note, "Sources&Sinks" can be given on tool- or execution-specific level, i.e. we can identify sources and sinks occurring in a certain scope such as a single app or just use a predefined list which is commonly provided along with the respective tool. Finally, we can infer our first soft and another hard requirement:

▷ **Req. 1:** Universal and tool-specific inputs should be configurable in the AQL-System.

▶ **Req. 2:** AQL-Queries must allow to model all execution-specific inputs.

Universal and tool-specific inputs could also be hard coded. In contrast, hard coding execution-specific inputs would obviously limit us, for example, we could only specify a single analysis target (APK).

Table 5 lists the tools "Jicer" and "Jicer-ICC", both entries in fact refer to the same tool, however, to trigger the ICC capabilities of Jicer an additional execution-specific input is required ("ICC Dependence Graph Edges"). Hence, whether a certain input is given or not, different tool configurations must be considered. Another example is embodied by the sources and sinks input. As stated above, sources and sinks can be provided on different levels. In consequence, different tool configurations are needed. This gives us another requirement:

▶ **Req. 3:** Tools and tool variants must be chosen with respect to the execution-specific inputs given in an AQL-Query.

Next, we take a look at the outputs produced by the different tools. Many tools output analysis information related to specific statements or program locations. As reference the respective class, method and statement is denoted. Additionally, bytecode location identifiers or source code line numbers are often provided.

> A **reference** or **to reference** (a statement) is defined by denoting the statement, method, class and app it belongs to. Optionally, further identifiers (e.g. line numbers) may be provided.

Table 5 contains in summary the following output elements (cf. column "Outputs"):

- Taint Flows (ICC/IAC): Most tools that output taint flows only reference the sources and sinks each taint flow connects. Only a few tools provide additional information about the path(s) between these pairs of sources and sinks.

- Intents (Intent Sinks): Any analysis tool, that outputs information about an intent, denotes the corresponding intent triple (action, category, data). If a related reference is given, for instance a statement that starts another component, we call this combination of intent triple and reference an *intent sink*.[12]

- Intent Filters (Intent Sources): In case of an intent filter the corresponding intent triple is also denoted. Analogously, if a related reference is given, for instance, a statement that extracts information from an intent, we call this combination an *intent source*.[12]
- Sources and Sinks: One reference for each source or sink is sufficient to fully describe it.
- Permissions: At least the name of a permission is denoted whenever a tool outputs a permission. If a reference is attached to such a permission, it references a statement that requires this permission.
- Slices: The only slicer in the given set of tools is JICER (and its ICC variant). It has multiple output options. The output variant, which is considered here, is the following: slices in form of `.apk` files wherein all the code that does not belong to the slice has been removed. In short, slice outputs are provided in form of `.apk` files.
- Preprocessed Apps: Just as slices, preprocessed apps are always given as `.apk` files.

The AQL-SYSTEM which is detailed in Section 3.4 comes with a couple of built-in auxiliary tools (see Subsection 3.4.3). The additional outputs produced by them are described below:

- Arguments: Any kind of textual data may be denoted to represent arguments. For example, the features an app comprises can be denoted in form of a comma-separated list of strings – keywords that unambiguously identify certain features.
- Converted Files: Some auxiliary tools fulfill the purpose of converters. Files of a certain type are converted into files of another type. Two short examples: (1.) a tool's result is converted into a generic result format (e.g. an AQL-Answer); or the other way around (2.) a generic result is converted into a tool specific input format. In the latter case, the output may be a file of any type.

The different outputs can be grouped as follows:

Raw:	Arguments
Files:	Slices, Preprocessed Apps, Converted Files
Analysis Information:	Taint Flows (ICC/IAC), Intents, Intent Filters, Intent Sinks, Intent Sources, Sources, Sinks

In a cooperative analysis one tool uses the analysis information of another or combines the information of multiple, hence, the analysis information that is output must be structured clearly.

▶ **Req. 4:** Any kind of *analysis information* must be encodable in an AQL-Answer.

Without knowing what can possibly be found in an AQL-Answer we would never be able to e.g. define how to combine certain information.

Files may be used whenever the given information cannot reasonably be encoded in a more generic way. For example, there is no generic format that is better to represent an entire app than an `.apk` file.

[12] An explanatory example in regard to intent sinks and intent sources can be found in Appendix A.3.

▶ **Req. 5:** Manually and automatically determined *files* must be referable in AQL-Queries.

Otherwise the output of one tool cannot be the input of another once more than structured analysis information (e.g. a preprocessed `.apk` file) is output.

Raw outputs may be used to influence the analysis task given in form of an AQL-Query. For example, the features to consider or the tool to be used may be influenced by a raw output such that we can pick the best tool with respect to a certain feature or simply by a tool's name. Since we want to be able to manually provide or automatically determine arguments such as features or tool names, we must be able to include any information in raw format directly in AQL-Queries. Hence, we get the following soft requirement:

▷ **Req. 2:** Raw outputs should directly be usable in AQL-Queries.

As an alternative we could mandatorily require a tool name, so this is only a soft requirement.

Lastly, with respect to analysis tools, we derive two more requirements from the tools' output formats. The output formats used by the individual tools are as different as their outputs themselves. As the column entitled with "Output Format" in Table 5 shows, some tools use custom textual formats and others well-known, structured formats like XML, JSON or ProtoBuf. Even though the textual outputs might be easier to read out, the structured ones clearly have other advantages. First, because of the available support (editors, viewers, parsers, ...) for these structured formats, the data contained can be managed comfortably. Second, meta-modeling instruments allow us to further specify the structure and possible contents of such files. Based on that we infer the next hard requirement:

▶ **Req. 6:** The AQL-Answer output format must be specified clearly.

This is a hard requirement since insufficiently specified formats would only allow simple operations to combine answers, for example, appending one answer to another. In the context of cooperative analysis we want to be able to more comprehensively combine answers with respect to their potential content.

Furthermore, most tools in our scope use the Jimple IR inside their own custom formats to reference statements, methods and classes. Consequently, it appears to be a best practice that we want to adopt.

▷ **Req. 3:** The AQL should use a mutual IR (Jimple) to reference statements, methods and classes.

This is only a soft requirement because there are alternatives available, for example, such references could be given on source or bytecode level.

In the next subsection we will infer more requirements from existing query languages and result formats.

3.1.2 Query Languages and Result Formats

There already exist many query languages, so the question arises whether we can use one of them instead of developing the AQL from scratch. A large portion of these existing query languages are used to interact with databases but there also exist a few that have been developed to be used in the context of querying programs and also analyses. In the following we are taking a closer look at especially the latter type in order to find out whether we can reuse one of them. Even though there are some promising candidates, all these existing languages have some issues that disqualify them from being used in our cooperative analysis context. Nonetheless, we will make use of these languages by deriving more requirements for the AQL based on the properties and best practices that these languages reveal.

After this inspection of query languages which are templates for AQL-Queries, we will briefly check some structured formats that may be usable to embody their counterparts: AQL-Answers. Lastly, a few more requirements will be inferred from related cooperative approaches.

PQL The *Program Query Language (PQL)* [19] can be used to instruct a static or a dynamic analyzer in order to check whether it is possible to identify specific properties in a program. The composable queries represent these specific program properties. For example, via the PQL it is possible to ask, if a program contains two specific statements (source and sink) and if there is a connection between them (a taint flow). However, it is not possible to generalize such a query. To ask for any taint flow we would need to formulate one query per pair of predefined source and sink or sources and sinks occurring in the app under analysis. This suits the idea behind the PQL: find one specific issue pattern, and write a query to find all instances of this particular pattern; but does not fit into our context. As a requirement we can infer that it must, for instance, be possible to ask for taint flows from any source to any sink.

▶ **Req. 7:** AQL-Queries must be generalizable. It must be possible to ask for specific program properties (e.g. a taint flow between a certain source and sink) as well as any property of a certain type (e.g. any taint flow).

In particular the latter makes this requirement become a hard one as we may not know if a certain app contains a taint flow at all, thus, we must be able to ask for any.

The PQL further defines its syntax via a grammar. We will do the same for the AQL with respect to AQL-Queries (see Subsection 3.2.2).

▷ **Req. 4:** The syntax of AQL-Queries should be clearly specified (e.g. via a grammar).

Alternatively, the syntax could be described in e.g. natural language.

To reference program locations the PQL makes use of source code descriptions that involve fully qualified package/class names.

▷ **Req. 5:** Fully qualified package/class names should always be used in references.

Shortened forms may introduce ambiguity. For instance, there could be two different classes (`package1.ClassA` and `package2.ClassA`) in two different packages that share the same name (`ClassA`). To solve this issue, we could also attach a unique identifier (an id) to each class, hence, this is only a soft requirement.

Blast Query Language As the name suggests, this query language, in the following abbreviated with *BQL* [15], can be used to interact with the model checker BLAST (Berkeley Lazy Abstraction Software Verification Tool). It simplifies the interaction with BLAST but *only* with BLAST.

▷ **Req. 6:** The AQL should be able to interact with arbitrary tools.

Alternatively we could also define a set of tools the AQL is compatible with.

The BQL was not designed to be used to interact with different tools, although it can be used to specify program verification tasks which can be split into subtasks that would possibly be answerable by different tools.

▷ **Req. 7:** A single AQL-Query may consist of multiple smaller queries.

This allows to formulate simple queries per task instead of being forced to formulate a complex one for all tasks at once. As the PQL, the BQL allows to formulate specific taint flow queries, but a general one cannot be formulated.

FQL The *FQL (FShell Query Language)* [23, 27] can be used to compose test cases with specific coverage goals in form of queries. Required paths can be specified which again would allow to model taint flows, however, the FQL focuses on tests and test coverage. For the latter a coverage criterion, given as a reference (statement, method, class), can be specified in the query.

▷ **Req. 8:** Criteria such as references should be specifiable in AQL-Queries.

We keep this as a soft requirement since e.g. bytecode identifiers or source code line numbers could fulfill the same purpose. As the above, the FQL does not allow to formulate a generalized taint flow query.

CodeQL Another example for a query language that allows us to query programs or "code as though it were data" [122] is *CodeQL*. It certainly allows to encode taint flows in queries and allows to configure sources and sinks to be considered when executing these. Because of these features CodeQL is a promising candidate. However, it does not allow to deal with different tools nor does it come with any instruments to combine results of different queries – a required feature in the context of cooperative analysis.

▶ **Req. 8:** AQL-Queries must describe how to combine or merge tool results.

Union and intersection could, for example, represent two merge operations, thus, we must be able to specify which operation to use in a query.

SQL The most commonly known query language is the *Structured Query Language (SQL)*. It is used to interact with databases, hence, it can hardly be treated as a candidate usable in cooperative analysis context. Nonetheless, the history of the SQL shows us that a language cannot be fixed once and for all. Its first definition was officially released back in 1987 [182] and it is still regularly updated [183, 184]. Accordingly, we must ensure that the AQL can be updated and extended easily.

▷ **Req. 9:** The definition and implementation of the AQL should be extendable.

All the languages above are tailored to their field of application: PQL and CodeQL to program properties, BQL to instructing BLAST, FQL to test cases and coverage criteria, and SQL to databases of course. In summary, we infer:

▶ **Req. 9:** A query language must be specific to its field of application.

For example, with respect to the AQL we must be able to model intent related information (e.g. intent triples) – analysis information that is only required in Android context. From an abstract perspective all these query languages (except SQL) were designed to ask for properties of a program. The tool that should be used to find these properties is never described in more detail, because there always is just one or a fixed set. In our case, we want to be able to interact with different tools depending on parameters given in queries.

▷ **Req. 10:** AQL-Queries should be able to hint at the tool(s) to be used.

Otherwise it might not be possible to explicitly employ a different tool for a certain purpose or task.

After this brief view at query languages, we are taking a look at structured formats that could be considered as candidates to encode AQL-Answers. Some have already been mentioned while discussing analysis tool output formats in the previous subsection. We will not go into more detail with respect to unspecified textual formats, instead we will only evaluate structured formats.

XML One of the most used structured formats is *XML*. XML stands for *Extensible Markup Language*. Any data kept in an `.xml` file is stored in a hierarchical, tree-like structure. Apart from this structure, plain XML does not provide any information about what contents we may find in each element. This can be changed via meta-modeling. For example, *XML schema definitions (XSDs)* [191] can be used to specify more precisely what is expected to be found in a document that adheres to such a schema.

▷ **Req. 11:** The possible contents of AQL-Answers should be specified via meta-modeling.

JSON *JSON (JavaScript Object Notation)* is another tree-like structured document type that is very similar to XML. Actually it is so similar that almost any XML document can be converted into a JSON document and vice versa. In the end, it seems to be only a matter of taste when choosing between XML and JSON. However, one clear advantage of JSON is its compactness – way less symbols are required to express the same content and structure as in equivalent XML documents. Meta-modeling is also supported by implementing a *JSON schema* [161]. Such schemas are JSON documents as well. As it is very similar to XML, we cannot infer any new requirements from the JSON standard.

ProtoBuf Google developed (since 2001) and publicly released (open source) another document type called *Protocol Buffers (ProtoBuf)* [174] in 2008. It has been adopted, for example, by the analysis tool IC3. Although it is very similar, it has unique features that separate it from the above. Most notably it can be stored, parsed and edited in binary format which speeds up processing data stored in ProtoBuf documents. However, this advantage comes at a cost: binary documents cannot be read directly by humans. In the end, it again occurs to be a matter of taste when choosing to adopt ProtoBuf or not, hence, we do not infer any new requirements.

SARIF By its name *SARIF (Static Analysis Results Interchange Format)* [177] seems to be a promising candidate to encode AQL-Answers. On the one hand, it allows to specify a lot of information about the analysis tool, the analyzed artifacts and other environmental information, but on the other hand, it is not designed to present details about the properties detected by an analysis. It rather represents the circumstances that led to warnings given by a certain analysis tool in a concrete situation. SARIF is, so to say, focused on capturing a complete analysis picture instead of archiving the detected properties in detail. Consequently, SARIF is *not* the perfect candidate since we require this functionality as stated before by ▶ Req. 4. Nonetheless, SARIF offers two interesting insights: (1.) the SARIF format is an instance of JSON, more precisely, it is meta-modeled via a JSON schema, and (2.) to reference certain program locations source code line numbers are used in SARIF. In terms of soft requirements this again provides us ▷ Req. 11 and an additional one:

▷ **Req. 12:** Unique identifiers, such as source code line numbers, should be used to more precisely reference statements in the AQL.

Without fulfilling this soft requirement we may face ambiguity. For example, there might be two statements in the same method that look exactly the same considering their textual representation only. Jimple most of the time implicitly prohibits this due to the textual format used to represent statements, hence, this is only a soft requirement.

Witnesses In the field of software verification an often used format to exchange information between different verifiers are witnesses [50, 79]. Witnesses come in XML format, more specifically, GraphML [145] which is perfectly suited to hold the content of a witness, an automaton. These automata encode, for example error paths – a program execution that follows such a path eventually reaches an error location. While taint flows could be modeled as error paths, it becomes difficult to model other analysis information such as permissions or intent related information (e.g. intent triples). Thereby ▶ **Req. 4** is not strictly violated but in order to fulfill it, non-standardized custom key-value-pairs must be added. Also references to code elements should be provided on token level, hence, more custom key-value-pairs would be required to fulfill ▷ **Req. 3** and ▷ **Req. 5**. Furthermore, ▶ **Req. 9** cannot be met since the format is neither tailored to Android nor to taint analysis. Lastly, the format is not clearly specified via a meta-model (cf. ▷ **Req. 11**, ▶ **Req. 6**), instead natural language is used [190]. Nonetheless, witnesses are flexible enough such that we could use them in AQL context if we attach a meta-model that is tailored to Android, however, this means that we would have to *bend* this format such that it fits our needs. Due to the argumentation above witnesses will not be used to encode AQL-Answers, although they have successfully been employed in cooperative software verification [79, 89].

Others There are many other similar languages, for example HTML, the standard language to model websites, or YAML, a more "human-friendly" [192] version of XML. And even more variants of XML and JSON that have been further specified through schemas. For the latter we already saw one example, namely SARIF. For brevity, we refrain from listing and describing more formats and conclude that the available formats are either to coarsely (e.g. plain XML or JSON) or to narrowly (e.g. SARIF) specified. However, the requirements determined will be used as a guideline while specifying the AQL-Answer format.

3.1.3 Cooperation Types

In many areas cooperative approaches already exist and Android app analysis is no exclusion in this context. Basically, there are two different types: black- and white-box cooperations [79].

- *Black-box* cooperations use various analyses implemented in different tools without adapting these tools. The inputs are given into the black box and the outputs are taken out of the black box once it finishes its computation. What happens inside the black box is irrelevant for the cooperation.

- *White-box* approaches may change the behavior of analyses by adapting the associated tools before using them. To do so, the developer of the cooperative approach must be able to look inside the white box as if it was made of glass, i.e. the developer must have access to the source code of the respective analysis tool(s) and the expertise to make the intended adaptations.

One example for a white-box approach is implemented in the analysis tool DidFail. As described in the proposing paper [35] DidFail internally uses Epicc and FlowDroid. It adapts (or instruments) the (byte-)code of the app under analysis before it is provided to Epicc and *an adapted version* of FlowDroid. The code changes made assign unique identifiers to each intent sink and intent source such that they can be identified in the results of Epicc and FlowDroid. DidFail analyzes the intent and intent filter information given by Epicc in order to find matching pairs. These pairs and their respective intent sink and intent source stand for inter-component flows. Finally, DidFail interweaves these inter-component flows with the intra-component taint flows found by FlowDroid to detect inter-component taint flows. To do so, FlowDroid had to be adapted such that intent sinks and intent sources mentioned in results of Epicc can be identified in results of FlowDroid. Because of this adaptation of FlowDroid, DidFail must be treated as a white-box cooperation.

A black-box cooperation example that is very similar to DidFail, is embodied in IccTA [45]. To cooperate IccTA takes the output of IC3 (or its predecessor Epicc) as input. On the basis of this input IccTA identifies intent sinks and intent sources. Then it replaces intent sinks (e.g. a `startActivity` statements) by method calls to methods holding matching intent sources (e.g. `getStringExtra` statements).

> An intent sink **matches** an intent source whenever the respective intent matches the respective intent filter.

Because of these replacements an intra-component taint analysis (as implemented in FlowDroid) is sufficient to implicitly but finally find inter-component taint flows. Nowadays IccTA is integrated into FlowDroid and its functionality is triggered by providing the additional input produced by IC3 (or Epicc). Thus, no tool must be adapted and IccTA can be counted as a black-box cooperation.[13]

The advantage of black-box approaches is that the different analyses can be exchanged without requiring further adaptations, i.e. when IC3 replaced Epicc, IccTA must not have been touched. To this end, black-box cooperations follow the *low coupling, high cohesion* [2] principle more strictly. We want to acknowledge this advantage by inferring the following soft requirement:

▷ **Req. 13:** Analysis tools used in a cooperative analysis should be treated as black boxes.

Alternatively, we could require that tools must be adapted to become usable in AQL context.

In the area of software verification, there exist various approaches to cooperate in terms of "sequential and parallel combinations" [65] of different verifiers. Most of the time the different verifiers are represented by different strategies implemented in the same tool – seldom different tools are involved. The goal of such combinations is to get the correct result as fast as possible by arranging a certain verification task order or by dividing verification tasks into subtasks that can be dealt with in parallel. However, to find the best strategy is a crucial problem. "A Simple but Effective Approach" to solve this problem is described in the paper "Strategy Selection for Software Verification Based on Boolean Features" [65]. Simple properties of the program to be verified are encoded in boolean features which are used to select the supposedly best strategy. For example, it is determined if the program contains a loop or if it uses arrays before a strategy is selected. To

[13]In the past, IccTA was an extension of FlowDroid, hence, FlowDroid was adapted and consequently IccTA would have been considered to be a white-box cooperation.

adopt this practice we formulate two more soft requirements:

▷ **Req. 14:** Features of the app under analysis should be enumerable in AQL-Queries.

▷ **Req. 15:** It should be possible to use different cooperative strategies depending on the features given in an AQL-Query.

Without fulfilling these two requirement only basic strategies that are always applied could be integrated. In the context of the AQL, cooperative strategies are query transformations as detailed in one of the following subsections (see Section 3.3).

In another recently proposed study [90], verifiers work together by being executed sequentially, in parallel, or with respect to a certain algorithm selection technique. When executed sequentially they work together by sharing computation time. Any verifier gets all the other resources (e.g. memory and cpu cores available). As soon as a result is successfully determined no more verifiers are executed. When executed in parallel the verifiers share all resources except time. Any verifier is assigned a fair amount of e.g. memory. All verifiers can be stopped at the moment one of them finishes successfully. These first two strategies divide the available computing resources (power and time). We want to adopt this technique by inferring the soft requirement:

▷ **Req. 16:** Cooperative analyses should share computing resources.

The third and last strategy (algorithm selection) first determines features with respect to the potential difficulty to analyze the target program. A machine learning based algorithm selector then picks the most promising candidate (verifier) in accordance to the extracted features. This last strategy again provides us ▷ **Req. 14** and ▷ **Req. 15**. Other cooperative verification approaches focus on combining soft- and hardware verification [85], or combine static with dynamic techniques [41], however, no further requirements can be inferred.

Tao Xie et al. list examples for existing “Cooperative [...] Analysis” approaches and discuss advantages and disadvantages in a survey [38]. Not only *tool-tool* or *analysis-analysis* cooperations are considered but also *human-tool* approaches. One important finding when dealing with tool-tool cooperations is: while combining tools, complementary effects must be taken into account. In an Android taint analysis scenario for example, one tool may find less taint flows than another, but it finds some taint flows no other tool is able to find. Thus, it is a better idea to probably use both tools instead of choosing e.g. the one that finds more taint flows.

▷ **Req. 17:** The AQL should allow to combine different analysis results with respect to their complementary properties.

There exist many more cooperative approaches [73, 80] of different types partially summarized in a survey [79], however, no additional requirements can be inferred. Hence, for the sake of brevity, we end the inspection of related work at this point.

3.1.4 Summary

All hard and soft requirements inferred above are listed in Table 6. For each item listed, it is denoted whether it should be applied to AQL-Queries, -Answers or the AQL's implementation. Accordingly, the items are sorted with respect to their area of application, type (hard or soft) and identifier. It becomes clearly visible, that most requirements are AQL-Query specific. The last column of the table refers to the (sub-)section(s) that deal with the respective requirement.

Table 6: Summary of Hard and Soft Requirements

Description			AQL-		Implemen-	Sec-
(▶ Hard / ▷ Soft Requirement, Explanation)			Query	Answer	tation	tion
▶	2	AQL-Queries must allow to model all execution-specific inputs.	✔			3.2.2, 3.4.1
▶	5	Manually and automatically determined *files* must be referable in AQL-Queries.	✔			3.2.2
▶	7	AQL-Queries must be generalizable. It must be possible to ask for specific program properties (e.g. a taint flow between a certain source and sink) as well as any property of a certain type (e.g. any taint flow).	✔			3.2.2
▶	8	AQL-Queries must describe how to combine or merge tool results.	✔			3.2.2
▷	2	Raw outputs should directly be usable in AQL-Queries.	✔			3.3, 3.4.4
▷	4	The syntax of AQL-Queries should be clearly specified (e.g. via a grammar).	✔			3.2.2
▷	7	A single AQL-Query may consist of multiple smaller queries.	✔			3.2.2
▷	8	Criteria such as references should be specifiable in AQL-Queries.	✔			3.2.2
▷	10	AQL-Queries should be able to hint at the tool(s) to be used.	✔			3.2.2, 3.4.2
▷	13	Analysis tools used in a cooperative analysis should be treated as black boxes.	✔			3.4.1
▷	14	Features of the app under analysis should be enumerable in AQL-Queries.	✔			3.2.2
▷	15	It should be possible to use different cooperative strategies depending on the features given in an AQL-Query.	✔			3.3, 3.4.4
▷	17	The AQL should allow to combine different analysis results with respect to their complementary properties.	✔			3.2.2
▶	9	A query language must be specific to its field of application.	✔	(✔)		3.2.2, 3.2.5
▷	3	The AQL should use a mutual IR (Jimple) to reference statements, methods and classes.	✔	✔		3.2.2, 3.2.5
▷	5	Fully qualified package/class names should always be used in references.	✔	✔		3.2.5
▷	12	Unique identifiers, such as source code line numbers, should be used to more precisely reference statements in the AQL.	✔	✔		3.2.2, 3.2.5
▶	4	Any kind of *analysis information* must be encodable in an AQL-Answer.		✔		3.2.5
▶	6	The AQL-Answer output format must be specified clearly.		✔		3.2.5
▷	11	The possible contents of AQL-Answers should be specified via meta-modeling.		✔		3.2.5
▶	1	Tools must be accessible in different execution environments.			✔	3.4.5
▶	3	Tools and tool variants must be chosen with respect to the execution-specific inputs given in an AQL-Query.			✔	3.4.2
▷	1	Universal and tool-specific inputs should be configurable in the AQL-System.			✔	3.4.1
▷	16	Cooperative analyses should share computing resources.			✔	3.4.2, 3.4.3
▷	6	The AQL should be able to interact with arbitrary tools.	✔		✔	3.4.1
▷	9	The definition and implementation of the AQL should be extendable.	✔	✔	✔	3.4.3

Next, we introduce the AQL, starting with the syntax and semantics of AQL-Queries and continuing with a description of the AQL-Answer result format. While doing so, we refer to the derived requirements whenever they are tackled.

3.2 The Android App Analysis Query Language (AQL)

A centerpiece of the cooperative analysis approach presented in this thesis is the AQL. Two examples will be presented as an intuitive introduction to the language (see Subsection 3.2.1). After that both parts of the AQL, i.e. AQL-Queries and AQL-Answers, are defined. First, the syntax of AQL-Queries is defined by a grammar (see Subsection 3.2.2). Second, the semantics of the AQL are specified by means of derivation rules (see Subsection 3.2.3). Lastly, after an illustrative example showing the progression of a query (see Subsection 3.2.4), the structure of AQL-Answers is specified by a meta-model given in form of an XML schema definition (see Subsection 3.2.5).

3.2.1 Running Example 1: Intuitive Introduction to the AQL (Part 5/5)

Before we formally introduce the AQL, we continue our first running example. This intuitive continuation of the example does not require recapitulating the first four parts. Two AQL-Queries and the associated AQL-Answers will be presented. To support explainability, the answers will be given in an illustrative format.[14]

Query/Answer-Example 1 The AQL is primarily designed to be used in the context of Android and taint analysis, hence, the first example AQL-Query asks for taint flows and (Android) permissions. This will allow us to see, if the targeted app contains any taint flows and which statements (e.g. source or sink of a taint flow) are protected by which permissions. Let us assume the example app is represented by the Android package `example1.apk`, then the AQL-Query to ask for permissions reads as follows:

```
Permissions IN App('example1.apk') ?
```

Once an analysis tool has answered this query, an AQL-Answer will become available that presents the result. Let us assume that two permission-protected statements are found. In that case the respective AQL-Answer can be depicted as follows (see Figure 6):

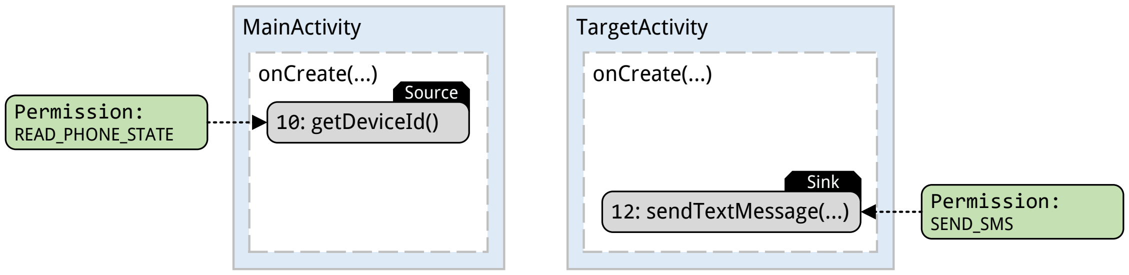


Figure 6: Illustration of AQL-Answer (`Permissions`-Query)

As we can see, the statement `getDeviceId()` requires the permission `READ_PHONE_STATE` and the statement `sendTextMessage()` the permission `SEND_SMS`. These two permissions indicate that sensitive data is read and that arbitrary data is sent.

The AQL-Query to ask for taint flows looks very similar. Only the subject for which we are asking is changed from `Permissions` to `Flows`:

```
Flows IN App('example1.apk') ?
```

The result of this query is shown below in form of an illustrated AQL-Answer in Figure 7. The answer shows a single taint flow that connects the `getDeviceId()` statement with

[14]Appendix A.6.2 partially contains an AQL-Answer in its non-illustrative form (see Listing 24).

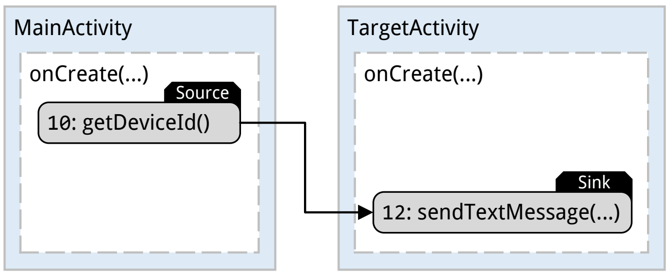


Figure 7: Illustration of AQL-Answer (`Flows`-Query)

the `sendTextMessage()` statement.

Most likely the answer created for both queries would be determined by different tools. To bring together the answers of both tools, or to build a simple cooperative analysis, we can ask the following AQL-Query:

```
UNIFY [
    Permissions IN App('example1.apk') ?,
    Flows IN App('example1.apk') ?
] ?
```

To construct the answer for this query, a third tool builds the union of the two AQL-Answers illustrated above. The final answer than contains both information:

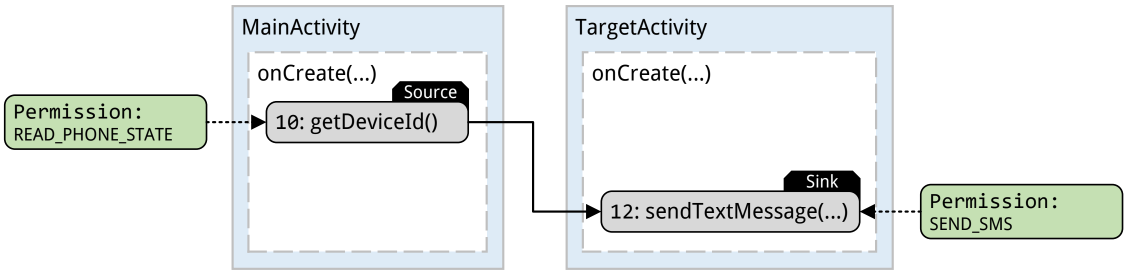


Figure 8: Illustration of AQL-Answer (`UNIFY`-Query)

It becomes visible that the taint flow connects a permission-protected statement with another statement which is protected by a second permission. Consequently, this simple cooperative analysis allows to conclude, that this taint flow might be security-critical since source and sink are protected by permissions.

To simplify the example we attached **source** and **sink** markers to all illustrations. However, this information is not explicitly provided in AQL-Answers unless we ask for it:

```
UNIFY [
    Permissions IN App('example1.apk') ?,
    Flows IN App('example1.apk') ?,
    Sources IN App('example1.apk') ?,
    Sinks IN App('example1.apk') ?
] ?
```

For better readability we could also encapsulate the `Sources-and-Sinks`-Query by applying the `UNIFY` operator twice:

```
UNIFY [
    Permissions IN App('example1.apk') ?,
    Flows IN App('example1.apk') ?,
    UNIFY [
        Sources IN App('example1.apk') ?,
        Sinks IN App('example1.apk') ?
    ] ?
] ?
```

Query/Answer-Example 2 During the example above we assumed that we have access to a tool that is capable of analyzing ICC directly. Often such a tool is not given. Thus, if we ask the `Flows`-Query (`Flows IN App('example1.apk') ?`) the answer is usually a different one (see Figure 9):

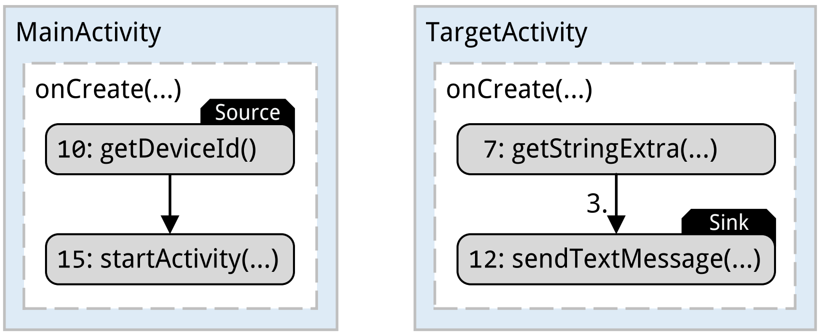


Figure 9: Illustration of AQL-Answer (`Flows`-Query – no ICC capabilities)

Only the intra-component taint flows are found. The connection between both components remains undetected. Consequently, the complete taint flow stretching from source to sink cannot be detected.

To change this we can insist on caring about ICC. To do so, we adapt the query by attaching the feature `ICC`[15]:

```
Flows IN App('example1.apk') FEATURING 'ICC' ?
```

Still, we may have no tool that is able to find taint flows that start and end in different components, however, we may have a tool that detects any kind of ICC. Such a tool could provide the following answer:

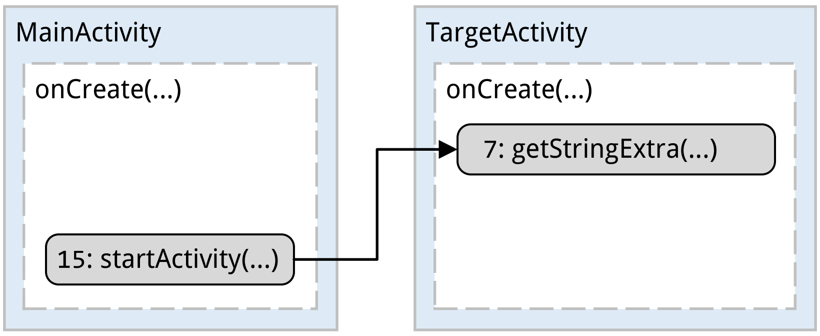


Figure 10: Illustration of AQL-Answer (`Flows`-Query – with ICC Capabilities)

[15]Later on it will be explained how to determine such features automatically (see Section 3.3).

To bring together both answers we use the `CONNECT` operator:

```
CONNECT [
    Flows IN App('example1.apk') ?,
    Flows IN App('example1.apk') FEATURING 'ICC' ?
] ?
```

As the `UNIFY` operator, the `CONNECT` operator gathers all detected flows in a single answer. Additionally, it computes the transitive closure for all detected flows such that we get the following answer:

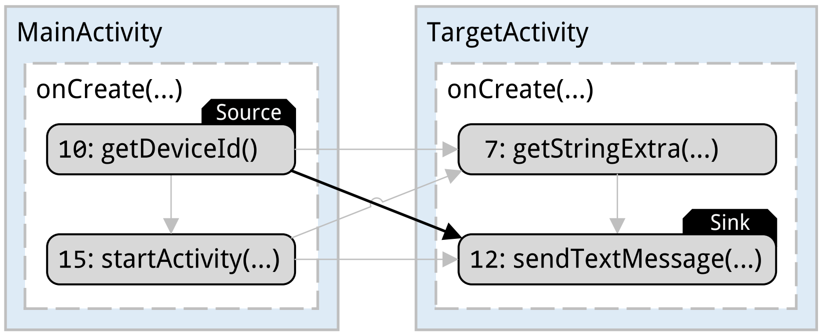


Figure 11: Illustration of AQL-Answer (`CONNECT`-Query)

The intra-component taint flows and the inter-component flow are stitched together. Finally, the taint flow starting at the **source** in the `MainActivity` and ending at the **sink** in the `TargetActivity` is successfully detected (see highlighted black edge in Figure 11).

These two examples provide a glimpse at the possibilities the AQL offers. The formal introduction will introduce all possibilities in the following.

3.2.2 AQL-Queries: Syntax

The syntax of AQL-Queries is described via grammar G which is introduced throughout this whole subsection, thereby ▷ **Req. 4** is met. Illustrations are partially used to intuitively but implicitly describe its terminals, non-terminals and production rules. Alternatively, the complete grammar can be found in the appendix (see Appendix A.5.1). The most general structure of a query that can be derived from the starting symbol of G (Query) is quickly described: it consists of an arbitrary long sequence of Query-Parts – as depicted in Figure 12[16] ($p_{0.1}$), i.e. a query may consist of multiple smaller queries (cf. ▷ **Req. 7**). In the context of the AQL, variables just fulfill the purpose of shortcuts to increase the readability of queries. Hence, the usage of variables and their definitions (see Figure 12 – $p_{0.2}$) is explained later on.

When describing the structure of a query part (Query-Part), it becomes more difficult. Figure 13[16] illustrates the derivation process for query parts. As illustrated, a query part can be derived to (1) an Analysis-Question, (2) an Operator-Question, (3) a Result or (4) a Variable-Usage. The latter two can be used to structure and shorten complex queries. The first two represent two of three centerpieces of the AQL, namely *questions*.

[16] `Rectangles` represent terminal symbols whereas rectangles with rounded corners stand for non-terminal symbols.

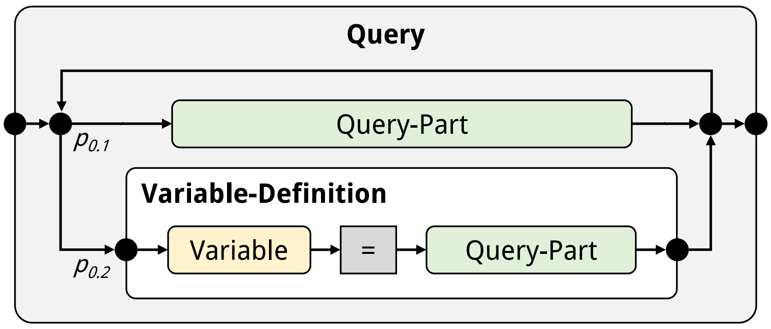


Figure 12: Visualization of a Query (Production Rule p_0)

On the one hand, an $\boxed{\text{Analysis-Question}}$ is mainly used for the interaction with arbitrary analysis tools (cf. ▷ **Req. 6**). On the other hand, an $\boxed{\text{Operator-Question}}$ was initially and primarily designed to combine results of analysis tools (cf. ▶ **Req. 8**).

Analysis-Questions must transfer all the information required to run an analysis tool in order to get a result that answers the question. According to ▶ **Req. 2** this includes all execution-specific inputs. To identify which tool is required, the subject of interest is always specified first in an analysis question.

> **Subjects of interest (SOIs)** in the context of the AQL are: flows, sources, sinks, permissions, intents, intent filters, intent sinks, intent sources, a slice or arguments.

Through this set of SOIs, AQL-Queries are already tailored to their application area (cf. ▶ **Req. 9**). However, additional SOIs could be added by adapting only one production rule of the grammar. The production rule (p_2) to derive a SOI is visualized in and defined by Figure 14. For example, if `Flows` is specified as SOI, a tool is required that outputs control, information or data flows such as taint flows. FlowDroid or Amandroid would be two exemplary tools that can be used to answer analysis questions that ask for flows.

The next part that has to be specified for an analysis question is the analysis target. An app must always be specified as a part of the analysis target. Furthermore, by specifying a precise statement, method or class a more precise scope may optionally be defined. A reference is used to specify both, the app and the scope if provided. Production rule p_3 (depicted in Figure 15) defines how to derive a reference (cf. ▷ **Req. 8**). With respect to analysis questions we can look for properties *in* a certain app and scope or *from* one *to* another. This allows us, for instance, to ask for any taint flows in an entire app or for a specific taint flow, from a certain source to a certain sink. Thus, it is possible to ask for general as well as specific properties as demanded by ▶ **Req. 7**. For example,

```
App('A.apk')
```

and

```
Statement('virtualinvoke r2.<android.content.Intent: java.lang.String
     getStringExtra(java.lang.String)>("Secret")')->App('A.apk')
```

both represent valid derivations of a reference. Note, the AQL does not confine the content (`'virtualinvoke [...]'`) that is provided to describe a statement by any means, however, to acknowledge ▷ **Req. 3** we will use the Jimple IR for statements, methods and classes in the context of this thesis. A source code line number or any integer number

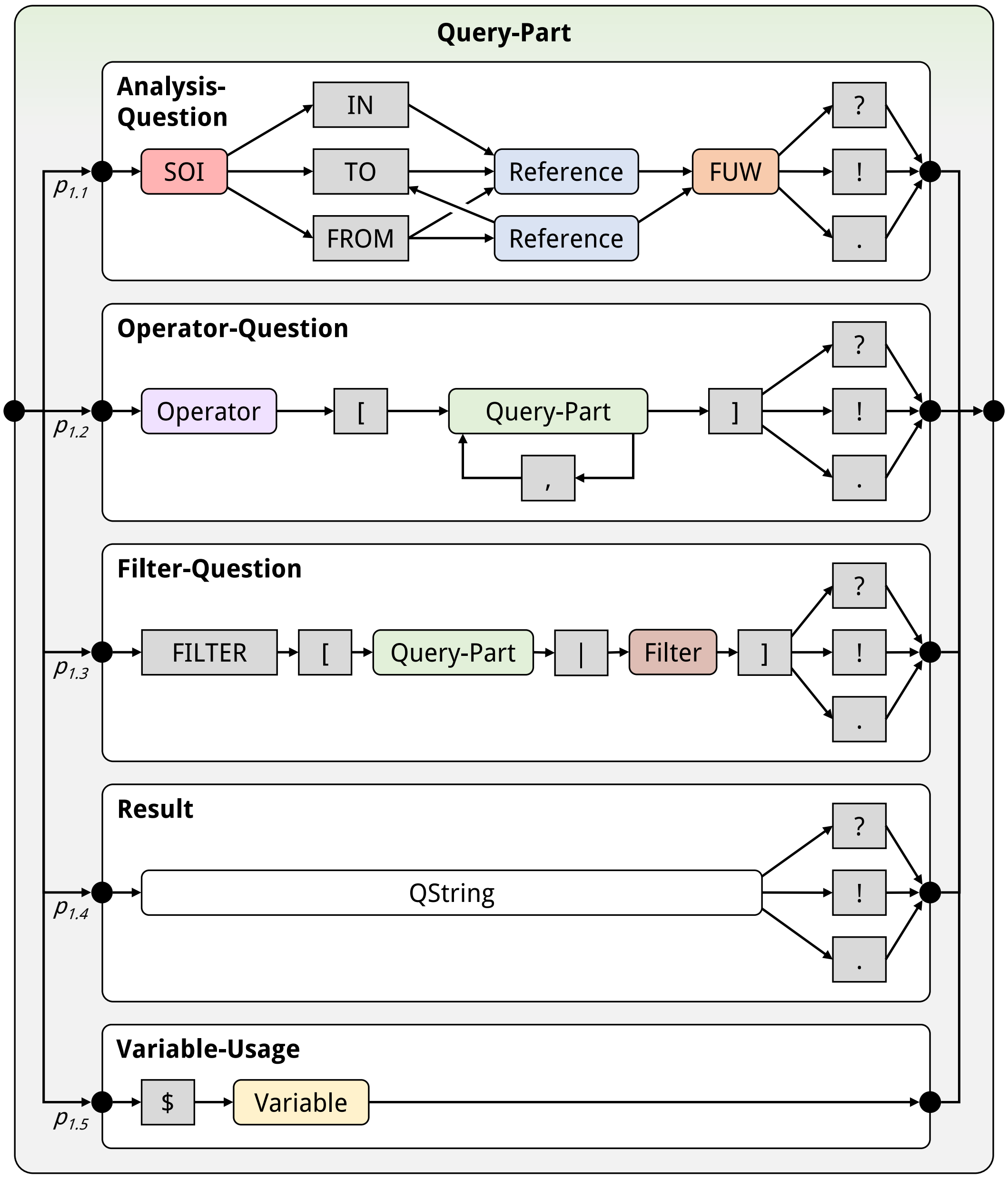


Figure 13: Visualization of a Query Part (Production Rule p_1)

SOI ::= (`Flows` | `Sources` | `Sinks` | `Permissions` | `Intents` | `IntentFilters` | `IntentSinks` | `IntentSources` | `Slice` | `Arguments`)

Figure 14: Production Rule p_2

associated with a statement can additionally be assigned to a statement (see p_3 in Figure 15). As observed in other languages, this often simplifies the identification process of individual statements (cf. ▷ **Req. 12**).

