Do Android Taint Analysis Tools Keep Their Promises?

Felix Pauck
University Paderborn
Paderborn, Germany
fpauck@mail.uni-paderborn.de

Eric Bodden
University Paderborn
Paderborn, Germany
eric.bodden@uni-paderborn.de

Heike Wehrheim
University Paderborn
Paderborn, Germany
wehrheim@uni-paderborn.de

ABSTRACT

In recent years, researchers have developed a number of tools to conduct taint analysis of Android applications. While all the respective papers aim at providing a thorough empirical evaluation, comparability is hindered by varying or unclear evaluation targets. Sometimes, the apps used for evaluation are not precisely described. In other cases, authors use an established benchmark but cover it only partially. In yet other cases, the evaluations differ in terms of the data leaks searched for, or lack a ground truth to compare against. All those limitations make it impossible to truly compare the tools based on those published evaluations.

We thus present ReproDroid, a framework allowing the accurate comparison of Android taint analysis tools. ReproDroid supports researchers in inferring the ground truth for data leaks in apps, in automatically applying tools to benchmarks, and in evaluating the obtained results. We use ReproDroid to comparatively evaluate on equal grounds the six prominent taint analysis tools Amandroid, DIALDroid, DidFail, DroidSafe, FlowDroid and IccTA. The results are largely positive although four tools violate some promises concerning features and accuracy. Finally, we contribute to the area of unbiased benchmarking with a new and improved version of the open test suite DroidBench.

CCS CONCEPTS

• **Software and its engineering → Empirical software validation**; *Software testing and debugging*;

KEYWORDS

Android Taint Analysis, Tools, Benchmarks, Empirical Studies, Reproducibility.

ACM Reference Format:
Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android Taint Analysis Tools Keep Their Promises?. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236024.3236029

1 INTRODUCTION

With smartphones becoming a part of everyone's daily life, users frequently downloading new apps to their phones and even performing security critical applications like banking, or the coordination of medical treatments, the probability of an undesired leak of private data increases steadily. Such data leaks may arise form coding mistakes but also may be the result of malicious attacks. According to Gartner [12], currently 86% of all mobile phones use Android as operating system. In the past, researchers have proposed a number of tools to detect data leaks in Android applications. The tools employ various analysis techniques, ranging from static [2, 7, 9, 11, 13, 15, 17, 19, 28] over dynamic [10, 29] and hybrid [1] analyses to methods built on logical reasoning [8]. Static analysis tools often employ a form of *taint* analysis: sensitive data (i.e., data from specific private sources) is tainted and then statically tracked through the application's data flow and sometimes control flow. Whenever tainted data reaches a pre-defined public sink, the taint analysis tool reports a privacy leak.

Since several decades, static analysis is known to be undecidable [24]. Undecidability forces static taint analysis tools to approximate the data flows they compute. In case of an over-approximation, the taint analysis might report spurious warnings, so-called *false positives*, while in the case of under-approximations it may miss actual data flows, resulting in *false negatives*. While all static taint analysis tools naturally share the idea of tracking taints, each tool has its own strengths and weaknesses. For instance, some but not all tools have good support for handling component lifecycles, callbacks or inter-component communication (ICC). Further, the tools provide different levels of precision by supporting (or not) an object, field or context-sensitive analysis [26]. Due to these various features, both researchers and practitioners wonder which tool is the optimal choice in which application context—a question that can only be answered through a comparative evaluation.

Many papers proposing taint-analysis tools evaluate those tools using benchmarks such as the open test suites DroidBench [2] or ICC-Bench [28]. These so-called *micro-benchmarks* consist of artifical mini-apps developed for benchmarking purposes only. Each app or a predefined combination of multiple apps represents one benchmark case. Benchmark cases contain intentionally encoded data leaks, e.g. flows of data from a statement accessing the device's serial number `getDeviceId()` to a statement sending SMS `sendTextMessage(...)`. While the usage of such common benchmark suites seems to provide a solid basis for an unbiased and systematic comparison, it suffers from some fundamental drawbacks: For instance, the micro-benchmarks mostly lack information about the exact data leaks contained in each test case, i.e., the so-called *ground truth*. Instead comparisons take place on the grounds of just *counting* the number of data leaks found and comparing it against the one given for the benchmark case. This hides incorrect leak detections in cases where these numbers match up coincidentally. As our experiments confirm, this problem exists not just in theory but has impaired past evaluations.