Production rule p_3 holds two more non-terminals, namely QString and Preprocessor-

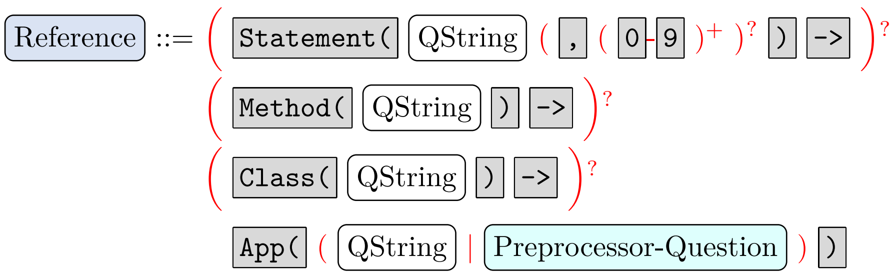

Reference ::= (Statement(QString (, (0-9)+)?) ->)?
(Method(QString) ->)?
(Class(QString) ->)?
App((QString | Preprocessor-Question))

Figure 15: Production Rule p_3

Question. The first is used to derive almost arbitrary strings – the only symbol that is not allowed is ' since any string must be surrounded by this symbol to mark its beginning and ending. A QString can also be replaced by a Query-Part, this allows us to provide queries that represent a dynamic value which is determined once the respective query part is answered instead of static values represented through strings. For example, we could write a query that targets a tool that produces an `.apk` file as output and use this file as app in our designated analysis target (cf. ▶ **Req. 5**). The production rule to further derive a QString is p_4 (see Figure 16).

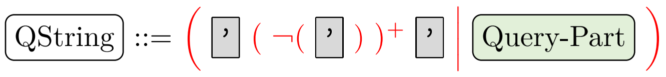


Figure 16: Production Rule p_4

A preprocessor question, the third and last question type to be introduced, can be further derived via production rule p_5 which is illustrated and defined by Figure 17. Once

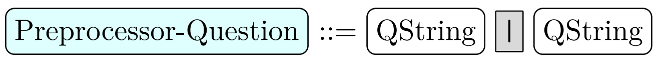


Figure 17: Production Rule p_5

p_5 is applied, two QStrings, separated by a | symbol, are derived. The first QString represents the app, i.e. it could stand for the path to an `.apk` file in the local file system or a query that asks for such an `.apk` file. The second QString comes into play to identify the preprocessor to be used. It embodies the so-called preprocessor keyword.

A **preprocessor keyword** unambiguously identifies one preprocessor.

For example, the keyword `COMBINE` may be associated with a tool like ApkCombiner which is used to merge multiple `.apk` files into one. Which preprocessors and preprocessor keywords are available depends on the implementation realizing the AQL and its configuration as described in detail in Section 3.4.

To further specify an analysis question, (1.) features, (2.) tools to be used or (3.) arguments to be considered with the analysis question, can be assigned after naming the analysis target. This is done by deriving the non-terminal FUW (see p_6 in Figure 18). Lists of QStrings may be used after `FEATURING` (or `FEATURES`) and `USING` (or `USES`) to assign features and tools that shall be regarded and preferred while answering this query (cf. ▷ **Req. 10**, ▷ **Req. 14**). Accordingly, a different tool may be chosen to answer an analysis question if a certain feature is assigned or the use of specific tools is enforced.

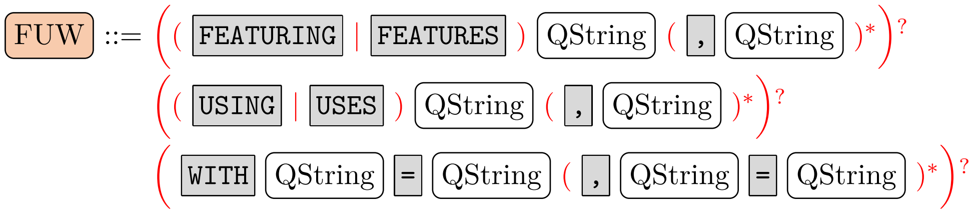


Figure 18: Production Rule p_6

Pairs of QStrings, separated by =, can be provided after WITH to assign key-value-pairs that may influence, for example, the configuration of a tool. Note that in any case the QStrings may also refer to query parts again and all values assigned may be determined dynamically by answering the associated query parts.

Finally, an analysis question is finished with an ending symbol which may either be ?, ! or . . The different ending symbols indicate what type of answer is expected.

> The **ending symbol** of an AQL-Query indicates the expected answer type for this query: ? is associated with an AQL-Answer, ! with an arbitrary file and . with raw data.

Operator-Questions are employed to specify operations on the results of one or more query parts, for instance, merge or combine operations as demanded by ▶ **Req. 8**. Therefore, operator questions can only be executed if all related results are available – all underlying questions have been answered. An operator question is derived via production rule $p_{1.2}$ of grammar G (see Figure 13). First the operator must be determined by deriving the Operator non-terminal also called *operator name*. This can only be done via p_7 (see Figure 19). It allows to derive the name of a default operator that must be implemented by

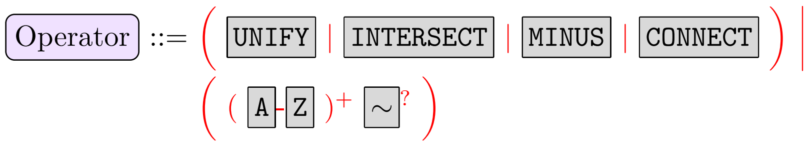


Figure 19: Production Rule p_7

any implementation of the AQL or a custom operator's name, which must consist of capital letters only. Optionally and only by convention the name of a custom operator should end with ~ if the associated operator is performing an over- or under-approximation.

> Any system implementing the AQL must support the **default operators**: unify, intersect, minus and connect.

Most of these default operators must perform the set operations their names suggest. Unify, for instance, collects all the contents of various AQL-Answers in a single one – it builds the union. Respectively, intersect and minus build the intersection and the difference. Note that the complement is not a default operator, since the complement of an AQL-Answer is not computable unless the full set of e.g. all existing permissions or possible flows is defined – which is not done in AQL context. The unify operator is already sufficient to fulfill ▷ **Req. 17**. The requirement states that complementary effects must be taken into account and this can be done by e.g. unifying results that hold different taint flows probably computed by different tools. Even more can be done via the default connect operator. It must perform three actions:

1. build the union of the given answers,
2. connect intent sinks and intent sources whenever their associated intent triples match (by adding flows from and to the respective references), and
3. compute the transitive closure of all flows given in the provided answers also by adding additional flows.

Custom operators may perform arbitrary operations and may overwrite default operators.

After denoting the name of the operator at least one query part enclosed by square brackets must be derived. If more than one query part is derived, they are separated by commas ($\boxed{\texttt{,}}$). These query parts represent the results to operate on. As analysis questions any operator question ends with an ending symbol.

Filter-Questions or Filter-Operator-Questions are operator questions tailored to a specific purpose: filtering a given answer. The operator name is fixed ($\boxed{\texttt{FILTER}}$) and inside the square brackets two parts (split by a $\boxed{\texttt{|}}$ symbol) can be found. In contrast to ordinary operator questions, the first part only allows to derive a *single* query part. The second part inside the square brackets consists of the non-terminal $\boxed{\text{Filter}}$. It can further be derived via production rule p_8 which is presented and defined in Figure 20. As indicated by p_8 there are three options to filter a result:

1. *By reference:* Any element in the given AQL-Answer, that is not related to the given reference, is filtered out (removed).
2. *By subject of interest:* Any element that is related to the given subject of interest is removed.
3. *By key-value-pair:* Only elements that hold a certain value for a specific attribute (key) are kept.

Filter-questions like any other question (except preprocessor questions) end with one of the three existing ending symbols ($\boxed{\texttt{?}}$, $\boxed{\texttt{!}}$, $\boxed{\texttt{.}}$). Usually the ending symbol will be a $\boxed{\texttt{?}}$ since most of the time AQL-Answers are meant to be filtered. However, there are examples (see Subsections 3.3.1, 3.4.4 or Chapter 5) where operator questions end with another symbol, hence, we do not want to constrain the language here.

> Systems employing the AQL must implement the **default filter operator** with all its options.

$$\boxed{\text{Filter}} ::= \boxed{\text{Reference}} \mid \boxed{\text{SOI}} \mid (\boxed{\text{QString}} \boxed{\texttt{=}} \boxed{\text{QString}})$$

Figure 20: Production Rule p_8

Variables As mentioned before, variables are used as shortcuts in the AQL to increase the readability of queries. A variable definition can only be placed before, after or in between two complete queries. To do so production rule $p_{0.2}$ has to be applied (see Figure 12 – Page 60). It expects a variable name to be derived for $\boxed{\text{Variable}}$ and a

$\boxed{\text{Query-Part}}$ as content of the variable. The $\boxed{\texttt{=}}$ terminal symbol between the variable name and the query part signals that the usage of this variable is equal to denoting the respective query part. The variable name is derived by applying production rule p_9 (see Figure 21). Any lower case letter and any number may be contained in a variable's name, however, it must not start with a number.

$$\boxed{\text{Variable}} ::= \boxed{\texttt{a}}\text{-}\boxed{\texttt{z}} \Big(\big(\boxed{\texttt{a}}\text{-}\boxed{\texttt{z}} \mid \boxed{\texttt{0}}\text{-}\boxed{\texttt{9}} \big) \Big)^{+}$$

Figure 21: Production Rule p_9

Once a variable has been defined, it can be used whenever a query part is derived. To use a variable a $\boxed{\texttt{\$}}$ sign must be derived followed by the variable's name as defined by production rule $p_{1.5}$ (see Figure 13 – Page 61).

We are not providing further example queries here as there are several throughout the thesis. For example, the two intuitive ones presented at the beginning of this section (see Subsection 3.2.1). The semantics of AQL-Queries are defined in the next subsection.

3.2.3 AQL-Queries: Semantics

The semantics of a language describe the meaning of each word (or sentence) in a language. Accordingly, the semantics of the AQL must describe the meaning of each query. Since we use queries to interact with tools, the results given by those tools stand for the meanings of such queries.

Definitions & Assumptions To specify the AQL's semantics, let us start by defining two sets: First, $Q = L(G)$ is the set of all queries that can be derived from grammar G which has been presented in the previous subsection. Second, the set R represents the set of all possible tool results an analysis tool, an operator or a preprocessor might output. Possible tool results are AQL-Answers ($\boxed{\texttt{?}}$), arbitrary files ($\boxed{\texttt{!}}$) or raw answers ($\boxed{\texttt{.}}$). The *ending symbol* ($\boxed{\texttt{?}}$, $\boxed{\texttt{!}}$, $\boxed{\texttt{.}}$) of an AQL-Query already indicates which result type is expected. AQL-Answers are represented through `.xml` files that follow the meta-model presented in Subsection 3.2.5. Raw answers can also be represented through text files in the sense that their only content is the raw answer. In conclusion, we assume:

> Any element $r \in R$ can be described by a file which is uniquely identified by its **filename** f_r.

An AQL query q can be in two different states: answered and unanswered. Unanswered queries still contain one or more questions that require tool execution(s) in order to be answered – answered queries do not.

> If the derivation of $q \in Q$ requires the application of at least one of the **crucial production rules** ($p_{1.1}$, $p_{1.2}$, $p_{1.3}$ or p_5 – derivation of an Analysis-, Operator-, Filter- or Preprocessor-Question as specified in the previous Subsection 3.2.2), it is categorized as ***unanswered*** – otherwise q counts as ***answered***.

Consequently, the function *ready* indicates if a query is ready to be answered. The function is declared as follows:

$$\begin{aligned} \mathit{ready} : Q &\rightarrow \{\mathit{true}, \mathit{false}\} \\ q &\mapsto \mathit{ready}(q) \end{aligned}$$

with

$$ready(q) = \begin{cases} true & \text{the derivation of } q \text{ starts with an application of} \\ & \text{a crucial production rule but requires no further} \\ & \text{application(s) of any crucial production rule.} \\ false & \text{otherwise} \end{cases}$$

In order to answer a query that is ready we have to run an analysis tool ($p_{1.1}$), an operator ($p_{1.2}$ or $p_{1.3}$) or a preprocessors (p_5) depending on the question(s) contained in the respective query. Later on we present how to select a tool with respect to a certain question (see Section 3.4). For now, we assume:

> There exists a **configuration** C that defines which tools are available, and that there always is a tool $t \in C$ such that $r_q \in R$ is the result of t for the query q.

Due to this assumption a configuration must not be provided as input to the function described next – a simplification to foster explainability. To model the outcome of any tool the function *ask* maps queries to results.

$$ask : Q \to R$$
$$q \mapsto r_q$$

(Assuming that $r_q \in R$ is the result of a tool answering q.)

Please note that there is no further input required to specify analysis targets (apps), since we assume that:

> **Analysis targets (apps)** are uniquely identified in queries.

For example, if a query uses `App('A.apk')` as reference, we assume that this reference unambiguously identifies a certain app.

Lastly, to foster readability, we introduce the following notation:

> Any question or sub-query s of a query $q \in Q$ ($q = ...s...$) whose derivation starts with a crucial production rule may be substituted by:
>
> $$\boxed{'}\ f_r\ \boxed{'}\ (\ \boxed{?}\ |\ \boxed{!}\ |\ \boxed{.}\ )$$
>
> where
>
> (1.) $r \in R$ refers to the result of a tool answering s represented by the filename f_r, and (2.) the ending symbol is equal to the one at the end of s ($\boxed{!}$ if there is none).
>
> We denote such a **substitution** by: $q_{[s \leftarrow r]}$

Derivation Rules Under these assumptions and with these definitions at hand, the semantics of the AQL can be described by the following two *derivation rules* that state how a query should be processed to find its final result (meaning):

$$\frac{q \neq s \land ready(s) = true}{q = ...s... \to q_{[s \leftarrow ask(s)]}} \qquad \texttt{(Query)}$$

$$\frac{ready(q) = true}{q \to ask(q)} \qquad \texttt{(Termination)}$$

The first rule (`Query`) will be applied until the query only holds a single question. Then, when the whole query is ready, the `Termination` rule can be applied. By applying this last rule the final result (the *answer* to the initial query) is determined. The next subsection explains the semantics of the AQL on an example query.

3.2.4 Example 2: Query Progression

To better understand the semantics of the AQL, let us have a look at the following example AQL-Query (q):

```
UNIFY [
    Permissions IN App('A.apk' | 'PP') ?,
    Flows IN App('A.apk' | 'PP') ?
] ?
```

It contains overall five questions (in terms of non-terminals): Two $\boxed{\text{Preprocessor-Question}}$s, two $\boxed{\text{Analysis-Question}}$s and one $\boxed{\text{Operator-Question}}$. The derivation sequence of q with respect to grammar G is shown in Figure 22 (Page 68). Clearly, the derivation sequence requires multiple applications of crucial production rules (p_5 and $p_{1.1}$ twice; $p_{1.2}$ once), hence, the `Termination` derivation rule cannot be applied in the first place, since $ready(q) = false$ holds. To apply the `Query` derivation rule we must first determine a sub-query s of the query such that $ready(s) = true$ holds. The easiest way to do this is to search the derivation sequence from end to start. Stop the search as soon as the first application of one of the crucial production rules has been found. This gives us derivation **17**, which applies the production rule p_5 (see Figure 22). The corresponding partial derivation result is:

$$\boxed{\texttt{'}}\boxed{\texttt{A.apk}}\boxed{\texttt{'}}\ \boxed{\texttt{|}}\ \boxed{\texttt{'PP'}} \qquad (s)$$

Since production rule p_5 was applied to derive a $\boxed{\text{Preprocessor-Question}}$, we know that we must execute a preprocessor. Furthermore, the preprocessor must be associated with the keyword $\boxed{\texttt{'PP'}}$ and the target to be preprocessed is the app represented through the `.apk` file $\boxed{\texttt{'}}\boxed{\texttt{A.apk}}\boxed{\texttt{'}}$. Let $r_1 \in R$ be the result given by the preprocessor which is represented by the filename $f_{r_1} =$ `B.apk`. Finally, to apply the `Query` rule the substitution $q_{[s \leftarrow r_1]}$ is employed. Thereby every occurrence of s is replaced by:

$$\boxed{\texttt{'}}\ f_{r_1}\ \boxed{\texttt{'}}\ \boxed{\texttt{!}} = \boxed{\texttt{'}}\boxed{\texttt{B.apk}}\boxed{\texttt{'}}\ \boxed{\texttt{!}}$$

This gives us the following query (q'):

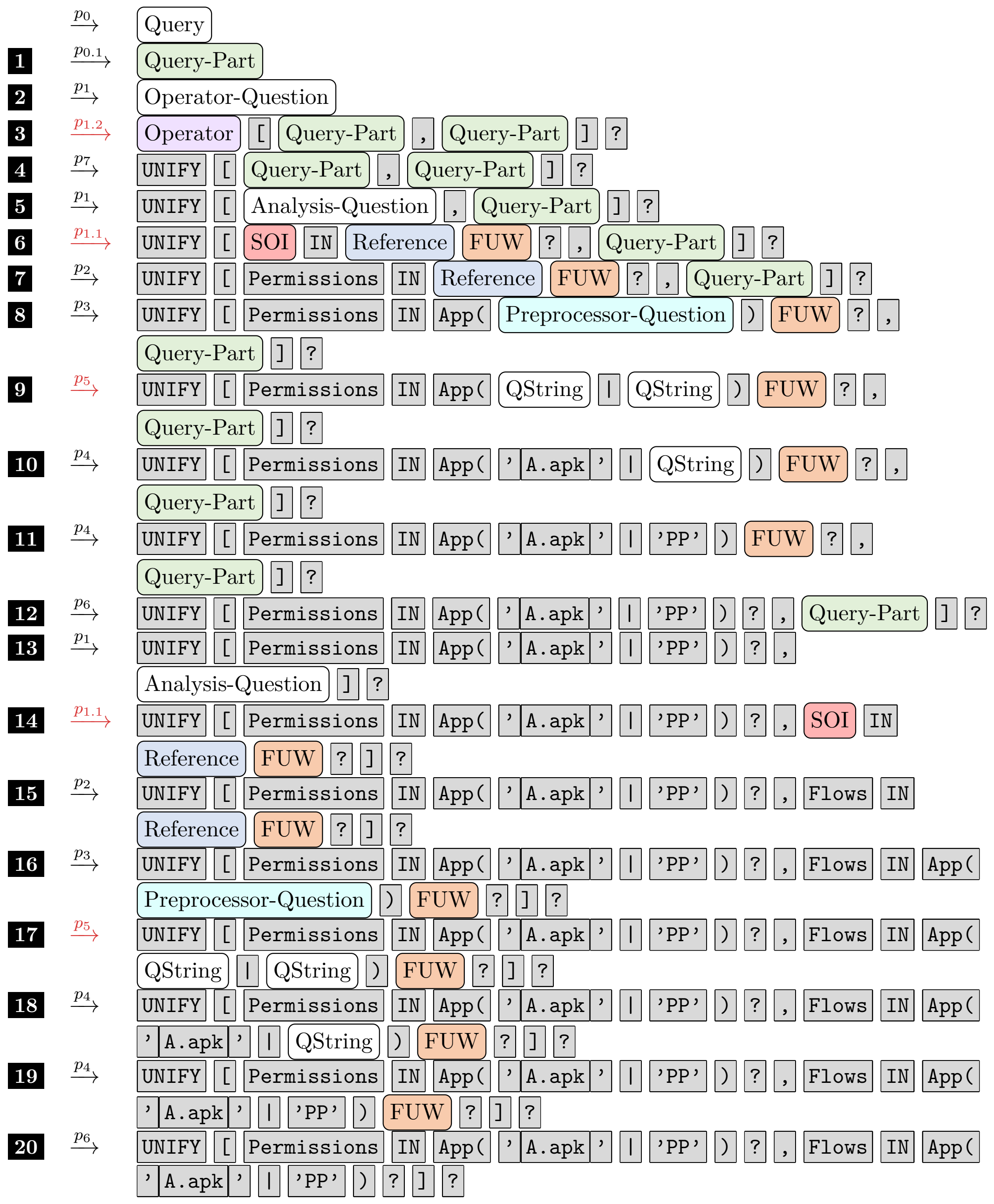
```
UNIFY [
    Permissions IN App('B.apk' !) ?,
    Flows IN App('B.apk' !) ?
] ?
```

To apply the `Query` derivation rule another time we must find a new sub-query s' ($q' = ...s'...$) that is ready to be answered. The derivation sequence of q' is shown in Figure 23 (Page 70). From bottom to top the first crucial production rule applied is $p_{1.1}$ during derivation **15'**. An $\boxed{\text{Analysis-Question}}$ is derived at this point. The respective partial derivation result is:

$$\boxed{\texttt{Flows}}\ \boxed{\texttt{IN}}\ \boxed{\texttt{App(}}\ \boxed{\texttt{'}}\boxed{\texttt{B.apk}}\boxed{\texttt{'}}\ \boxed{\texttt{!}}\ \boxed{\texttt{)}}\ \boxed{\texttt{?}} \qquad (s')$$

Since it is an $\boxed{\text{Analysis-Question}}$ we are looking for an analysis tool that allows us to detect intra-app (taint) flows ($\boxed{\texttt{Flows}}$) inside ($\boxed{\texttt{IN}}$) one application ($\boxed{\texttt{B.apk}}$). Let us assume we got such a tool in our quiver and it gives us the result $r_2 \in R$ with $f_{r_2} =$ `flows.xml`. Thus, we have to substitute s' by:

$$\boxed{\texttt{'}}\boxed{\texttt{flows.xml}}\boxed{\texttt{'}}\ \boxed{\texttt{?}}$$



$\xrightarrow{p_0}$ Query

1 $\xrightarrow{p_{0.1}}$ Query-Part

2 $\xrightarrow{p_1}$ Operator-Question

3 $\xrightarrow{p_{1.2}}$ Operator [Query-Part , Query-Part] ?

4 $\xrightarrow{p_7}$ UNIFY [Query-Part , Query-Part] ?

5 $\xrightarrow{p_1}$ UNIFY [Analysis-Question , Query-Part] ?

6 $\xrightarrow{p_{1.1}}$ UNIFY [SOI IN Reference FUW ? , Query-Part] ?

7 $\xrightarrow{p_2}$ UNIFY [Permissions IN Reference FUW ? , Query-Part] ?

8 $\xrightarrow{p_3}$ UNIFY [Permissions IN App(Preprocessor-Question) FUW ? , Query-Part] ?

9 $\xrightarrow{p_5}$ UNIFY [Permissions IN App(QString | QString) FUW ? , Query-Part] ?

10 $\xrightarrow{p_4}$ UNIFY [Permissions IN App(' A.apk ' | QString) FUW ? , Query-Part] ?

11 $\xrightarrow{p_4}$ UNIFY [Permissions IN App(' A.apk ' | 'PP') FUW ? , Query-Part] ?

12 $\xrightarrow{p_6}$ UNIFY [Permissions IN App(' A.apk ' | 'PP') ? , Query-Part] ?

13 $\xrightarrow{p_1}$ UNIFY [Permissions IN App(' A.apk ' | 'PP') ? , Analysis-Question] ?

14 $\xrightarrow{p_{1.1}}$ UNIFY [Permissions IN App(' A.apk ' | 'PP') ? , SOI IN Reference FUW ?] ?

15 $\xrightarrow{p_2}$ UNIFY [Permissions IN App(' A.apk ' | 'PP') ? , Flows IN Reference FUW ?] ?

16 $\xrightarrow{p_3}$ UNIFY [Permissions IN App(' A.apk ' | 'PP') ? , Flows IN App(Preprocessor-Question) FUW ?] ?

17 $\xrightarrow{p_5}$ UNIFY [Permissions IN App(' A.apk ' | 'PP') ? , Flows IN App(QString | QString) FUW ?] ?

18 $\xrightarrow{p_4}$ UNIFY [Permissions IN App(' A.apk ' | 'PP') ? , Flows IN App(' A.apk ' | QString) FUW ?] ?

19 $\xrightarrow{p_4}$ UNIFY [Permissions IN App(' A.apk ' | 'PP') ? , Flows IN App(' A.apk ' | 'PP') FUW ?] ?

20 $\xrightarrow{p_6}$ UNIFY [Permissions IN App(' A.apk ' | 'PP') ? , Flows IN App(' A.apk ' | 'PP') ?] ?

Figure 22: Derivation Sequence of the Initial Example Query q

By this substitution we get (q''):

```
UNIFY [
    Permissions IN App('B.apk' !) ?,
    'flows.xml' ?
] ?
```

For brevity, we skip the consequent derivation rule applications. In the end, the whole process gives us the sequence of derivation rule applications summarized on Page 71. After the application of the `Termination` derivation rule the final answer ($f_{r'''} =$ `answer.xml`) becomes available.

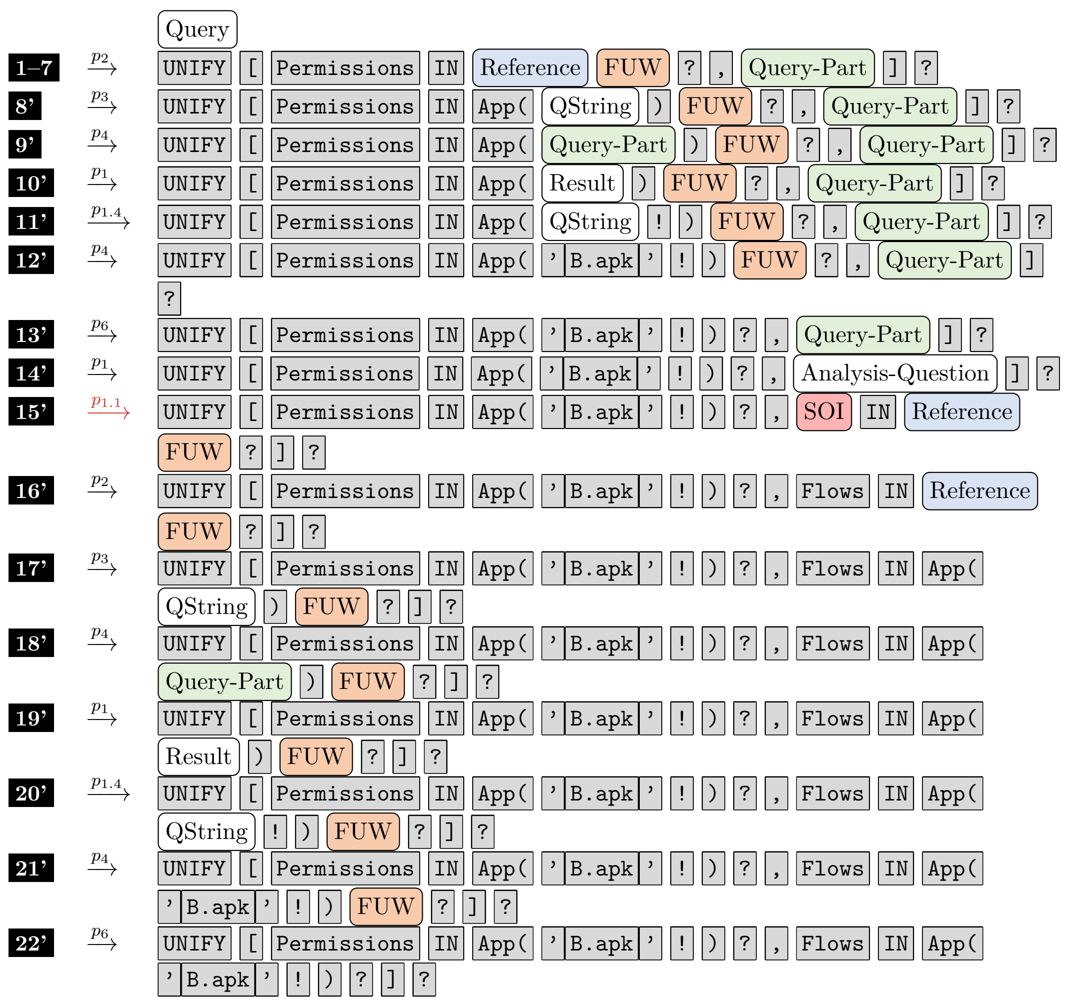

Query
1–7 $\xrightarrow{p_2}$ UNIFY [Permissions IN Reference FUW ? , Query-Part] ?
8’ $\xrightarrow{p_3}$ UNIFY [Permissions IN App(QString) FUW ? , Query-Part] ?
9’ $\xrightarrow{p_4}$ UNIFY [Permissions IN App(Query-Part) FUW ? , Query-Part] ?
10’ $\xrightarrow{p_1}$ UNIFY [Permissions IN App(Result) FUW ? , Query-Part] ?
11’ $\xrightarrow{p_{1.4}}$ UNIFY [Permissions IN App(QString !) FUW ? , Query-Part] ?
12’ $\xrightarrow{p_4}$ UNIFY [Permissions IN App(’ B.apk ’ !) FUW ? , Query-Part] ?
13’ $\xrightarrow{p_6}$ UNIFY [Permissions IN App(’ B.apk ’ !) ? , Query-Part] ?
14’ $\xrightarrow{p_1}$ UNIFY [Permissions IN App(’ B.apk ’ !) ? , Analysis-Question] ?
15’ $\xrightarrow{p_{1.1}}$ UNIFY [Permissions IN App(’ B.apk ’ !) ? , SOI IN Reference FUW ?] ?
16’ $\xrightarrow{p_2}$ UNIFY [Permissions IN App(’ B.apk ’ !) ? , Flows IN Reference FUW ?] ?
17’ $\xrightarrow{p_3}$ UNIFY [Permissions IN App(’ B.apk ’ !) ? , Flows IN App(QString) FUW ?] ?
18’ $\xrightarrow{p_4}$ UNIFY [Permissions IN App(’ B.apk ’ !) ? , Flows IN App(Query-Part) FUW ?] ?
19’ $\xrightarrow{p_1}$ UNIFY [Permissions IN App(’ B.apk ’ !) ? , Flows IN App(Result) FUW ?] ?
20’ $\xrightarrow{p_{1.4}}$ UNIFY [Permissions IN App(’ B.apk ’ !) ? , Flows IN App(QString !) FUW ?] ?
21’ $\xrightarrow{p_4}$ UNIFY [Permissions IN App(’ B.apk ’ !) ? , Flows IN App(’ B.apk ’ !) FUW ?] ?
22’ $\xrightarrow{p_6}$ UNIFY [Permissions IN App(’ B.apk ’ !) ? , Flows IN App(’ B.apk ’ !) ?] ?

Figure 23: Derivation Sequence of q' (Shortened)

Example Summary (sequence of derivation rule applications)

$$q \xrightarrow{\texttt{Query}} q_{[s \leftarrow r]} = q' \xrightarrow{\texttt{Query}} q'_{[s' \leftarrow r']} = q'' \xrightarrow{\texttt{Query}} q''_{[s'' \leftarrow r'']} = q''' \xrightarrow{\texttt{Termination}} r'''$$

with

$q =$

```
UNIFY [
    Permissions IN App('A.apk' | 'PP') ?,
    Flows IN App('A.apk' | 'PP') ?
] ?
```

$$s = \boxed{\texttt{'}}\boxed{\texttt{A.apk}}\boxed{\texttt{'}}\ \boxed{\texttt{|}}\ \boxed{\texttt{'PP'}},$$
$$r = run(s),$$
$$f_r = \texttt{B.apk},$$
$$q' =$$

```
UNIFY [
    Permissions IN App('B.apk' !) ?,
    Flows IN App('B.apk' !) ?
] ?
```

$$s' = \boxed{\texttt{Flows}}\ \boxed{\texttt{IN}}\ \boxed{\texttt{App(}}\boxed{\texttt{'}}\boxed{\texttt{B.apk}}\boxed{\texttt{'}}\ \boxed{\texttt{)}}\ \boxed{\texttt{?}},$$
$$r' = run(s'),$$
$$f_{r'} = \texttt{flows.xml},$$
$$q'' =$$

```
UNIFY [
    Permissions IN App('B.apk' !) ?,
    'flows.xml' ?
] ?
```

$$s'' = \boxed{\texttt{Permissions}}\ \boxed{\texttt{IN}}\ \boxed{\texttt{App(}}\boxed{\texttt{'}}\boxed{\texttt{B.apk}}\boxed{\texttt{'}}\ \boxed{\texttt{)}}\ \boxed{\texttt{?}},$$
$$r'' = run(s''),$$
$$f_{r''} = \texttt{permissions.xml},$$
$$q''' =$$

```
UNIFY [
    'permissions.xml' ?,
    'flows.xml' ?
] ?
```

and

$$r''' = run(q'''),$$
$$f_{r'''} = \texttt{answer.xml}.$$

3.2.5 AQL-Answers: Structure

In most cases an AQL-Answer is the expected result when an analysis or operator question is derived as a query part. Therefore, most subjects of interest can be modeled in AQL-Answers. The two subjects of interest that cannot be modeled are slices and arguments. While slices are expected to be delivered in `.apk` format, arguments should lead to a textual, raw result. In both cases, conversion into an AQL-Answer is neither reasonable nor possible, because raw answers probably cannot be structured and `.apk` files must be retained to ensure the respective app's analyzability and functionality. The subjects of interest that can be represented by AQL-Answers are the remaining eight: flows, intents, intent filters, intent sinks, intent sources, permissions, sources and sinks.

The whole structure of AQL-Answers is summarized in Figure 24. We present the structure of AQL-Answers *only* on the basis of this figure and the description below. This allows us to focus on the structure without impeding readability due to lengthy technical definitions. Nonetheless, the precise structure of AQL-Answers is defined by an XML schema definition (`.xsd` file) that can be found in Appendix A.5.2. Thereby it is clearly specified (cf. ▶ **Req. 6**) through a meta-model (cf. ▷ **Req. 11**). Since this precise structure is given in form of an `.xsd` file, AQL-Answers are stored in form of `.xml` files that follow this definition.

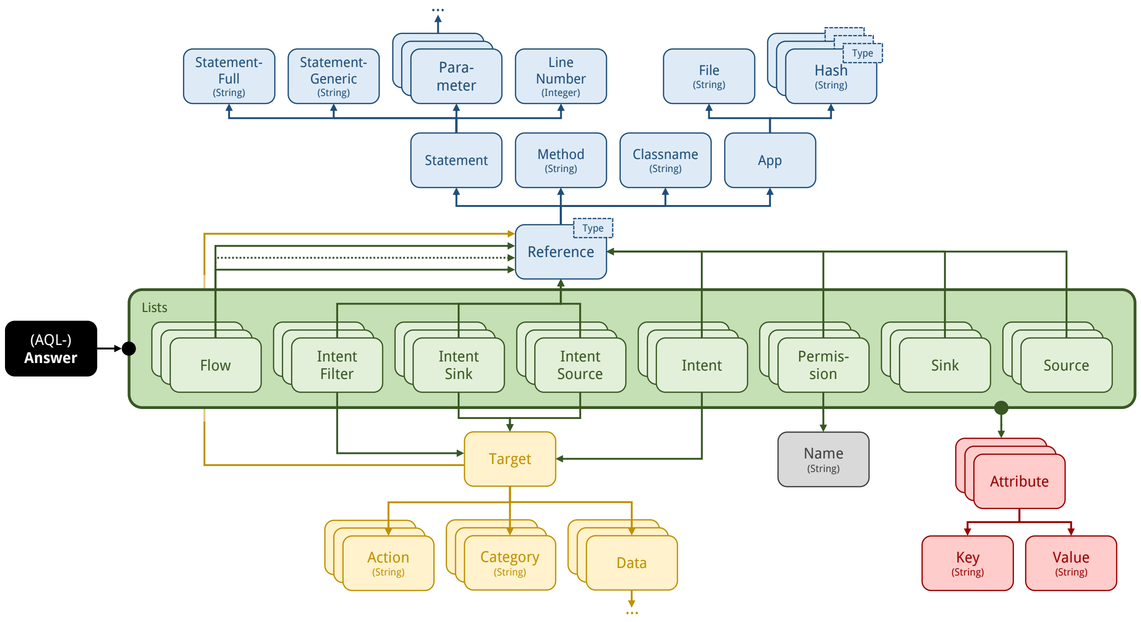


Figure 24: Visualization of the Structure of AQL-Answers

In the center of the figure above the eight subjects of interest (green) can be found. For each subject of interest an AQL-Answer holds one list in which all elements of the respective subject type can be found. For any element it is depicted what kind of sub-elements it may hold. In summary, there are four different kinds of sub-elements: References (blue), Targets (yellow), Names (gray) and Attributes (red):

- **References:** Each reference denotes a statement, method, class (classname) and an app. A statement is mainly described by two textual elements. The first refers to the full statement and the second to a generic version of this statement. For example, the full statement that refers to an intent source (e.g. `getStringExtra`) might be the complete Jimple string representation of such a statement:

```
r1 = virtualinvoke r2.<android.content.Intent: java.lang.String
        getStringExtra(java.lang.String)>("Secret")
```

and the generic one a cutout of it so that only generic information is contained (no variable identifiers or values, i.e. no notation of `r1`, `r2` or `"Secret"`):

```
android.content.Intent: java.lang.String
    getStringExtra(java.lang.String)
```

A specification of the full statement is optional. By using the Jimple representation we adhere to ▷ **Req. 3**. Furthermore, the parameters of an included function call can be described to define a statement. Each parameter can be specified by a pair of strings: its type and variable identifier or value. With respect to the example above one parameter (type: `java.lang.String`; value: `"Secret"`) could be specified. Lastly, a line number may be attached as intended by ▷ **Req. 12**.

Method and class name are simply represented by a string, for instance, if a reference refers to a statement in method `a()` of class `A` in package `pkg`, we denote `<pkg.A: void a()>` as method and `pkg.A` as class name. In accordance to ▷ **Req. 5** we only use fully qualified package/class names.

An app is described by its path to the respective `.apk` file and hashes of this file. On the one hand, the path allows a quick identification of the app on the local file system. On the other hand, the hashes allow the identification of the same app on different file systems. Implicitly, the hashes also allow to detect if an `.apk` file has been manipulated. Thus, an analysis result in form of an AQL-Answer is only valid for certain `.apk` files if the respective hashes match.

Finally, an optional attribute, that describes the `type` of reference, can be attached. A flow can consequently be denoted via two references which use this type attribute to declare where the flow starts and ends, i.e. one reference specifies the type `from` whereas the other specifies the type `to`.

- **Targets:** To specify their designated receivers *implicit* intents must describe an intent triple – *explicit* intents a component. Both can be done via a target element. For an intent triple, actions and categories are denoted as lists of strings. The data part is further structured with respect to the structure of intent triples in the context of Android. The technical details are omitted in this brief description but included in the schema (see Appendix A.5.2). To specify the component that is targeted by an explicit intent, a reference can be specified partially, i.e. app and component only.

- **Names:** A simple string argument that will only be used to specify unique identifiers such as the name of a permission.

- **Attributes:** To ensure a certain level of flexibility, any additional information may be attached to any element of an AQL-Answer. To do so, attributes come into play. Each attribute represents one key-value-pair. Key and value may hold the textual representation of any property.

With the description of these four sub-elements, the elements that represent the different subjects of interests can be described:

- **Flows:** In the context of this thesis, flow elements are most of the time used to encode taint flows. Nonetheless, any kind of information, data or control flow may be modeled by flow elements. To do so, arbitrarily many (but usually only two) references must be specified. The `type` attributes of these reference elements specify where a flow starts and ends, for example, the type could be binary (`from`, `to`) or specify a sequence of references (`1`, `2`, ...).

- **Sources & Sinks:** A single reference is sufficient to fully describe a source or a sink, since the type of element (source or sink) implicitly defines all additional information needed.

- **Permissions:** Only the name of a permission must be declared in order to unambiguously specify it. In addition, a reference can be provided that identifies the app, component, method or the exact statement that requires this permission.

- **Intents (& Intent Sinks):** A target and a reference must be specified for each intent sink. The reference must clearly define the statement that launches the intent, for instance, a call of the function `startActivity` that uses the respective intent as parameter. In case of an intent, the reference is optional – it may also be incomplete, i.e. it may only define a component, or refer to any statement that deals with the intent (e.g. its constructor call).

  In contrast, it is the other way around when it comes to the target element. It is optional in case of intent sink elements but required with respect to intent elements. In both cases the target element should describe which component(s) can be reached by the associated intent. This is either done via an intent triple (implicit intent) or a component (explicit intent) – both can be modeled via a target element.

- **Intent Filters (& Intent Sources):** Most intent filter definitions can be found in the Android manifest of an app, however, they can also be instantiated programmatically. Typically, the definitions only provide a component (app and class name) as reference, since they are defined per component. However, in case of an intent source it is mandatory that a precise statement is referenced, i.e. the statement that extracts information from an intent (e.g. `getStringExtra`).

  The target element, that describes which intent may match an intent filter or an intent source, is required in case of any intent filter specification but optional while specifying intent source elements.

These conventions, that certain elements must and must not be described, embody the difference between intents and intent sinks as well as intent filters and intent sources. The differentiation is required to clearly distinguish what a tool may and may not output. For example, AMANDROID's results include flows, intents, intent sinks and intent filters but no intent sources. The example in Appendix A.3 details the differences between intents and intent sinks, and intent filters and intent sources.

The structure of AQL-Answers as described above only allows to model analysis information (as specified in Subsection 3.1.1) in a structured way. Any other information can and should only be attached in form of attributes (cf. ▶ **Req. 4**). Thereby AQL-Answers are tailored to their application area as requested by ▶ **Req. 9** but still provide a certain amount of flexibility.

A concrete example of an AQL-Answer can be found in Appendix A.6.2.

3.3 Strategies

AQL-Queries show a wide range in terms of their complexity. While simple queries can easily be written and read by humans, more complex cooperative analysis queries are sometimes harder to handle. Furthermore, queries should be automatically adaptable with respect to features available in the app(s) under analysis (see ▷ **Req. 15**). In consequence, we propose (cooperative) strategies in this section. They allow us to automatically transform simple initial queries into complex cooperative analysis queries as follows.

> Strategies are realized through **transformation rules** that take one AQL-Query as input and output another.

We differentiate between two kinds of transformation rules: input/output and conditional transformation rules. An *input/output transformation rule* is specified by two queries, the input and the output query. Such a rule may be applied, if the actual input query matches the input query specified along with the rule. A *conditional transformation rule* is specified by only one output query. Conditional rules may only be applied for single analysis questions if a certain feature is present in this question as demanded by ▷ **Req. 15**. In both cases, the output query represents the outcome after applying the respective transformation rule. An example is explained in the following to further illustrate the functionality of transformation rules.

3.3.1 Running Example 3: Strategy Application (Part 1/2)

DidFail and IccTA embody two existing cooperative analyses in our focus area Android taint analysis. Both use a combination of FlowDroid and IC3 (or its predecessor Epicc) to enhance the intra-component taint analysis performed by FlowDroid so that it is possible to analyze inter-component scenarios. The strategy behind DidFail: use the information about intent sinks and intent sources as provided by IC3 to determine matching pairs which stand for ICC flows, then connect these flows with the intra-component taint flows determined by FlowDroid.

With the AQL at hand, we can adopt this strategy and formulate a query q_{coop} that targets the same cooperation (let `A.apk` stand for an arbitrary app that involves ICC):

```
1  CONNECT [
2      Flows IN App('A.apk') ?,
3      CONNECT [
4          IntentSinks IN App('A.apk') ?,
5          IntentSources IN App('A.apk') ?
6      ] ?
7  ] ?
```

In Line 2 we ask for flows. Once this analysis question is answered by FlowDroid, it will give us an AQL-Answer, as indicated by the ending symbol ($\boxed{?}$), that holds the detected intra-component taint flows. Similarly, we ask for intent sinks and intent sources in Line 4 and 5. These two analysis questions may be answered by IC3. The connect operator that surrounds these two lines is meant to call another tool that (1.) takes the provided information about intent sinks and intent sources, (2.) determines matching pairs and finally (3.) provides an AQL-Answer that holds the concluded ICC flows. The second connect operator that surrounds the whole query then combines these ICC flows with the intra-component flows found by FlowDroid. Since the connect operator is required to be implemented by any system using the AQL, it must be available (see Subsection 3.2.2).

Finally, we expect an AQL-Answer that holds inter-component taint flows (if there are any) as response to the whole query.

Anyway, one question arises: why do we have to formulate such a complex query if we just want to ask for inter-component taint flows in one app? The answer: we do *not* have to! With the help of two transformation rules[17], we can instead just ask the following query q_{simple}:

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

The first transformation rule is used to get information about the features used by the app under analysis. Thus, it automatically inserts an analysis question that asks for features

```
Arguments IN App('A.apk') .
```

into the input query only if it has the structure of q_{simple}. The result is $q_{features}$:

```
Flows IN App('A.apk') FEATURING
    Arguments IN App('A.apk') .
?
```

The associated input/output transformation rule is specified via two AQL-Queries. The first query specifies the structure the actual input query must have. It is defined as follows by making use of a variable placeholder (%FILE_1%):

```
Flows IN App(%FILE_1%) ?
```

The second query defines the outcome and uses the same variable placeholder twice:

```
Flows IN App(%FILE_1%) FEATURING
    Arguments IN App(%FILE_1%) .
?
```

By matching the input query of the transformation rule with the actual input query q_{simple}, we infer that %FILE_1% is equal to `'A.apk'`, which then can be used to determine the actual output query $q_{features}$ by replacing the variable placeholder.

Once the inner question of $q_{features}$, which is asking for arguments, is answered by a tool that determines the features of an app, we will receive a raw answer as response (see ending symbol $\boxed{.}$). This allows us to directly integrate the answer into the query (cf. ▷ **Req. 2**). Let us assume that only the feature `ICC` is determined for `A.apk`. Consequently, we get the query q_{icc} by replacing the `Arguments`-query with the raw content of the given answer:

```
Flows IN App('A.apk') FEATURING 'ICC' ?
```

Now the second transformation rule comes into play. It is a conditional transformation rule that is only applied if the feature `ICC` is present (cf. ▷ **Req. 15**). The output query is specified including predefined variable placeholders, for example, the variable placeholder %APP_APK_IN% may match the app that is given as part of a reference after the keyword $\boxed{\texttt{IN}}$. Which predefined placeholders are available is dependent on the individual implementation – more details, in regard to the AQL-System, are provided later on (see Subsection 3.4.1). Accordingly, the second rule is defined by the following output query:

```
CONNECT [
    Flows IN App(%APP_APK_IN%) ?,
    CONNECT [
        IntentSinks IN App(%APP_APK_IN%) ?,
        IntentSources IN App(%APP_APK_IN%) ?
    ] ?
] ?
```

[17] Both transformation rules are denoted in Appendix A.5.5 in the format used by the AQL-System.

In this example `%APP_APK_IN%` assumingly also matches `'A.apk'`. Once the rule is applied on q_{icc} we get the final query q_{coop}.

For brevity, we neither formally define transformation rules nor do we provide more examples. However, several examples are implicitly mentioned while introducing novel cooperative strategies in the evaluation chapter (see Chapter 5).

3.4 AQL-System

The original system that implements the AQL is the system that shares the language's name: the AQL-System. From an abstract perspective it can be viewed as a black box (depicted in Figure 25). This black box takes an AQL-Query as input from the user

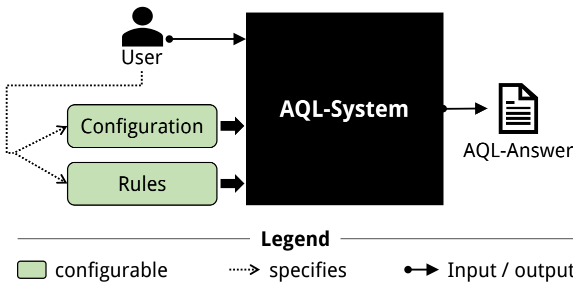


Figure 25: AQL-System Overview

and outputs an AQL-Answer as response.[18] The system on its own comes with a couple of built-in tools, the so-called *default tools* (see Subsection 3.4.3). Without configuring the system, only queries that can be answered by these default tools can be answered at all. Hence, before diving deeper into the functionality of the black box or the AQL-System, we take a closer look at the two options to configure it. On the one hand, the system can be configured to use arbitrary analysis tools, operators and preprocessors via its *configuration*. The configuration also holds information about converters that may be required to translate a tool's result into an AQL-Answer and information about the environment of the system, i.e. hardware information (e.g. maximal usable memory) or the availability of resources such as the Android SDK. On the other hand, *rules* can be provided to an AQL-System. They specify which (cooperative) strategies in form of transformation rules (input/output or conditional) are available. After describing the configuration options, the workflow of the AQL-System is described as it is implemented without going into detail about the implementation itself. The implementation is fully accessible as an open source project on Github [114]. Several tutorials and a user manual can also be found in this project. Lastly, the running example started before is continued to illustrate the system's workflow.

3.4.1 Configuration & Rules

In this subsection the options to configure an AQL-System are briefly described, especially technical details, that are not important to understand the concepts behind the AQL-System, are omitted.

The configuration of an AQL-System must be provided in form of an `.xml` file that follows the schema definition (`.xsd`) given in Appendix A.5.3. Figure 26 shows the structure demanded by the schema file. Each configuration comes with three environment describing elements (gray). The first element must be provided in form of an integer number which describes how much memory at maximum (in gigabytes) may be used by the tools when run via the associated AQL-System. The other two elements should denote paths to two directories (build tools and platforms) of the Android SDK.

[18]Depending on the type of query it might also be a raw ($\boxed{.}$) or a file ($\boxed{!}$) answer. For simplicity this is not mentioned in the following anymore.

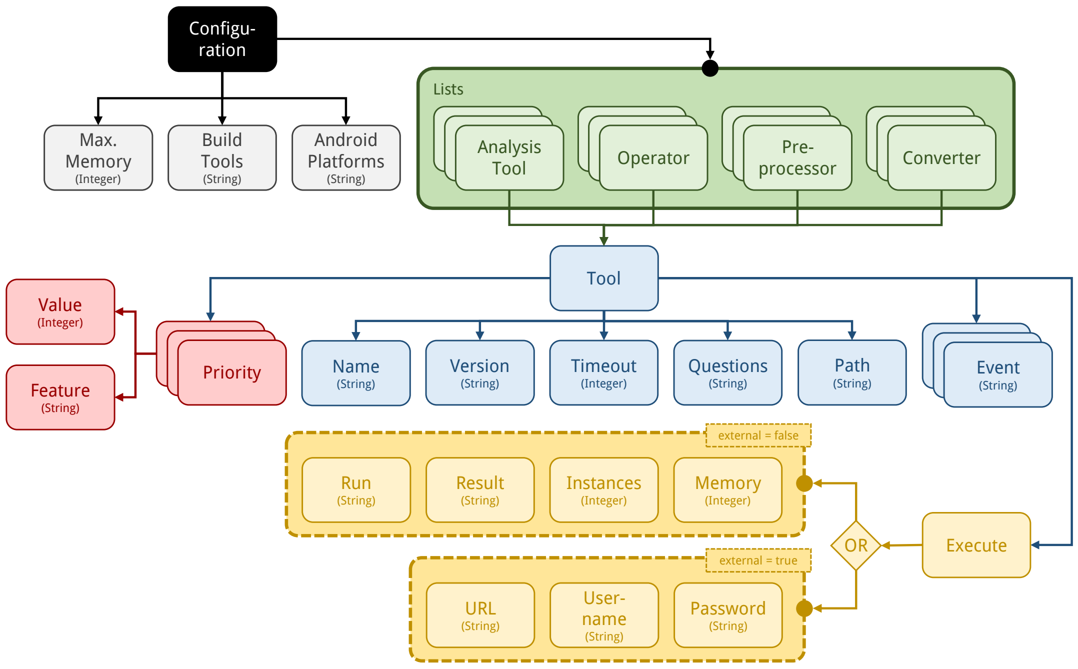


Figure 26: AQL-System Configuration

The **build tools** represent a collection of command line tools that are required to compile, sign and finally build an Android app in form of an `.apk` file.

Among others, the tools ZIPALIGN and APKSIGNER are included. They have to be executed in sequence in order to sign an app. Signing an app describes the process of attaching a digital signature to an `.apk` file. A signature uniquely identifies a developer. Only signed apps can be executed on Android devices.

The Android **platforms** (also called **platform directory** or **platform files**) provide API-specific materials and development instruments. Most importantly, API-specific versions of the Android library (`android.jar`) are contained.

Each of these `.jar` files holds Java classes and interfaces that implement how to compose or build apps, how to access hardware or software components typically available in and on Android devices and various other information. However, in the end it is a Java library, hence, it can be used as any other library. From here on we will refer to this library as *Android library*. For example, to construct an intent the class `android.content.Intent`, which belongs to the Android library, must be instantiated. In consequence, most analysis tools require access to the Android platforms in order to analyze an app, since the Android library may hold a (often large) portion of an app's code that is not available directly in the respective `.apk` file.

The available analysis tools, operators, preprocessors and converters can be configured by specifying four lists (green elements in Figure 26). Thereby, arbitrary tools can be configured as demanded by ▷ **Req. 6**. The tools are treated as black boxes, hence, their inputs and outputs must be declared in the configuration (cf. ▷ **Req. 13**). Each list holds arbitrarily many tool elements (blue). Each "Tool" element clearly identifies the associated tool by denoting its name and version. The value specified through the "Timeout" sub-element specifies the maximal execution time a tool may use before it is canceled – this value is optional as a timeout can also be specified on a global level. The sub-element entitled with "Questions", describes what the tool may be used for. For each type of tool, different information must be provided as detailed in the list below:

- **Analysis tools:** the subjects of interests a tool supports are listed here.
- **Operators:** the associated operator's name is denoted.
- **Preprocessors:** the preprocessor's keyword is provided.
- **Converter:** an identifier (name and version – the latter is optional) of the analysis tool, for which the respective converter shall be used, is named.

The "Path" sub-element describes where (in which directory) a specific tool or event is executed. How to execute a tool is described by the "Execute" sub-element (yellow). A tool can be executed locally or remotely. A boolean flag (`external`) determines how to execute a specific tool.

> **Internal tools** (`external = false`) are executed locally, **external tools** (`external = true`) are executed remotely.

For internal tools four elements have to be specified as sub-elements of the tool's execute element:

- **Run:** Determines the command to execute the respective tool. It is executed in the directory at which "Path" points. In the following, we refer to this command as a tool's *run command*.
- **Result:** Denotes where (on which path) the result of the tool will become available.
- **Instances:** The respective integer number specifies how many instances of the tool can be executed concurrently (0 stand for arbitrarily many).[19]
- **Memory:** Each instance of the respective tool should not use more memory than specified by this integer number (in gigabytes).

External tools are not executed on the local system, hence, we do not need to know how to run them, instead we need to know where to find them and how to access them. For that purpose three different sub-elements must be specified:

- **URL:** The URL of an AQL-WebService (see Subsection 3.4.5).
- **Username & Password:** Valid credentials to access this webservice.

Up to five "Events" can be attached to each tool. These five events are triggered before and after a tool is run (`onEntry`, `onExit`), when a tool finishes its execution (`onSuccess`, `onFail`) or when it is aborted (`onAbort`). The respective strings denote the commands to execute. They are also executed in the directory specified in the "Path" sub-element.

Lastly, each tool comes with at least one "Priority" sub-element (red). These elements assign a certain priority to a tool that may decide which tool is used to answer a question if multiple tools are able to do so. Whenever a feature is assigned to a priority element, the respective priority's value is only considered if this feature is present in the current analysis question. How exactly the priority is computed, is explained when the tool selection process is detailed (see Subsection 3.4.2).

[19] A few tools produce (intermediate) results that are always stored in the same file. Because of that they cannot be run in parallel as each instance would overwrite the result of another which often leads to data races and invalid results.

The configuration of tools together with the elements describing the environment allow us to configure all universal and tool-specific inputs as requested by ▷ **Req. 1**.

The (cooperative) strategies or (transformation) rules available in an AQL-System are specified in another `.xml` file that follows another schema definition (see Appendix A.5.4). Figure 27 illustrates the structure of such a file. It mainly consists of a list of rule elements

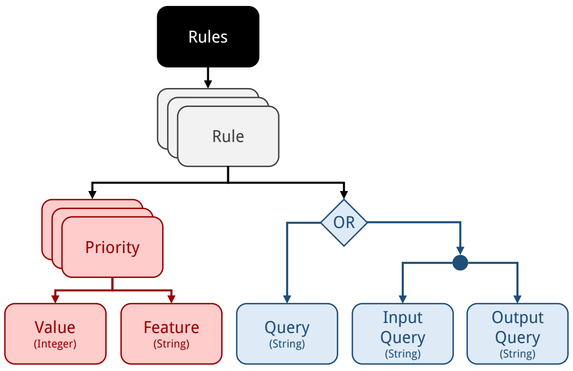


Figure 27: AQL-System Strategies / Transformation Rules

(gray). Each rule in this list might be an input/output or a conditional transformation rule. The required input and output queries are specified as strings (blue). Similar to each tool in the configuration of an AQL-System, priorities can be attached to each rule. Thus, if two or more rules are applicable the rule with the highest priority is chosen.

Variables & Placeholders Various variables may be used inside the different elements of a configuration or a rules file. A complete list of all *predefined variables* (and *placeholders*) and their meaning can be obtained from the AQL-System's documentation [117, 118]. In addition, *custom variables* refer to values set in a query via key-value-pairs after the keyword `WITH`. The query below, for example, assigns the value `SourcesAndSinks.txt` to the custom variable `%SourcesAndSinks%` that is implicitly declared.

```
Flows IN App('A.apk') WITH 'SourcesAndSinks' = 'SourcesAndSinks.txt' ?
```

Not least because of custom variables ▶ **Req. 2** is met – all execution-specific inputs can be modeled via the AQL. To foster brevity, no further details are provided here, however, for explanation purposes let us have a look at one example.

The "Run" element of an internal tool may hold the following command to run FlowDroid (2.9.0):

```
java -Xmx%MEMORY%g -jar soot-infoflow-cmd-2.9.0-jar-with-dependencies.jar
        -a %APP_APK% -p %ANDROID_PLATFORMS% -s %SourcesAndSinks% -o
                results/%APP_APK_FILENAME%_result.xml -ol
```

Five variables appear in this command. Once the command is executed, variable `%MEMORY%` is replaced by the memory value given for FlowDroid (2.9.0) in the same configuration. `%APP_APK%` refers to the `.apk` file of the app under analysis and is replaced accordingly. The variable `%ANDROID_PLATFORMS%` refers to the Android platforms specified as environmental information in the AQL-System's configuration. `%SourcesAndSinks%` stands for the value of the associated custom variable (e.g. `SourcesAndSinks.txt`). Lastly, `%APP_APK_FILENAME%` refers to the filename of the `.apk` which is often required to identify a tool's result file.

3.4.2 Workflow

In this subsection we open up the black box that represents the AQL-System. The six steps that embody the workflow inside this black box are explained in detail. Figure 28 shows all six steps (**1.** – **6.**) and provides an overview of the whole workflow.

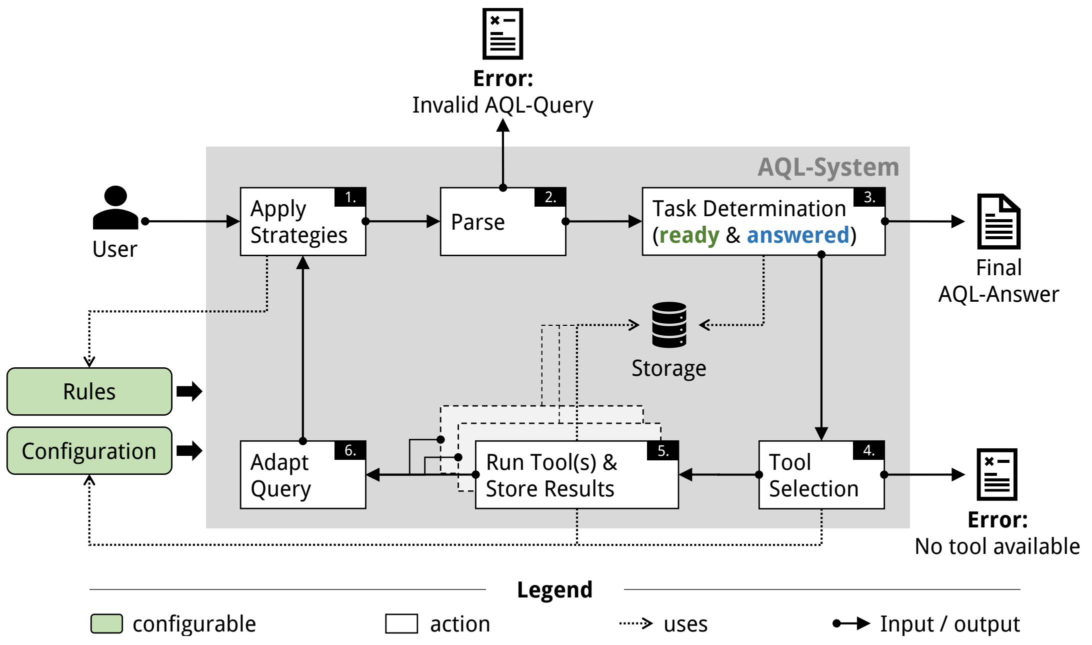


Figure 28: AQL-System Workflow

Close to the center of this figure an element can be found labeled with "Storage". As the name suggests, this element stores data, more precisely, arbitrary tool results. It allows us to track if a query or a (query-)part has been answered before, and if a tool has been run before for the same analysis target and under the exact same configuration, i.e. the same tool with the same launch parameters – the same run command after replacing all variables.[20] In case of an external tool an artificial run command is used to achieve the same effect. In the end, a tool which has been executed before is not executed again, instead the previously computed result is remembered. Even if the same run command is scheduled upon two distinct queries, the AQL-System will simply remember the result but probably take different elements of it into account.

Initially the user must input an AQL-Query to trigger the AQL-System's workflow:

1. **Apply Strategies**

(*Input*: AQL-Query q, *Output*: AQL-Query q', *Uses*: Rules R)

In order to find out if a cooperative strategy can be applied, more precisely, if the current query or a (query-)part of it can be transformed, the system checks whether there is at least one rule that has a priority greater than 0 and is applicable. The priority of a rule is equal to the sum of all priorities specified for it that have either no feature assigned or a feature that appears in the respective query or a part of it.[21] An input/output transformation rule is *applicable* if the rule's input query matches the whole query or a part of it. A conditional transformation rule is *applicable* if its

[20]Technically, the comparison of run commands happens on a level such that, for example, moving a file provided as launch parameter does not impair the comparison.

[21]An example can be found in Appendix A.5.5.

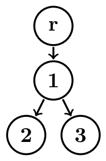
priority is greater than 0 for the whole query or a part of it. In both cases the part for which the rule becomes applicable is called *matching part* – even if it stands for the whole query. The applicable rule with the highest priority is determined and applied first. Thereby the matching part is replaced by the output query specified for the applied rule. While doing so, the included variables are resolved. On basis of the transformed query this process is repeated until no more rules can be applied. If there are no applicable rules available, the given query is simply forwarded to the next step without transformation.

2. Parse
(*Input*: AQL-Query q, *Output*: Tasktree t, *Uses*: $-$)
The query given as input is mapped to a tasktree while parsing it. A *tasktree* is a data structure that models the different questions contained in a query. Considering the grammar behind AQL-Queries, it is similar to a derivation tree for the input query that only includes the nodes that represent the application of a crucial production rule (a question). Technical details are skipped for brevity, but an example is given:

The query

```
CONNECT [
    IntentSinks IN App('A.apk') ?,
    IntentSources IN App('A.apk') ?
] ?
```

would be mapped to the following tasktree



where ⓡ represents an artificial root node, ① stands for the operator question (`CONNECT [ ... ] ?`) and ②, ③ for the two analysis questions dealing with intent sinks and intent sources.

If the query cannot be parsed, since it is not valid with respect to the AQL-Query grammar, the AQL-System outputs an error that points at the erroneous part of the query and aborts its computation.

3. Task Determination
(*Input*: Tasktree t, *Output*: Tasktree t', *Uses*: Storage S)
During this step it is determined which parts of the query are answered and which parts are ready with respect to the definitions given in Subsection 3.2.3 (Page 65). Analogously, a node in a tasktree is answered, if the respective question has been answered before according to the AQL-System's storage; a node is ready, if it is a leaf or if all its children are answered. The artificial root node counts as answered once all its children are marked as answered. In the latter case the computation stops successfully and the final answer is output.

4. Tool Selection
(*Input*: Tasktree t, *Output*: Tasktree t', *Uses*: Configuration C)
During this step a tool is assigned to each node of the tasktree if possible. To

do so, all tools given in the system's configuration are filtered until only a set of tools, which are able to answer the respective question, is left. First, the type of question is used to filter: obviously an analysis question must be answered by an analysis tool specified in the configuration, an operator question by an operator and a preprocessor question by a preprocessor. Then the content of the question is used to filter further: the subject of interest must match a subject of interest listed in the configuration ("Question" element) of an analysis tool. Similarly, the operator name or preprocessor keyword must match respectively in case of an operator or a preprocessor. Thereafter, each tool in the set of filtered tools is able to answer the associated question.

To decide which tool is ultimately selected from this set and assigned to the tasktree node, each tool's priority to answer the current question must be taken into account. The priority is determined just as it is done in case of transformation rules (cf. Step **1.**): the sum of all given priorities specified for the respective tool in the system's configuration is calculated. However, a priority is only summed up, if no feature or a feature, which is also present in the respective question, is attached.[22] Thus, the actual priority can only be determined if the respective question is complete, i.e. there are no information missing, for example, the features mentioned in an analysis question are known and must not be determined via a yet unanswered question.

If there are two tools with the same priority, the following tiebreakers are used one after another:

- Internal tools are favored over external ones.
- Tools for which more variables, mentioned in their respective run commands, can be resolved are preferred.
- If less custom variables, that have to be specified in a query, are involved in tools' run commands, they are selected.
- On another tie, tools that use more default variables in their run commands are favored.
- Ultimately, tools mentioned first in the configuration are preferred.

If the current question names a specific analysis tool after the keyword `USES` (or `USING`), this particular tool is preferred independent of its actual priority (cf. ▷ **Req. 10**). This selection process shows that tools are chosen with respect to execution-specific inputs (SOI, features, variables, ...) present in the respective questions, thus, ▶ **Req. 3** is met.

If the tasktree contains a node that is ready but has no tool assigned to it, the computation is aborted and an error is output that states that there is no tool configured to answer the respective part of the query.

5. Run Tool(s)

(*Input*: Tasktree t, *Output*: Tasktree t', *Uses*: Configuration C, Storage S)

In contrast to what has been explained in the semantics subsection (3.2.3), tools are not executed one after another by the AQL-System. Instead all tools that are assigned to nodes which are marked as ready are executed concurrently unless the maximal usable memory is exceeded or a tool which is needed to be run in parallel cannot be run in parallel (cf. ▷ **Req. 16**). In case of external tools, it is up to the

[22] An example can be found in Appendix A.5.5.

remote system when a tool can be executed. Locally it is assumed that external tools do not require any memory and can always be run in parallel. The memory an AQL-System should maximally use is specified as one of the environmental values in a system's configuration. To detect if a certain tool execution would possibly exceed this limit, the AQL-System keeps track of the memory that may be occupied by tools that are currently running. To do so, it takes the memory value specified for any internal tool (in the configuration) into account. The AQL-System also records how many instances of a certain tool are currently running and prohibits that this number exceeds the number specified in the respective tool's configuration ("Instances" element).

Whenever the execution of a tool finishes, the produced result is attached to the tasktree's node and stored in the system's storage. Then the next step is triggered. Note that Step **6.** consequently may be triggered multiple times, if multiple tasktree nodes were ready and multiple tools were executed. It is also possible that Step **6.** is not triggered since no tool was executed, however, this can only be the case if another tool is still running that will trigger this step once it finishes.

6. Adapt Query

(*Input*: Tasktree t, *Output*: AQL-Query q, *Uses*: $-$)

By a lookup in the map, that maps tasktrees to queries and vice versa, the original query is recovered. Then the query is adapted according to the derivation rules defined in the semantics subsection (see Subsection 3.2.3) and with respect to the answers available in the tasktree.

Note that a new tasktree will only be computed during Step **2.** if a strategy is applied on the start of the next iteration (Step **1.**). More precisely, if a transformation rule has become applicable by the query adaptation made with respect to the new results. Otherwise, the current tasktree is still valid to represent the current query and parsing the query can be skipped.

These six steps are repeated until an error occurs or the final answer has been computed. Before we explain this whole workflow on a concrete example, a few details about the implementation of the AQL-System are presented (see Subsection 3.4.3), since these are partially required to fully understand the example (see Subsection 3.4.4).

3.4.3 Implementation Details

For brevity, we only explain implementation details that represent important concepts used in the following. A comprehensive documentation of the AQL-System's implementation along with tutorials and further information is publicly available online [114].

Parser and Data Structure JavaCC [156] is used to generate the parser, that is employed in Step **2.** (Parse) of the AQL-System's workflow, directly on the basis of the AQL-Query grammar. The Java classes representing the exact structure of AQL-Answers are generated from the schema that defines them with the help of JAXB [157]. Thereby, the AQL-System strictly follows the language's definition and any adaptation of the language can easily be integrated into the implementation (cf. ▷ **Req. 9**).

Default Tools The AQL-System comes with several built-in tools (the so-called *default tools*), including the five necessary operators (unify, intersect, minus, connect, filter). A list of all default tools can be found in Table 7. The table is sorted by the different types of *tools* (analysis tools, operators, preprocessors, converters). Note that converters for the most prominent analysis tools are shipped with the AQL-System.

Table 7: Default Tools of the AQL-System

Analysis Tools		
Name	**Supported Subjects of Interest**	**Description**
FeatureFinder	Arguments	Provides a raw answer that lists features found in a reference. To do so, it is checked whether Jimple statements, that indicate the usage of certain features, can be found in the reference. The statements and features to look for are defined in a configurable list.
IntentInformationFinder	Intents, IntentFilters, IntentSinks, IntentSources	Finds information about intent filters by parsing the target's Android manifest and gains information about references involved with intents, intent filters, intent sinks and intent sources by comparison of all Jimple statements with a predefined, extendable list of such.
PermissionFinder	Permissions	The manifest is parsed to get to know which permissions are used by the referenced app.
SourceSinkFinder	Sources, Sinks	A predefined and adaptable list of Jimple statements that categorize certain method calls as sources and sinks is used in order to detect instances of such in the targeted reference.

Operators	
Name	**Description**
UNIFY	Collects all information from different AQL-Answers and puts it into one.
INTERSECT	Extracts the information that appears in all provided AQL-Answers.
MINUS	Removes all information given in the first AQL-Answer if they appear in any other.
CONNECT	Works as UNIFY, however, it additionally computes transitive flows and flows that can be determined by connecting intent sinks with intent sources. Additionally, incomplete intent sources (only naming a component) are completed by matching them with intent filters (naming the same component). Lastly, it adds backward flows whenever there is an intent sink connected to an intent source, e.g. from `setResult(...)` in the intent source's component to another intent source in the `onActivityResult(...)` method of the intent sink's component.
CONNECT∼	Same as CONNECT but while connecting intent sinks with intent sources only the intent's and intent filter's action is taken into account (category and data are ignored).

FILTER	This operator can be used in various ways: • When used without specifying a specific filter, it outputs the input set, but beforehand it removes all elements (permissions, intent sinks and intent sources, ...) whose reference does not appear in any flow contained in the answer. Elements without a reference are kept. • When an subject of interest is given as filter, the operator filters out all elements of the selected subject. • When using a name-value-pair as filter, only the elements that have this name-value-pair attached as attribute are kept. • A reference can also be used as filter. Only elements that refer to this reference are kept in this case. (Takes only one AQL-Answer as input.)
SIGN	Automatically signs the referenced app. (Uses a configurable signature/key and takes only one AQL-Answer as input.)
SIMPLIFY	Removes classes that represent common compatibility libraries from the referenced app. As second argument a file, that holds a list of package or class names, can be provided. Otherwise, a configurable default list is used. (Takes up to two arguments as input.)
SIMPLIFY∼	Removes all classes that do not belong to the referenced app's main package.
TOAD	Converts the sources and sinks given in an AQL-Answer into a source and sink list as it is used by Amandroid. (takes only one AQL-Answer as input)
TOFD	Same as TOAD but compatible with FlowDroid instead of Amandroid.

Preprocessors	
Keyword	**Description**
SIMPLIFY	Removes classes that represent common compatibility libraries from the referenced app. A configurable list of package or class names is used to identify the classes to remove.
SIMPLIFY∼	Removes all classes that do not belong to the referenced app's main package.

Converters		
Converter for	**Supported Versions**	**Remarks**
Amandroid	3.1.2, 3.2.0, 3.2.1	Different converters are used for older and never versions.
DIALDroid	September 2017	A database configuration must be available such that the converter gains access to DIALDroid's database.
DidFail	March 2015	—
DroidSafe	June 2016 (Final)	—
FlowDroid	April 2017 (Nightly), 2.7.1, 2.9.0, 2.10.0	Different converters are used for older and never versions. From 2.7.1 on the `.xml` output format of FlowDroid is required.
HornDroid	0.0.1	—
IC3	0.2.1	—
IccTA	February 2016, 2.9.0, 2.10.0	Analogous to FlowDroid different converters are used and the `.xml` output format is required for version 2.9.0 and 2.10.0.

The default tool PermissionFinder shares its name with a non-default tool presented before (see Subsection 2.4.2). Please note that the default version is less precise as it only identifies permissions listed in the manifest without referencing statements requiring this permission. If we refer to the tool PermissionFinder in the following, we always refer to the non-default one.

Retry Mechanism While describing the semantics of the AQL we assumed that tools do not fail, however, in reality tools may fail for various reasons, for example, a tool may crash due to a bug in its implementation or because it attempts to use more memory or time than given. To cope with (unexpected) fails, a retry mechanism is implemented in the AQL-System. If the tool with the highest priority fails, the tool with the next highest priority takes over (cf. ▷ **Req. 16**). If a tool fails and there is no other tool, which is able to answer the respective question, an error is output.

Expandability The AQL-System is available as an open-source project (on Github [114]) and usable as a library (e.g. via Maven [116]).Thereby other projects may use the AQL-System to interact with arbitrary tools, especially analysis tools. To encourage and simplify the usage and extension of the AQL-System, interfaces to extend and use the implementation have been integrated. For instance, tools and especially converters can also and easily be added as default tools, and events as well as hooks allow the integration of custom code (tutorial available online [115]). In conclusion, the AQL-System can be interpreted as extendable, thus, ▷ **Req. 9** is met.

Along with the implementation details explained above, everything necessary to understand the following example has been introduced.

3.4.4 Running Example 3: Workflow Run-Through (Part 2/2)

The example presented in this subsection exemplifies the workflow of an cooperative analysis realized via the AQL-System. The cooperative strategies introduced in Section 3.3 are reused and we assume that the AQL-System is configured to use FlowDroid and IC3 in addition to the default tools implemented in the AQL-System.

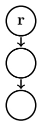
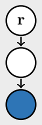
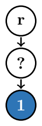
The example is mainly visible in the long table below (see Table 8). Its four columns show a step id, the current workflow position (with respect to Figure 28 – Page 82), and the current query and tasktree. For better visibility only changes are shown in the table.

The initial query is asking for flows in a single app and initially no tasktree is available (see Step 1). Again we assume that the app (`A.apk`) only implements one feature which is ICC. To get to know this feature a first transformation rule is applied (see Step 2). An analysis question that asks for arguments, in this case features, is inserted into the initial query. Since there is no other rule that can be applied, this query is given to the next step. By parsing the query a first tasktree is created. It consists of three nodes. The artificial root node (ⓡ), one node for the flows question and a third node for the arguments question – the latter two are depicted as unlabeled nodes (◯). During the task determination workflow step (**3.**) the questions (or tasks of the tasktree) that are ready, are determined. In this case (Step 4) only the last task (●: the only leaf of the tasktree), that represents the arguments question, is ready. Step 5 in the table shows the outcome after two workflow steps (**4.**, **5.**), since **5.** does not change the query or the tasktree. However, during tool selection (**4.**) the default tool for argument questions is assigned to the only tasktree node that was marked as being ready (❶: FeatureFinder). No tool could be assigned to the node representing the flows question since it is not complete yet (?). More precisely, the answer to the arguments question may influence which tool is picked to answer the flows question. After executing the FeatureFinder the result, given in form of a raw answer (`'ICC'`), is directly integrated into the query (see Step 6; cf. ▷ **Req. 2**). Thereafter another workflow iteration starts as marked in the table by the repetition symbol (↻).

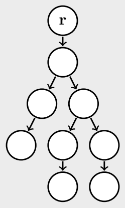
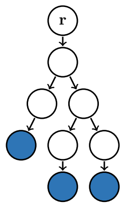
Table 8: Workflow Run-Through Overview (Changes only)

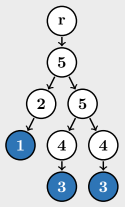
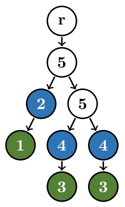
Step (id)	**Workflow Position**	**Query**	**Task-tree**
1	Initial	`Flows IN App('A.apk') ?`	–

2	**1.**	`Flows IN App('A.apk') FEATURING`
		`    Arguments IN App('A.apk') .`
		`?`
			–

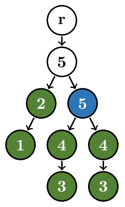
3	**2.**	(see above)	
4	**3.**	(see above)	
5	**4.**, **5.**	(see above)	
6	**6.**	`Flows IN App('A.apk') FEATURING 'ICC' ?`	(see above)

7	↻ **1**	`CONNECT [`
		`    Flows IN App('A.apk') ?,`
		`    CONNECT [`
		`        IntentSinks IN App('A.apk') ?,`
		`        IntentSources IN App('A.apk') ?`
		`    ] ?`
		`] ?`
			(see above)

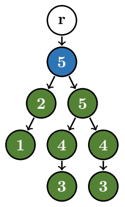
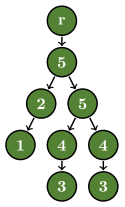
8	**2.**	(see above)	
9	**3.**	(see above)	

Step (id)	Workflow Position	Query	Task-tree
10	4., 5., 6. ↻ 1., 2.	(see above)	
11	3., 4., 5.	(see above)	

12	6.
	↻
	1., 2.
		`CONNECT [`
		`    ’flows.xml’ ?,`
		`    CONNECT [`
		`        ’intentSinks.xml’ ?,`
		`        ’intentSources.xml’ ?`
		`    ] ?`
		`] ?`
			(see above)

13	3., 4., 5.	(see above)	

14	6.
	↻
	1., 2.
		`CONNECT [`
		`    ’flows.xml’ ?,`
		`    ’iccFlows.xml’ ?`
		`] ?`
			(see above)

15	3., 4., 5.	(see above)	
16	6. ↻ 1., 2.	`’final.xml’ ?`	(see above)
17	3., Final	(see above)	

Because of the attached feature (cf. ▷ **Req. 15**), another transformation has become applicable, consequently, the query is transformed again (see Step 7). By parsing this new query a much bigger tasktree is constructed (see Step 8). Since it does not hold any incomplete questions/nodes, a tool can be assigned to each of the nodes (see Step 10). All remaining steps, depict four more iterations of the workflow. Progressively all the tools of the tasks that become ready are executed and more and more tasktree nodes become answered. In the first batch (Step 10), FlowDroid (❶) and IC3 (❸) are executed. Note, IC3 is only executed once even though there are two nodes labeled with 3, as the AQL-System recognizes that the run command is the same in both cases. After the execution of this batch the query cannot be adapted yet. The tools' results are only available in tool specific formats. The results have been stored in the AQL-System's storage such that the respective tasktree nodes will be marked as answered (❶, ❸) by the next task determination (**3.**).

During the next iteration the converters for FlowDroid (❷) and IC3 (❹) are executed. With respect to the two answers expected from IC3, the AQL-System will only gather intent sinks in one answer in intent sources in the other. Finally, at the end of this iteration the query can be adapted further (see Step 12). To do so, the derivation rules specified in the semantics subsection are applied (see Subsection 3.2.3) as visible in the table whenever a question is replaced by a filename in the query. Two more iterations are required to answer the remaining two operator questions (❺: CONNECT). Finally, since the artificial root node is marked as answered (ⓡ) in Step 17, the computation ends and the final answer, in this case an AQL-Answer (`final.xml`), is output.

3.4.5 AQL-WebService

The AQL-System interacts with so-called internal tools, installed on the local system, and external tools, accessed remotely. Up to this point we have not specified how an external tool can be accessed or how it is run. In an AQL-System's configuration each tool comes with an execute element that holds information about how to run it (see Subsection 3.4.1). In case of an external tool a URL, a username and a password have to be specified. This information tells the system where to find the external tool (URL) and how to access it (username, password). Thus, whenever an AQL-System reaches out to an external tool it puts together a package consisting of the given credentials and the analysis task. The *analysis task*, in this context, is modeled by another AQL-Query and the files used in it. This query is constructed with respect to the type of tool to execute:

- **Analysis tools:** The query consists of the respective analysis question. Local files used in this question (e.g. the app – `A.apk`) are replaced by placeholders (e.g. `%FILE_1%`).

- **Operators:** The associated operator question is employed as query. At this point all arguments the operator takes must be available. For instance, any analysis question used as an argument must be answered. The local files that represent the respective answers are again replaced in the query and sent along with it.

- **Preprocessors, Converters:** An artificial query is constructed. Its only purpose is to tell the receiver that the provided file should be preprocessed or converted with respect to the parameters (e.g. features) present in this query.

This package is then sent to the external tool, more precisely, an AQL-WebService that checks the attached credentials and runs the respective tool. The placeholders in the query are resolved again once the files have been transmitted to this webservice and have become available on its local file system. To send the package a HTTP(S) POST request is formulated and sent to the specified URL. Figure 29 illustrates the AQL-System as

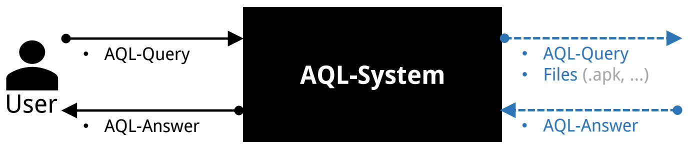


Figure 29: The AQL-System's REST API

a black box again. This black box takes an AQL-Query as input from the user and responds with an AQL-Answer. To do so, it may execute external tools. The blue dashed arrows on the right hand side of the figure depict the associated communication with a webservice over an REST API interface. The other ends of these arrows are depicted in Figure 30. The figure also depicts the AQL-System as a black box, however, this time

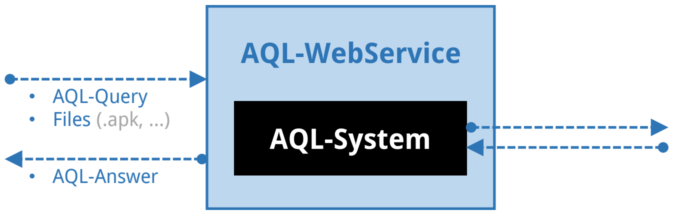


Figure 30: AQL-WebService Wrapper

it is encapsulated in an AQL-WebService. Accordingly, an AQL-WebService can be interpreted as a wrapper for an AQL-System. This wrapper allows to reach out to the wrapped system via REST API requests. The outgoing edges on the right hand side of the figure illustrate that the wrapped system may reach out to another webservice. This way webservices can be stacked and networks of connected AQL-System can be constructed.

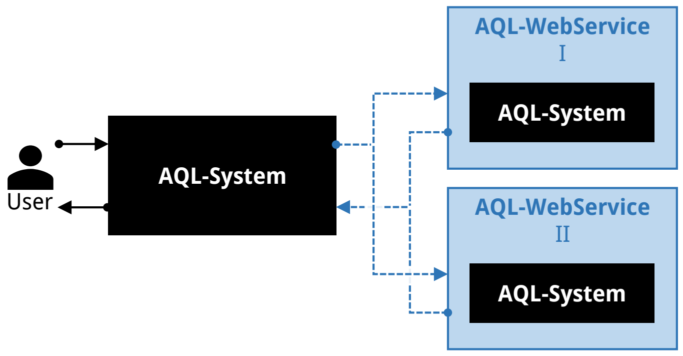


Figure 31: The CoDiDroid Instance

One example instance of such a network is known as CoDiDroid [75, 123]. This instance is depicted in Figure 31. It consists of an AQL-System (front end) that is used as interface by the user and two AQL-WebServices (back ends). The first webservice (AQL-WebService I) executes several analysis tools. The second webservice (II) only hosts a single analysis tool, due to very specific environmental requirements, this tool must be executed on another system. In the CoDiDroid study all tools were deployed on a Linux virtual machine. However, one dynamic tool, namely PIM, could not be deployed in a virtual space, since it requires to efficiently run an Android emulator which is only doable with hardware support that is not fully available in virtual environments. Accordingly, AQL-WebServices allow us to fulfill ▶ **Req. 1** since different tools can be executed in different execution environments.

We refrain from describing further features or details of AQL-WebServices, as these are of a purely technical nature and do not contain any conceptual novelties. For example, credentials used to access a webservice are checked by comparing them with accounts managed by a configurable account manager that tracks stats for each account. Thereby certain users may, for instance, only ask a specific number of queries per day. While such a feature is vital from the perspective of a webservice's host, it is negligible with respect to its conceptual importance.

4 Automatic and Reproducible Benchmarks

Before taking a look at various instances of cooperative analyses, we need to specify how to compare and assess different (cooperative) approaches so that we can argue which are more or less beneficial than others. Our instrument of choice to do so are automatic and reproducible benchmarks. To construct and use such benchmarks we developed an approach on the basis of the AQL that allows us to refine and execute novel and existing benchmark suites.[23] *Refining* in this context means that we semi-automatically specify a precise, unambiguous and machine-readable ground truth for a set of benchmark apps. *Executing* stands for the following three-step procedure that our approach performs automatically: (1.) run the analysis under evaluation for each benchmark case, (2.) compare its results against the given ground truth, and (3.) measure the benchmark's outcome in terms of precision, recall and F-measure.

Before we describe our approach in detail, we motivate why ground truth refinements are required to achieve automated reproducible benchmarks. For our experiments, the benchmark suites DroidBench and (in some cases) ICC-Bench as well as TaintBench are used. The ground truth of the two micro benchmarks, DroidBench and ICC-Bench, is only available in an imprecise format. Source code comments are used to describe it. These comments include natural language descriptions and a single machine-readable information that states how many taint flows should be found for a certain benchmark app.

```
1  ...
2  /**
3  * @testcase_name StrongUpdate1
4  * @version 0.1
5  * ...
6  * @description Sensitive data is assigned to a heap object, but
       → then overwritten before it is leaked
7  * @dataflow source -> heap object -> alias -> nothing
8  * @number_of_leaks 1
9  * @challenges The alias analysis must support strong updates for
       → not causing a false positive.
10 */
11 public class MainActivity extends ActionBarActivity {
12 ...
```

Listing 14: Source Code Comment in a Benchmark App of DroidBench (Category: Aliasing; Name: StrongUpdate1)

Listing 14 shows such a comment – extracted from the source code of the DroidBench app *StrongUpdate1* [129]. The only machine-readable information it contains is visible on Line 8: `number_of_leaks` specifies how many taint flows (in this case: `1`) are comprised in this app. This is, for example, the only information which is checked in the associated test case of FlowDroid [140]. The natural language `description` (see Line 6) indicates that the respective taint flow should be documented as a *not*-expected case. Only together with this description, the ground truth can be understood completely and correctly. Since a machine cannot flawlessly do so, a more precise and machine-readable description, that

[23] All benchmark terms and concepts have been introduced in the background chapter (see Section 2.3).

categorizes the associated taint flow as not-expected, is required. In conclusion, an imprecise ground truth must be refined in order to use the respective benchmark suite in an automatic, unambiguous and reproducible fashion.

In case of the real-world benchmark TaintBench a (most-likely incomplete) ground truth is already given. The taint flows contained are originally defined on the basis of decompiled (via JADX [154]) source code statements that have manually been identified as connected pairs of sources and sinks. Since most analysis tools operate on bytecode or intermediate representations of it (e.g. Jimple), the definitions may have to be adapted or extended in order to be usable to benchmark arbitrary tools. Luckily, each taint flow documentation of the TaintBench suite states whether the respective taint flow is an expected (`"isNegative": false`) or a not-expected (`"isNegative": true`) one [186].

Along with the ReproDroid study [67] we proposed the Benchmark Refinement and Execution Wizard (BREW). With the help of BREW we refined DroidBench and ICC-Bench to overcome their limitations described above. For the TaintBench study [91] we extended BREW such that it is capable of semi-automatically refining benchmark suites that are available in TaintBench's format, namely *TAF (Taint Analysis Benchmark Format)*. With these three benchmark suites becoming available in refined form, we are ready to automatically evaluate Android taint analysis tools. In doing so, we focus on measuring effectiveness instead of efficiency. BREW primarily records precision, recall and F-measure (effectiveness) but also execution times (efficiency). For the latter it only tracks per benchmark case how long it takes until an AQL-Answer becomes available after issuing an AQL-Query. Related approaches such as BenchExec [39] focus on efficiency by gathering detailed information about e.g. the cpu and memory workload.

In the following, we present BREW in detail (see Section 4.1). Thereafter, first evaluation results achieved with BREW for experiments dealing with DroidBench, ICC-Bench and TaintBench are presented (see Section 4.2). Note that the results are more comprehensive and precise as in the original studies since a newer version of BREW has been used and more tools (and different versions) are included.[24] The results are nonetheless comparable to those of the original studies – reproducibility is *not* impaired. Any changes applied on the benchmarks themselves have been documented online [176] and the original benchmark results are compatible with the newest version of BREW. In Chapter 5, BREW is used again to automatically evaluate various cooperative approaches.

4.1 BREW

To benchmark an analysis tool or cooperative approach, we must execute the tool or approach for each of the benchmark suite's apps or cases. To guarantee reproducibility, we want to execute and evaluate the benchmarks automatically and always with respect to the exact same ground truth. Therefore, the ground truth must be precisely defined.

The *Benchmark Refinement and Execution Wizard (BREW)* helps us to refine a benchmark suite such that an imprecise ground truth becomes a precise one, and to execute and evaluate benchmarks automatically. The term wizard stands for the specific type of GUI that is typically used to guide a user step by step through a process, e.g. the refinement of a benchmark's ground truth. For benchmark execution and evaluation BREW's GUI is not needed, although it helps interpreting and inspecting results – a often disregarded task. Not least, artifact evaluations, which are meanwhile established in the field of software engineering, show the importance of this task [96]. Interpreting and

[24]In comparison to the ReproDroid study, up-to-date tools have been added. With respect to the TaintBench study, tools apart from FlowDroid and Amandroid were used.

inspecting results can become a time-consuming task, so means to accelerate and simplify this process can decide over the successful evaluation of an artifact [81].

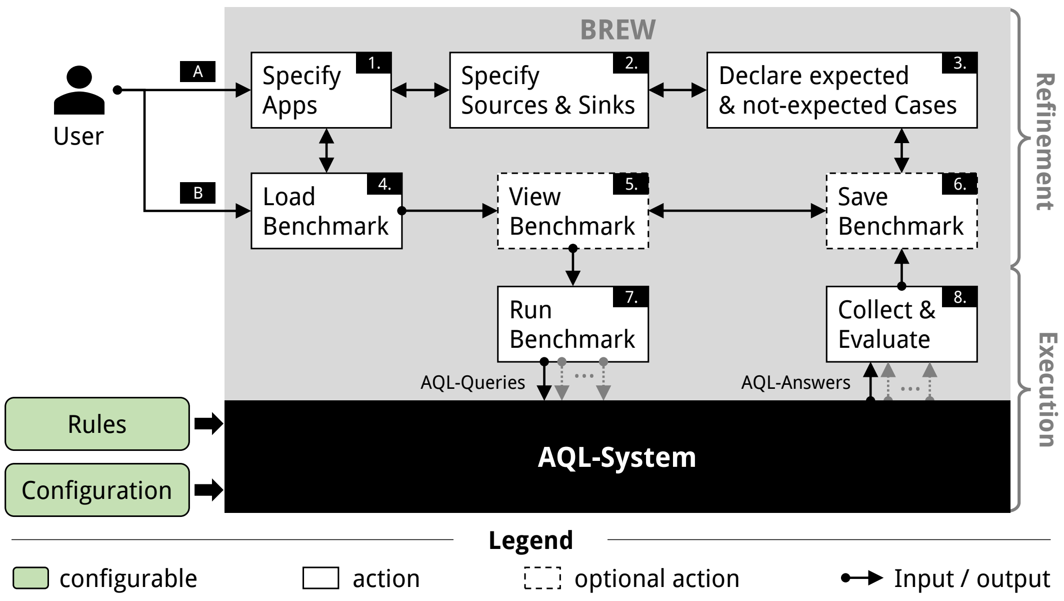


Figure 32: BREW Workflow

An overview of BREW's workflow is given in Figure 32. As illustrated in the figure, BREW is based on the AQL-System, more precisely, BREW uses the AQL-System as a library (Maven dependency). The depicted steps, in particular Step **1.** – **3.**, show the three steps that embody the refinement process. Steps **4.** – **6.** also belong to this process, but they only serve to load, save and view the refined benchmark (suite). Step **7.** and **8.** together with the AQL-System represent the workflow steps dealing with the execution of a refined benchmark.

Once the user launches BREW, two options are given to proceed: Option **A** create a new benchmark suite and Option **B** load an existing benchmark – probably including results of a previous evaluation. If Option **B** is chosen, the main refinement steps (**1.** – **3.**) may be skipped. Below all eight steps of BREW's workflow are described in detail:

1. Specify Apps

During this first step, the benchmark apps must be specified. To do so, the respective `.apk` files must be selected. They form the corpus of the benchmark suite being created or refined. BREW additionally allows to assign features to each app such that these can be taken into account when analyzing the respective apps. With the help of the FeatureFinder, a default tool implemented in the AQL-System, this process of assigning features can also be performed automatically. Furthermore, if certain benchmark apps are related to each other, for example because they comprise an inter-app taint flow that starts in one app and ends in another, they can be marked as belonging together.

While switching from Step **1.** to **2.** the statements that appear in the code of each specified benchmark app are extracted. To do so, Soot is employed to iterate through all the classes, methods and finally statements of each app. The extracted (Jimple-)statements are the candidates that can be specified as sources and sinks during the next step. BREW allows to ignore certain classes and the associated statements by excluding them due to efficiency. For example, most of the time we are not interested in statements that belong to the Android library but in statements that call methods of this library, hence, the library classes themselves can typically be excluded.

2. Specify Sources & Sinks

Next, the statements representing sources and sinks must be selected from the set of statements extracted during transition into this step. This process can be done manually by the user or automatically by comparing each statement against a predefined but configurable list of generic sources and sinks. Whenever a statement matches the generic representation of a listed source or sink it is marked respectively. For example, we know that the generic statement `sendTextMessage(..., String, ...)` is on our list and should be considered as a sink. Thus, we can automatically select a concrete statement like `sendTextMessage(..., "Hello world!", ...)` as a sink.

BREW allows to specify multiple statements as a single source or sink. For example, to get the last known location of a device, the method `getLastKnownLocation` may be called. This method can therefore be considered as a source. However, this method returns a `Location` object that holds various information, i.e. sensitive information like latitude and longitude or less sensitive information such as the GPS sensor's current accuracy. Accordingly, instead of `getLastKnownLocation` methods like `getLongitude` or `getLatitude` are frequently declared as sources. With respect to a taint flow, the same sensitive information is accessed, hence, the different statements can be treated as a single source. Another example: a sink may be accompanied by a logging statement directly occurring before or after the sink statement. In this case both statements, the sink and the logging statement, would leak the same information, thus, it is valid to interpret both statements as the same sink.

Note, a generic list of sources and sinks, that represent the specified ones, may be exported at the end of this step. Such a list may then be used by a taint analysis tool in order to identify benchmark suite specific sources and sinks.

3. Declare expected & not-expected Cases

While switching to this step, the benchmark's cases are generated if they have not been generated before. Initially, every source-sink combination that belongs to the same benchmark app (or app combination specified during Step **1.**) is added as an expected benchmark case. The benchmark's creator or the person who is refining it, has to decide whether a certain benchmark case actually is an expected or a not-expected one. For example, if one source and two sinks were specified during the previous step, two cases are generated. Let us assume that the first sink is embodied by the concrete statement `sendTextMessage(..., "Hello world!", ...)` and the second sink by `sendTextMessage(..., imei, ...)`. Then the source-sink combination including the first sink should be marked as a not-expected benchmark case, since it only sends a static string (`"Hello world!"`). In contrast, the second case which is involving the second sink may leak sensitive data, namely the `imei`. Note that the first sink can also be able to leak information if data is attached elsewhere, or if just sending a message reveals information (implicit information flow).

By constructing one benchmark case per source and sink combination, every taint flow possibly detected by an analysis tool can be associated with one case. Any detected taint flow for which no counterpart in form of a benchmark case exists is ignored and not taken into account during benchmark evaluation (see Step **8.**) as it must deal with an irrelevant source or sink that has not been specified during

Step **2.**. This might be the case if a tool outputs partial flows. For example, if a taint flow involves ICC, the partial flow between source and intent sink may be output although we are only interested in the bigger flow that stretches from source to sink. Since the partial flow represents a correct but irrelevant finding, it should be counted neither as true positive nor as false positive. Therefore, ignoring this flow is in line with our expectations. Note, without specifying any not-expected cases the precision metric is not meaningful (always 100%) since false positives (and true negatives) cannot be determined.

After specifying the expected and not-expected cases the refinement process is finished.

4., **6.** **Load & Save Benchmark**

Refined benchmark suites can be loaded from and saved to a compressed file in form of a serialized object. Thereby, all the information gathered during Step **1.**–**3.** are stored in a single file. If the benchmark has already been run, the actual results determined are also stored in this file. Hence, since hundreds or thousands of results may be attached to one file, it is compressed to save space. The user decides whether a benchmark is saved during Step **6.** as it is optional to do so.

The benchmark can also be saved by exporting it as JUnit tests. These tests may, for example, be deployed in the continuous integration pipeline of a taint analysis tool to detect and avoid regressions. The results of a benchmark can also be saved or exported in form of AQL-Answers.

5. **View Benchmark**

Viewing a benchmark presents another optional step in BREW's workflow. However, it might also be the only step performed via BREW after loading a benchmark to simply take a look at the outcome of a benchmark run. BREW presents statistics in terms of countings (TP, FP, TN, FN) and accuracy metrics (Precision, Recall, F-measure) as well as information about (overall) found flows, occurred timeouts or crashes and run-/analysis-times.

Furthermore, any benchmark case can be inspected in comparison to its expected (or not-expected) counter part, to get an idea why it failed, for example. Options are given that allow to filter the viewable results such that, for example, each category of DroidBench can be inspected separately. The currently viewed benchmark and its results may also be compared to another benchmark run, i.e. two benchmarks executed for two different sets of tools (and cooperative strategies) can be compared. The outcome will inform the user about the differences with respect to the involved benchmark cases, countings and accuracy metrics.

7. **Run Benchmark**

Once a benchmark is run via BREW, one query is given to the underlying AQL-System per benchmark case. Basically this query asks for flows in the respective benchmark case's app (e.g. `A.apk`):

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

Dependent on BREW's setup the reference may also be given as a combination of statement, method, class and app or the query may be formulated as a from-to query similar to the following example:

```
Flows FROM
    Statement('source()')->...->App('A.apk')
TO
    Statement('sink()')->...->App('A.apk')
?
```

Furthermore, features assigned during Step **1.** may also be attached to the issued query, for instance, `FEATURING 'ICC'` may be attached if the inter-component communication feature (ICC) is present:

```
Flows IN App('A.apk') FEATURING 'ICC' ?
```

Every query that arrives at the AQL-System is processed as defined in Subsection 3.4.2. In particular, transformation rules may be applied that further adapt the query. Not seldom two queries issued for two distinct benchmark cases, that deal with the same benchmark app, are the same. In such cases the AQL-System will run the query only once. For the second case, it will simply load the result as also described in Subsection 3.4.2.

8. Collect & Evaluate
During this last step the actual answers replied by the AQL-System are collected and compared to the expected answers. In case of an expected benchmark case it is checked whether the actual result holds the expected taint flow. If so, the associated case is marked and counted as a TP. If the expected taint flow is not present in the actual result, it is marked as FN. Respectively, in case of a not-expected benchmark case, it is marked as FP if the taint flow is included and as a TN if it is not. Finally, the accuracy metrics (Recall, Precision and F-measure) are computed.

In the next section, BREW is utilized to conduct a first set of experiments.

4.2 The Baseline

Before taking a look at cooperative analyses, we must determine the performance of individual tools – the baseline to compare against. By using BREW to do so, we determine the performance of existing standalone tools for various benchmark suites measured in terms of accuracy by automatically calculating precision, recall and F-measure, however, we also take a look at e.g. analysis execution times.

Table 9: Baseline Tool Overview

Name	**Version** / Date	**Availability** (Commit)	
AMANDROID*	3.1.2	✓ (65aec77)	[99]
AMANDROID	3.2.0	✓ (415ad9f)	[100]
AMANDROID	3.2.1	✓ (06596c6)	[101]
DIALDROID	September 2017	✓ (5df5734)	[126]
DIDFAIL	March 2015	✓	[127]
DROIDSAFE	June 2016 (Final)	✓ (1eab2fc)	[131]
FLOWDROID*	April 2017 (Nightly)	✗	[137]
FLOWDROID	2.7.1	✓ (72734bd)	[135]
FLOWDROID	2.9.0	✓ (e17e615)	[136]
FLOWDROID	2.10.0	✓ (0174ec4)	[134]
ICCTA*	February 2016	✓ (831afaa)	[151]
ICCTA	2.9.0	✓ (e17e615)	[150]
ICCTA	2.10.0	✓ (0174ec4)	[149]

*: oldest tool variant

The tools for which the baseline is computed are listed in Table 9. In the following, we refer to any tool's oldest version (marked with * in the table) by using "old" as version identifier.[25] We only consider static analysis tools that claim to be at least flow- or context-sensitive, to assure that all tools are competitors within the same league. The selected tools have been introduced in the background chapter (see Section 2.4) as well as other tools that have not been selected even though they would fit into our scope. The reasons for their exclusion are also discussed in the background chapter.

As determined by the REPRODROID study [67] and another independent study [68], AMANDROID and FLOWDROID are considered to be the state-of-the-art. Both studies thoroughly evaluated the accuracy that static taint analysis tools can actually achieve. Since both tools (AMANDROID and FLOWDROID) were updated after the publication of these studies, we add all versions published since then to our set of tools. Please note that all tools are used in their default configuration. Furthermore, to permit a fair comparison of all tools, BREW's option to consider line numbers is turned off, since not all tools output line numbers. It is mentioned explicitly, if we deviate from this default by adapting a tool's or BREW's configuration.[26]

By computing this baseline, we also want to compare the selected tools against each other in order to find out under which circumstances which tool should be included into a cooperative approach. To do so, we replicate the results of the REPRODROID study and

[25]The old versions match those used for the REPRODROID study [67].

[26]We refer to the work conducted by Mordahl and Wei [86] for more detailed information about how to configure specific tools with respect to certain analysis challenges.

check whether these tools keep their promises. In this context, we consider two kinds of promises: feature and accuracy promises. *Accuracy promises* refer to the precision, recall and F-measure values reported in the respective papers proposing these tools. *Feature promises* reflect the claims that a certain tool is, for example, object-sensitive or ready to analyze real-world apps. Alongside (mostly technical) limitations such as Android API level restrictions are identified.

The specifications of the execution environment that has been used to carry out the following experiments can be found in Appendix A.4.1.

4.2.1 Experiment 1: Feature Promises

To check which feature promises are kept by the tools we use the Feature-Checking benchmark suite first introduced in Section 2.4. It consists of 18 apps that comprise 21 expected and 6 not-expected benchmark cases. All these benchmark cases exploit only one specific feature at a time in an iconic way. Thus, each comprised case can be used to explicitly check the handling of a dedicated feature.

We originally developed the Feature-Checking benchmark suite for the ReproDroid study. At that time the current Android API version was 26, however, some analysis tools have been developed targeting even older API versions. Hence, we had developed two versions of the benchmark suite via Android Studio [105]. The first is targeting Android API 19, the second 26. The nowadays up-to-date Android API version is 30. Accordingly, we developed a new version targeting API 30. Additionally, we also recreated the suite targeting API 19 by employing an up-to-date version of Android Studio and its integrated build tools.

Since the analysis tools in our scope were designed for different API versions, we get different results with respect to the benchmark suite version used. In summary, not only the API version targeted by an app, but also the build tools used to construct it, influence the outcome of some analysis tools. To be fair we only report the optimal results achieved by a tool in Table 10.[27]

Table 10: Experiment 1: Results for the FeatureChecking Benchmark Suite

	AmAndroid			DIAL-	Did-	Droid-	FlowDroid				IccTA		
Feature	3.1.2	3.2.0	3.2.1	Droid	Fail	Safe	old	2.7.1	2.9.0	2.10.0	old	2.9.0	2.10.0
Aliasing	◯✓	◯✓	◯✓		◯✓	◯✓	◯✓	◯✓	◯✓	◯✓	◯✓	◯✓	◯✓
Static	◯✔	◯✗	◯✗		◯✗	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔
Callbacks	◯✔	◯✗	◯✗		◯✗	◯✗	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔
Life-Cycle	◯✔	◯✗	◯✗		◯✔	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔
Inter-Procedural	◯✔	◯✔	◯✔		◯✔	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔
Inter-Class	◯✔	◯✔	◯✔		◯✗	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔
IAC	◯✗*	◯✗*	◯✗*	◯✗	◯✗*	◯✗*	✗	✗	✗	✗	◯✗*	◯✗*	◯✗*
ICC (Explicit)	◯✓	◯✗	◯✗	◯✔	◯✗	◯✓	✗	✗	✗	✗	◯✔	◯✗	◯✗
ICC (Implicit)	◯✓	◯✗	◯✗	◯✓	◯✗	◯✓	✗	✗	✗	✗	◯✓	◯✓	◯✓
Flow-Sensitivity	◯✗	◯✔	◯✔		◯✔	✗	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔
Context-Sensitivity	◯✔	◯✔	◯✔		◯✔	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔	◯✔
Field-Sensitivity	◯✓	◯✓	◯✓		◯✓	◯✓	◯✓	◯✓	◯✓	◯✓	◯✓	◯✓	◯✓
Object-Sensitivity	◯✓	◯✗	◯✗		◯✗	◯✓	◯✓	◯✓	◯✗	◯✓	◯✓	◯✓	◯✓
Path-Sensitivity	✗	✗	✗		✗	✗	✗	✗	✗	✗	✗	✗	✗
Thread-Awareness	◯✔	◯✔	◯✔		✗	◯✔	✔	✔	✔	✗	◯✔	✔	✗
Reflection	✓	✓	✓		✗	✓	✗	✗	✗	✗	✗	✗	✗

◯ supported, ✔ confirmed, ✓ partially confirmed, ✗ not confirmed, * not promised without cooperation

[27] All individual results for each version of the Feature-Checking benchmark suite can be found in Appendix A.4.2.

If a promise is displayed as confirmed (✔) all the respective benchmark cases were correctly handled – expected taint flows were detected while not-expected taint flows were not. In case of a partially confirmed (✔) promise at least one expected case was handled correctly. In all other cases the feature promise was counted as not confirmed (✘).

As the result shows, most feature promises are kept. Nonetheless, it is striking that the more recent versions of AMANDROID do not keep six feature promises. The old version of AMANDROID did keep most of its promises but still appears to be flow-*in*sensitive. DIDFAIL also does not hold six promises. DIALDROID could not detect taint flows in the iconic inter-app communication case. The empty cells in the DIALDROID column reflect that DIALDROID is only usable in the context of ICC and IAC scenarios.[28] DROIDSAFE does not keep its promise with respect to callbacks. While most versions of FLOWDROID do not violate any promise, version 2.9.0 appears to be object-*in*sensitive – version 2.10.0 again fulfills this promise. Similarly, the more recent versions of ICCTA are not able to detect the taint flows involving ICC via explicit intents. FLOWDROID (except the old version) and ICCTA are the only tools outputting line numbers, and if BREW is configured to consider those, these tools not only partially but fully fulfill their promises with respect to Aliasing, Field- and Object-Sensitivity.

Most feature promises are kept. Different tool versions violate different promises.

4.2.2 Experiment 2: Tool Capabilities

We observed unequal tool behavior when confronted with different app versions during Experiment 1. Hence, we take a closer look at tool capabilities with respect to API and build tool versions in Experiment 2. To do so we developed a special purpose micro benchmark suite, the DIRECTLEAK suite. It consists of eight apps that are semantically equivalent to the `DirektLeak1` app of DROIDBENCH. Accordingly, each app comprises the same expected taint flow. The differences between these eight apps are caused by the instruments and settings used to built them. To understand the differences in detail, we take a brief look at some Android milestones that may influence an analysis:

- In **2014, Android 5.0 (API 21)** was released. Android 5.0 brought two major changes that may influence the outcome of an analysis. First, the Dalvik VM was replaced by ART (Android Runtime) [163]. Second, multiple `.dex` files packaged in the same `.apk` were allowed [167]. The latter has also immediately become the default. While the first may only influence dynamic analysis tools, the second might particularly influence static tools which now have to analyze all `.dex` file instead of just one – a simple but important technical requirement.

- Along with **Android 6.0 (2015, API 23)** *runtime permissions* were introduced [165, 172]. Instead of asking for permissions to certain resources at install time, permissions should be requested once the associated resources are accessed. To handle runtime permissions correctly, newly introduced callback functions have to be implemented and, from the perspective of an analysis, these callback functions must be treated explicitly.

- Along with **Android 8.0 (2017, API 26)** the AAPT (Android Asset Packaging Tool) was upgraded to its successor AAPT2 [120]. Thereby, the encoding of e.g.

[28] DIALDROID relies on FLOWDROID when dealing with intra-component cases.

layout files was slightly changed which may influence the capabilities of analysis tools to read such files.

Considering this history, we developed seven versions of the `DirectLeak1` app targeting Android API <21 (19), 21, 23, ≥26 (30). The apps targeting version 21, 23 and 30 have been added twice – with and without multiple `.dex` files.[29] As eighth app the original version of DroidBench (DB) has been included. The latter also targets an API < 21 (17[30]). Furthermore, all apps were developed and built in Android Studio (build tools version: 31), only the original `DirectLeak1` app was built with Eclipse.

We again use BREW to execute the experiment. Whenever a tool successfully detects the taint flow in a certain app, we mark the app as analyzable by this tool. All results can be found in Table 11.

Table 11: Experiment 2: Results for the DirectLeak Benchmark Suite

DirectLeak	Amandroid			DIAL-	Did-	Droid-	FlowDroid				IccTA		
Version	3.1.2	3.2.0	3.2.1	Droid	Fail	Safe	old	2.7.1	2.9.0	2.10.0	old	2.9.0	2.10.0
DB	✔	✔	✔	✘	✔	✔	✔	✔	✔	✔	✔	✔	✔
19	✔	✔	✔	✘	✔	✘	✔	✔	✔	✔	✔	✔	✔
21	✔	✔	✔	✘	✔	✘	✔	✔	✔	✔	✔	✔	✔
23	✔	✔	✔	✘	✔	✘	✔	✔	✔	✔	✔	✔	✔
30	✘	✘	✘	✘	✘	✘	✔	✔	✔	✔	✘	✔	✔
21 *	✔	✔	✔	✘	✘	✘	✘	✓	✓	✓	✘	✓	✓
23 *	✔	✔	✔	✘	✘	✘	✘	✓	✓	✓	✘	✓	✓
30 *	✘	✘	✘	✘	✘	✘	✘	✓	✓	✓	✘	✓	✓

DB: The original version of the `DirectLeak1` app taken from DroidBench.
✔: analyzable, ✓: analyzable (special configuration needed), ✘: not analyzable
*: Comprising multiple `.dex` files

Since we use DIALDroid only for ICC and IAC cases, it is – as expected – not successful in any case. Amandroid seems to be unable to handle API 30. DidFail can neither handle API 30 nor multiple `.dex` files. DroidSafe seems to be incompatible with recent build tools. The old versions of FlowDroid and IccTA show issues when dealing with multiple `.dex` files and API 30. The latter only holds for the old version of IccTA. More recent versions of FlowDroid and IccTA successfully handle any app in our DirectLeak suite, the non-default configuration option to handle multiple `.dex` files must therefore be activated via a command line parameter (`-d`). Due to this result, we always use this option from now on, when using FlowDroid or IccTA (≥2.7.1).

Amandroid and especially FlowDroid are the most successful tools with respect to handling different Android API versions and build tools.

[29] A build tool option allows to disable the default (multile `.dex` files).
[30] For all newly developed apps the targeted API version is equal to the one allowed as minimum. The original `DirectLeak1` app targets API 17 while allowing API 8 as minimum.

4.2.3 Experiment 3: Accuracy Promises

The previous experiment indicates that all tools are able to handle the original DroidBench apps, thus, we focus on this benchmark suite in order to check whether the tools hold their promises with respect to accuracy. More precisely, we are using DroidBench 3.0.[31] To determine what has been promised we trawled through the proposing papers for denoted values of precision, recall and F-measure – the harmonic mean of the first two. For the sake of clarity, and because it best represents the overall accuracy, we only report on the F-measure values here. Precision and recall can additionally be found in the digital Appendix (A.6). Figures 33a–33e (Page 107) show our results by depicting the promised (dark bars) and actually achieved values (light bars). Whenever there was no promise a promised value of 0.00 is reported.

DroidBench 3.0 This is always the case for DroidBench 3.0 (see Figure 33a). All the tools were proposed prior to the release of DroidBench 3.0 or have been evaluated on a subset only to foster comparability to earlier reported values. Most tools have an accuracy of more than 60% apart from DIALDroid, DidFail and newer versions of Amandroid which have less. 60% does not sound confidence inspiring, but a lot of distinct features are exploited in DroidBench 3.0, specifically such features designed to challenge existing tools. Thus, it was to be expected that each tool makes mistakes at some point. We use DIALDroid only for its designed purpose (ICC and IAC), hence, its value for the complete set is low (14%) as expected. DidFail is the least updated tool in our scope but still scores an F-measure of 52%. The more recent versions of Amandroid appear to be – by far – less accurate. This regression, however, is no surprise once we recall the results of Experiment 1, where we already saw that the more recent versions support less features.

DroidBench 2.0 Most promises can be found in form of values reported for DroidBench 2.0 or a subset of it. Figure 33b shows the promised and actual values achieved for the complete set (DroidBench 2.0). Still, none of the tools actually fulfill their promises.

DroidBench (FlowDroid/Amandroid Subset) Amandroid and FlowDroid precisely report the apps they took into account. We call the respective subset the FlowDroid/Amandroid subset. The results are presented in Figure 33c. The bars colored in green highlight kept promises while bars highlighted with yellow color refer to almost kept promises. As it becomes visible, for this subset, any version of FlowDroid is able to keep its promises. The old version of Amandroid is almost able to do so, however, the newer versions are not.

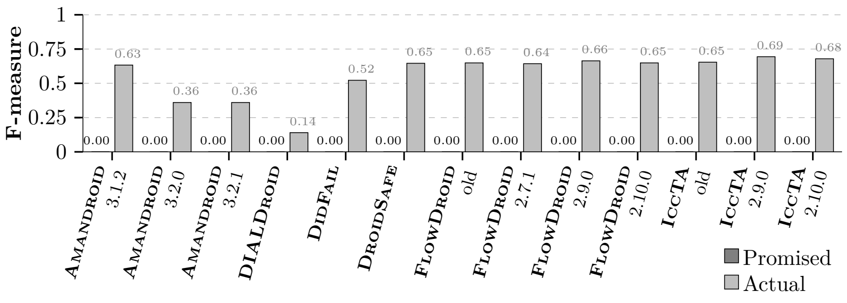
ICC-Bench 2.0 Some tools have also given accuracy promises for ICC-Bench (2.0). However, no tool could keep its promises with respect to this suite (see Figure 33d). Closest to keeping it is the old version of Amandroid with an F-measure of 94% instead of promised 100%. FlowDroid does only achieve 0 values since it has no ICC capabilities. DidFail does, however, it renders ineffective as no expected taint flow is detected. Similarly, DroidSafe does not achieve an F-measure greater than 0. Finally, in case of IccTA we can observe a positive trend – more recent versions of the tool score better F-measure values, however, the promised value of 94% is never reached.

[31] Whenever claims are made for DroidBench 2.0 or another subset of 3.0, this has been achieved by filtering the results respectively.

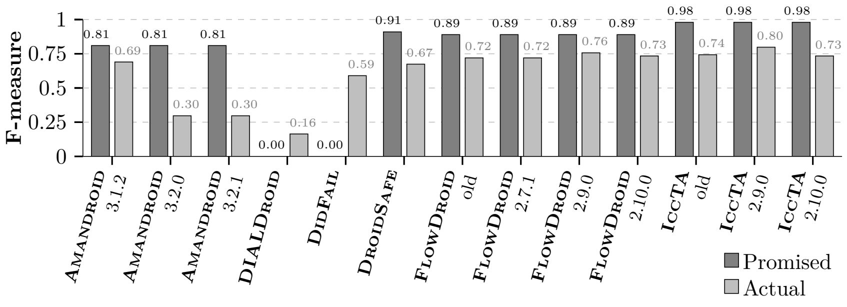
Intent-Matching We also evaluated the benchmark suite Intent-Matching although no promises are available for it. The associated results can be inspected in Figure 33e. Similar to ICC-Bench, the old version of Amandroid scores best (94%).

The accuracy values promised can often not be reached or reproduced. Only FlowDroid keeps its accuracy promises.

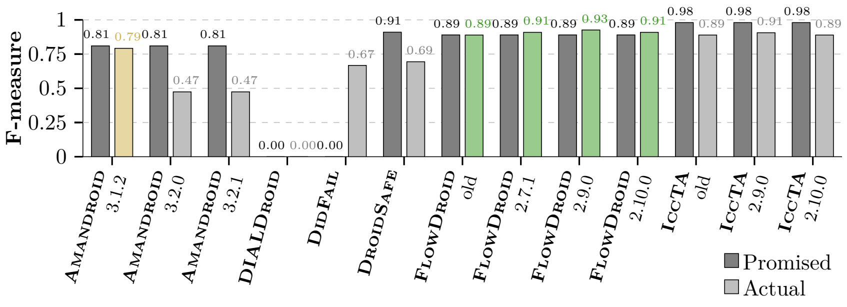
(a) DroidBench 3.0



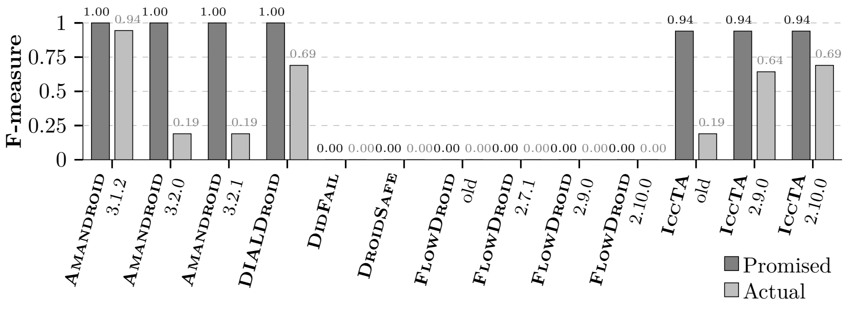
(b) DroidBench 2.0



(c) DroidBench (FlowDroid/Amandroid Subset)



(d) ICC-Bench 2.0



(e) Intent-Matching

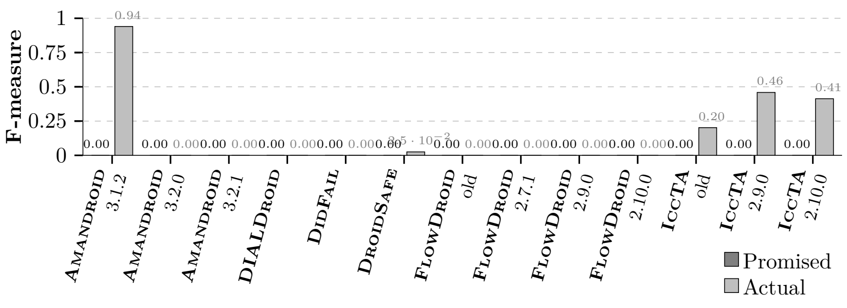


Figure 33: Experiment 3: Accuracy Promises

4.2.4 Experiment 4: Accuracy Comparison

Next, we want to compare the tools with each other to determine whether certain tools are more accurate than others with respect to certain features. To do so, we determine the tools' accuracy for each category of DroidBench (3.0).

Table 12: Experiment 4: Results for DroidBench

Category	**DIALDroid**	**AmAndroid** 3.2.0	**AmAndroid** 3.2.1	**DroidSafe**	**DidFail**	**IccTA** old	**FlowDroid** old	**FlowDroid** 2.9.0	**IccTA** 2.9.0	**AmAndroid** 3.1.2	**FlowDroid** 2.10.0	**IccTA** 2.10.0	**FlowDroid** 2.7.1	∅
FieldAndObjectSensitivity	0.000	1.000	1.000	0.667	0.800	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	**0.882**
EmulatorDetection	0.000	0.929	0.929	0.846	0.889	0.966	0.966	0.966	0.966	0.966	0.966	0.966	0.966	**0.871**
AndroidSpecific	0.000	0.533	0.533	0.900	0.429	0.842	0.900	0.952	0.952	0.706	0.952	0.952	0.842	**0.730**
Callbacks	0.000	0.235	0.235	0.667	0.769	0.897	0.897	0.897	0.897	0.692	0.857	0.857	0.897	**0.677**
Lifecycle	0.000	0.452	0.452	0.933	0.400	0.737	0.737	0.769	0.769	0.737	0.769	0.769	0.800	**0.640**
GeneralJava	0.000	0.286	0.286	0.821	0.595	0.762	0.810	0.810	0.810	0.703	0.810	0.810	0.684	**0.630**
Threading	0.000	0.286	0.286	0.000	0.667	0.667	1.000	0.909	0.909	0.800	0.667	0.667	0.909	**0.597**
Aliasing	0.000	0.667	0.667	0.000	0.667	0.667	0.667	0.667	0.667	0.500	0.667	0.667	0.667	**0.552**
ArraysAndLists	0.000	0.222	0.222	0.667	0.444	0.500	0.615	0.727	0.727	0.545	0.615	0.615	0.615	**0.501**
ICC	0.538	0.100	0.100	0.364	0.500	0.706	0.273	0.348	0.690	0.750	0.348	0.690	0.348	**0.443**
ImplicitFlows	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000	1.000	1.000	**0.231**
Reflection	0.000	0.200	0.200	0.615	0.200	0.200	0.200	0.200	0.200	0.364	0.200	0.200	0.200	**0.229**
DynamicLoading	0.000	0.500	0.500	0.000	0.000	0.000	0.000	0.000	0.000	0.500	0.000	0.000	0.000	**0.115**
IAC	0.625	0.000	0.000	0.000	0.533	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.089**
Reflection_ICC	0.000	0.000	0.000	0.533	0.167	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.054**
Native	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.333	0.000	0.000	0.333	**0.051**
SelfModification	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.000**
UnreachableCode	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**0.000**
Average (∅)	0.065	0.301	0.301	0.390	0.392	0.441	0.448	0.458	0.477	0.478	0.492	0.511	0.515	–
Overall	0.140	0.360	0.360	0.646	0.522	0.654	0.649	**0.664**	**0.694**	0.633	0.649	**0.679**	0.643	–
	13.	11.	11.	7.	10.	4.	6.	**3.** 🏆	**1.** 🏆	9.	5.	**2.** 🏆	8.	

The results can be found in Table 12. Each row in the table stands for one category in DroidBench, each column refers to one tool. Rows and columns are sorted with respect to the average per category and tool. Additionally, a color scheme has been added to emphasize each tool's performance: the darker the background of a cell is, the higher the F-measure. The last two rows of the table show the overall F-measure computed for the whole benchmark and a ranking with respect to that value. Please note that the overall F-measure differs from the average F-measure value used for sorting.

For the first ten categories of DroidBench most tools provide decent results. On average an F-measure of more than 0.4 is reached. For the remaining 8 categories some tools outperform others. In case of category "SelfModification" and "UnreachableCode" no tool finds any taint flow, therefore the F-measure in these categories is 0. Finally DIALDroid shows promising results in its designated categories (ICC, IAC). In particular, in the IAC category it outperforms all other tools and reaches an F-measure of 0.625. In the "Reflection" category most tools score an F-measure value of around 0.2, only DroidSafe manages to reach an F-measure of 0.615. With respect to the "ImplicitFlows" category the newest versions of FlowDroid (and version 2.7.1) and consequently IccTA manage to achieve an F-measure of 1.

On the micro benchmark DroidBench (3.0) high F-measure values are scored frequently by various tools. IccTA (0.694) and FlowDroid (0.664) perform best.

Lastly, a comment on the analysis times. Any version of FlowDroid requires less than 10 seconds per app on average. The old version of FlowDroid, in comparison to the more recent ones, needs about a second longer per app. All other tools except DroidSafe do not exceed a minute on average. DroidSafe needs about 4–5 minutes per app and timeouts (after 20 minutes) for 12 apps. None of the other tools ever reaches the timeout.

The analysis of a benchmark app as comprised in DroidBench takes less than a minute on average. Fastest is FlowDroid which does not even require 10 seconds per app.

4.2.5 Experiment 5: Real-World Readiness

To determine the real-world readiness of the different tools we use two different app sets and one benchmark suite:

- Top15 (see Appendix A.4.3): The 15 most downloaded apps in Google's Play-Store [142].
- DIALDroid-Bench [125]: 30 apps that have been collected for and published with the DIALDroid study [58].
- TaintBench [186]: A benchmark suite that comprises 39 real-world benchmark apps for which 249 benchmark cases (203 expected and 46 not-expected) have been manually specified [91].

Only the latter one (TaintBench) actually is a benchmark suite since it comes with a ground truth. Hence, we will use the first two only in order to see if the analysis tools can finish an analysis of the respective apps. In case of TaintBench we will evaluate their accuracy precisely.

As a first result we document the success rate of the different tools for the different app sets. Thus, we want to know for how many apps of each app set the analysis finished successfully. However, it is unclear what defines a successful finish. Some tools always output that they finished successfully (process exit code 0) even though the respective analysis has failed – e.g. encountered an exception or a timeout. Other tools indicate a successful execution but simply do not output a result.

To approximate we define that an analysis tool was **executed successfully** if it outputs a non-empty AQL-Answer.

This definition only allows an approximation since we exclude that a successful execution may correctly lead to an empty answer. We accept this limitation since most apps should include *must-have* taint flows. For example, a messenger app most of the time comes with the ability to share one of your contacts, hence, it must have at least a taint flow from a source accessing the contacts to a sink used for sharing it.

The chart displayed in Figure 34 shows the success rate of all 13 tools with respect to the definition above for the three different app sets. The timeout threshold was set to 60 minutes in case of the Top15 and DIALDroid-Bench set. For TaintBench it was only 20 minutes since it is the largest set. The best relative success rate (79%) is reached by the old version of FlowDroid for TaintBench. Next is the newest version of FlowDroid which reached 77% in case of DIALDroid-Bench. For the Top15 set almost all answers were empty. Only the oldest version of Amandroid (3.1.2) has output two non-empty answers for this set.

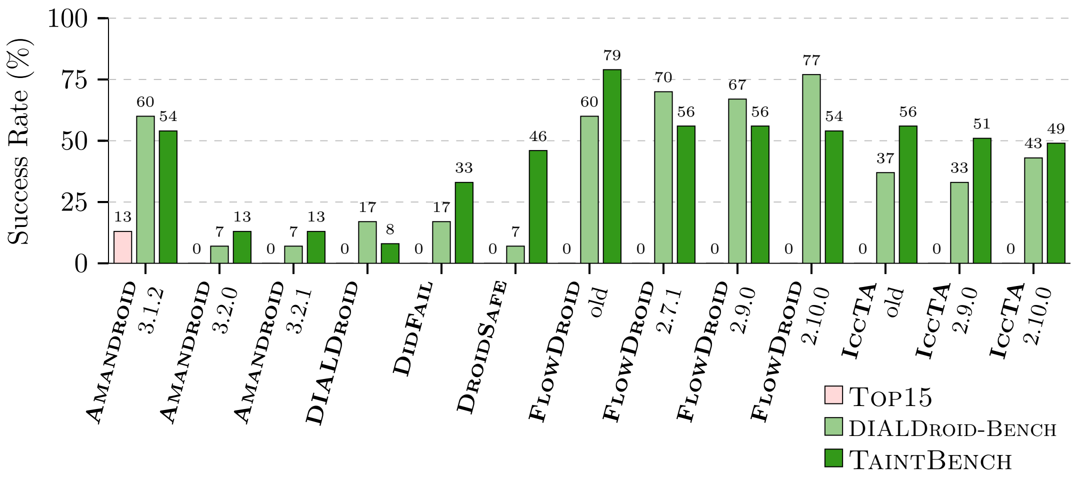


Figure 34: Experiment 5: Successful Analyses (Real-World Apps)

All 13 tools are able to successfully analyze some real-world apps.

For TaintBench we got a ground truth, hence, we want to take a closer look and evaluate the tools' accuracies. In Table 13 the F-measure values all tools achieved for the 39 benchmark apps (rows) are shown. Similar to the DroidBench experiment result presentation, the rows and columns are sorted with respect to the average F-measure per row and column. Additionally, the two last rows report the overall F-measure value for the whole set and a ranking according to this value.

Again the old version of FlowDroid, followed by the more recent versions of FlowDroid, performs best. IccTA falls behind because it relies on IC3 to extract intent and intent filter information, but IC3 is unable to complete its analysis in many cases. For a large portion of apps (23 of 39) the F-measure value is always 0 (see red area in Table 13). The main reason is that the tools failed to successfully analyze the apps. In addition, the analysis tools often did not detect a single expected taint flow, which again results in an F-measure value of 0. Surprisingly, for some apps only a single tool is able to reach an F-measure of more than 0:

- only DidFail reaches an F-measure of 80% in case of `fakeappstore`,
- the old version of FlowDroid achieves 75% for `slocker_android_samp`,
- the most recent version of FlowDroid 74% for the `backflash` app,
- DroidSafe 67% for the `sms_send_locker_qqmagic` app, and
- Amandroid (3.1.2) 40% in case of the `smssilience_fake_vertu`.

This clearly shows, that there is no standalone tool that is suitable for an arbitrary real-world case. In general, FlowDroid seems to be most accurate with respect to TaintBench, however, even FlowDroid does not reach an F-measure above 0 in most cases (30 of 39 cases show an F-measure of 0).

Please note again that the results above are part of the baseline. No cooperative strategies were used and all tools have been executed in their default configuration. In the next chapter we will see that cooperative strategies can improve the presented results.

Table 13: Experiment 5: Results for TaintBench

Category	DIALDroid	AmAndroid 3.2.0	AmAndroid 3.2.1	DroidSafe	AmAndroid 3.1.2	DidFail	IccTA old	IccTA 2.10.0	IccTA 2.9.0	FlowDroid 2.7.1	FlowDroid 2.9.0	FlowDroid 2.10.0	FlowDroid old	∅
chat_hook	0.000	0.429	0.429	0.000	0.000	0.462	0.333	0.462	0.462	0.778	0.778	0.778	0.778	0.438
exprespam	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.667	0.667	0.667	0.667	0.667	0.667	0.308
smssend_packageInstaller	0.000	0.000	0.000	0.000	0.333	0.000	0.571	0.333	0.333	0.571	0.571	0.333	0.571	0.278
proxy_samp	0.000	0.000	0.000	0.000	0.000	0.476	0.476	0.400	0.476	0.476	0.476	0.400	0.400	0.275
overlay_android_samp	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.400	0.400	0.667	0.667	0.667	0.000	0.215
cajino_baidu	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.500	0.500	0.500	0.500	0.500	0.000	0.192
stels_flashplayer_android...	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.500	0.500	0.500	0.500	0.500	0.000	0.192
roidsec	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000	0.000	0.500	0.115
fakedaum	0.000	0.000	0.000	0.000	0.000	0.000	0.667	0.000	0.000	0.000	0.000	0.000	0.667	0.103
save_me	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.167	0.167	0.167	0.667	0.090
fakeappstore	0.000	0.000	0.000	0.000	0.000	0.800	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.062
samsapo	0.000	0.000	0.000	0.000	0.000	0.400	0.000	0.000	0.000	0.000	0.000	0.000	0.400	0.062
slocker_android_samp	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.750	0.058
backflash	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.737	0.000	0.057
sms_send_locker_qqmagic	0.000	0.000	0.000	0.667	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.051
smssilience_fake_vertu	0.000	0.000	0.000	0.000	0.400	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.031
beita_com_beita_contact	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
chulia	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
death_ring_materialflow	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
dsencrypt_samp	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
fakebank_android_samp	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
fakemart	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
fakeplay	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
faketaobao	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
godwon_samp	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
hummingbad_android_samp	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
jollyserv	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
overlaylocker2_android_samp	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
phospy	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
remote_control_smack	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
repane	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
scipiex	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
smsstealer_kysn_assassin...	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
sms_google	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
tetus	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
the_interview_movieshow	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
threatjapan_uracto	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
vibleaker_android_samp	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
xbot_android_samp	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Average (∅)	0.000	0.011	0.011	0.017	0.019	0.055	0.078	0.084	0.086	0.111	0.111	0.122	0.138	–
Overall	0.000	0.032	0.032	0.021	0.021	0.112	0.158	0.149	0.158	**0.233**	**0.233**	**0.269**	**0.292**	–
	13.	9.	9.	11.	11.	8.	5.	7.	5.	**3.** 🏆	**3.** 🏆	**2.** 🏆	**1.** 🏆	

5 Evaluation

Finally, we are ready to evaluate whether and which cooperative analyses pay off. Therefore, different cooperative analyses bringing together various tools are proposed, explained and evaluated. For each cooperative analysis (1.) the tools that are involved are listed, (2.) the employed cooperative strategies are explained to describe how these tools are combined or how they cooperate with each other, and (3.) the respective evaluation results are presented in comparison to the baseline (see Section 4.2). Thereby, we repeatedly deal with our *main research question*:

RQ$_{\text{MAIN}}$: To which extent can cooperative analyses outperform standalone taint analyses?

To answer this question we focus on performance in terms of accuracy (precision, recall, F-measure); comments about execution or analysis times and other aspects can be found in Section 5.7. In the following six sections, we take a look at six cooperative analyses:

1. Inter-Component Communication (ICC) & Slicing ...... Section 5.1 (Page 115 ff.)
2. Inter-App Communication (IAC) & App Merging .... Section 5.2 (Page 121 ff.)
3. Reflection & Native Code .... Section 5.3 (Page 125 ff.)
4. False Positive Elimination .... Section 5.4 (Page 128 ff.)
5. Sources & Sinks .... Section 5.5 (Page 131 ff.)
6. Backward Compatibility .... Section 5.6 (Page 137 ff.)

Remarks

For brevity, the three remarks below are only discussed once to avoid repetitions in the sections associated with the individual cooperative analyses.

Remark 1 (Initial AQL-Queries) Please note that most cooperative strategies are initially triggered by raising the following AQL-Query:

```
Flows IN App(%APP_APK_IN%) ?
```

Placeholders like `%APP_APK_IN%` are replaced by the AQL-System or BREW, for example, with the path to the `.apk` file representing the respective benchmark case's app. Whenever another initial query is required, it is mentioned in the associated section. When different taint analysis tools are involved, `Flows`-questions are usually transformed into:

```
Flows IN App(%APP_APK_IN%) USING '%TOOL%' ?
```

The placeholder `%TOOL%` is then replaced by the respective tool's identifier (e.g. FlowDroid-2.10.0).

Remark 2 (Reported Results) For the evaluation of several cooperative analyses, benchmarks are only used in parts. For example, often only a single category of DroidBench is used. Some of these categories only contain positive benchmark cases. Hence, since there are no negative cases, false positives and true negatives cannot be evaluated. Whenever this is the case, only true positives (and implicitly false negatives) are reported.

All reported results and the associated raw data of all experiments is collected in an artifact as described in Appendix A.6.

Remark 3 (Tool Introductions & Execution Environment) All tools involved in the cooperative analyses presented in the following, are briefly described again in the respective sections. An initial description of all tools can be found in the background chapter (see Section 2.4). Default tools that belong to the AQL-System are introduced in Subsection 3.4.3.

The specifications of the execution environments used to carry out all experiments are attached in Appendix A.4.1. However, since we focus on accuracy and largely disregard analysis time, the specifications of these execution environments are of secondary importance.

5.1 Cooperative Analysis 1: Inter-Component Communication (ICC) & Slicing

Many challenges for Android app analyses are introduced by the Android system, in particular the communication between different components (ICC) or apps (IAC) is often hard to handle. This first cooperative analysis deals with ICC only. Different strategies are used to analyze diverse ICC cases. These strategies involve static and dynamic analysis tools and even a slicer in order to get the most accurate results.

5.1.1 Cooperative Analysis 1: Tools

Table 14 lists all the tools and benchmark suites that are employed for the evaluation of Cooperative Analysis 1. Amandroid and FlowDroid, which are identified as state-of-the-art by two recent and independent studies [67, 68], are used as intra-component taint analysis tools. Their results are connected by gathering information about inter-component flows. To do so, IC3 or a combination of Amandroid and the IntentInformationFinder provide information about intent sinks and intent sources. This information is combined either by the AQL-System's CONNECT operator or PIM. Jicer comes into play to slice across component boundaries in order to eliminate false positives that were initially introduced by the cooperative analysis.

Table 14: Cooperative Analysis 1: Tools and Benchmark Suites

Tools (cooperative)	**Benchmark Suites**
• FlowDroid (2.10.0)	• DroidBench (ICC only)
• Amandroid (3.1.2)	• ICC-Bench
• IC3	• Intent-Matching
• IntentInformationFinder^d	
• CONNECT^d	
• PIM	
• Jicer	
Tools (standalone for comparison)	
• FlowDroid (2.10.0)	• IccTA (2.10.0)
• Amandroid (3.1.2)	

^{d}: Default tool (see Section 3.4.3, Page 85)

As benchmark suites we partially use DroidBench, more precisely, only its ICC category, ICC-Bench and the Intent-Matching benchmark. While the cases of DroidBench and ICC-Bench allow us to get a general impression of an approach's ability to handle ICC, the Intent-Matching benchmark allows us to assess its ability to match intent sinks and intent sources.

For comparison we considered all tools that claim to support ICC as evaluated in the baseline (see Section 4.2). However, for brevity only the three listed in Table 14 are actually used for comparison. Amandroid (3.1.2) scored the best results with respect to the ICC category of DroidBench in the baseline. FlowDroid (2.10.0) embodies the basis for the cooperative strategies introduced in the following. Lastly, IccTA (2.10.0) is used for comparison since it is an ICC-enabled extension of this basis.

5.1.2 Cooperative Analysis 1: Strategies

Overall four different cooperative strategies with respect to ICC are evaluated. Each strategy is described briefly in the following.

Strategy 1: Default The default cooperative ICC strategy transforms the initial AQL-Query (`Flows IN App(%APP_APK_IN%) ?`) into the following one:

```
1  CONNECT [
2      Flows IN App(%APP_APK_IN%) ?,
3      CONNECT [
4          IntentSinks IN App(%APP_APK_IN%) ?,
5          IntentSources IN App(%APP_APK_IN%) ?
6      ] ?
7  ] ?
```

In the context of this strategy, flows are determined by the up-to-date version of FlowDroid (2.10.0 – see Line 2 in the listing above). Whenever intent sinks or intent sources are to be determined, IC3 is used (see Lines 4, 5). As suggested by its name, the CONNECT operator connects or combines the findings of FlowDroid and IC3 (see Lines 1, 3). To do so, it is called twice. The inner call makes use of its ability to match intent sinks and intent sources in order to determine inter-component flows. The outer call then merges the intra-component results given by FlowDroid with these inter-component flows.

Strategy 2: +PIM By replacing the inner `CONNECT` with `MATCH`, we trigger PIM to perform the intent sink/source matching task (see Line 3 in the listing below).

```
1  CONNECT [
2      Flows IN App(%APP_APK_IN%) ?,
3      MATCH [
4          IntentSinks IN App(%APP_APK_IN%) ?,
5          IntentSources IN App(%APP_APK_IN%) ?
6      ] ?
7  ] ?
```

Strategy 3: Amandroid Amandroid is also capable of providing information about intent sinks and intent sources. Hence, the following strategy employs Amandroid instead of IC3 to answer the inner questions (see Lines 4–8 in the listing below).

```
1   CONNECT [
2       Flows IN App(%APP_APK_IN%) ?,
3       MATCH [
4           IntentSinks IN App(%APP_APK_IN%) USING 'Amandroid-312' ?,
5           CONNECT [
6               IntentFilters IN App(%APP_APK_IN%) USING 'Amandroid-312' ?
7               IntentSources IN App(%APP_APK_IN%) USING 'Default' ?
8           ] ?
9       ] ?
10  ] ?
```

Please note that Amandroid only provides information about intent filters (Line 6 in the listing above) but not about intent sources, i.e. the intent triples provided are exact but the references given for intent filters are components only – no exact statements (e.g. `getStringExtra(...)` instances) are provided. To overcome this limitation, we also ask the AQL-System's default tool IntentInformationFinder for intent sources (Line 7).

The latter are precise with respect to the statements identified but imprecise when it comes to intent triples – the opposite of the intent filters given by Amandroid. To get the most out of both tools, we connect or merge their results (Lines 5–8). Thereby the referenced components are compared, i.e. if the component of the intent filter (given by Amandroid) matches the component of an intent source (given by the IntentInformationFinder), the exact intent triple information (Amandroid) is merged with the exactly determined statements (IntentInformationFinder). In the end, this allows us to get precise and complete intent sources.

Strategy 4: +Jicer The last cooperative ICC strategy makes use of Jicer to slice from source to sink. First, inter-component flows are determined (see Lines 10–13 in Listing 15). Second, these flows are forwarded to Jicer as input edges (see variable `'InputEdges'` in Line 9). The static, inter-procedural slicer operates on an *app dependence graphs (ADG)*, which is an extended version of an SDG (see Subsection 2.4.2). Whenever Jicer receives an AQL-Answer as input edges, it takes all flows that are contained in this answer and adds them as edges to its ADG before its slicing procedure starts. Consequently, Jicer gains the ability to slice across component boundaries, whenever ICC edges are provided. The latter is extensively described in the proposing paper [87].

```
 1  app = Slice FROM
 2      Statement(%STATEMENT_FROM%, %LINENUMBER_FROM%)->
 3      Method(%METHOD_FROM%)->Class(%CLASS_FROM%)->
 4      App(%APP_APK_FROM%)
 5  TO
 6      Statement(%STATEMENT_TO%, %LINENUMBER_TO%)->
 7      Method(%METHOD_TO%)->Class(%CLASS_TO%)->
 8      App(%APP_APK_TO%)
 9  WITH 'InputEdges' = {
10      MATCH [
11          IntentSinks IN App(%APP_APK_FROM%) ?,
12          IntentSources IN App(%APP_APK_TO%) ?
13      ] ?
14  } !
15
16  CONNECT [
17      Flows IN App($app) ?,
18      MATCH [
19          IntentSinks IN App($app) ?,
20          IntentSources IN App($app) ?
21      ] ?
22  ] ?
```

Listing 15: Strategy 4: +Jicer (AQL-Query)

The slice is computed from source (Lines 2–4) to sink (Lines 6–8), i.e. source and sink are used as slicing criteria. For this purpose, source and sink must be described unambiguously. To get the required information about the respective statements, an AQL-Query that asks for sources and sinks inside the targeted app could be issued. Luckily, this is not needed since the benchmark cases of all considered benchmark suites already provide this information. Hence, when using BREW to carry out the evaluation, it can be configured to ask a more comprehensive initial AQL-Query that provides all required information:

```
Flows FROM
    Statement(%STATEMENT_FROM%, %LINENUMBER_FROM%)->
    Method(%METHOD_FROM%)->Class(%CLASS_FROM%)->
    App(%APP_APK_FROM%)
TO
    Statement(%STATEMENT_TO%, %LINENUMBER_TO%)->
    Method(%METHOD_TO%)->Class(%CLASS_TO%)->
    App(%APP_APK_TO%)
?
```

The involved placeholders (e.g. `%STATEMENT_FROM%`) are then resolved by BREW with respect to each benchmark case. Finally the file answer given for the slice query (see Lines 1–14 in Listing 15) is associated with the variable `app` (Line 1).

The main query (Lines 16–22) uses this variable to reference the sliced app (`$app`). Apart from this change the main query is equal to the query of Strategy 2, thus, the final AQL-Answer is computed as described for Strategy 2.

5.1.3 Cooperative Analysis 1: Results

All results determined for the four cooperative ICC strategies are summarized in Table 15. The first column denotes the benchmark suite. The next three columns partially repeat the baseline results given for the respective benchmark. The individual columns under the "Strategies" headline provide the results for the different ICC strategies. The last three columns compare the results reported for the baseline against those reported for the cooperative strategies by displaying their differences. In each result cell the achieved F-measure value is reported along with the true and false positives counted.

Table 15: F-measure, True Positives / False Positives for Cooperative Analysis 1

Benchmark Suite	**Baseline**			**Strategies**				**Difference**		
				1.	2./3.	4.		*= Best strategy − Baseline*		
(Cases)	FD	IccTA	Best^1	Default	+PIM	+JICER	Best	FD	IccTA	Best
DroidBench (ICC only)	0.348	0.690	0.750	0.727	0.690	—*	0.727	▲ **0.379**	▲ **0.037**	▼ -0.023
(19 expected, 9 not-expected)	4 / 0	10 / 0	12 / 1	12 / 2	10 / 0	—/—*				
ICC-Bench	0.000	0.690	0.940	0.417	0.480	—*	0.480	▲ **0.480**	▼ -0.210	▼ -0.460
(19 expected, 0 not-expected)	0 / 0	10 / 0	17 / 0	5 / 0	6 / 0	—/—*				
Intent-Matching	0.000	0.413	0.940	0.152	0.542	0.706	0.706	▲ **0.706**	▲ **0.293**	▼ -0.234
(79 expected, 146 not-expected)	0 / 0	32 / 44	79 / 10	7 / 6	48 / 50	48 / 9				

Version 2.10.0 of FD (FlowDroid) and IccTA, ^{1}: Amandroid 3.1.2, *: No false positives before

Baseline Recap FlowDroid without any sort of cooperation is not able to detect any ICC related flows that are based on intents. Hence, FlowDroid alone does not detect any flows in case of ICC-Bench and the Intent-Matching benchmark. In case of DroidBench's ICC category some benchmark cases are not based on intent communication so that FlowDroid is actually able to analyze the associated cases and to achieve an F-measure greater than zero (0.348).

IccTA and DidFail are two extensions of FlowDroid that use IC3 (or its predecessor Epicc) internally. Since our cooperative ICC strategies also involve FlowDroid and IC3 it appears to be natural to compare them against these two tools. However, with respect to the baseline IccTA outperforms DidFail, thus, we do not repeat the results for DidFail here – DidFail's results can be found in the baseline (see Section 4.2).

The tool that performs best without cooperation was the old version of Amandroid (3.1.2). The associated results are repeated in the "Best" column below the "Baseline" headline (see Table 15).

Cooperative Results Since FlowDroid alone is not able to handle ICC based on intents, Strategy 1 achieves better results than FlowDroid for all three benchmark suites. In comparison to IccTA, this strategy falls behind in case of ICC-Bench and the Intent-Matching benchmark. For these two suites the CONNECT operator's intent sink and intent source matching capabilities are not sufficient. The CONNECT operator only detects a connection between an intent sink and an intent source if the respective intent triples are equal with respect to their string representation. For example, if `image/jpeg` is set as an intent sink's data element (MIME-Type) and `image/*` as the intent source's element, the CONNECT operator does not correctly interpret the wildcard (`*`) and falsely assumes that there is no ICC possible, since `image/jpeg` is not equal to `image/*`. Furthermore, if IC3 outputs only a wildcard (`*`) as action, category or data element for an intent triple, the CONNECT operator assumes that it matches any string. For instance, IC3 over-approximates and reports a wildcard as action attribute if the action of an intent is set as follows:

```
Intent intent = new Intent();
intent.setAction("prefix.action.string".substring(7));
```

In case of Strategy 2 the CONNECT operator is replaced by PIM. PIM compares intent sinks to intent sources by instantiating the associated intent triples on an Android (virtual) device, and by comparing these instances via the installed Android system. Whenever this system reports a match, PIM adds the respective inter-component flow from intent sink to intent source to its AQL-Answer output. This allows PIM to operate more precisely than the CONNECT operator, however, it cannot handle wildcards in action or category elements because those are not allowed by the Android framework. Consequently, Strategy 2 performs worse than Strategy 1 with respect to the DroidBench benchmark. Whenever IC3 reports a wildcard as action or category element, PIM does not detect a connection – this also leads to the elimination of the two false positives detected before. Due to this issue, we attempted to employ Amandroid to extract intent sinks and intent sources (Strategy 3), however, the results are identical, hence, the information about intent sinks and intent sources determined by IC3 and Amandroid seem to be equally (im-)precise.

As visible in Table 15, for ICC-Bench and the Intent-Matching benchmark better results are achieved. Most recognizable, the number of true positives detected in case of the Intent-Matching benchmark has been increased by 41. Sadly, the number of false positives has also been increased by 44. Still, Strategy 2 scores a better F-measure value than IccTA for this benchmark suite.

Strategy 4 is only applied in case of the Intent-Matching benchmark since the strategy aims at eliminating false positives but there are none with respect to Strategy 2 and the other two benchmark suites. The strategy succeeds as 41 false positives are eliminated by slicing prior to the analysis.

Figure 35 shows an example that illustrates how these false positives were introduced and avoided. Let us assume that the sensitive information extracted at the source (`getImei()`) is only attached to intent `i1`. This source is connected to the intent sink (`startActivity(...)`). This intent sink again is connected to the intent source (`getStringExtra(...)`) since intent `i2` matches intent filter `if2`. Lastly, the intent source is connected to a sink (`Log.i(...)`). Anyway, intent `i1`, the only intent that carries the sensitive data, never reaches the intent source. Accordingly, a taint flow that stretches from source to sink is a false positive.

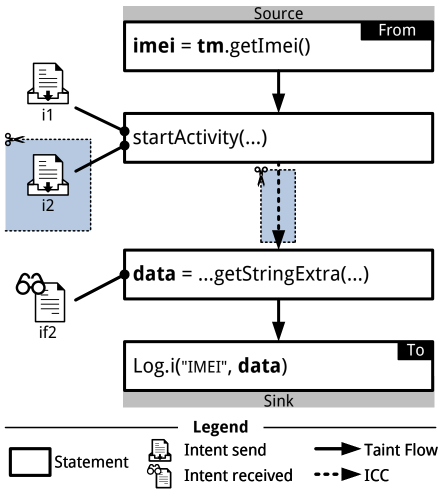


Figure 35: False Positive Elimination by Slicing (Cooperative Analysis 1)

Strategy 2 wrongly detects these false positives although all cooperating parties function as intended. FlowDroid detects both intra-component taint flows and IC3 together with PIM correctly determines the inter-component flow. Additionally, these flows can be stitched together since the inter-component flow starts at the end of the first intra-component flow and ends at the start of the second one.

To avoid such false positives, Jicer is added (Strategy 4). Considering the example again, it takes the given inter-component flow into account as input edge in order to slice from source to sink. In the end, intent `i2` and consequently the inter-component flow is sliced off (as illustrated in Figure 35). Thus, the false positive is avoided since the intra-component taint flows cannot be stitched together anymore. This effectively improves the cooperative analysis's performance as it achieves an F-measure value of 0.706.

Conclusion The columns entitled with "Difference" (see Table 15) give an answer to our main research question with respect to Cooperative Analysis 1. The differences denoted show that the cooperative strategies outperform FlowDroid and IccTA in most scenarios, however, the best tool in the baseline (Amandroid 3.1.2) still scores better F-measure values for all three benchmark suites. Since Strategy 2 and 3 deliver identical results the reason cannot be that Amandroid has more accurate information with respect to intent sinks and intent sources, however, it may handle wildcards in intent triples more explicitly and thereby over-approximate the given information in order to recognize more matches. Once IC3 (or another tool) provides more accurate information, PIM will also be able to correctly identify such matches without any approximation. We already got promising results by employing ICCBot instead on IC3 in first experiments. The wildcard issues seem to be solved – a thorough evaluation will be conducted in the future.

Furthermore, AQL-Answers currently do not include information about extras (the data, that is attached to and transported by intent sinks and intent sources) even though this information is provided by tools like IC3. Adding this information to the AQL-Answer data structure would allow tools like PIM to determine whether an inter-component flow actually carries sensitive data [78]. Thereby, further false positives could be avoided in the future.

5.2 Cooperative Analysis 2: Inter-App Communication (IAC) & App Merging

In the previous section, various approaches to handle ICC were presented. There are several methods to extend these approaches so that IAC scenarios can be analyzed. In the following two of these methods are evaluated. On the one hand, we extend two cooperative ICC strategies (1 & 2). On the other hand, we employ a method that relies on merging (or combining) Android apps (`.apk` files).

5.2.1 Cooperative Analysis 2: Tools

The tools used while evaluating Cooperative Analysis 2 are listed in Table 16. All 13 taint analysis tools that have been used in the baseline are used by at least one cooperative strategy. IC3, the CONNECT operator and PIM fulfill the same purpose as before, which is finding flows between components or, in this case, apps. ApkCombiner and AMT are two candidates that can be used to merge two (or more) `.apk` files into a single one.

As benchmark suites we employ the IAC category of DroidBench and the Feature-Checking benchmark. For the latter, only the IAC related part is re-evaluated.

Table 16: Cooperative Analysis 2: Tools and Benchmark Suites

Tools	Benchmark Suites
• All 13 baseline tools*	• DroidBench (IAC only)
• ApkCombiner	• Feature-Checking
• AMT	
• IC3	
• CONNECT^d	
• PIM	

*: Also as standalone tools for comparison (see Table 9, Page 101),
^{d}: Default tool (see Section 3.4.3, Page 85)

5.2.2 Cooperative Analysis 2: Strategies

Before introducing the strategies considering IAC, we must mention that BREW asks different questions whenever the expected or not-expected taint flow of a benchmark case starts in one app and ends in another. In fact, instead of asking for flows `IN` one app, it asks a `FROM-TO`-question:

```
Flows FROM App(%APP_APK_FROM%) TO App(%APP_APK_TO%) ?
```

BREW then replaces `%APP_APK_FROM%` and `%APP_APK_TO%` with the respective apps. Also, when the expected or not-expected taint flow starts and ends in the same app but another app, the so-called *bridge app*, is assigned to the same benchmark case, BREW asks the following initial query to find flows to and back from the bridge app:

```
CONNECT [
    Flows FROM App(%APP_APK_FROM%) TO App(%APP_APK_TO%) ?,
    Flows FROM App(%APP_APK_TO%) TO App(%APP_APK_FROM%) ?
] ?
```

The individual strategies are then applied twice – to both inner questions (see Line 2 and 3 in the listing above).

Strategy 1: Default The first and default strategy in the context of IAC is very similar to the default strategy for ICC. Instead of asking for flows, intent sinks and intent sources of the same app, we ask for the same subjects of interest of all involved apps:

```
CONNECT [
    Flows IN App(%APP_APK_FROM%) ?,
    Flows IN App(%APP_APK_TO%) ?,
    CONNECT [
        IntentSinks IN App(%APP_APK_FROM%) ?,
        IntentSinks IN App(%APP_APK_TO%) ?,
        IntentSources IN App(%APP_APK_FROM%) ?,
        IntentSources IN App(%APP_APK_TO%) ?
    ] ?
] ?
```

Because it is possible for a taint flow to go back and forth between apps, we do not only ask for intent sinks in one app and intent sources in the other, but for both subjects of interest in both apps. This always gives us all possible inter-app flows, no matter what direction they are oriented.

Strategy 2: +PIM Analogously to the second ICC strategy, we are only replacing the inner operator for Strategy 2. Thereby, PIM is employed to connect intent sinks and intent sources instead of the AQL-System's CONNECT operator.

```
CONNECT [
    Flows IN App(%APP_APK_FROM%) ?,
    Flows IN App(%APP_APK_TO%) ?,
    MATCH [
        IntentSinks IN App(%APP_APK_FROM%) ?,
        IntentSinks IN App(%APP_APK_TO%) ?,
        IntentSources IN App(%APP_APK_FROM%) ?,
        IntentSources IN App(%APP_APK_TO%) ?
    ] ?
] ?
```

Strategy 3: Combining with ApkCombiner Strategy 3 is the first of two strategies that rely on app merging. With respect to this strategy ApkCombiner is triggered as a preprocessor by the keyword `COMBINE`:

```
Flows IN App(%APP_APK_FROM%, %APP_APK_TO% | 'COMBINE') ?
```

Thereby the apps referenced by `%APP_APK_FROM%` and `%APP_APK_TO%` are merged into a single one such that we are able to ask an `IN`-question again as we now only have ICC.

Strategy 4: Combining with AMT The last IAC strategy is equal to Strategy 3 with one exception: the keyword that triggers ApkCombiner as preprocessor is replaced by `MERGE`, the keyword that triggers AMT instead.

```
Flows IN App(%APP_APK_FROM%, %APP_APK_TO% | 'MERGE') ?
```

5.2.3 Cooperative Analysis 2: Results

Before having a look at the taint analysis results produced by all strategies, we compare ApkCombiner and AMT with respect to their ability to combine apps that belong to individual benchmark cases. Table 17 lists all IAC benchmark cases of DroidBench – one per row. For each of these benchmark cases, the app that holds the respective source is denoted as well as the app that contains the respective sink. Whenever a benchmark case includes a bridge app, this app is denoted as well. The last two columns show whether ApkCombiner and AMT are able to merge the respective apps.

Table 17: DroidBench Cases (IAC only) *Merge-able* by ApkCombiner and AMT (Cooperative Analysis 2)

DroidBench ID	Apps Source	Bridge	Sink	Apk-Combiner	AMT
101	`DeviceId_Broadcast1`	—	`Collector`	✗	✓
102	`DeviceId_contentProvider1`	—	`Collector`	✗	✓
103	`DeviceId_OrderedIntent1`	—	`Collector`	✗	✓
104	`DeviceId_Service1`	—	`Collector`	✓	✓
105	`Location1`	—	`Collector`	✗	✓
106	`Location_Broadcast1`	—	`Collector`	✗	✓
107	`Location_Service1`	—	`Collector`	✗	✓
108	`SendSMS`	`Echoer`	`SendSMS`	✓	✓
109	`SendSMS`	—	`Echoer`	✓	✓
110	`StartActivityForResult1`	`Echoer`	`StartActivityForResult1`	✓	✓
111	`StartActivityForResult1`	—	`Echoer`	✓	✓
✗: failed merge, ✓: successful merge			Number of successful merges (✓):	5	11

While ApkCombiner is only able to merge apps that belong to 5 different benchmark cases, AMT successfully merges apps in all 11 cases. ApkCombiner fails for 6 benchmark cases since it cannot handle apps that were built for more recent Android versions – similar to some taint analysis tools that are not updated anymore (see Subsection 4.2.2). In addition, the paper that proposes AMT [76] shows that it is as accurate as ApkCombiner while merging apps, i.e. the taint analysis results produced for apps merged by both tools are identical. For these reasons, we abandon Strategy 3 (ApkCombiner) and only continue with the remaining strategies.

Table 18 summarizes the results for all strategies in relation to both benchmarks. The first column of each row refers to one strategy and the taint analysis tool that has been employed as basis to compose the cooperative analysis. For Strategy 1 and 2 this basis is FlowDroid as before and discussed in the context of ICC (see Section 5.1). However, for this cooperative analysis the up-to-date version of FlowDroid could not be employed since the newer versions ($\geq$ 2.7.1) do not report decisive flows regarding IAC. For example, in case of the `DeviceId_Broadcast1` app (DroidBench benchmark case 101 – see Table 17) the newer versions of FlowDroid do not report an intra-component taint flow from a source (`getDeviceId()`) to an intent sink (`startActivityForResult(...)`) – no matter what list of sources and sinks is given. Presumably, FlowDroid recognizes no intent source in the same app and, hence, discards flows to this intent sink. The old version of FlowDroid does report such flows and is consequently chosen as basis for Strategy 1 and 2. In case of Strategy 4 all 13 taint analysis tools are individually employed as basis. The column entitled "Tool(s) added" refers to the tools added to this basis to form the cooperative analysis.

As the title "Baseline" suggests, the associated column repeats the respective baseline results. The column entitled with "Coop. Analysis" shows the results of the cooperative

Table 18: Feature Promises (F.-P.) Regarding the FEATURE-CHECKING Benchmark Suite, and F-measure (F-m.), True Positives (TPs) with Respect to DROIDBENCH (IAC only; 11 Positive Cases – Cooperative Analysis 2)

Strategy:	**Baseline**			**Tool(s) added**	**Coop. Analysis**		
Taint Analysis Tool (Version)	F.-P.	F-m.	TPs	(for the coop. analysis)	F.-P.	F-m.	TPs
1: FLOWDROID (old)	×	0.000	0	+ IC3 + CONNECT[d]	×	▲ 0.308	2
2: FLOWDROID (old)	×	0.000	0	+ IC3 + PIM	✔	▲ **0.625**	**5**
4: AMANDROID (3.1.2)	◯×*	0.000	0	+ AMT	◯✔	▲ 0.429	3
4: AMANDROID (3.2.0)	◯×*	0.000	0	+ AMT	◯×	0.000	0
4: AMANDROID (3.2.1)	◯×*	0.000	0	+ AMT	◯×	0.000	0
4: DIALDROID	◯×	**0.625**	**5**	+ AMT	◯✓	**0.625**	**5**
4: DIDFAIL	◯×*	0.533	4	+ AMT	◯✔	0.533	4
4: DROIDSAFE	◯×*	0.000	0	+ AMT	◯✔	▲ 0.167	1
4: FLOWDROID (old)	×	0.000	0	+ AMT	×	0.000	0
4: FLOWDROID (2.7.1)	×	0.000	0	+ AMT	×	0.000	0
4: FLOWDROID (2.9.0)	×	0.000	0	+ AMT	×	0.000	0
4: FLOWDROID (2.10.0)	×	0.000	0	+ AMT	×	0.000	0
4: ICCTA (old)	◯×*	0.000	0	+ AMT	◯✔	0.000	0
4: ICCTA (2.9.0)	◯×*	0.000	0	+ AMT	◯✓	0.000	0
4: ICCTA (2.10.0)	◯×*	0.000	0	+ AMT	◯✓	0.000	0

[d]: Default tool (see Section 3.4.3, Page 85)

◯ supported, ✔ confirmed, ✓ partially confirmed, × not confirmed, * not promised without cooperative analysis

analysis. Both columns contain three elements: the IAC feature promise (as in Subsection 4.2.1) and the F-measure value as well as the true positive count.

Regarding the IAC feature promise, we can observe that many tools (partially) fulfill their promise once the apps to be analyzed are merged. Only FLOWDROID and two versions of AMANDROID (3.2.0, 3.2.1) fail to do so, however, FLOWDROID in contrast to AMANDROID never promised to support IAC. With respect to the reported F-measure values, the baseline suggests that DIALDROID, a white-box cooperative analysis that uses adapted versions of FLOWDROID and IC3, is the best standalone tool for IAC scenarios (F-measure: 0.625). Since Strategy 2 performs as well as DIALDROID, we may conclude that the black-box cooperative analysis (Strategy 2) is as performant as the white-box cooperative analysis (DIALDROID). DIDFAIL reaches an F-measure of 0.533 and thereby functions best as a basis for Strategy 4. DIDFAIL is also implemented through a white-box cooperative analysis between EPICC and FLOWDROID, so the best truly standalone tool again is AMANDROID (3.1.2 – F-measure: 0.429).

Conclusion In summary, both cooperative methods – combining results (Strategy 1 & 2) and merging apps (Strategy 3 & 4) – enable taint analysis tools to be able to analyze IAC scenarios. To further answer the main research question: one cooperative IAC strategy (Strategy 2) outperforms any truly standalone taint analysis tool. In terms of F-measure, the standalone white-box cooperative analysis DIALDROID keeps up and appears to be equally precise. If we also take the FEATURE-CHECKING benchmark into account, the cooperative analysis (Strategy 2) performs slightly better than DIALDROID as it successfully analyzes all related benchmark cases.

5.3 Cooperative Analysis 3: Reflection & Native Code

Apart from challenges such as ICC and IAC, which are introduced by the Android system, other challenges have their origin in the programming language used to implement apps, namely Java (or Kotlin – see Section 2.1). Two examples are reflection and native code. Both are often extensively used by malicious developers for a single purpose: hiding frauds (see Subsection 2.1.4). Analyses are also negatively impacted by such concepts. In the following two cooperative strategies are presented that allow to build a cooperative analysis which is able to deal with reflection and native code.

5.3.1 Cooperative Analysis 3: Tools

The tools employed in Cooperative Analysis 3 are listed in Table 19. NOAH comes into play to detect sources and sinks in native code. To do so NOAH takes an app and optionally a list of sources and sinks as input and outputs:

- flows *to* native method calls *from* sources in the native code (*native sources*),
- flows *from* native method calls *to* sinks in the native code (*native sinks*), and
- an adapted list of sources and sinks that contains the respective native method calls.

The flows are output in form of an AQL-Answer, however, the output list of sources and sinks follows FlowDroid's format. The sources and sinks list is created by extending the given input list, if present. Otherwise, a configurable default list is used and extended. Due to the output format of this list, NOAH is limited to be used together with FlowDroid. The CONNECT operator is used to stitch together flows found by NOAH and FlowDroid.

DroidRA is added to resolve reflection. It can be employed as a preprocessor as it takes an `.apk` file as input and also produces an `.apk` file as output. Because of that it is also compatible with any taint analysis tool.

Two categories of DroidBench tailored to reflection and native code are used as benchmark suites.

Table 19: Cooperative Analysis 3: Tools and Benchmark Suites

Tools	Benchmark Suites
• All 13 baseline tools*	• DroidBench (Native only)
• NOAH	• DroidBench (Reflection only)
• DroidRA	
• CONNECT^d	

*: Also as standalone tools for comparison (see Table 9, Page 101),

^{d}: Default tool (see Section 3.4.3, Page 85)

5.3.2 Cooperative Analysis 3: Strategies

The names of the two strategies presented in the following refer to the language features they deal with.

Strategy 1: Native Code The strategy denoted in the listing below uses NOAH twice (Lines 3, 5) and FlowDroid once (Line 2):

```
1 CONNECT [
2     Flows IN App(%APP_APK_IN%) WITH 'SourcesAndSinks' = {
3         Arguments IN App(%APP_APK_IN%) FEATURING 'Native' !
4     } ?,
5     Flows IN App(%APP_APK_IN%) FEATURING 'Native' ?
6 ] ?
```

By answering the question asking for `Arguments`, NOAH computes the extended list of sources and sinks and forwards it to FlowDroid (Line 2–4). Thereby, the detected native sources and native sinks can be taken into account by FlowDroid during its analysis. By answering the question in Line 5, NOAH reports the respective flows between native method calls and native sources or native sinks. In the end, the flows determined by FlowDroid and NOAH are connected via the surrounding `CONNECT` operator.

Strategy 2: Reflection The strategy to handle reflection produces a simple query which is similar to the default one:

```
Flows IN App(%APP_APK_IN% | 'RESOLVE') ?
```

Only the keyword `RESOLVE` has been added. This keyword triggers DroidRA in order to resolve reflective calls in the given app before it is analyzed.

5.3.3 Cooperative Analysis 3: Results

The results with respect to Strategy 1 (Native Code) can be found in Table 20 whereas the results in regard to Strategy 2 (Reflection) are visible in Table 21. Both tables adhere to the same three-column structure. The first column names a tool, the second repeats the baseline results and the last column presents the results achieved by the cooperative analysis. For both, the repeated baseline results and the cooperative analysis results, the F-measure (F-m.) as well as the number of true positives (TPs) is denoted.

Table 20: F-measure (F-m.), True Positives (TPs) for DroidBench (Native only; 5 Positive Cases – Cooperative Analysis 3)

Tool	Baseline		Strategy 1	
(Version)	F-m.	TPs	F-m.	TPs
FlowDroid (old)	0.000	0	▲ **0.889**	**4**
FlowDroid (2.7.1)	**0.333**	**1**	▲ 0.750	3
FlowDroid (2.9.0)	0.000	0	▲ 0.571	2
FlowDroid (2.10.0)	0.000	0	▲ 0.571	2
AmanDroid (3.1.2)	**0.333**	**1**	—	—

The baseline results show that only FlowDroid (2.7.1) and AmanDroid (3.1.2) were able to detect one out of five taint flows with respect to native code (see Table 20). The best cooperative approach clearly performs better by only missing one taint flow. In terms of F-measure this is equivalent to an improvement of ~56% ($0.889 - 0.333 = 0.556 \approx 56\%$).

Further investigation shows that the only missed taint flow is actually found as well, however, BREW does not count it as detected since it defines taint flows purely on the level of statements in the non-native code. Thereby, BREW expects a flow from the native method call to the exact same native method call – a tautology. Obviously, no analysis reports a connection from and to the same statement, instead the cooperative analyses report a flow from source to sink in the native code via the native method call.

Table 21: F-measure (F-m.), True Positives (TPs) for DroidBench (Reflection only; 9 Positive Cases – Cooperative Analysis 3)

Tool	**Baseline**		**Strategy 2**	
(Version)	F-m.	TPs	F-m.	TPs
Amandroid (3.1.2)	0.364	2	▲ **0.800**	**6**
Amandroid (3.2.0)	0.200	1	▲ **0.800**	**6**
Amandroid (3.2.1)	0.200	1	▲ **0.800**	**6**
DIALDroid	0.000	0	0.000	0
DidFail	0.200	1	▼ 0.000*	0
DroidSafe	**0.615**	**4**	▲ 0.615	4
FlowDroid (old)	0.200	1	▲ **0.800**	**6**
FlowDroid (2.7.1)	0.200	1	▲ **0.800**	**6**
FlowDroid (2.9.0)	0.200	1	▲ **0.800**	**6**
FlowDroid (2.10.0)	0.200	1	▲ **0.800**	**6**
IccTA (old)	0.200	1	▲ 0.615	4
IccTA (2.9.0)	0.200	1	▲ **0.800**	**6**
IccTA (2.10.0)	0.200	1	▲ **0.800**	**6**

*: 6 crashes

Similarly, a cooperative analysis using Strategy 2 outperforms standalone analysis tools. The best standalone tool (DroidSafe) detects four out of nine taint flows that involve reflection. With cooperation in place DroidSafe is unable to detect more taint flows, however, all versions of Amandroid, FlowDroid and IccTA (except the old version of IccTA) are enabled to detect six taint flows. No matter which taint analysis tools is used, some taint flows always remain undetected because DroidRA is not able to fully resolve the included reflection.[32] DidFail is not able to analyze any of the preprocessed apps. It has shown difficulties before while dealing with different versions during the baseline experiments, thus, we assume a similar reason here. Comparing the F-measure scored by DroidSafe (0.615) with one of the best cooperative approaches (0.8), we get an improvement of 0.185. The individual improvement of certain tools is much higher, for example, FlowDroid detects five more taint flow which is equal to an improvement of 0.6.

Conclusion The two cooperative analyses using Strategy 1 and 2 clearly outperform all standalone tools. In case of Strategy 2, ten analysis tools scored better results with the cooperative strategy in place. Strategy 1, which is currently only applicable to FlowDroid, enhances the results for all four versions of it. In the future, NOAH will be adapted to not only output flows but also sources and sinks in AQL format. Then, with a little help of the AQL, the strategy can also be applied to Amandroid. The latter will be possible, since both tools (FlowDroid and Amandroid) use configurable lists of sources and sinks. This feature will also be discussed and brought to use by the cooperative analysis presented in the Section 5.5.

[32]Detailed results, including in which cases the reflection could not be resolved, can be found in Appendix A.6.2.

5.4 Cooperative Analysis 4: False Positive Elimination

Taint analyses, in particular static ones, often (over-)approximate in order to detect as many taint flows as possible, however, this comes at a cost: most of the time not only the number of true positives but also the number of false positives grows. With Cooperative Analysis 4 we tackle this issue and try to eliminate false positives by comparing the results of a typical taint analysis tool with the results of a specialized tool.

5.4.1 Cooperative Analysis 4: Tools

As taint analysis tools we employ all 13 tools in our scope (see Table 22). The AQL-CheckOperator is used to check the taint flows found by each of these tools, i.e. each taint flow that is confirmed by another tool is not removed from the first tool's result. Optimally, the second tool would be able to soundly analyze which of the detected taint flows are actually feasible. For example, a dynamic taint analysis tool that takes taint flows as input to guide its app exploration would be a perfect match, however, such a tool does currently not exist. An overview of the existing dynamic taint analysis tools is provided in the background chapter (see Subsection 2.4.2). Hence, instead of a dynamic analysis tool we employ HornDroid, an atypical taint analysis tool. Atypical since it is based on logical reasoning and since it does not output complete taint flows but sinks that truly leak sensitive data.

Table 22: Cooperative Analysis 4: Tools and Benchmark Suites

Tools	Benchmark Suites
• All 13 baseline tools*	• DroidBench
• HornDroid	
• AQL-CheckOperator	

*: Also as standalone tools for comparison (see Table 9, Page 101)

The AQL-CheckOperator in combination with HornDroid treats a taint flow as confirmed or checked, if HornDroid declares the respective sink as (potentially) leaking. Thus, whenever HornDroid declares that a sink cannot leak sensitive data, all associated taint flows are removed from the AQL-Answer given by the initial taint analysis tool.

5.4.2 Cooperative Analysis 4: Strategy

In regard to Cooperative Analysis 4 only one cooperative strategy is composed:

```
1  CHECK [
2      Flows IN App(%APP_APK_IN%) ?,
3      Flows IN App(%APP_APK_IN%) USING 'HornDroid' ?
4  ] ?
```

The AQL-CheckOperator is triggered by the operator keyword `CHECK` (see Line 1 in the listing above). While the first inner question (Line 2) is answered by one of our 13 taint analysis tools, the second is always answered by HornDroid (Line 3).

5.4.3 Cooperative Analysis 4: Results

All result of Cooperative Analysis 4 for the DroidBench benchmark are presented in Table 23. The first column refers to the taint analysis tool that has been employed. The second, third and fourth column are structured the same way. Each subcolumn entitled with “Baseline” refers to the baseline results of the respective tool. Each subcolumn entitled with “Coop.” refers to the results of Cooperative Analysis 4. The subcolumns entitled with “Δ” show the relative difference between both results, for example, DidFail found 21 false positive taint flows of which 17 could be eliminated due to the cooperative analysis, hence, almost $\Delta_{FP} \approx 80.95\%$ of these 21 false positives were eliminated. In the last column we compare the relative number of removed false positives with the relative number of removed true positives.

Table 23: False Positives, True Positives and F-measure for DroidBench (Cooperative Analysis 4)

Taint Analysis	**False Positives**			**True Positives**			**F-measure**			Δ_{FP} −
Tool (Version)	Baseline	Coop.	Δ_{FP}	Baseline	Coop.	Δ_{TP}	Baseline	Coop.	Δ	Δ_{TP}
Amandroid (3.1.2)	17	6	**64.71%**	82	33	59.76%	0.633	0.332	47.55%	4.95%
Amandroid (3.2.0)	13	5	**61.54%**	38	25	34.21%	0.360	0.263	26.94%	27.33%
Amandroid (3.2.1)	13	5	**61.54%**	38	25	34.21%	0.360	0.263	26.94%	27.33%
DIALDroid	0	0	0.00%	12	0	100.00%	0.140	0.000	100.00%	-100.00%
DidFail	21	4	**80.95%**	64	30	53.13%	0.522	0.309	40.80%	27.83%
DroidSafe	18	7	**61.11%**	85	47	44.71%	0.646	0.439	32.04%	16.41%
FlowDroid (old	15	5	**66.67%**	84	42	50.00%	0.649	0.406	37.44%	16.67%
FlowDroid (2.7.1)	15	5	**66.67%**	83	43	48.19%	0.643	0.413	35.77%	18.47%
FlowDroid (2.9.0)	13	4	**69.23%**	86	43	50.00%	0.664	0.415	37.50%	19.23%
FlowDroid (2.10.0)	15	5	**66.67%**	84	42	50.00%	0.649	0.406	37.44%	16.67%
IccTA (old)	19	6	**68.42%**	87	41	52.87%	0.654	0.396	39.45%	15.55%
IccTA (2.9.0)	13	4	**69.23%**	92	43	53.26%	0.694	0.415	40.20%	15.97%
IccTA (2.10.0)	15	5	**66.67%**	90	42	53.33%	0.679	0.406	40.21%	13.33%

The strategy shows great success with respect to the number of eliminated false positives. For any tool at least 61% of false positives are eliminated. However, the cooperative strategy unexpectedly also decreased the number of true positives such that up to almost 60% (in case of Amandroid 3.1.2) of all true positives were not found anymore. DIALDroid was ignored in this evaluation since HornDroid is not able to detect ICC or IAC related flows, thus, no flow found by DIALDroid could ever be confirmed by HornDroid. The same argument holds for any other tool that found ICC or IAC related flows with respect to the baseline. Furthermore, HornDroid timed out for six benchmark apps after 20 minutes, thus, in these cases no flows were confirmed.[33] Another reason for the large number of unconfirmed true positives may be caused by the absence of sinks in HornDroid’s results, i.e. HornDroid categorizes sinks as (potentially) leaking or definitely not leaking, however, it may also not categorize some sinks at all. The AQL-System converts HornDroid’s results into AQL-Answers by collecting all sinks that were categorized as leaking or potentially leaking. In consequence, a sink that has not been mentioned in HornDroid’s result was treated the same way as a sink for which HornDroid found that it cannot be leaking. As a result, too many taint flows may have been removed. In the future, we will attempt to only eliminate flows that are declared as not leaking. To do so, the AQL-System’s converter for HornDroid must be replaced with a customized one and the AQL-CheckOperator must be adapted. For those reasons, the F-measure values determined for the cooperative analysis are lower than those determined in the baseline.

[33]Details, including in which cases HornDroid timed out, can be found in the artifact (see Appendix A.6).

Conclusion In the end, the last column of Table 23 gives an answer to our main research question. With respect to any tool and relative numbers, the cooperative analysis removed more false than true positives (in absolute numbers more true positives were removed). Nonetheless, Cooperative Analysis 4 already indicates the potential of cooperative analyses with respect to the elimination of false positives.

5.5 Cooperative Analysis 5: Sources & Sinks

The goal of this cooperative analysis is to increase the accuracy of standalone taint analysis tools by adapting the source and sink lists they use. Often analysis tools are less accurate, since they consider too many or too few sources and sinks. Taint analysis tools should only take sources and sinks into account that are actually relevant with respect to feasible taint flows. We set up different cooperative strategies triggering multiple tools to respect only relevant sources and sinks.

5.5.1 Cooperative Analysis 5: Tools

This cooperative analysis employs the taint analysis tools AMANDROID and FLOWDROID. These two allow us to adapt the sources and sinks considered during analysis by changing a single text file, the *source and sink list*. The sources and sinks considered by other tools, such as DROIDSAFE, are hard coded in the tool's source code. Furthermore, we only consider these two mature tools and ignore variants of it, since this cooperative analysis does not focus on a feature that is particularly supported by a variant. For example, DIALDROID uses FLOWDROID internally and ICCTA is an extension of FLOWDROID, however, both tools are tailored to ICC or IAC scenarios.

Overall three versions of AMANDROID and FLOWDROID are considered, the version that showed the best performance in the baseline (AMANDROID 3.1.2, FLOWDROID old), the most recent version (AMANDROID 3.2.1, FLOWDROID 2.10.0) and, for reproducibility, another version that has been used in the TAINTBENCH study [91] (AMANDROID 3.2.0, FLOWDROID 2.7.1). In addition to these two tools, overall four default tools and one additional tool come into play. A list of all tools involved in this cooperative analysis can be found in Table 24. The two default operators TOAD (to AMANDROID) and TOFD

Table 24: Cooperative Analysis 5: Tools and Benchmark Suites

Tools	Benchmark Suites
• FLOWDROID (old, 2.7.1, 2.10.0)	• DROIDBENCH
• AMANDROID (3.1.2, 3.2.0, 3.2.1)	• TAINTBENCH
• TBSASMAPPER	
• SOURCESINKFINDER[d]	
• TOFD[d]	
• TOAD[d]	
• UNIFY[d]	

[d]: Default tool (see Section 3.4.3, Page 85)

(to FLOWDROID) are used as input converters, i.e. both take an AQL-Answer as input and convert all sources and sinks contained in it into a tool-specific sources and sink list. TBSASMAPPER and the SOURCESINKFINDER are employed to gather information about sources and sinks that can be found in a single benchmark app or a specific benchmark case. To do so, the default tool SOURCESINKFINDER compares all statements inside an app with a configurable list of sources and sinks. Whenever a statement included in the code of an app matches a statement in this list, the statement is added respectively as source or sink to the output AQL-Answer. TBSASMAPPER also outputs an AQL-Answer that holds a list of source and sinks, however, it constructs that list on the basis of the information given in TAINTBENCH's ground truth definition, more precisely, for a given benchmark app or case it collects all the documented sources and sinks.

5.5.2 Cooperative Analysis 5: Strategies

With these tools at hand the following eight experiments (see Table 25) for the two benchmarks DroidBench and TaintBench can be conducted by employing different cooperative strategies.

Table 25: Cooperative Analysis 5: Experiments

Identifier	Benchmark Suite	Strategy	Sources and Sinks
DB1	DroidBench	1	Default List
DB2	"	2	Benchmark suite specific
DB3	"	3	Benchmark app specific
DB4	"	4	Benchmark case specific
TB1	TaintBench	1	Default List
TB2	"	2	Benchmark suite specific
TB3	"	3	Benchmark app specific
TB4	"	4	Benchmark case specific

Strategy 1: Default For Experiment 1 the default list of sources and sinks, each tool is shipped with, is used. The associated strategy simply transforms the query given by BREW into the following:

```
Flows IN App(%APP_APK_IN%) WITH 'SourcesAndSinks' = 'DefaultList.txt' ?
```

`DefaultList.txt` is a placeholder here. At the time of analysis, a tool-specific list is used. The results determined for experiments related to this strategy match the results determined in the baseline.[34]

Strategy 2: Suite-Level BREW allows us to export sources and sinks that belong to a whole benchmark suite. Thus, we used BREW to export one list that is specific to DroidBench and another one that is specific to TaintBench. For Experiment 2 these specific lists are used:

```
Flows IN App(%APP_APK_IN%) WITH 'SourcesAndSinks' = 'SpecificList.txt' ?
```

Again `SpecificList.txt` is a placeholder which will be replaced by the benchmark suite-specific list – in Amandroid's or FlowDroid's format respectively.

Strategy 3: App-Level In context of Strategy 3 we want to use lists of sources and sinks that are specific with respect to the benchmark app analyzed. To get this list the query is transformed into the following when dealing with DroidBench:

```
Flows IN App(%APP_APK_IN%) WITH 'SourcesAndSinks' = {
    TOAD [
        UNIFY [
            Sources IN App(%APP_APK_IN%) ?,
            Sinks IN App(%APP_APK_IN%) ?
        ] ?
    ] !
} ?
```

[34]In case of FlowDroid an older version of the default list of sources and sinks has been used for Cooperative Analysis 5, therefore, the results are slightly different (Appendix A.6 contains the exact lists used in both cases).

The two inner queries (Line 4, 5 in the listing above) asking for sources and sinks trigger the default tool SourceSinkFinder. The `UNIFY` operator (Line 3) merges the answers this tool replies into a single one and the `TOAD` operator (Line 2) transforms this answer into a tool-specific source and sink list. In case of FlowDroid, `TOFD` is used instead of `TOAD`.

When dealing with TaintBench only the inner queries must be adapted by appending `USING 'TBSaSMapper'` (see Lines 4, 5 in the listing below):

```
1  Flows IN App(%APP_APK_IN%) WITH 'SourcesAndSinks' = {
2      TOAD [
3          UNIFY [
4              Sources IN App(%APP_APK_IN%) USING 'TBSaSMapper' ?,
5              Sinks IN App(%APP_APK_IN%) USING 'TBSaSMapper' ?
6          ] ?
7      ] !
8  } ?
```

Thereby, the TBSaSMapper is used instead of the default tool SourceSinkFinder.

Strategy 4: Case-Level Lastly, for Strategy 4 we want to use a similar strategy but provide more information to the tools determining sources and sinks such that we get case-specific source and sink lists instead of app-specific ones. In the context of DroidBench this is realized by asking for sources and sinks with respect to a certain statement (see Line 4 and 5 in the listing below):

```
1  Flows IN App(%APP_APK_IN%) WITH 'SourcesAndSinks' = {
2      TOAD [
3          UNIFY [
4              Sources IN Statement(%STATEMENT_FROM%,
                   → %LINENUMBER_FROM%)->App(%APP_APK_IN%) ?,
5              Sinks IN Statement(%STATEMENT_TO%, %LINENUMBER_TO%
                   → )->App(%APP_APK_IN%) ?
6          ] ?
7      ] !
8  } ?
```

The content for the statement placeholders (`%STATEMENT_...%`) as well as for the linenumber placeholders (`%LINENUMBER_...%`) is provided by BREW. For this purpose BREW must be configured to initially ask a from-to query with up to statement-level references:

```
Flows FROM
    Statement(%STATEMENT_FROM%, %LINENUMBER_FROM%)->
    Method(%METHOD_FROM%)->Class(%CLASS_FROM%)->
    App(%APP_APK_FROM%)
TO
    Statement(%STATEMENT_TO%, %LINENUMBER_TO%)->
    Method(%METHOD_TO%)->Class(%CLASS_TO%)->
    App(%APP_APK_TO%)
?
```

For TaintBench we must forward TaintBench's case ID to TBSaSMapper. This can be done via a strategy that produces the following query:

```
Flows IN App(%APP_APK_IN%) WITH 'SourcesAndSinks' = {
    TOAD [
        UNIFY [
            Sources IN App(%APP_APK_IN%) USING 'TBSaSMapper'
                → WITH 'TaintBenchID' = '%ID%' ?,
            Sinks IN App(%APP_APK_IN%) USING 'TBSaSMapper'
                → WITH 'TaintBenchID' = '%ID%' ?
        ] ?
    ] !
} ?
```

While loading TaintBench into BREW, BREW tracks the ids of all imported benchmark cases such that the replacement for the `%ID%` placeholder can also be provided by BREW.

5.5.3 Cooperative Analysis 5: Results

Next we will have a look at the results achieved under these cooperative strategies.

Table 26: Intersection (∩) and Difference (\) of Source and Sink Sets Used by Analysis Tools (Amandroid, FlowDroid) and Involved in Benchmark Cases of DroidBench and TaintBench.

$B =$ / $A =$	Amandroid		FlowDroid		DroidBench		TaintBench	
	Sources	Sinks	Sources	Sinks	Sources	Sinks	Sources	Sinks
Intersection ($\|A \cap B\|$)								
Amandroid	*30*	*42*	24	38	**4**	**8**	**6**	4
FlowDroid	24	38	*89*	*133*	**7**	**9**	**12**	**8**
DroidBench	**4**	**8**	**7**	**9**	*15*	*23*	7	4
TaintBench	**6**	**4**	**12**	**8**	7	4	*44*	*44*
Difference ($\|A \setminus B\|$)								
Amandroid	*0*	*0*	6	4	26	34	24	38
FlowDroid	65	95	*0*	*0*	82	124	77	125
DroidBench	**11**	**15**	**8**	**14**	*0*	*0*	8	19
TaintBench	**38**	**40**	**32**	**36**	37	40	*0*	*0*

Baseline Recap The results of DB1 & TB1 are visualized in three bar charts (see Figure 36). These results are a replication of the respective baseline findings (see Subsection 4.2.4, 4.2.5). Due to the visualization, differences between the results for DB1 and TB1 can easily be identified. In particular, the recall values determined for DroidBench and TaintBench deviate from each other.

The main reason for this deviation may be embodied in the sources and sinks list used. Table 26 shows the intersection and difference of the involved source and sink lists. For example, the cursive number (in the intersection area) show how many sources and sinks are considered by FlowDroid and Amandroid as well as how many appear in DroidBench and TaintBench. The bold numbers (in this area) reveal that the sources and sinks considered by tools have only minor commonalities with the sources and sinks that appear in benchmark suites. Most importantly, the bold numbers (in the difference area) show how many sources and sinks appear in the individual benchmark suites that are not recognized by the analysis tools. From a relative perspective more sources and sinks occurring in TaintBench are ignored than occurring in DroidBench. Hence, we

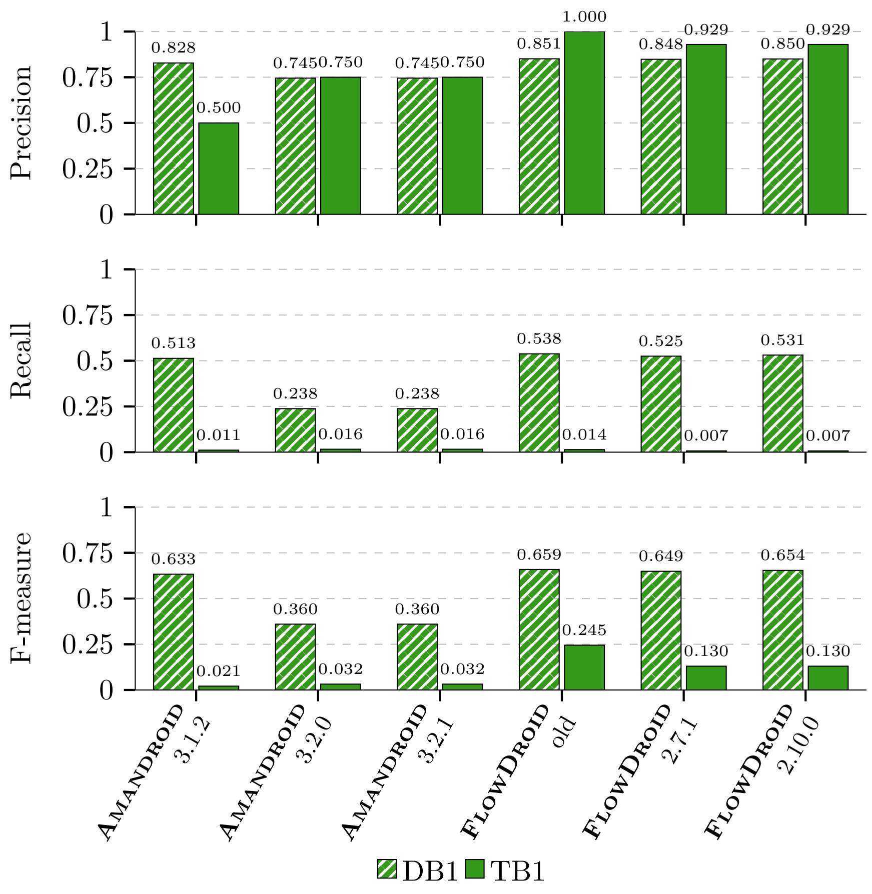


Figure 36: Precision / Recall / F-measure for Cooperative Analysis 5 (DB1 & TB1)

expect a slight improvement in the results determined for DB2, DB3 and DB4 if compared to DB1, but a large improvement for TB2, TB3 and TB4 in comparison to TB1. If the tools are not over-adapted in any other way to micro benchmarks like DroidBench, the tools should be comparably accurate on DroidBench and TaintBench.

Cooperative Results The bar charts in Figure 37 visualize the results for all experiments. The striped bars refer to DroidBench results and solid bars to TaintBench results. By comparing striped and solid bars, it becomes visible that most tools score better values with respect to DroidBench. Only the old version of FlowDroid manages to achieve an F-measure value for TaintBench (∼68%) which is almost as high as the best measured F-measure for DroidBench (∼69%). All other tools score significantly worse values for TaintBench no matter which cooperative strategy is in place. In particular, the recall of all other tools deviates from the recall measurable in DroidBench context. Furthermore, a trend becomes visible in case of FlowDroid: the more accurate the source and sink lists are, the higher the accuracy scores are.

The overall best F-measure score is also reached by FlowDroid (old) in case of experiment DB2 (∼69%). As a visual baseline we added a gray, dotted line in each bar chart in regard to this best result (see Figure 37). It becomes visible, that Amandroid (3.1.2) and FlowDroid (all versions) achieve comparable results for DB2. The newer versions of Amandroid (3.2.0, 3.2.1) fall behind. In fact, all newer tool versions score

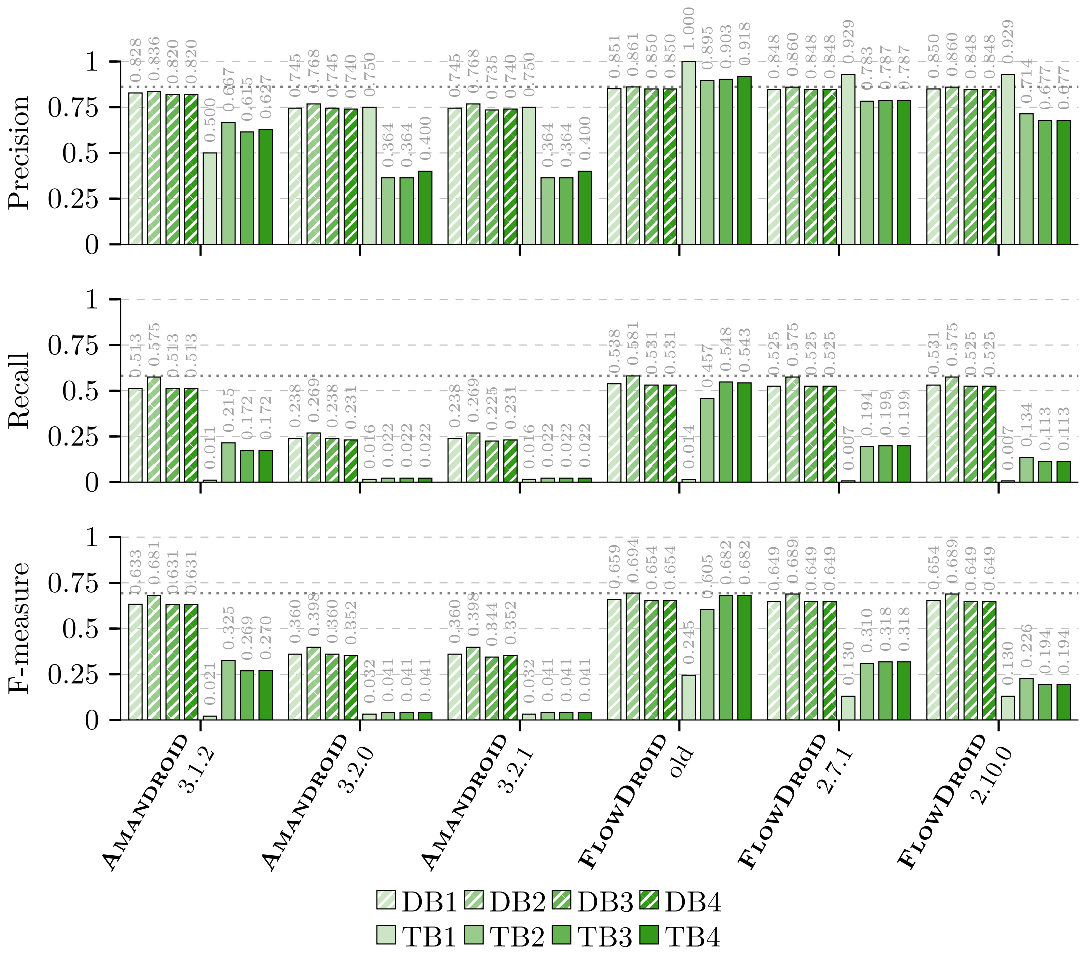


Figure 37: Precision / Recall / F-measure for Cooperative Analysis 5 (DB1–4 & TB1–4)

worse than their predecessors (Amandroid 3.1.2 and FlowDroid old). This regression is striking in the results obtained for TaintBench. The DroidBench results also allow identifying this regression in case of Amandroid, however, FlowDroid's regression in DroidBench context appears negligible. Perhaps FlowDroid was over-adapted to the mirco benchmark DroidBench since it seems to be tested regularly only by employing DroidBench as a benchmark [138].

Please note that some of these results can only be achieved by including knowledge about the benchmark suites. We knew the sources and sinks to be considered in advance – before executing the analysis. In a real-world scenario we cannot always get this information, especially if yet unknown sources or sinks come into play.

Conclusion Considering our main research question, the cooperative analysis (Strategies 2–4) clearly outperforms standalone analysis tools with default source and sink lists (Strategy 1). Furthermore, Cooperative Analysis 5 allows drawing two conclusions: (1.) Source and sink list are crucial: the more accurate the list are, the more accurate the results are, however, a perfect static list does not seem to exist – the lists often are either too coarse or too narrow. (2.) Regressions that may remain unnoticed with respect to DroidBench are clearly visible once TaintBench is employed as a benchmark.

5.6 Cooperative Analysis 6: Backward Compatibility

Analyses often run into scalability issues while dealing with huge amounts of support library code. The cooperative analysis presented in this section allows taint analysis tools to scale with respect to support libraries.

5.6.1 Cooperative Analysis 6: Tools

A typical approach to deal with support libraries is excluding them during analysis. To do so we must be able to differentiate app code from library code. This differentiation can be made by name – a simple but frequently applied approach [139, 160]. For example, all classes in the package `android.support` are considered to be support libraries that can be trusted and safely excluded from analysis. However, in case of analyses dealing with security aspects, e.g. a taint analysis, we may not be allowed to ignore libraries this way. For instance, a malicious developer could have hidden his malicious behavior in a class deliberately placed in a support package. Thus, we employ APK-Simplifier to identify *trusted* library classes (see Section 2.4.2). The AQL-System's default SIMPLIFY preprocessor and operator is applied to remove library classes from apps. Its simpler relative (SIMPLIFY∼) is also used for this purpose, but instead of removing classes it only keeps classes that belong to an app's main package.

Table 27: Cooperative Analysis 6: Tools and Benchmark Sets

Tools	Benchmark Sets
• All 13 baseline tools*	• Top15 (partially)
• APK-Simplifier	
• SIMPLIFY^d	
• SIMPLIFY∼^{d}	

*: Also as standalone tools for comparison (see Table 9, Page 101), ^{d}: Default tool (see Section 3.4.3, Page 85)

Whenever the cooperative analysis uses APK-Simplifier, it can only be applied to up-to-date apps since APK-Simplifier relies on the availability of `.version` files. Such files can nowadays be found inside `.apk` files. Their names refer to library classes and their contents identify the version of the library used. Most benchmark apps either do not use support libraries (e.g. DroidBench – at least not extensively) or are to old (e.g. TaintBench – all apps published before 2017), thus, APK-Simplifier cannot be applied effectively. Only the Top15 app set (see Subsection 4.2.5 and Appendix A.4.3) contains 11 apps that are recent enough to hold the required `.version` files. Hence, the evaluation of this cooperative analysis will only deal with this subset.

5.6.2 Cooperative Analysis 6: Strategies

Three strategies are introduced below. All three fulfill the same task: remove support libraries prior to the analysis of an app.

Strategy 1: Maximal The first strategy removes all classes from an app that are not located in the app's main package or any of its sub-packages. To do so, the SIMPLIFY∼ preprocessor is applied. It is expected that this strategy will remove more classes than the other two, thus, it is called the maximal strategy.

```
Flows IN App(%APP_APK_IN% | 'SIMPLIFY~') ?
```

Strategy 2: Coarse The coarse strategy produces queries that are almost equal to the queries created by the maximal strategy. Only the missing "∼" symbol distinguishes them. Thereby the SIMPLIFY preprocessor is employed instead of SIMPLIFY∼.

```
Flows IN App(%APP_APK_IN% | 'SIMPLIFY') ?
```

Strategy 3: Precise The first two strategies could also be realized without preprocessors by using their operator counterparts. For example, the coarse strategy could be implemented as follows:

```
Flows IN App(SIMPLIFY [ %APP_APK_IN% ! ] !) ?
```

The precise strategy also uses the SIMPLIFY operator, however, in this case it takes two input arguments, namely the app and a list of classes. This list of classes is determined by APK-Simplifier, the tool that answers the most inner question (see Line 4 in the listing below). Since this strategy only removes classes which are denoted on this list, it is called precise strategy.[35]

```
1 Flows IN App( {
2     SIMPLIFY [
3         %APP_APK_IN% !,
4         Arguments IN App(%APP_APK_IN%) USING 'APK-Simplifier' !
5     ] !
6 } ) ?
```

In summary, all strategies are supposed to remove at least all trusted support library classes. The maximal strategy is expected to remove most classes including third-party libraries since those are typically not located in the main package. The precise strategy removes least classes as it only considers classes that are definitively removed under the coarse strategy. In contrast to Strategy 1 and 2, Strategy 3 will only remove *trusted* support library classes, that have, for instance, not been manipulated as determined by APK-Simplifier.

5.6.3 Cooperative Analysis 6: Results

Before we answer our main research question with respect to Strategy 1, 2 and 3, we take a look at the classes removed by each strategy. Table 28 shows the outcome. The first two columns identify the app analyzed. The column entitled with "Original" denotes how many classes the not-simplified app holds. The remaining columns show how many classes are left after simplification with respect to the strategy denoted at the top of each column. Additionally, the relative difference is denoted in % (always in comparison to the original app).

While applying the precise strategy (Strategy 3) up to 15% of all classes are removed. At least 133 classes are removed, which is exactly the case for the TikTok app (`com.zhiliaoapp.musically`). As expected, under Strategy 2 more classes (up to 19%) are removed. By applying the maximal strategy (Strategy 1) two apps could not be output since no classes were located in the main package of these apps and in case of the

[35]Strategy 3 could also be realized using a preprocessor, however, the list or classes must then be provided via a specific variable which appears less natural.

Table 28: Classes Removed by Cooperative Analysis 6

	App (main package)	Number of classes; Reduction (in %) wrt. original **Original**	**3: Precise**		**2: Coarse**		**1: Maximal**	
1	`com.gamma.scan`	12 701	10 974	14 %	10 320	19 %	0	100 %
2	`com.lidl.eci.lidlplus`	19 693	19 034	3 %	18 554	6 %	0	100 %
3	`com.paypal.android.p2pmobile`	73 948	73 443	1 %	72 768	2 %	3 368	95 %
4	`com.sec.android.easyMover`	13 631	11 959	12 %	11 511	16 %	4 240	69 %
5	`com.starfinanz.mobile.android.pushtan`	6 365	6 076	5 %	6 109	4 %	475	93 %
6	`com.zhiliaoapp.musically`	121 162	121 029	0 %	120 882	0 %	1	100 %
7	`de.cellular.ottohybrid`	11 445	10 024	12 %	9 994	13 %	2 210	81 %
8	`de.hafas.android.db`	20 495	17 355	15 %	16 940	17 %	13	100 %
9	`de.komoot.android`	26 562	25 633	3 %	24 362	8 %	9 302	65 %
10	`de.rki.coronawarnapp`	11 228	11 029	2 %	9 644	14 %	5 512	51 %
11	`org.telegram.messenger`	14 596	14 243	2 %	13 984	4 %	1 931	87 %

TikTok app only one class remained. Naturally, if the maximal strategy is applied, way more classes are removed in comparison to both other strategies.

With the coarse strategy (Strategy 2) in place all support libraries are rigorously removed. The precise strategy (Strategy 3) will at best also remove all support libraries. By comparing the amount of classes removed under these two strategies, we can conclude that most support libraries were also removed with respect to the precise strategy. Still, the numbers differ slightly. This difference is caused by the employment of different build tools or configurations while building the app and the library, for example, different source code optimizations might have been applied. APK-Simplifier then detects that certain support library classes extracted from the app are not equal to its counterpart extracted from a trusted library and concludes that these classes cannot be trusted. This also shows that APK-Simplifier is able to detect manipulated classes so that these are not removed and will be analyzed.

In summary, all three strategies allow us to identify and remove library classes from up-to-date, real-world apps prior to analysis. To remove significantly more classes trusted third-party library classes must also be taken into account by APK-Simplifier or a related tool (see Section 2.4.2) and eventually be removed as well.

Cooperative Results The results discussed in the following represent an answer to our main research question. Table 29 lists the number of timeouts reached by all tools after 60 minutes and how many timeouts could be prevented by applying a cooperative strategy. The full Top15 set could be used for this evaluation, as APK-Simplifier does not reach a timeout, even though there are no `.version` files in four apps. Overall 14 timeouts could be prevented under Strategy 2 and 3 and 24 in case of Strategy 1. Only in five scenarios, while the maximal strategy was applied, one version of FlowDroid and all versions of IccTA timed out in cases they did not timeout prior to simplification. However, without simplification the respective analyses ran into errors with respect to these five scenarios, thereby the analyses were unsuccessfully stopped before a timeout was reached. Hence, these timeouts could also be interpreted as an improvement since these errors appear to be circumvented.

Table 30 shows how many apps were successfully analyzed (✔) before and after simplification. Please note that an app is counted as *successfully analyzed* only if the analysis result holds at least one taint flow. One could argue that some apps simply do not hold any taint flows, however, since all apps allow to knowingly share information, at least benign taint flows with respect to that purpose should be found. The empty cells (—) in

Table 29: Number of Timeouts Prevented and Introduced (Cooperative Analysis 6)

Tool	Original	3: Precise	2: Coarse	1: Maximal
Amandroid (3.1.2)	5	0	0	▼ 1
Amandroid (3.2.0)	10	▼ 1	▼ 1	▼ 4
Amandroid (3.2.1)	10	▼ 1	▼ 1	▼ 4
DIALDroid	0	0	0	0
DidFail	0	0	0	0
DroidSafe	1	0	0	0
FlowDroid (old)	1	0	0	0
FlowDroid (2.7.1)	12	▼ 3	▼ 3	▲ 1, ▼ 4
FlowDroid (2.9.0)	4	▼ 2	▼ 2	▼ 3
FlowDroid (2.10.0)	13	▼ 4	▼ 4	▼ 6
IccTA (old)	0	0	0	▲ 1
IccTA (2.9.0)	2	▼ 2	▼ 2	▲ 1, ▼ 1
IccTA (2.10.0)	1	▼ 1	▼ 1	▲ 2, ▼ 1
Sum (Σ)	**59**	**▼ 14**	**▼ 14**	**▲ 5, ▼ 24**

▼: Number of timeouts prevented, ▲: Number of timeouts introduced

Table 30 refer to scenarios in which a simplified version of the original app could not be created. In case of the precise and coarse strategy, this was the case when Soot could not write its output `.apk` file since it ran out of memory while trying to do so. In regard to the maximal strategy, for two apps (`com.gamma.scan`, `com.lidl.eci.lidlplus`) all classes are removed, thus, no output is generated.

As visible in Table 30 only one of the 11 original apps is analyzable by a standalone analysis tool, namely Amandroid 3.1.2. After applying a cooperative strategy four to five apps could be analyzed depending on the strategy chosen. In summary, more apps could successfully be analyzed after simplification. DIALDroid, DidFail and DroidSafe never delivered a non-empty analysis result, which was expected since these tools are not maintained anymore – no updates for years but officially announced final versions can be found.

Table 30: Successful Analyses (Cooperative Analysis 6)

	App (main package)	Baseline	Cooperative analysis (Strategy) 3: Precise	2: Coarse	1: Maximal
1	`com.gamma.scan`	✓✗✗✗...	✗✗✗✗✗✗✗✓✓✓✗✗✗	✗✗✗✗✗✗✗✓✓✓✗✗✗	—[1]
2	`com.lidl.eci.lidlplus`	✗✗✗✗...	—[2]	—[2]	—[1]
3	`com.paypal.android.p2pmobile`	✗✗✗✗...	—[2]	—[2]	✗✗✗✗✗✗✗✓✓✓✗✗✗
4	`com.sec.android.easyMover`	✗✗✗✗...	—[2]	—[2]	✗✗✗✗✗✗✗✗✗✗✗✗✗
5	`com.starfinanz.mobile.android.pushtan`	✗✗✗✗...	✗✗✗✗✗✗✗✗✗✗✗✗✗	✗✗✗✗✗✗✗✗✗✗✗✗✗	✗✗✗✗✗✗✗✗✗✗✗✗✗
6	`com.zhiliaoapp.musically`	✗✗✗✗...	—[2]	—[2]	✗✗✗✗✗✗✗✗✗✗✗✗✗
7	`de.cellular.ottohybrid`	✗✗✗✗...	✗✓✓✗✗✗✗✓✓✓✗✗✗	✗✓✓✗✗✗✗✓✓✓✗✗✗	✓✓✓✗✗✗✗✗✗✗✗✗✗
8	`de.hafas.android.db`	✗✗✗✗...	✗✗✗✗✗✗✓✗✗✗✗✗✗	✗✗✗✗✗✗✓✗✗✗✗✗✗	✗✗✗✗✗✗✗✗✗✗✗✗✗
9	`de.komoot.android`	✗✗✗✗...	—[2]	—[2]	✗✗✗✗✗✗✓✗✗✗✗✗✓
10	`de.rki.coronawarnapp`	✗✗✗✗...	✗✗✗✗✗✗✗✗✗✗✗✗✗	✗✗✗✗✗✗✗✗✗✗✗✗✗	✗✗✗✗✗✗✗✗✗✗✗✗✗
11	`org.telegram.messenger`	✗✗✗✗...	✗✗✗✗✗✗✗✗✗✗✗✗✗	✗✗✗✗✗✗✗✗✗✗✗✗✗	✗✗✗✗✗✗✗✗✗✗✗✗✗
	Σ	1	9	9	8

Tool order: Amandroid (3.1.2), Amandroid (3.2.0), Amandroid (3.2.1), DIALDroid, DidFail, DroidSafe, FlowDroid (old), FlowDroid (2.7.1), FlowDroid (2.9.0), FlowDroid (2.10.0), IccTA (old), IccTA (2.9.0), IccTA (2.10.0)

[1]: All classes removed, [2]: APK-Simplifier (Soot) throws `OutOfMemoryException` while writing output

✓: Successful analysis, ✗: *Un*successful analysis

Please note that the taint flows found upon a successful analysis have not been checked for completeness or if they are actually critical from a security viewpoint. If anything, a closer look at some samples revealed that most detected taint flows were simple ones (e.g. source and sink can be found in the same method).

Conclusion Referring to the main research question, we can conclude that all three cooperative strategies outperform standalone tools with respect to prevented timeouts and the increased number of successful analyses. However, the results also point out that alarmingly many up-to-date, real-world apps cannot be analyzed by any of the 13 tools considered – no matter if cooperative strategies are applied or not.

5.7 Supplementary Discussion

Lastly, to complete the evaluation chapter we discuss (1.) to which extent cooperative strategies can be applied automatically, and (2.) secondary performance aspects not related to accuracy.

5.7.1 Automatic Applicability of Cooperative Strategies

All cooperative strategies presented can be applied automatically. In the strategies section (Section 3.3) we exemplified how to automatically select and apply (cooperative) strategies by always appending an `Arguments`-query asking for features (`... FEATURING Arguments IN App('A.apk') .`). Depending on the features found and the number of considered apps, different strategies can be applied.

Cooperative Analysis 1, 2 and 3 deal with different categories of DroidBench, namely ICC, IAC, reflection and native code. With respect to the DroidBench apps of these categories the AQL-System's default tool FeatureFinder can, for instance, detect the following features in order to answer the respective `Arguments`-query:

- **ICC, IAC:** If an intent sending or receiving method is called (e.g. `startActivity(...)` or `getStringExtra(...)`).
- **Reflection:** If a class or a method is accessed reflectively (e.g. `Class.forName(...)` or `invoke(...)`).
- **Native Code:** If a native library is loaded via `loadLibrary(...)`.

Whenever the "ICC, IAC" features are recognized, a strategy from Cooperative Analysis 1 or 2 can be applied depending on the number of apps involved, i.e. if a `FROM-TO`-query is issued initially, an IAC strategy can be applied – otherwise, if an `IN`-query is issued, an ICC strategy can be applied instead. Upon the detection of feature "Reflection" or "Native Code" the respective strategy of Cooperative Analysis 3 can be applied. In doing so, the baseline results for DroidBench could be improved with respect to these categories.

In Table 31 the baseline results for DroidBench are compared to the cooperative analysis results that can be reached by applying this *mixed* strategy. The columns entitled with "Best" refer to the best results that were achieved by at least one of our 13 taint analysis tools. "FD" stands for FlowDroid (2.9.0) in this case. This version of FlowDroid was selected since it represents the overall best, truly standalone tool with respect to the DroidBench benchmark.

On average (see last row of Table 31) the mixed cooperative strategies outperform FlowDroid (2.9.0). In regard to the "Best" tool per category the mixed cooperative strategies fall slightly behind, however, selecting the best tool per category (e.g. via algorithm selection [90]) is only possible if we know which tool performs best in advance, hence, this comparison lacks fairness. To further improve the cooperatively achieved results we could always apply a cooperative strategy that optimizes the sources and sinks considered as introduced in Cooperative Analysis 5.

Since these cooperative strategies can be selected and applied automatically, they are not limited to be used for benchmark apps with known features. Instead, they can be applied to arbitrary apps whenever they are needed. To do so, the user initiating the analysis must only ask a very simple AQL-Query (e.g. `Flows IN App('A.apk') ?`), i.e. the user only needs basic knowledge about the AQL to employ a complex cooperative analysis. This way the burden of setting up analysis tools and creating cooperative strategies is shifted to domain experts.

Table 31: F-measure for DroidBench (Mixed Cooperative Strategies)

DroidBench Category	**Baseline** Best	FD	**Coop. Ana.**	**Difference to** Best	FD
Aliasing	0.667	0.667	0.667	0.000	0.000
AndroidSpecific	0.952	0.952	0.952	0.000	0.000
ArraysAndLists	0.727	0.727	0.727	0.000	0.000
Callbacks	0.897	0.897	0.897	0.000	0.000
DynamicLoading	0.500	0.000	0.000	▼ 0.500	0.000
EmulatorDetection	0.966	0.966	0.966	0.000	0.000
FieldAndObjectSensitivity	1.000	1.000	1.000	0.000	0.000
GeneralJava	0.821	0.810	0.810	▼ 0.011	0.000
ImplicitFlows	1.000	0.000	0.000	▼ 1.000	0.000
IAC	0.625	0.000	0.625	0.000	▲ -0.625
ICC	0.750	0.348	0.727	▼ 0.023	▲ -0.379
Lifecycle	0.933	0.769	0.769	▼ 0.164	0.000
Native	0.333	0.000	0.889	▲ -0.556	▲ -0.889
Reflection	0.615	0.200	0.800	▲ -0.185	▲ -0.600
Reflection_ICC	0.533	0.000	0.000	▼ 0.533	0.000
SelfModification	0.000	0.000	0.000	0.000	0.000
Threading	1.000	0.909	0.909	▼ 0.091	0.000
UnreachableCode	0.000	0.000	0.000	0.000	0.000
Average	**0.684**	**0.458**	**0.597**	▼ **0.088**	▲ **-0.139**

FD: FlowDroid (2.9.0)

5.7.2 Secondary Performance Aspects

With the evaluation of all six cooperative analyses, we attempted to answer our main research question. In doing so, we focused on performance in terms of accuracy only. Since there are many other relevant performance aspects, we briefly discuss the most important ones in the following.

Cooperative Analysis 1 Cooperative Analysis 1 deals with ICC and the strategy, that partially performed best, involved slicing. Please note that slicing also reduces the amount of time an analysis requires to finish, however, the slicing process itself takes a similar amount of time as most analyses require in order to finish their analysis of the unsliced app. A detailed runtime evaluation can be found in the paper proposing the employed slicer [87].

Cooperative Analysis 2 Different cooperative methods have been presented along with Cooperative Analysis 2. For brevity, neither analysis time nor scalability was evaluated. A proper evaluation of such aspects can be found in prior work [75]. Still, an important finding we want to document becomes visible when IAC scenarios are extended. Let us assume that we want to detect inter-app taint flows in a set of apps that consists of x apps. Not seldom merging these x apps and analyzing the merged app is faster than computing and combining the individual results per app. However, at the moment we add app $x+1$, an analysis based on app merging has to be restarted from scratch. In case of a strategy based on combining results, only the additional app must be analyzed. The additional analysis result can then be combined with the previous result computed for x apps.

Cooperative Analysis 3 The two strategies presented along with Cooperative Analysis 3 are primarily based on two tools: DroidRA and NOAH. While NOAH does not create a larger overhead with respect to analysis time, since it is a flow-insensitive and consequently a fast analysis tool, DroidRA may cause a non-negligible overhead. However, both tools were similarly fast as the employed taint analysis tools (Amandroid and FlowDroid) in case of all experiments conducted which only dealt with micro benchmarks.

Cooperative Analysis 4 HornDroid is employed together with all 13 taint analysis tools in our scope for Cooperative Analysis 4. In comparison HornDroid often requires way more resources, especially with respect to analysis time, to finish its analysis, since HornDroid implements a sound approach which is based on logical reasoning. As mentioned before, HornDroid timed out six times even though 20 minutes were provided in order to analyze each micro benchmark case – real-world scenarios were not evaluated in this context.

Cooperative Analysis 5 The experiments conducted for Cooperative Analysis 5 involve two benchmark suites, namely DroidBench and TaintBench. Since DroidBench is a micro benchmark whereas TaintBench is a real-world benchmark that includes larger benchmark apps with respect to size (e.g. lines of code), all analysis tools in general required more time to finish their analyses when dealing with TaintBench's apps. A more detailed evaluation of these differences (app size and analysis time) can be found in our prior work proposing TaintBench [91].

Cooperative Analysis 6 APK-Simplifier is the primary tool that drives one of the strategies behind Cooperative Analysis 6. With respect to runtime APK-Simplifier may create a recognizable overhead, since it will compare support library classes extracted from the app under analysis against library classes that must be downloaded from an online repository, extracted and decompiled. However, APK-Simplifier keeps track of all comparisons performed in the past such that it is not required to e.g. download the same library multiple times. With respect to our experiments we executed APK-Simplifier for each app that was analyzed before we conducted the experiments such that the overhead created by APK-Simplifier actually was negligible.

6 Conclusion

The cooperative analysis approach presented embodies the first contribution and the centerpiece of this thesis (see Chapter 3). It has been realized by designing and implementing the AQL. While AQL-Queries have been introduced to interact with arbitrary analyses and to compose cooperative analyses, AQL-Answers came into play to unify and generalize analysis results so that they can be combined or used to enhance other analyses. Strategies have been proposed as a means to automatically transform simple initial queries into complex queries representing cooperative analyses, whenever certain conditions are fulfilled, for example, if a query has a certain form or if it denotes certain features. Finally, we implemented a system using the AQL (the AQL-System) and a wrapper for this system (AQL-WebService) that allows us to access analyses in distinct environments. To design the AQL and to develop the associated implementations, we first derived appropriate requirements from related work. After an exploration of available analysis tools, query languages and result formats as well as existing cooperative approaches, we ended up with 26 requirements. Thereafter, we explained how to fulfill these requirements while describing the AQL in detail. For this purpose, we presented two meta-models (a grammar and a schema) that respectively define the syntax of AQL-Queries and the structure of AQL-Answers. A formal description of the AQL's semantics defines how to process AQL-Queries in order to get the associated AQL-Answers. By describing the workflow of the AQL-System we detailed this process and explained how it is implemented.

As the second contribution of this thesis we presented automatic and reproducible benchmarks (see Chapter 4). To realize such benchmarks we proposed BREW. Based on its workflow, we showed that it can be used to create and refine new and existing benchmark suites, to execute benchmarks, and to evaluate their outcome. To create and refine benchmark suites, BREW allows us to semi-automatically define their ground truth. For this purpose the expected results of each benchmark case are described in the form of an AQL-Answer. To execute benchmarks, BREW automatically constructs one AQL-Query per benchmark case and forwards it to an AQL-System. Therefore, it can be used to evaluate arbitrary analyses. To evaluate a benchmark's outcome, BREW compares the actual answers received for each query against the expected answers defined in a suite's ground truth. The result of each comparison decides whether the respective benchmark case is counted as a true or false positive or negative. Because of the machine-readable format used to describe expected and actual results (AQL-Answers), these two can automatically be compared against each other without providing space for (mis-)interpretation. Consequently, benchmarks executed via BREW finally become reproducible. The final result of a benchmark is summarized with the computation of precision, recall and F-measure scores. To complete this second contribution, we employed BREW to thoroughly evaluate 13 standalone Android taint analysis tools. We showed that these tools mostly keep their feature-support and accuracy promises, and that all tools are able to analyze (some) real-world apps.

The evaluation of six cooperative analyses represents our third and last contribution (see Chapter 5). We described each cooperative analysis by explaining its purpose, i.e. which Android framework or programming language feature it deals with. The tools used in each cooperative analysis were listed along with an explanation of their purpose in this context. The different strategies employed per cooperative analysis were also explained in detail. Along with these strategies, BREW could be used to evaluate all six cooperative analyses. By comparing the results achieved for cooperative and standalone (see above) analyses, we could answer our main research question:

RQ$_{\text{MAIN}}$: To which extent can cooperative analyses outperform standalone taint analyses?

Cooperative analyses outperform their standalone counterparts with respect to accuracy most of the time – to which extent depends on the individual strategies and tools forming the cooperative analysis. To complete the third contribution, we briefly discussed the automatic applicability of the presented cooperative strategies as well as secondary performance aspects (e.g. efficiency in terms of analysis time).

6.1 Future Work

In the future, we will maintain and extend the AQL and the frameworks implementing it to be able (1.) to pick low-hanging fruits, (2.) to run a competition, and (3.) to realize the vision that motivated this research from its beginning.

Low-hanging fruits To pick low-hanging fruits, more tools, better configured tool variants and further benchmark suites and approaches could be integrated in AQL context. With each new or improved tool being released, cooperative analyses can potentially be upgraded by replacing the tool currently used with the new one. For example, early experiments indicate that the employment of ICCBot [97] instead of IC3 would improve cooperative analyses tailored to ICC and IAC. Many analysis tools can be configured differently to handle certain features more accurately. Hence, configuring an AQL-System to select tool variants with respect to the features denoted in a query could increase the accuracy of analyses executed via this AQL-System. It has already been shown that case-specific configurations can achieve better results if certain features are present in the app under analysis [86]. Recently more (real-world) benchmark suites (e.g. FossDroid [86]) and benchmark approaches (e.g. benchmark fuzzing [94]) tailored to Android taint analysis have been released. With their integration in BREW, we can potentially gain new, yet unknown insights.

Competitions We want to encourage the community to start a first competition in the field of Android taint analysis as it is known to often push research to the next level and to promote open science. From a technical perspective, automatic and reproducible benchmarks as implemented in BREW can be used to run such competitions. Once participants submit configurations for BREW (or AQL-Systems) that allow us to access their (standalone or cooperative) analyses, these can be evaluated automatically and compared competitively against previously evaluated analyses. Due to the reproducibility of benchmarks executed via BREW we would not even be forced to rerun previous evaluations. A competition would allow us to detect and also measure improvements and regressions perhaps on an annual basis.

Although we might be ready from a technical perspective, open organizational questions must first be answered in order to successfully bring a competition to life. For example: When, where and how often should the competition take place? How to motivate potential participants to take part in the competition and how to reimburse them for their effort? The histories of successful competitions provide potential answers to such questions. For example, the SV-Comp [31, 185] takes place annually as a satellite event of the TACAS conference. The SV-Comp participants are motivated by offering them a trophy and reimbursed with a citation in the competition's annual report.

The Vision With the AQL and the frameworks implementing it, the foundations are laid to create an entire cooperative analysis infrastructure that follows the principles of on-the-fly computing [82]. Analysis experts and developers could offer their tools (along with configurations for AQL-Systems) as *services* on software markets. *Compute centers*, providing extensive resources in regard to computation power and time, could host AQL-WebServices in order to execute these services. Cooperative analysis *providers* could implement and share strategies that on-the-fly combine different services hosted in distinct compute centers. Automated and reproducible benchmarks could be used to assess and consequently rate the quality of single services (standalone tools) or compositions (cooperative analyses). Finally, the user could access everything by formulating a single AQL-Query. The user in this context may be a security expert, an analysis or app developer as well as an everyday smartphone user.

Let us recapitulate the arms race between attackers and defenders mentioned in our introduction. In this race, attackers attempt to e.g. steal data and defenders try to prohibit such malicious behavior. Having an infrastructure (as described above) in place would empower the defenders as they could focus on their subtasks while silently contributing to cooperative analyses composed by third parties. Additionally, resource constraints could become less restrictive since the execution of analyses can be outsourced to compute centers. Most importantly, the third participant who is unwillingly taking part in this race, the non-expert user, could eventually catch up by gaining access to the defenders' instruments. In conclusion this infrastructure would realize the vision of:

Cooperative Analysis **for Everyone**

6.2 Discussion

When presenting the cooperative analysis approach (or automatic and reproducible benchmarks) in the context of Android taint analysis, we frequently get a question afterwards that asks whether the approach can also be applied to other types of analyses, other programs than Android apps or in the context of other programming languages than Java. In short, the generalizability of the approach is questioned.

The answer to this question is twofold. On the one hand, an approach must be tailored to a specific field of application as demanded by ▶ **Req. 9**. On the other hand, the approach should be extendable (cf. ▷ **Req. 9**). Accordingly, what we are dealing with is a tradeoff – a tradeoff between generalizability and specificity. A strictly generalizable approach would only offer little benefit as we would have to broaden the meta-models defining AQL-Queries and AQL-Answers such that we might end up with AQL-Queries being similar to plain command line instructions and AQL-Answers being indistinguishable from arbitrary XML documents. A strictly specific approach would not be applicable to other types of analyses, programs or programming languages.

To deal with this tradeoff, we designed the AQL and the AQL-System so that both can be adapted easily to be employable in different contexts. We only have to adapt the meta-models defining AQL-Queries and AQL-Answers to adapt the language. This is also sufficient to adapt the implementation since parsers and data structures are generated on the basis of these meta-models. Still, the implementation of default analysis tools, operators and preprocessors must be adapted once the data structures are changed. Additionally, new converters must be created when dealing with yet unsupported tools.[36]

[36]The difficulty to create a converter always depends on the input given. A structured format can easily be parsed and converted. Unstructured formats or formats relying on natural language are harder to convert. As a rule of thumb: 219 lines of code were required on average to construct the 11 converters currently shipped with the AQL-System.

To summarize, the AQL and its implementations can be used in the context of other types of analyses, programs and programming languages, however, few manual adaptations are required.

The AQL in its current shape has very little in common with its first version proposed back in 2017 [63]. The language has grown with the objectives we chased. If we had known all objectives from the beginning, the language would probably be defined differently. New objectives and ideas required new language elements and adaptations of the associated implementations. In case of the AQL, denoting features in queries, for example, was not possible from the beginning, hence, selecting tools or strategies depending on the features given was not possible either. In regard to the AQL-System, all tools, for example, were executed locally, thus, we did not think about executing tools in distinct environments. In particular the latter led to a non-negligible engineering overhead because we had to largely refactor the AQL-System's implementation and had to implement AQL-WebServices. However, we initially did all this to solely use a single dynamic tool (PIM), that required an Android emulator and consequently a distinct environment, in a cooperative analysis.

Regardless of all this engineering effort spent, some features and functionalities are still missing. For instance, it is not possible to issue a query per taint flow contained in an answer. This would be required to dynamically check statically found taint flows. Moreover, for the integration of dynamic taint analysis tools there are no specific instruments available to e.g. describe execution traces in queries or answers.

In contrast, dozens of features exist which might be considered as nice-to-have without any scientific value. For example, the AQL-System comes with a GUI that provides a lot of usability (e.g. a graphical viewer for AQL-Answers and an editor for AQL-Queries with auto-completion) but does not improve or influence the cooperative analysis approach at all. The same holds for BREW. We implemented, for instance, the capability to export benchmarks as JUnit tests such that benchmarks can easily be integrated in continuous integration pipelines. Even though this is certainly a useful feature for analysis developers, it is absolutely irrelevant for our research on automatic and reproducible benchmarks.

In conclusion, we can infer another tradeoff between engineering effort and scientific relevance. Only the future can tell if the effort spent has additional value in form of impact in research or industry. Anyhow, we think the implementations developed in AQL context cannot be treated as research prototypes anymore – we handle them as mature tools ready to be employed in production.

6.3 Summary

In this thesis, we presented a cooperative analysis approach realized by means of the AQL. We syntactically and semantically specified the AQL and described the entire approach including its implementation in detail (see Chapter 3). With the introduction of automatic and reproducible benchmarks, we showed how to use the AQL for benchmarking taint analyses (see Chapter 4). In a thorough evaluation, we presented six cooperative analyses (see Chapter 5) and found out that cooperative analyses most of the time outperform their standalone counterparts.

Appendix

A.1 Code Comparison (Source Code, Jimple, Bytecode)

The three code snippets below contain the same class, namely `A`, in three different formats: source code, Jimple (IR) and bytecode. Listing 16 shows the source code of class `A`. The class only holds one method (`increment`). It takes a single integer input (`value`) and returns the same value increased by one. The increment happens in Line 3: `value + 1;`

In Jimple we can suddenly see two methods (see Listing 17). The first (`<init>`) is the default constructor. In source code the default constructor is always implicitly given if no other constructor is specified. In Jimple any constructor is explicitly denoted. Furthermore, additional variables are introduced. For example, the return value is explicitly stored in `$i1` in Line 14. The increment happens in the same line.

The bytecode version (see Listing 18) also holds the default constructor (Lines 2–6). Variables do not appear. Instead, we have one instruction loading a certain value (Line 10) and one instruction representing a constant (Line 11). The addition is performed by another instruction (Line 12).

As the comparison shows, all versions contain the same information in different formats. Source code appears to be optimized for humans as it is the shortest and probably the easiest to read. The opposite holds for bytecode. It appears lengthy and hard to read, however, this will not bother a machine. Finally, Jimple can be considered as a mixture of source code and bytecode as it shows commonalities with both.

```
1  class A {
2      public int increment(int value) {
3          return value + 1;
4      }
5  }
```

Listing 16: Class `A` (Source Code)

```
1  class A extends java.lang.Object {
2      void <init>() {
3          A r0;
4          r0 := @this: A;
5          specialinvoke
               → r0.<java.lang.Object:
               → void <init>()>();
6          return;
7      }
8
9      public int increment(int) {
10         A r0;
11         int i0, $i1;
12         r0 := @this: A;
13         i0 := @parameter0: int;
14         $i1 = i0 + 1;
15         return $i1;
16     }
17 }
```

Listing 17: Class `A` (Jimple)

```
1  class A {
2      A();
3          Code:
4              0: aload_0
5              1: invokespecial #1 // Method java/lang/Object."<init>":()V
6              4: return
7
8      public int increment(int);
9          Code:
10             0: iload_1
11             1: iconst_1
12             2: iadd
13             3: ireturn
14 }
```

Listing 18: Class `A` (Bytecode)

A.2 Framework, Tool and Benchmark Suite Contribution Summary

Table 32 lists all frameworks, tools and benchmark suites that have (partially) been developed as a part of this thesis. Note that the AQL-System includes all the default tools mentioned throughout the thesis.

Table 32: Framework, Tool and Benchmark Suite List

Framework	Version	References*
AQL-SYSTEM	$2.0.1_t$	[63] [114]
CODIDROID	—	[75] [123]
BREW (REPRODROID)	$2.0.1_t$	[67] [175]

Tool	Version	References*
AMT	$2.0.1_t$	[76] [102]
JICER	2.0.0	[87] [159]
APK-SIMPLIFIER	$2.0.1_t$	[92] [108]
AQL-CHECKOPERATOR	2.0.0	[75] [112]
NOAH	$2.0.1_t$	[75] [168]
PERMISSIONFINDER	$2.0.1_t$	— [170]
PIM	$2.0.1_t$	[75] [171]

Benchmark Suites	Version	References*
TAINTBENCH	1.0	[91] [186]
FEATURE-CHECKING	$2.0.1_t$	[67] [175]
INTENT-MATCHING	$2.0.1_t$	[67] [175]

$_t$: to be released. Currently, only included in Appendix A.6.

*: Publication, Link.

A.3 Additional Example (Intents, Intent Filters, Intent Sinks, Intent Sources)

Figure 38 shows an example that includes one intent sink and two intent sources. For all three an intent triple (action, category, data) and a statement (e.g. `startActivity(...)`) is depicted. Once the Android system attempts to deliver the intent to the correct intent source, the system needs to find out which intents and intent filters match. In this example the intent sent by the intent sink of `MainActivity` can only be matched by the intent filter of `TargetActivityA`. The intent filter of `TargetActivityB` denotes a different intent triple. For example, the action string (`STORE`) does not match the intent's one (`SEND`).

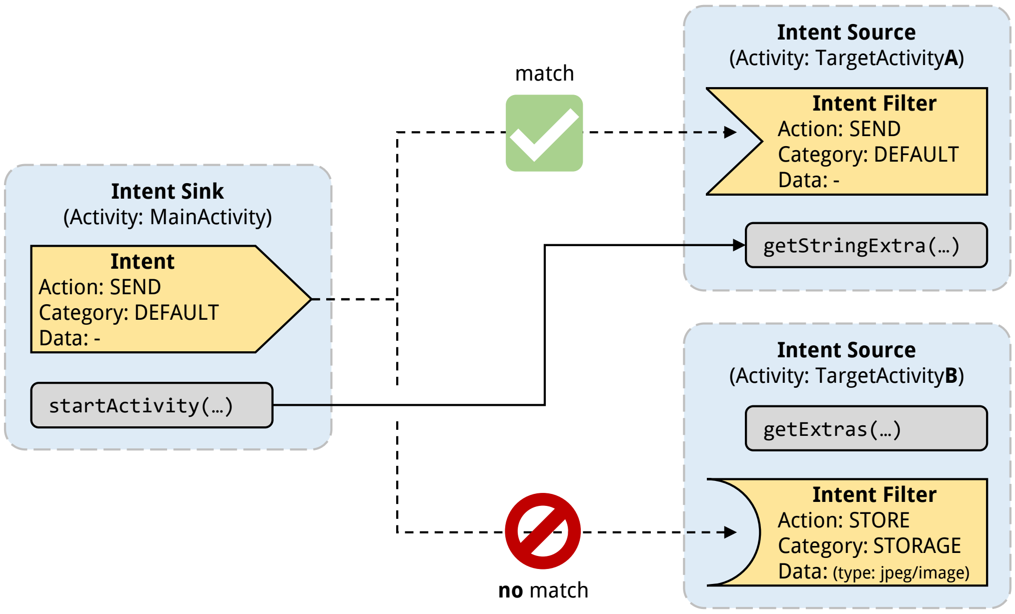


Figure 38: Example Showing One Intent Sink and Two Intent Sources

An analysis, that attempts to find inter-component flows, therefore requires both: First, the intent triple to find out which intents and intent filters match. Second, the statement to identify and document inter-component flows.

As visible in Figure 38 intents and intent filters are complete without a reference that identifies the statement they are associated with. For intent sinks and intent sources the intent triples are optional but the statement (reference) is required. Hence, if no intent triples would be given in the example above, we could still compute all possible inter-component flows by connecting all intent sinks with all intent sources, however, this would give us an infeasible flow from `startActivity(...)` to `getExtras(...)`.

A.4 Experimental Details

A.4.1 Execution Environment: Specifications

The execution environments described in the following have been used to carry out all experiments with respect to the baseline (see Section 4.2) and the evaluation (see Chapter 5).

Execution Environment 1: All tools except PIM have been executed on a Debian 9 (Stretch) virtual machine (VM). Java 17 (Oracle; 17+35-LTS-2724) *and* Java 8 (Oracle; 1.8.0_331-b09) have been installed since some analysis tools require a Java version < 9. The VM was granted access to two cores of an Intel© Xeon® CPU (E5-2695 v3 @ 2.30GHz) and 128 GB memory. 96 out of 128 GB were assigned to all memory-intensive analysis tools.

Execution Environment 2: PIM, the only dynamic tool in our scope, requires an Android (virtual) device to be executed. To run a virtual device (e.g. the Android emulator) efficiently, hardware support is required. Thus, running a virtual device inside a virtual machine is barely possible. Consequently, we used another execution environment for the execution of PIM. With the help of AQL-WebServices the two environments (or the AQL-Systems employed) were interacting with each other (see Subsection 3.4.5). The system executing PIM was a Windows 10 laptop (Dell XPS 13 9310) supplied with an Intel© Core® i7 CPU (i7-1185G7 @ 3.00GHz), 32 GB memory and Java 17 (Oracle; 17+35-LTS-2724) installed.

A.4.2 Experiment 1: Details

Table 33 (see Page 155) shows a more detailed version of Table 10 (see Page 102). Each cell of this table shows the results for all four versions of the Feature-Checking benchmark suite. It becomes visible that some tools cannot handle certain versions at all. No version of Amandroid is able to successfully analyze any app that targets API 26 or 30. Same holds for the old version of IccTA. DidFail fails to analyze any app except those that target API 19 and have been built with up-to-date build tools. DroidSafe is also only able to analyze apps that target API 19 but fails if up-to-date build tools are used to construct the apps. In general, we observe deviating behavior across the board. Experiment 2 is conducted to determine what causes this behavior (see Subsection 4.2.2).

Table 33: Experiment 1: Results for the FEATURE-CHECKING Benchmark Suite

Feature	AMANDROID 3.1.2	AMANDROID 3.2.0	AMANDROID 3.2.1	DIAL-DROID	DID-FAIL	DROID-SAFE	FLOWDROID old	FLOWDROID 2.7.1	FLOWDROID 2.9.0	FLOWDROID 2.10.0	ICCTA old	ICCTA 2.9.0	ICCTA 2.10.0
Aliasing	✓⃝,×⃝†,×⃝†,✓⃝	×⃝,×⃝†,×⃝†,✓⃝	×⃝,×⃝†,×⃝†,✓⃝		×⃝†,×⃝†,×⃝†,✓⃝	✓⃝,×⃝†,×⃝†,×⃝†	✓⃝,✓⃝,✓⃝,✓⃝	✓⃝,✓⃝,✓⃝,✓⃝	✓⃝,✓⃝,✓⃝,✓⃝	✓⃝,✓⃝,✓⃝,✓⃝	✓⃝,×⃝†,×⃝†,✓⃝	✓⃝,✓⃝,✓⃝,✓⃝	✓⃝,✓⃝,✓⃝,✓⃝
Static	✔⃝,×⃝†,×⃝†,✔⃝	×⃝,×⃝†,×⃝†,×⃝	×⃝,×⃝†,×⃝†,×⃝		×⃝†,×⃝†,×⃝†,×⃝	✔⃝,×⃝†,×⃝†,×⃝†	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,×⃝†,×⃝†,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝
Callbacks	✔⃝,×⃝†,×⃝†,×⃝	×⃝,×⃝†,×⃝†,×⃝	×⃝,×⃝†,×⃝†,×⃝		×⃝†,×⃝†,×⃝†,×⃝	×⃝,×⃝†,×⃝†,×⃝†	✔⃝,✔⃝,×⃝,×⃝	✔⃝,✔⃝,×⃝,×⃝	✔⃝,✔⃝,×⃝,×⃝	✔⃝,✔⃝,×⃝,×⃝	✔⃝,×⃝†,×⃝†,×⃝	✔⃝,✔⃝,×⃝,×⃝	✔⃝,✔⃝,×⃝,×⃝
Life-Cycle	✔⃝,×⃝†,×⃝†,✔⃝	×⃝,×⃝†,×⃝†,×⃝	×⃝,×⃝†,×⃝†,×⃝		×⃝†,×⃝†,×⃝†,✔⃝	✔⃝,×⃝†,×⃝†,×⃝†	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,×⃝†,×⃝†,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝
Inter-Procedural	✔⃝,×⃝†,×⃝†,✔⃝	✔⃝,×⃝†,×⃝†,✔⃝	✔⃝,×⃝†,×⃝†,✔⃝		×⃝†,×⃝†,×⃝†,✔⃝	✔⃝,×⃝†,×⃝†,×⃝†	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,×⃝†,×⃝†,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝
Inter-Class	✔⃝,×⃝†,×⃝†,✔⃝	✔⃝,×⃝†,×⃝†,✔⃝	✔⃝,×⃝†,×⃝†,✔⃝		×⃝†,×⃝†,×⃝†,×⃝	✔⃝,×⃝†,×⃝†,×⃝†	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,×⃝†,×⃝†,×⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝
IAC	×⃝*,×⃝*†,×⃝*†,×⃝*	×⃝*,×⃝*†,×⃝*†,×⃝*	×⃝*,×⃝*†,×⃝*†,×⃝*	×⃝,×⃝,×⃝,×⃝	×⃝*†,×⃝*†,×⃝*†,×⃝*	×⃝*,×⃝*†,×⃝*†,×⃝*†	×,×,×,×	×,×,×,×	×,×,×,×	×,×,×,×	×⃝*,×⃝*†,×⃝*†,×⃝*	×⃝*,×⃝*,×⃝*,×⃝*	×⃝*,×⃝*,×⃝*,×⃝*
ICC (Explicit)	✓⃝,×⃝†,×⃝†,×⃝	×⃝,×⃝†,×⃝†,×⃝	×⃝,×⃝†,×⃝†,×⃝	✔⃝,✔⃝,×⃝,×⃝	×⃝†,×⃝†,×⃝†,×⃝	✓⃝,×⃝†,×⃝†,×⃝†	×,×,×,×	×,×,×,×	×,×,×,×	×,×,×,×	✔⃝,×⃝†,×⃝†,×⃝	×⃝,×⃝,×⃝,×⃝	×⃝,×⃝,×⃝,×⃝
ICC (Implicit)	✓⃝,×⃝†,×⃝†,✓⃝	×⃝,×⃝†,×⃝†,×⃝	×⃝,×⃝†,×⃝†,×⃝	×⃝,×⃝,✓⃝,✓⃝	×⃝†,×⃝†,×⃝†,×⃝	✓⃝,×⃝†,×⃝†,×⃝†	×,×,×,×	×,×,×,×	×,×,×,×	×,×,×,×	✓⃝,×⃝†,×⃝†,✓⃝	✓⃝,×⃝,×⃝,×⃝	✓⃝,×⃝,×⃝,×⃝
Flow-Sensitivity	×⃝,×⃝†,×⃝†,×⃝	✔⃝,×⃝†,×⃝†,✔⃝	✔⃝,×⃝†,×⃝†,✔⃝		×⃝†,×⃝†,×⃝†,✔⃝	×,×†,×†,×†	✔⃝,✔⃝,×⃝,✔⃝	✔⃝,×⃝,×⃝,✔⃝	✔⃝,×⃝,×⃝,✔⃝	✔⃝,×⃝,×⃝,✔⃝	✔⃝,×⃝†,×⃝†,✔⃝	✔⃝,×⃝,×⃝,✔⃝	✔⃝,×⃝,×⃝,✔⃝
Context-Sensitivity	✔⃝,×⃝†,×⃝†,✔⃝	✔⃝,×⃝†,×⃝†,✔⃝	✔⃝,×⃝†,×⃝†,✔⃝		×⃝†,×⃝†,×⃝†,✔⃝	✔⃝,×⃝†,×⃝†,×⃝†	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,×⃝†,×⃝†,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝	✔⃝,✔⃝,✔⃝,✔⃝
Field-Sensitivity	✓⃝,×⃝†,×⃝†,✓⃝	×⃝,×⃝†,×⃝†,✓⃝	×⃝,×⃝†,×⃝†,✓⃝		×⃝†,×⃝†,×⃝†,✓⃝	✓⃝,×⃝†,×⃝†,×⃝†	✓⃝,✓⃝,✓⃝,✓⃝	✓⃝,✓⃝,✓⃝,✓⃝	✓⃝,✓⃝,✓⃝,✓⃝	✓⃝,✓⃝,✓⃝,✓⃝	✓⃝,×⃝†,×⃝†,✓⃝	✓⃝,✓⃝,✓⃝,✓⃝	✓⃝,✓⃝,✓⃝,✓⃝
Object-Sensitivity	✓⃝,×⃝†,×⃝†,✓⃝	×⃝,×⃝†,×⃝†,×⃝	×⃝,×⃝†,×⃝†,×⃝		×⃝†,×⃝†,×⃝†,×⃝	✓⃝,×⃝†,×⃝†,×⃝†	✓⃝,✓⃝,×⃝,×⃝	✓⃝,✓⃝,✓⃝,✓⃝	×⃝,×⃝,×⃝,×⃝	✓⃝,✓⃝,✓⃝,✓⃝	✓⃝,×⃝†,×⃝†,×⃝	✓⃝,×⃝,×⃝,×⃝	✓⃝,✓⃝,✓⃝,✓⃝
Path-Sensitivity	×,×†,×†,×	×,×†,×†,×	×,×†,×†,×		×†,×†,×†,×	×,×†,×†,×†	×,×,×,×	×,×,×,×	×,×,×,×	×,×,×,×	×,×†,×†,×	×,×,×,×	×,×,×,×
Thread-Awareness	✔,×†,×†,✔	✔,×†,×†,✔	✔,×†,×†,✔		×†,×†,×†,×	✔,×†,×†,×†	✔,✔,✔,✔	✔,✔,✔,✔	✔,✔,✔,✔	×,×,×,×	✔,×†,×†,×	✔,✔,✔,✔	×,×,×,×
Reflection	✓,×†,×†,✓	✓,×†,×†,✓	✓,×†,×†,✓		×†,×†,×†,×	✓,×†,×†,×†	×,×,×,×	×,×,×,×	×,×,×,×	×,×,×,×	×,×†,×†,×	×,×,×,×	×,×,×,×

◯ supported, ✔ confirmed, ✓ partially confirmed, × not confirmed, * not promised without cooperation, † aborted

App version (API) order: 19, 26, 30, 19' (19' was created with up-to-date build tools - API 31.)

A.4.3 Benchmark Set: Play Store Top15 Free

This section identifies the apps that belong to the *Play Store Top 15 Free (Top15)* benchmark corpus. Since it is only a collection of apps without any form of ground truth, it does *not* constitute a benchmark suite. All the recorded statistics are listed in Table 34.

Table 34: Play Store Top-15 Free Benchmark Set Statistics

			Source Code			Android API	
	Logo	**Name**	**Classes**	**Methods**	**Statements**	**Minimal**	**Target**
1.		TikTok	121 162	559 665	6 987 257	19	30
		Package: com.zhiliaoapp.musically, Hash (SHA-1): 351af2d407e4ae7be434ce3d0152e09bc41b4936, Date: 22.04.2022 06:38:24					
2.		Corona-Warn-App	11 227*	48 762	742 030	23	29
		Package: de.rki.coronawarnapp, Hash (SHA-1): bfc67f00896241dec51a1a964a005290e1f04a07, Date: 25.04.2022 16:35:02					
3.		QR&Barcode-Scanner	12 701*	63 529	1 037 444	16	29
		Package: com.gamma.scan, Hash (SHA-1): 921f31b9e90deec06752503e5a8f20dbbdb80c94, Date: 22.04.2022 21:09:14					
4.		PayPal	73 948	356 409	4 761 000	23	30
		Package: com.paypal.android.p2pmobile, Hash (SHA-1): 2d879bd79eb2ec9d72fe8af0c2325a838f19cc8c, Date: 19.03.2022 14:35:27					
5.		eBay-Kleinanzeigen	39 217	225 259	2 846 580	23	31
		Package: com.ebay.kleinanzeigen, Hash (SHA-1): 3367e05701c8655457b7f5310168f98bed72f3cb, Date: 25.04.2022 16:39:43					
6.		DB-Navigator	20 494	111 514	1 478 644	21	29
		Package: de.hafas.android.db, Hash (SHA-1): 7575dc6e35eeb12b5454bf8b1c16f25a37c738ad, Date: 22.04.2022 18:26:28					
7.		Samsung-Smart-Switch-Mobile	13 630	86 333	1 443 881	14	29
		Package: com.sec.android.easyMover, Hash (SHA-1): b79f856ec5c1ce8bd1fda882600117dc758d94d2, Date: 24.04.2022 04:12:02					
8.		VR-SecureGo-plus			Error†		
		Package: de.fiduciagad.securego.vr, Hash (SHA-1): –, Date: –					
9.		WhatsApp	16 667	56 403	1 405 338	16	29
		Package: com.whatsapp, Hash (SHA-1): e8315435487e16d1cb549567a66dee6e482b5758, Date: 23.04.2022 23:04:57					
10.		OTTO	11 444	59 728	778 473	21	30
		Package: de.cellular.ottohybrid, Hash (SHA-1): 52a0d96680abe7f11da0566a9fa53f538e1fc9eb, Date: 25.04.2022 01:41:09					
11.		Telegram	14 595	79 500	1 530 637	23	29
		Package: org.telegram.messenger, Hash (SHA-1): bb8385e6850a169e5e519fd87119ce56c1c18a91, Date: 24.04.2022 00:15:07					
12.		Instagram	72 911	228 920	3 868 328	23	30
		Package: com.instagram.android, Hash (SHA-1): 7b94886774199bcbd836339e5bada1fc5c653fed, Date: 23.04.2022 23:19:59					
13.		S-pushTAN	6 365*	25 251	502 947	23	30
		Package: com.starfinanz.mobile.android.pushtan, Hash (SHA-1, shortened): d11358...1bd9a7, Date: 25.04.2022 17:32:23					
14.		Komoot	26 367	138 176	1 832 701	21	30
		Package: de.komoot.android, Hash (SHA-1): 357dd4a2ec2f3b0b2fdc4bc3b4a0369b96d117fc, Date: 19.04.2022 00:22:46					
15.		Disney+	34 470	176 077	2 125 684	21	30
		Package: com.disney.disneyplus, Hash (SHA-1): 8ca88b8b9306e63d2ffe0397bebb96fb40ec7c73, Date: 24.04.2022 02:08:54					
16.		Lidl	19 692‡	100 020‡	1 287 780‡	21	30
		Package: com.lidl.eci.lidlplus, Hash (SHA-1): e075cdb5f077ec2e4fe8b2ab215dfeded480bd95, Date: 22.04.2022 21:58:18					
Sum			494 890	2 315 546	**32 628 724**	–	–
Average			32 993	154 370	2 175 248	**20**	**29**
Median			19 692	100 020	1 478 644	21	30

*: A single `.dex` file holds all classes.

†: Could not be downloaded due to the following error: "invalid package/non-free/not compatible"

‡: The class `com.salesforce.marketingcloud.w.a.e` was skipped while counting.

The apps were chosen by selecting the top 15 apps in the "Top-Charts" (free) category of Google's Play Store [142].[37,38] To avoid legal issues the apps (`.apk` files) are not part of the digital appendix. However, given the information (package, hash and date) in the table

[37]The Play Store was accessed from within Germany on the 25th of April 2022 (10:30 am).

[38]Since one app (VR-SecureGo-plus) could not be downloaded, the top 16 apps were actually selected.

above, all `.apk` files can be downloaded again via APK-Downloader [107].[39] The logos were freely available and have been taken from Google's Play Store.[40] While Soot [179] has been used to count the number of classes, methods and statements, ApkParser [110] was employed to extract manifest information (minimal and targeted Android API version).

The VR-SecureGo-plus app could not be downloaded due to access restrictions of APK-Downloader [107]. Statistics could initially not be recorded for (at least) the eBay-Kleinanzeigen app while employing Soot in version 3.3.0 [180]. After switching to a more recent version of Soot (4.3.0 [181]) the statistics for all apps could be recorded. Consequently, we expect that some tools, which also rely on older Soot versions, will struggle to analyze the respective apps – at least the eBay-Kleinanzeigen app.

The class `com.salesforce.marketingcloud.w.a.e` of the Lidl app was skipped while counting classes, methods and statements since its method `b` caused an `OutOfMemoryException` by exceeding 8 GB of memory. Accordingly, we foresee that some tools, also employing Soot, will fail to analyze the Lidl app for the same reason.

[39]To download all 15 apps about 600 MB disk space is required.

[40]The copyright belongs to the respective vendors.

A.5 Grammar & Schemas

A.5.1 AQL-Query Grammar

The grammar in JavaCC format comprises more than 700 lines of code. The respective `.jj` file is available online [113]. An excerpt that represents production rule p_0 can be found in Listing 19.

```
void queries() :
{
  Token variable;
}
{
  {
    questionHandler.startQueries();
  }
  (
    {
      questionHandler.startQuery();
    }
    (
      query()
    |
      (
        variable = < VARIABLE > "="
        {
          questionHandler.setVariable(variable.toString());
        }
        query()
      )
    )
    {
      questionHandler.endQuery();
    }
  )+
  {
    questionHandler.endQueries();
  }
  < EOF >
}
```

Listing 19: Excerpt of `QuestionGrammar.jj`

A.5.2 AQL-Answer XSD

The XSD that further specifies the content of AQL-Answers is available online [111]. For brevity only an excerpt is presented in this appendix (see Listing 20). The excerpt shows the schema definition's content with respect to “Flow” elements.

```
<xs:element name="flow">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="reference" maxOccurs="unbounded"
                ↪ minOccurs="0"/>
            <xs:element ref="attributes" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Listing 20: Excerpt of `answer.xsd`

A.5.3 Configuration XSD

The configuration of an AQL-System must adhere to the schema specified in an XSD [124]. Here we only show an excerpt that provides a glimpse at the definition of "Execute" elements (see Listing 21).

```
<xs:element name="execute">
    <xs:complexType>
        <xs:choice>
            <xs:sequence>
                <xs:element ref="run"/>
                <xs:element ref="result"/>
                <xs:element ref="instances"/>
                <xs:element ref="memoryPerInstance"/>
            </xs:sequence>
            <xs:sequence>
                <xs:element ref="url"/>
                <xs:element ref="username"/>
                <xs:element ref="password"/>
            </xs:sequence>
        </xs:choice>
    </xs:complexType>
</xs:element>
```

Listing 21: Excerpt of `config.xsd`

A.5.4 Rules XSD

Cooperative strategies in form of transformation rules can be provided to an AQL-System in form of `.xml` files that follow the schema defined via an XSD [187]. Listing 22 shows a excerpt of the respective `.xsd` file. The excerpt shows how input/output (`inputQuery`, `outputQuery`) and conditional (`query`) transformation rules are specified.

```
    <xs:element name="rule">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="priority" maxOccurs="unbounded"/>
                <xs:choice minOccurs="1">
                    <xs:sequence>
                        <xs:element ref="inputQuery"/>
                        <xs:element ref="outputQuery"/>
                    </xs:sequence>
                    <xs:element ref="query"/>
                </xs:choice>
            </xs:sequence>
            <xs:attribute type="xs:string" name="name" use="optional"/>
        </xs:complexType>
    </xs:element>
```

Listing 22: Excerpt of `rules.xsd`

A.5.5 Exemplary Schema Instance

Listing 23 shows an instance of this latter schema. It denotes the two strategies employed in Example 3 (see Subsection 3.3.1). The first rule (Lines 2–12) appends the Arguments-Query. The second (Lines 13–25) embodies the respective ICC strategy. The feature `NoICC` (see Line 18) comes into play so that the first rule is not applied again after applying the second one.

```
 1 <rules>
 2     <rule name="AskForFeaturesRule">
 3         <priority>1</priority>
 4         <inputQuery>
 5             Flows IN App(’%FILE_1%’) ?
 6         </inputQuery>
 7         <outputQuery>
 8             Flows IN App(’%FILE_1%’) FEATURING
 9                 Arguments IN App(’%FILE_1%’) .
10             ?
11         </outputQuery>
12     </rule>
13     <rule name="IccRule">
14         <priority>0</priority>
15         <priority feature="ICC">1</priority>
16         <query>
17             CONNECT [
18                 Flows IN App(’%APP_APK_IN%’) FEATURING ’NoICC’ ?,
19                 CONNECT [
20                     IntentSinks IN App(’%APP_APK_IN%’) ?,
21                     IntentSources IN App(’%APP_APK_IN%’) ?
22                 ] ?
23             ] ?
24         </query>
25     </rule>
26 </rules>
```

Listing 23: Example Rules File

The first rule is an input/output transformation rule. It always has a priority of 1 but can only be applied if a query matches the input query as denoted in Line 5. The second rule is a conditional transformation rule which has a default priority of 0 and a priority of 1 for the feature ICC (see Lines 14, 15). Thus, it is not applied unless the feature ICC is present in the given query. For example, for the two queries below it has a priority of 0:

```
Flows IN App('A.apk') ?
Flows IN App('A.apk') FEATURING 'Reflection' ?
```

However, for the next two queries it has a summed-up priority of 1 (0 + 1) since the feature ICC is enumerated:

```
Flows IN App('A.apk') FEATURING 'ICC' ?
Flows IN App('A.apk') FEATURING 'ICC', 'Reflection' ?
```

The priority of tools is computed analogously.

A.6 Artifact (Digital Appendix)

The artifact (or digital appendix) of this thesis can be found at Zenodo (`https://doi.org/10.5281/zenodo.7404900`). All released and future versions of all frameworks, tools and benchmark suites developed partially for this thesis can be found under the links denoted in Appendix A.2.[41] Additional information about the contributions and artifacts of this thesis can be found at `https://FelixPauck.de`.

A.6.1 Table of Contents

The artifact's table of contents can be derived from its directory structure as depicted in Figure 39. On the first level the directories `private` and `public` can be found. Only the `public` directory is actually available at Zenodo. The private part is not publicly available for legal reasons as we refrain from redistributing commercial software. Its only content are the app sets that belong to DIALDroid-Bench and the Top15 set.

The `public` directory holds three sub-directories, namely `Baseline`, `Evaluation` and `Implementation`. The first two in turn hold one directory per experiment or cooperative analysis. Each of these `Experiment` or `Cooperation` directories holds the benchmark suites used and raw results achieved. Any other material, required to replicate the results, are also attached. For example, the source and sink lists used by Cooperative Analysis 5 are attached to the `Cooperation5` directory. The `Implementation` sub-directory holds all frameworks, tools and benchmarks suites as listed in Appendix A.2 – including versions that are not publicly released yet.

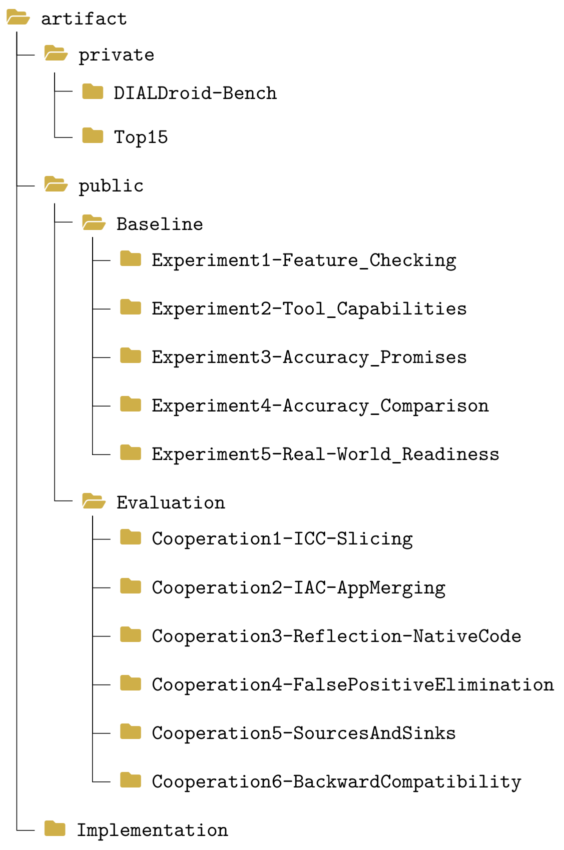

artifact
private
DIALDroid-Bench
Top15
public
Baseline
Experiment1-Feature_Checking
Experiment2-Tool_Capabilities
Experiment3-Accuracy_Promises
Experiment4-Accuracy_Comparison
Experiment5-Real-World_Readiness
Evaluation
Cooperation1-ICC-Slicing
Cooperation2-IAC-AppMerging
Cooperation3-Reflection-NativeCode
Cooperation4-FalsePositiveElimination
Cooperation5-SourcesAndSinks
Cooperation6-BackwardCompatibility
Implementation

Figure 39: Artifact Directory Structure

A.6.2 Raw Results

Raw results mostly come in two formats: `result_*.zip` files *(result files)* represent benchmark runs stored in BREW's format, whereas `log_*.txt` are BREW's *log files*. Once we load a result file with BREW we can inspect the benchmark including the expected answers belonging to the suite's (incomplete) ground truth and the actual answers the respective (cooperative) analysis determined. Both types of answers come in AQL-Answer format. These can be viewed as plain `.xml` files or as a graph. Furthermore, all answers belonging to a benchmark can be exported via BREW. Listing 24 shows a shortened (see Line 23) AQL-Answer determined for one app of the DirectLeak suite (API 19). The listing shows half of the definition of a flow by denoting the reference where the flow starts (see Lines 5 ff.). The content of this example follows the schema presented

[41] Some frameworks, tools and benchmark suites are not yet released but included in the artifact.

in Appendix A.5.2. The graphical representation of the complete flow can be viewed in Figure 40.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<answer>
  <flows>
    <flow>
      <reference type="from">
        <statement>
          <statementfull>$r4 = virtualinvoke r2.&lt;android.telephony.TelephonyManager: java.lang.String getLine1Number()&gt;()</statementfull>
          <statementgeneric>android.telephony.TelephonyManager: java.lang.String getLine1Number()</statementgeneric>
          <linenumber>21</linenumber>
        </statement>
        <method>&lt;de.foellix.aql.aqlbench.directleak.MainActivity: void taint()&gt;</method>
        <classname>de.foellix.aql.aqlbench.directleak.MainActivity</classname>
        <app>
          <file>/home/fpauck/benchmarks/DirectLeakSet/DirectLeak_19.apk</file>
          <hashes>
            <hash type="MD5">87fba7d4e21a614471a7ff79bd413355</hash>
            <hash type="SHA-1">651bcc08b7920cb161ae8679fc31255b6d83f405</hash>
            <hash type="SHA-256">2163bb0a930f8712aa09e75c3815b30c5cddfe2e976e1b4bfef19b09779f83ad</hash>
          </hashes>
        </app>
      </reference>
      <reference type="to">
...
      </reference>
    </flow>
  </flows>
</answer>
```

Listing 24: Excerpt of an AQL-Answer

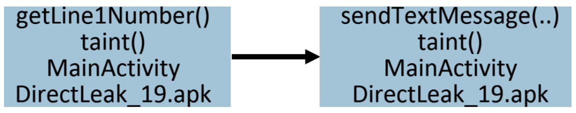


Figure 40: AQL-Answer Example Graph

A shortened excerpt of an example log file is shown in Listing 25. It first shows when and which benchmark case is started (see Line 1), the associated initial query (Line 2) and the transformed query (Line 5). Here the query has been transformed such that FlowDroid (old) and a certain source and sink list is used explicitly. Second, the tools that start and finish are denoted (Line 7, 10, 12, 14). How exactly a tool is launched is shortened here but normally shown inside the square brackets. Finally, in Line 15 the analysis is finished. Its result is output in form of an AQL-Answer (Line 17 ff.). The last line, shows the current status of the benchmark. After finishing the first benchmark case, the first true positive is found (Line 21).

```
BREW 05/09/2022 - 13:42:34 *** Starting Benchmark Case 1/8 *** (ID: 1)
BREW 05/09/2022 - 13:42:34 Starting: Flows IN App('/home/fpauck/benchmarks/DirectLeakSet/DirectLeak_19.apk') ?
...
BREW 05/09/2022 - 13:42:34 Transformation-rule applied: ToolSelection
BREW 05/09/2022 - 13:42:34 Transformed: Flows IN App('/home/fpauck/benchmarks/DirectLeakSet/DirectLeak_19.apk') USES 'FlowDroid-1' WITH 'SourcesAndSinks' = '/home/fpauck/tools/FlowDroid/env/SourcesAndSinks.txt' ?
...
BREW 05/09/2022 - 13:42:34 Starting execution of internal FlowDroid (1). [...]
...
BREW 05/09/2022 - 13:42:38 Result available: /home/fpauck/tools/FlowDroid/old/results/DirectLeak_19_result.txt
BREW 05/09/2022 - 13:42:38 Finished execution of internal FlowDroid (1) after 4.01 seconds.
...
BREW 05/09/2022 - 13:42:39 Starting execution of default DefaultConverter for FlowDroid (2.0.1-SNAPSHOT). [...]
BREW 05/09/2022 - 13:42:39 Storing answered task: /home/fpauck/BREW/data/storage/00002.xml [...]
BREW 05/09/2022 - 13:42:39 Finished execution of default DefaultConverter for FlowDroid (2.0.1-SNAPSHOT) after 0.27 seconds.
BREW 05/09/2022 - 13:42:39 Finished (after 5.523s): Flows IN App('/home/fpauck/benchmarks/DirectLeakSet/DirectLeak_19.apk') ?
BREW 05/09/2022 - 13:42:39
***** Answer (/home/fpauck/BREW/answers/answer_09_05_2022-13_42_39-001.xml) *****
...
AQL-Answer:
...
BREW 05/09/2022 - 13:42:39 *** Finished Benchmark Case 1/8 *** (ID: 1) "getLine1Number() -> sendTextMessage(java.lang.String,java.lang.String,java.lang.String,android.app.PendingIntent,android.app.PendingIntent)": Successful! (Testcases: 8 -> Sources: 8, Sinks: 8 -> Positive cases: 8 (8), Negative cases: 0 (0) -> Flows found (per Query): 0 (0), Analysis time without Timeouts/Crashes: 5s (0m), Analysis time with Timeouts/Crashes: 5s (0m), Timeouts/Crashes (per App): 0 (0) -> True Positive: 1, False Positive: 0, True Negative: 0, False Negative: 7 -> Precision: 1.0, Recall: 0.125, F-Measure: 0.222)
```

Listing 25: Excerpt of `log_DirectLeak_flowdroid-1.txt`

Index

References

[1] Frances E. Allen. "Control flow analysis." *Proceedings of Symposium on Compiler Optimization, 1970.* ACM, 1970. URL: https://doi.org/10.1145/800028.808479.

[2] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. "Structured Design." *IBM Syst. J.* 13.2 (1974), pp. 115–139. URL: https://doi.org/10.1147/sj.132.0115.

[3] Mark Weiser. "Program Slicing." *IEEE Trans. Software Eng.* 10.4 (1984), pp. 352–357.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986. ISBN: 0-201-10088-6. URL: https://www.worldcat.org/oclc/12285707.

[5] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization." *ACM Trans. Program. Lang. Syst.* 9.3 (1987), pp. 319–349.

[6] Susan Horwitz, Thomas W. Reps, and David W. Binkley. "Interprocedural Slicing Using Dependence Graphs." *ACM Trans. Program. Lang. Syst.* 12.1 (1990), pp. 26–60.

[7] Thomas W. Reps and Genevieve Rosay. "Precise Interprocedural Chopping." *Proceedings of the 3rd FSE, 1995.* ACM, 1995.

[8] Frank Tip. "A survey of program slicing techniques." *J. Prog. Lang.* 3.3 (1995). URL: https://dl.acm.org/doi/10.5555/869354.

[9] Clemens A. Szyperski. *Component software - beyond object-oriented programming.* Addison-Wesley-Longman, 1998. ISBN: 978-0-201-17888-3.

[10] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis.* Springer, 1999. ISBN: 978-3-540-65410-0. URL: https://doi.org/10.1007/978-3-662-03811-6.

[11] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. "Soot - a Java bytecode optimization framework." *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, 1999.* IBM, 1999. URL: https://dl.acm.org/citation.cfm?id=782008.

[12] George T Heineman and William T Councill. *Component-based software engineering.* Addison Wesley, 2001.

[13] Jens Krinke. "Evaluating Context-Sensitive Slicing and Chopping." *Proceedings of the 18th ICSM, 2002.* IEEE, 2002. URL: https://doi.org/10.1109/ICSM.2002.1167744.

[14] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. "Finding Plagiarisms among a Set of Programs with JPlag." *Journal of Universal Computer Science* 8 (2002). URL: https://doi.org/10.3217/jucs-008-11-1016.

[15] Dirk Beyer, Adam Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. "The Blast Query Language for Software Verification." *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings.* Lecture Notes in Computer Science. Springer, 2004. URL: https://doi.org/10.1007/978-3-540-27864-1_2.

[16] David Hovemeyer and William W. Pugh. “Finding bugs is easy.” *Proceedings of OOPSLA, 2004*. ACM, 2004. URL: https://doi.org/10.1145/1028664.1028717.

[17] Bert Lagaisse and Wouter Joosen. “Component-Based Open Middleware Supporting Aspect-Oriented Software Composition.” *Proceedings of the 8th CBSE, 2005*. Lecture Notes in Computer Science. Springer, 2005. URL: https://doi.org/10.1007/11424529_10.

[18] V. Benjamin Livshits and Monica S. Lam. “Finding Security Vulnerabilities in Java Applications with Static Analysis.” *Proceedings of the 14th USENIX Security Symposium, 2005*. USENIX Association, 2005. URL: https://www.usenix.org/conference/14th-usenix-security-symposium/finding-security-vulnerabilities-java-applications-static.

[19] Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. “Finding application errors and security flaws using PQL: a program query language.” *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. ACM, 2005. URL: https://doi.org/10.1145/1094811.1094840.

[20] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. “A brief survey of program slicing.” *ACM SIGSOFT Software Engineering Notes* 30.2 (2005), pp. 1–36. URL: https://doi.org/10.1145/1050849.1050865.

[21] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. “The DaCapo benchmarks: java benchmarking development and analysis.” *Proceedings of the 21th OOPSLA, 2006*. ACM, 2006. URL: https://doi.org/10.1145/1167473.1167488.

[22] Arni Einarsson and Janus Dam Nielse. *A Survivor’s Guide to Java Program Analysis with Soot*. Tech. rep. University of Aarhus, Denmark, 2008. URL: https://www.brics.dk/SootGuide/sootsurvivorsguide.pdf (visited on 07. 12. 2022).

[23] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. “Query-Driven Program Testing.” *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*. Lecture Notes in Computer Science. Springer, 2009. URL: https://doi.org/10.1007/978-3-540-93900-9_15.

[24] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. “Static data race detection for concurrent programs with asynchronous calls.” *Proceedings of the 7th ESEC/FSE, 2009*. ACM, 2009. URL: https://doi.org/10.1145/1595696.1595701.

[25] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. *SCanDroid: Automated security certification of Android applications*. Tech. rep. University of Maryland, 2009.

[26] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick D. McDaniel, and Anmol Sheth. “TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones.” *Proceedings of the 9th OSDI, 2010*. USENIX Association, 2010. URL: http://www.usenix.org/events/osdi10/tech/full_papers/Enck.pdf.

[27] Andreas Holzer, Michael Tautschnig, Christian Schallhart, and Helmut Veith. “An Introduction to Test Specification in FQL.” *Hardware and Software: Verification and Testing - 6th International Haifa Verification Conference, HVC 2010, Haifa, Israel, October 4-7, 2010. Revised Selected Papers*. Lecture Notes in Computer Science. Springer, 2010. URL: `https://doi.org/10.1007/978-3-642-19583-9_5`.

[28] Dirk Beyer and M. Erkan Keremoglu. “CPAchecker: A Tool for Configurable Software Verification.” *Proceedings of the 23rd CAV, 2011*. Lecture Notes in Computer Science. Springer, 2011. URL: `https://doi.org/10.1007/978-3-642-22110-1_16`.

[29] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. “Pick your contexts well: understanding object-sensitivity.” *Proceedings of the 38th POPL, 2011*. ACM, 2011.

[30] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. “PScout: analyzing the Android permission specification.” *Proceedings of CCS, 2012*. ACM, 2012. URL: `https://doi.org/10.1145/2382196.2382222`.

[31] Dirk Beyer. “Competition on Software Verification - (SV-COMP).” *Proceedings of the 18th TACAS, 2012*. Lecture Notes in Computer Science. Springer, 2012. URL: `https://doi.org/10.1007/978-3-642-28756-5_38`.

[32] Damien Octeau, Patrick D. 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.” *Proceedings of the 22th USENIX Security Symposium, 2013*. USENIX Association, 2013. URL: `https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/octeau`.

[33] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. “FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps.” *Proceedings of PLDI, 2014*. ACM, 2014. URL: `https://doi.org/10.1145/2594291.2594299`.

[34] Yu Feng, Isil Dillig, Saswat Anand, and Alex Aiken. “Apposcopy: automated detection of Android malware (invited talk).” *Proceedings of the 2nd DeMobile, 2014*. ACM, 2014.

[35] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. “Android taint flow analysis for app sets.” *Proceedings of the 3rd SOAP, 2014*. ACM, 2014. URL: `https://doi.org/10.1145/2614628.2614633`.

[36] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. “A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks.” *Proceeding of the 21st NDSS, 2014*. The Internet Society, 2014. URL: `https://www.ndss-symposium.org/ndss2014/machine-learning-approach-classifying-and-categorizing-android-sources-and-sinks`.

[37] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. “Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps.” *Proceedings of CCS, 2014*. ACM, 2014. URL: `https://doi.org/10.1145/2660267.2660357`.

[38] Tao Xie, Lu Zhang, Xusheng Xiao, Yingfei Xiong, and Dan Hao. “Cooperative Software Testing and Analysis: Advances and Challenges.” *J. Comput. Sci. Technol.* 29.4 (2014), pp. 713–723. URL: `https://doi.org/10.1007/s11390-014-1461-6`.

[39] Dirk Beyer, Stefan Löwe, and Philipp Wendler. “Benchmarking and Resource Measurement.” *Proceedings of the 22nd SPIN, 2015.* Lecture Notes in Computer Science. Springer, 2015. URL: `https://doi.org/10.1007/978-3-319-23404-5_12`.

[40] Xingmin Cui, Jingxuan Wang, Lucas Chi Kwong Hui, Zhongwei Xie, Tian Zeng, and Siu-Ming Yiu. “WeChecker: efficient and precise detection of privilege escalation vulnerabilities in Android apps.” *Proceedings of the 8th Conference on Security & Privacy in Wireless and Mobile Networks, 2015.* ACM, 2015.

[41] Mike Czech, Marie-Christine Jakobs, and Heike Wehrheim. “Just Test What You Cannot Verify!” *Proceedings of FASE, 2015.* Lecture Notes in Computer Science. Springer, 2015. URL: `https://doi.org/10.1007/978-3-662-46675-9_7`.

[42] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. “Information Flow Analysis of Android Applications in DroidSafe.” *Proceedings of the 22nd NDSS, 2015.* 2015. URL: `https://www.ndss-symposium.org/ndss2015/information-flow-analysis-android-applications-droidsafe`.

[43] Wei Huang, Yao Dong, Ana L. Milanova, and Julian Dolby. “Scalable and precise taint analysis for Android.” *Proceedings of ISSTA, 2015.* ACM, 2015. URL: `https://doi.org/10.1145/2771783.2771803`.

[44] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. “ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis.” *Proceedings of SEC, 2015.* Springer, 2015. URL: `https://doi.org/10.1007/978-3-319-18467-8_34`.

[45] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick D. McDaniel. “IccTA: Detecting Inter-Component Privacy Leaks in Android Apps.” *Proceedings of the 37th ICSE, 2015.* IEEE, 2015. URL: `https://doi.org/10.1109/ICSE.2015.48`.

[46] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick D. McDaniel. “Composite Constant Propagation: Application to Android Inter-Component Communication Analysis.” *Proceedings of ICSE, 2015.* IEEE, 2015. URL: `https://doi.org/10.1109/ICSE.2015.30`.

[47] Steven Arzt and Eric Bodden. “StubDroid: automatic inference of precise data-flow summaries for the android framework.” *Proceedings of the 38th ICSE, 2016.* ACM, 2016. URL: `https://doi.org/10.1145/2884781.2884816`.

[48] Michael Backes, Sven Bugiel, and Erik Derr. “Reliable Third-Party Library Detection in Android and its Security Applications.” *Proceedings of CCS, 2016.* ACM, 2016. URL: `https://doi.org/10.1145/2976749.2978333`.

[49] Hamid Bagheri, Alireza Sadeghi, Reyhaneh Jabbarvand Behrouz, and Sam Malek. “Practical, Formal Synthesis and Automatic Enforcement of Security Policies for Android.” *Proceedings of the 46th DSN, 2016.* IEEE Computer Society, 2016. URL: `https://doi.org/10.1109/DSN.2016.53`.

[50] Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. "Correctness witnesses: exchanging verification results between verifiers." *Proceedings of the 24th FSE, 2016.* ACM, 2016. URL: `https://doi.org/10.1145/2950290.2950351`.

[51] Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. "HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving." *Proceedings of EuroS&P, 2016.* IEEE, 2016. URL: `https://doi.org/10.1109/EuroSP.2016.16`.

[52] Lisa Nguyen Quang Do, Michael Eichberg, and Eric Bodden. "Toward an automated benchmark management system." *Proceedings of the 5th SOAP@PLDI, 2016.* ACM, 2016. URL: `https://doi.org/10.1145/2931021.2931023`.

[53] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. "LAVA: Large-Scale Automated Vulnerability Addition." *Proceedings of SP, 2016.* IEEE Computer Society, 2016. URL: `https://doi.org/10.1109/SP.2016.15`.

[54] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. "DroidRA: taming reflection to support whole-program analysis of Android apps." *Proceedings of the 25th ISSTA, 2016.* ACM, 2016. URL: `https://doi.org/10.1145/2931037.2931044`.

[55] Bradley Reaves, Jasmine D. Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, Byron Wright, Kevin R. B. Butler, William Enck, and Patrick Traynor. "*droid: Assessment and Evaluation of Android Application Analysis Tools." *ACM Comput. Surv.* 49.3 (2016), 55:1–55:30. URL: `https://doi.org/10.1145/2996358`.

[56] Daojuan Zhang, Rui Wang, Zimin Lin, Dianjie Guo, and Xiaochun Cao. "IacDroid: Preventing Inter-App Communication capability leaks in Android." *Proceedings of ISCC, 2016.* IEEE Computer Society, 2016. URL: `https://doi.org/10.1109/ISCC.2016.7543779`.

[57] Maqsood Ahmad, Valerio Costamagna, Bruno Crispo, and Francesco Bergadano. "TeICC: targeted execution of inter-component communications in Android." *Proceedings of SAC, 2017.* ACM, 2017. URL: `https://doi.org/10.1145/3019612.3019797`.

[58] Amiangshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. "Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications." *Proceedings of AsiaCCS, 2017.* ACM, 2017. URL: `https://doi.org/10.1145/3052973.3053004`.

[59] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. "CodeMatch: obfuscation won't conceal your repackaged app." *Proceedings of ESEC/FSE, 2017.* ACM, 2017. URL: `https://doi.org/10.1145/3106237.3106305`.

[60] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. "Static analysis of android apps: A systematic literature review." *Inf. Softw. Technol.* 88 (2017), pp. 67–95. URL: `https://doi.org/10.1016/j.infsof.2017.04.001`.

[61] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. “LibD: scalable and precise third-party library detection in android markets.” *Proceedings of the 39th ICSE, 2017.* IEEE / ACM, 2017. URL: `https://doi.org/10.1109/ICSE.2017.38`.

[62] Björn Mathis, Vitalii Avdiienko, Ezekiel O. Soremekun, Marcel Böhme, and Andreas Zeller. “Detecting information flow by mutating input data.” *Proceedings of the 32nd ASE, 2017.* IEEE Computer Society, 2017. URL: `https://doi.org/10.1109/ASE.2017.8115639`.

[63] Felix Pauck. “Cooperative static analysis of Android applications.” MA thesis. Germany: Paderborn University, 2017.

[64] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. “A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software.” *IEEE Trans. Software Eng.* 43.6 (2017), pp. 492–530. URL: `https://doi.org/10.1109/TSE.2016.2615307`.

[65] Dirk Beyer and Matthias Dangl. “Strategy Selection for Software Verification Based on Boolean Features - A Simple but Effective Approach.” *Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II.* Lecture Notes in Computer Science. Springer, 2018. URL: `https://doi.org/10.1007/978-3-030-03421-4_11`.

[66] Dirk Beyer, Marie-Christine Jakobs, Thomas Lemberger, and Heike Wehrheim. “Reducer-based construction of conditional verifiers.” *Proceedings of the 40th ICSE, 2018.* ACM, 2018. URL: `https://doi.org/10.1145/3180155.3180259`.

[67] Felix Pauck, Eric Bodden, and Heike Wehrheim. “Do Android taint analysis tools keep their promises?” *Proceedings of the 26th ESEC/FSE, 2018.* ACM, 2018. URL: `https://doi.org/10.1145/3236024.3236029`.

[68] Lina Qiu, Yingying Wang, and Julia Rubin. “Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe.” *Proceedings of the 27th ISSTA, 2018.* ACM, 2018. URL: `https://doi.org/10.1145/3213846.3213873`.

[69] Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. “ORLIS: obfuscation-resilient library detection for Android.” *Proceedings of the 5th MOBILESoft@ICSE, 2018.* ACM, 2018. URL: `https://doi.org/10.1145/3197231.3197248`.

[70] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. “JN-SAF: Precise and Efficient NDK/JNI-aware Inter-language Static Analysis Framework for Security Vetting of Android Applications with Native Code.” *Proceedings of CCS, 2018.* ACM, 2018. URL: `https://doi.org/10.1145/3243734.3243835`.

[71] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. “A Systematic Evaluation of Static API-Misuse Detectors.” *IEEE Trans. Software Eng.* 45.12 (2019), pp. 1170–1188. URL: `https://doi.org/10.1109/TSE.2018.2827384`.

[72] Tanzirul Azim, Arash Alavi, Iulian Neamtiu, and Rajiv Gupta. “Dynamic slicing for Android.” *Proceedings of the 41st ICSE, 2019.* IEEE / ACM, 2019. URL: `https://doi.org/10.1109/ICSE.2019.00118`.

[73] Dirk Beyer and Marie-Christine Jakobs. “CoVeriTest: Cooperative Verifier-Based Testing.” *Proceedings of FASE@ETAPS, 2019.* Springer, 2019.

[74] Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. "Performance-Boosting Sparsification of the IFDS Algorithm with Applications to Taint Analysis." *Proceedings of the 34th ASE, 2019*. IEEE, 2019. URL: `https://doi.org/10.1109/ASE.2019.00034`.

[75] Felix Pauck and Heike Wehrheim. "Together strong: cooperative Android app analysis." *Proceedings of ESEC/FSE, 2019*. ACM, 2019. URL: `https://doi.org/10.1145/3338906.3338915`.

[76] Felix Pauck and Shikun Zhang. "Android App Merging for Benchmark Speed-Up and Analysis Lift-Up." *Proceedings of A-Mobile@ASE, 2019*. IEEE, 2019. URL: `https://doi.org/10.1109/ASEW.2019.00019`.

[77] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. "PhASAR: An Inter-procedural Static Analysis Framework for C/C++." *Proceedings of the 25th TACAS, 2019*. Ed. by Tomás Vojnar and Lijun Zhang. Lecture Notes in Computer Science. Springer, 2019. URL: `https://doi.org/10.1007/978-3-030-17465-1_22`.

[78] Abhishek Tiwari, Sascha Groß, and Christian Hammer. "IIFA: Modular Inter-app Intent Information Flow Analysis of Android Applications." *Proceedings of SecureComm, 2019*. Springer, 2019.

[79] Dirk Beyer and Heike Wehrheim. "Verification Artifacts in Cooperative Verification: Survey and Unifying Component Framework." *Proceedings of the 9th ISoLA, 2020*. Springer, 2020. URL: `https://doi.org/10.1007/978-3-030-61362-4_8`.

[80] Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. "Modular collaborative program analysis in OPAL." *Proceedings of ESEC/FSE, 2020*. ACM, 2020.

[81] Ben Hermann, Stefan Winter, and Janet Siegmund. "Community expectations for research artifacts and evaluation processes." *Proceedings of the 28th ESEC/FSE, 2020*. ACM, 2020. URL: `https://doi.org/10.1145/3368089.3409767`.

[82] Holger Karl, Dennis Kundisch, Friedhelm Meyer auf der Heide, and Heike Wehrheim. "A Case for a New IT Ecosystem: On-The-Fly Computing." *Bus. Inf. Syst. Eng.* 62.6 (2020), pp. 467–481. URL: `https://doi.org/10.1007/s12599-019-00627-x`.

[83] Wei You, Bin Liang, Wenchang Shi, Peng Wang, and Xiangyu Zhang. "TaintMan: An ART-Compatible Dynamic Taint Analysis Framework on Unmodified and Non-Rooted Android Devices." *IEEE Trans. Dependable Secur. Comput.* 17.1 (2020), pp. 209–222. URL: `https://doi.org/10.1109/TDSC.2017.2740169`.

[84] Khaled Ahmed, Mieszko Lis, and Julia Rubin. "Mandoline: Dynamic Slicing of Android Applications with Trace-Based Alias Analysis." *Proceedings of ICST, 2021*. IEEE, 2021. URL: `https://doi.org/10.1109/ICST49551.2021.00022`.

[85] Marie-Christine Jakobs, Felix Pauck, Marco Platzner, Heike Wehrheim, and Tobias Wiersema. "Software/Hardware Co-Verification for Custom Instruction Set Processors." *IEEE Access* 9 (2021), pp. 160559–160579. URL: `https://doi.org/10.1109/ACCESS.2021.3131213`.

[86] Austin Mordahl and Shiyi Wei. "The impact of tool configuration spaces on the evaluation of configurable taint analysis for Android." *Proceedings of ISSTA, 2021*. ACM, 2021. URL: `https://doi.org/10.1145/3460319.3464823`.

[87] Felix Pauck and Heike Wehrheim. “Jicer: Simplifying Cooperative Android App Analysis Tasks.” *Proceedings of the 21st SCAM, 2021.* IEEE, 2021. URL: `https://doi.org/10.1109/SCAM52516.2021.00031`.

[88] Xian Zhan, Lingling Fan, Sen Chen, Feng Wu, Tianming Liu, Xiapu Luo, and Yang Liu. “ATVHUNTER: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications.” *Proceedings of the 43rd ICSE, 2021.* IEEE, 2021. URL: `https://doi.org/10.1109/ICSE43902.2021.00150`.

[89] Dirk Beyer, Jan Haltermann, Thomas Lemberger, and Heike Wehrheim. “Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR.” *Proceedings of the 44th ICSE, 2022.* ACM, 2022. URL: `https://doi.org/10.1145/3510003.3510064`.

[90] Dirk Beyer, Sudeep Kanav, and Cedric Richter. “Construction of Verifier Combinations Based on Off-the-Shelf Verifiers.” *Proceedings of FASE@ETAPS, 2022.* Lecture Notes in Computer Science. Springer, 2022. URL: `https://doi.org/10.1007/978-3-030-99429-7_3`.

[91] Linghui Luo, Felix Pauck, Goran Piskachev, Manuel Benz, Ivan Pashchenko, Martin Mory, Eric Bodden, Ben Hermann, and Fabio Massacci. “TaintBench: Automatic real-world malware benchmarking of Android taint analyses.” *Empir. Softw. Eng.* 27.1 (2022), p. 16. URL: `https://doi.org/10.1007/s10664-021-10013-5`.

[92] Felix Pauck. “Scaling Arbitrary Android App Analyses.” *Proceedings of A-Mobile@ASE, 2022.* ACM, 2022. URL: `https://doi.org/10.1145/3551349.3561339`.

[93] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. “JuCify: A Step Towards Android Code Unification for Enhanced Static Analysis.” *Proceedings of the 44th ICSE, 2022.* ACM, 2022. URL: `https://doi.org/10.1145/3510003.3512766`.

[94] Stefan Schott and Felix Pauck. “Benchmark Fuzzing for Android Taint Analyses.” *Proceedings of the 22nd SCAM, 2022.* To appear. IEEE, 2022. URL: `https://doi.org/10.1109/SCAM55253.2022.00007`.

[95] Sinan Wang, Yibo Wang, Xian Zhan, Ying Wang, Yepang Liu, Xiapu Luo, and Shing-Chi Cheung. “APER: Evolution-Aware Runtime Permission Misuse Detection for Android Apps.” *Proceedings of the 44th ICSE, 2022.* ACM, 2022. URL: `https://doi.org/10.1145/3510003.3510074`.

[96] Stefan Winter, Christopher Steven Timperley, Ben Hermann, Jürgen Cito, Jonathan Bell, Michael Hilton, and Dirk Beyer. “A retrospective study of one decade of artifact evaluations.” *Proceedings of the 30th ESEC/FSE, 2022.* ACM, 2022. URL: `https://doi.org/10.1145/3540250.3549172`.

[97] Jiwei Yan, Shixin Zhang, Yepang Liu, Jun Yan, and Jian Zhang. “ICCBot: Fragment-Aware and Context-Sensitive ICC Resolution for Android Applications.” *Proceedings of the 44th ICSE, 2022.* ACM/IEEE, 2022. URL: `https://doi.org/10.1145/3510454.3516864`.

[98] *Amandroid.* URL: `https://github.com/arguslab/Argus-SAF` (visited on 07. 12. 2022).

[99] *Amandroid (3.1.2).* URL: `https://github.com/arguslab/Argus-SAF/releases/tag/v3.1.2` (visited on 07. 12. 2022).

[100] *Amandroid (3.2.0)*. URL: https://github.com/arguslab/Argus-SAF/releases/tag/v3.2.0 (visited on 07. 12. 2022).

[101] *Amandroid (3.2.1)*. URL: https://github.com/arguslab/Argus-SAF/blob/master/binaries/argus-saf-3.2.1-SNAPSHOT-assembly.jar (visited on 07. 12. 2022).

[102] *AMT (2.0.0)*. URL: https://github.com/FoelliX/AMT/releases/tag/2.0.0 (visited on 07. 12. 2022).

[103] *Android Components*. URL: https://developer.android.com/guide/components/fundamentals (visited on 07. 12. 2022).

[104] *Android CVE*. URL: https://www.cvedetails.com/product/19997/Google-Android.html (visited on 07. 12. 2022).

[105] *Android Studio*. URL: https://developer.android.com/studio (visited on 07. 12. 2022).

[106] *APER-mapping*. URL: https://github.com/sqlab-sustech/APER-mapping (visited on 07. 12. 2022).

[107] *APK Downloader*. URL: https://apps.evozi.com/apk-downloader (visited on 07. 12. 2022).

[108] *APK-Simplifier (2.0.1)*. URL: https://github.com/FoelliX/APK-Simplifier/releases/tag/2.0.1-SNAPSHOT (visited on 07. 12. 2022).

[109] *ApkCombiner (1.0.1)*. URL: https://github.com/lilicoding/ApkCombiner/blob/master/release/ApkCombiner-1.0.1.jar (visited on 07. 12. 2022).

[110] *ApkParser*. URL: https://github.com/hsiafan/apk-parser (visited on 07. 12. 2022).

[111] *AQL-Answer (XSD)*. URL: https://github.com/FoelliX/AQL-System/blob/master/schemas/answer.xsd (visited on 07. 12. 2022).

[112] *AQL-CheckOperator (2.0.0)*. URL: https://github.com/FoelliX/AQL-CheckOperator/releases/tag/2.0.0 (visited on 07. 12. 2022).

[113] *AQL-Query (Grammar)*. URL: https://github.com/FoelliX/AQL-System/blob/master/src/de/foellix/aql/datastructure/handler/QuestionGrammar.jj (visited on 07. 12. 2022).

[114] *AQL-System*. URL: https://FoelliX.github.io/AQL-System (visited on 07. 12. 2022).

[115] *AQL-System (Development Tutorial)*. URL: https://github.com/FoelliX/AQL-System/wiki/Install_Compile (visited on 07. 12. 2022).

[116] *AQL-System (Maven)*. URL: https://mvnrepository.com/artifact/de.foellix/AQL-System (visited on 07. 12. 2022).

[117] *AQL-System (Rules Tutorial)*. URL: https://github.com/FoelliX/AQL-System/wiki/Rules (visited on 07. 12. 2022).

[118] *AQL-System (Variables Tutorial)*. URL: https://github.com/FoelliX/AQL-System/wiki/Variables (visited on 07. 12. 2022).

[119] *AsyncTask*. URL: https://developer.android.com/reference/android/os/AsyncTask (visited on 07. 12. 2022).

[120] *Build Tools*. URL: https://developer.android.com/studio/releases/build-tools (visited on 07. 12. 2022).

[121] *CameraAPI*. URL: https://developer.android.com/guide/topics/media/camera (visited on 07. 12. 2022).

[122] *CodeQL*. URL: https://codeql.github.com (visited on 07. 12. 2022).

[123] *CoDiDroid*. URL: https://FoelliX.github.io/CoDiDroid (visited on 07. 12. 2022).

[124] *Configuration (XSD)*. URL: https://github.com/FoelliX/AQL-System/blob/master/schemas/config.xsd (visited on 07. 12. 2022).

[125] *DIALDroid-Bench*. URL: https://github.com/amiangshu/dialdroid-bench (visited on 07. 12. 2022).

[126] *DIALDroid (September 2017)*. URL: https://github.com/dialdroid-android/DIALDroid (visited on 07. 12. 2022).

[127] *DidFail (March 2015)*. URL: https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=508078 (visited on 07. 12. 2022).

[128] *DroidBench*. URL: https://github.com/secure-software-engineering/DroidBench/tree/develop (visited on 07. 12. 2022).

[129] *DroidBench's StrongUpdate1 (Source Code)*. URL: https://github.com/secure-software-engineering/DroidBench/blob/9898d71ca9053fc0c918a0aac7a768746d6738b2/eclipse-project/Aliasing/StrongUpdate1/src/de/ecspride/MainActivity.java (visited on 07. 12. 2022).

[130] *DroidRA (April 2017)*. URL: https://github.com/lilicoding/DroidRA (visited on 07. 12. 2022).

[131] *DroidSafe (June 2016 (Final))*. URL: https://github.com/MIT-PAC/droidsafe-src (visited on 07. 12. 2022).

[132] *F-Droid*. URL: https://www.f-droid.org (visited on 07. 12. 2022).

[133] *FlowDroid*. URL: https://github.com/secure-software-engineering/FlowDroid/releases (visited on 07. 12. 2022).

[134] *FlowDroid (2.10.0)*. URL: https://github.com/secure-software-engineering/FlowDroid/releases/tag/v2.10 (visited on 07. 12. 2022).

[135] *FlowDroid (2.7.1)*. URL: https://github.com/secure-software-engineering/FlowDroid/releases/tag/v2.7.1 (visited on 07. 12. 2022).

[136] *FlowDroid (2.9.0)*. URL: https://github.com/secure-software-engineering/FlowDroid/releases/tag/v2.9 (visited on 07. 12. 2022).

[137] *FlowDroid (April 2017 (Nightly))*. URL: https://github.com/secure-software-engineering/soot-infoflow-android/wiki (visited on 08. 03. 2018).

[138] *FlowDroid (DroidBench only)*. URL: https://github.com/secure-software-engineering/FlowDroid/tree/develop/soot-infoflow-android/test/soot/jimple/infoflow/android/test (visited on 07. 12. 2022).

[139] *FlowDroid (Ignoring Libraries)*. URL: https://github.com/secure-software-engineering/FlowDroid/blob/3e531f8d560672e78b113ca825a686979c9d719d/soot-infoflow-android/src/soot/jimple/infoflow/android/config/SootConfigForAndroid.java (visited on 07. 12. 2022).

[140] *FlowDroid Test Case for DroidBench's StrongUpdate1 (Source Code)*. URL: https://github.com/secure-software-engineering/FlowDroid/blob/27b642da5ad5aa19e33dd990a460b30cbc4a7667/soot-infoflow-android/test/soot/jimple/infoflow/android/test/droidBench/AliasingTest.java (visited on 07. 12. 2022).

[141] *FossDroid*. URL: https://www.FossDroid.com (visited on 07. 12. 2022).

[142] *Google Play Store*. URL: https://play.google.com (visited on 07. 12. 2022).

[143] *Google's Security Report 2021*. URL: https://storage.googleapis.com/android-com/resources/enterprise/pdfs/AE%20Security%20Paper_V6%20CM.pdf (visited on 07. 12. 2022).

[144] *Google's Security Reports*. URL: https://source.android.com/docs/security/overview/reports (visited on 07. 12. 2022).

[145] *GraphML*. URL: http://graphml.graphdrawing.org (visited on 07. 12. 2022).

[146] *HornDroid (0.0.1)*. URL: https://github.com/ylya/horndroid (visited on 07. 12. 2022).

[147] *IC3 (0.2.1)*. URL: https://github.com/FoelliX/ic3 (visited on 07. 12. 2022).

[148] *ICC-Bench*. URL: https://github.com/fgwei/ICC-Bench (visited on 07. 12. 2022).

[149] *IccTA (2.10.0)*. URL: https://github.com/secure-software-engineering/FlowDroid/releases/tag/v2.10 (visited on 07. 12. 2022).

[150] *IccTA (2.9.0)*. URL: https://github.com/secure-software-engineering/FlowDroid/releases/tag/v2.9 (visited on 07. 12. 2022).

[151] *IccTA (February 2016)*. URL: https://github.com/lilicoding/soot-infoflow-android-iccta (visited on 07. 12. 2022).

[152] *Instructions*. URL: https://docs.oracle.com/javase/specs/jvms/se12/html/jvms-6.html#jvms-6.5 (visited on 07. 12. 2022).

[153] *Intents and Intent Filters*. URL: https://developer.android.com/guide/components/intents-filters (visited on 07. 12. 2022).

[154] *JADX*. URL: https://github.com/skylot/jadx (visited on 07. 12. 2022).

[155] *Java Native Interface*. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jni (visited on 07. 12. 2022).

[156] *JavaCC*. URL: https://javacc.github.io/javacc (visited on 07. 12. 2022).

[157] *JAXB*. URL: https://javaee.github.io/jaxb-v2 (visited on 07. 12. 2022).

[158] *Jetpack (androidx.)* URL: https://developer.android.com/jetpack/androidx (visited on 07. 12. 2022).

[159] *Jicer (2.0.0)*. URL: https://github.com/FoelliX/Jicer/releases/tag/2.0.0 (visited on 07. 12. 2022).

[160] *Jicer (Ignoring Libraries)*. URL: https://github.com/FoelliX/Jicer/blob/f5138df9b93521393f7b394984e5c109233aae85/src/de/foellix/aql/jicer/config/Config.java (visited on 07. 12. 2022).

[161] *JSON Schema*. URL: https://json-schema.org (visited on 07. 12. 2022).

[162] *Kotlin for Android*. URL: https://kotlinlang.org/docs/android-overview.html (visited on 07. 12. 2022).

[163] *Lollipop.* URL: `https://developer.android.com/about/versions/lollipop` (visited on 07. 12. 2022).

[164] *Luca-App: Sicherheitslücke ermöglicht Angriffe auf Gesundheitsämter.* URL: `https://www.computerbase.de/2021-05/luca-app-sicherheitsluecke-ermoeglicht-angriffe-auf-gesundheitsaemter` (visited on 07. 12. 2022).

[165] *Marshmallow.* URL: `https://developer.android.com/about/versions/marshmallow/android-6.0-changes` (visited on 07. 12. 2022).

[166] *Mobile app developers' misconfiguration of third party services leave personal data of over 100 million exposed.* URL: `https://research.checkpoint.com/2021/mobile-app-developers-misconfiguration-of-third-party-services-leave-personal-data-of-over-100-million-exposed` (visited on 07. 12. 2022).

[167] *MultiDex.* URL: `https://developer.android.com/studio/build/multidex` (visited on 07. 12. 2022).

[168] *NOAH (2.0.0).* URL: `https://github.com/FoelliX/NOAH/releases/tag/2.0.0` (visited on 07. 12. 2022).

[169] *Operating System Market Share.* URL: `https://gs.statcounter.com/os-market-share#quarterly-200901-202204` (visited on 07. 12. 2022).

[170] *PermissionFinder (to be released).* URL: `https://github.com/FoelliX/PermissionFinder` (visited on 07. 12. 2022).

[171] *PIM (2.0.0).* URL: `https://github.com/FoelliX/PIM/releases/tag/2.0.0` (visited on 07. 12. 2022).

[172] *Platform Releases.* URL: `https://developer.android.com/studio/releases/platforms` (visited on 07. 12. 2022).

[173] *Popular Android apps with 142.5 million collective installs leak user data.* URL: `https://cybernews.com/security/research-popular-android-apps-with-142-5-million-collective-downloads-are-leaking-user-data` (visited on 07. 12. 2022).

[174] *Protocol Buffers.* URL: `https://developers.google.com/protocol-buffers` (visited on 07. 12. 2022).

[175] *ReproDroid.* URL: `https://FoelliX.github.io/ReproDroid` (visited on 07. 12. 2022).

[176] *ReproDroid Errata.* URL: `https://foellix.github.io/ReproDroid/#errata` (visited on 07. 12. 2022).

[177] *SARIF.* URL: `https://docs.oasis-open.org/sarif/sarif/v2.0/sarif-v2.0.html` (visited on 07. 12. 2022).

[178] *Securibench.* URL: `https://too4words.github.io/securibench-micro` (visited on 07. 12. 2022).

[179] *Soot.* URL: `https://soot-oss.github.io/soot` (visited on 07. 12. 2022).

[180] *Soot 3.3.0 (Maven).* URL: `https://mvnrepository.com/artifact/ca.mcgill.sable/soot/3.3.0` (visited on 07. 12. 2022).

[181] *Soot 4.3.0 (Maven).* URL: `https://mvnrepository.com/artifact/org.soot-oss/soot/4.3.0` (visited on 07. 12. 2022).

[182] *SQL (1987).* URL: `https://www.iso.org/standard/16661.html` (visited on 07. 12. 2022).

[183] *SQL (2016)*. URL: `https://www.iso.org/standard/63555.html` (visited on 07. 12. 2022).

[184] *SQL (under development)*. URL: `https://www.iso.org/standard/76583.html` (visited on 07. 12. 2022).

[185] *SV-Comp*. URL: `https://sv-comp.sosy-lab.org` (visited on 07. 12. 2022).

[186] *TaintBench*. URL: `https://TaintBench.github.io` (visited on 07. 12. 2022).

[187] *Transformation Rules (XSD)*. URL: `https://github.com/FoelliX/AQL-System/blob/master/schemas/rules.xsd` (visited on 07. 12. 2022).

[188] *Using Java Reflection*. URL: `https://www.oracle.com/technical-resources/articles/java/javareflection.html` (visited on 07. 12. 2022).

[189] *WALA*. URL: `http://wala.sourceforge.net` (visited on 07. 12. 2022).

[190] *Witnesses*. URL: `https://sv-comp.sosy-lab.org/2015/witnesses` (visited on 07. 12. 2022).

[191] *XSD*. URL: `https://www.w3.org/TR/xmlschema11-1` (visited on 07. 12. 2022).

[192] *YAML*. URL: `https://yaml.org` (visited on 07. 12. 2022).

FOELLIX.DE