The comparison gets even less systematic when moving to *real-world apps*, i.e., apps that can be downloaded from an app market such as Google's Play store.[1] The inclusion of such apps in experiments is indispensable when it comes to evaluating the tools for scalability. Yet, in most recent works reporting the evaluation of Android taint analysis tools it is quite unclear which apps exactly have been used in the respective evaluation. To give one example of many, Li et al. state "We randomly selected 50 apps from our Googleplay set for our study." without naming them [17]. An additional related problem is that it is unclear which exact data leaks are to be found and have been found. Since for real-world apps not even the number of actual data leaks is known (let alone their exact data flows), the evaluations simply report the number of leaks found, without being able to assess which fraction of the tools' warnings might be false, and how many leaks are missed. In fact, *this way of measuring success rewards rather than penalizes false positives*, as higher numbers are frequently seen as better results. In summary, tool benchmarks frequently lack confirmability and reproducibility.

The goal of this work is to remedy this unsatisfactory situation by enabling a reproducible, fair and unbiased comparison. Specifically, we present ReproDroid, a framework allowing for the accurate and systematic benchmarking of Android taint analysis tools. The approach comprises the following original contributions. First, we introduce the *Android App Analysis Query Language (*AQL*)* which is used to precisely define analysis questions and answers. In questioning, the usage of AQL allows us to run all examined tools on the same target, i.e., inspect the existence of the same flow of data. On the answer side, AQL acts as a standardized language for describing the flows found. Second, the associated AQL-*System* delegates analysis questions to appropriate tools by matching the question subject against tool capabilities, and converts the produced answers into the AQL format. Through those unique features, the AQL-System supports the completely automatic benchmarking of tools. Third, we present the *Benchmark Refinement and Execution Wizard* (Brew), which helps one to refine, execute and evaluate precisely formulated benchmarks. Brew allows us and others to more easily determine the ground truth of data leaks for test apps.

As another major contribution of this work, we use ReproDroid to carry out a *reproducible* comparison of some of the most prominent taint analysis tools for Android apps: Amandroid, DIALDroid, DidFail, DroidSafe, FlowDroid and IccTA, on 265 apps (235 micro-benchmarks and 30 real-world apps). We find that the experiments reproduce most but not all results of previously published evaluations. To further contribute to the area of systematic benchmarking, we provide all benchmarks, now *precisely* defined, within a new version of the open test suite DroidBench.[2]

To summarize, this paper presents the following contributions:

- the Android App Analysis Query Language (AQL), a mechanism to precisely define taint-analysis queries and responses,
- the AQL-*System*, which dispatches AQL queries to tools and consolidates their responses,
- the *Benchmark Refinement and Execution Wizard* (Brew), a tool to refine, execute and precisely evaluate formulated benchmarks, and
- a large-scale, comparative empirical evaluation of Amandroid, DIALDroid, DidFail, DroidSafe, FlowDroid and IccTA on DIALDroid-Bench, DroidBench, ICC-Bench and 21 newly developed test apps.

The remainder of this paper is structured as follows. Section 2 introduces some basic concepts and a running example. Section 3 details ReproDroid's three major components. Section 4 and 5 present our large-scale comparative evaluation and its results for the six mentioned taint analysis tools. We discuss related work in Section 6 and conclude in Section 7.

2 BACKGROUND

In this section, we start with introducing basic terminology and concepts, in particular explain taint analysis and the current form of benchmark suites.

2.1 Taint Analysis

The purpose of taint analysis is to track the flow of sensitive data within programs. For smart-phone apps, a data leak occurs when private data (phone numbers, device identifiers, contact data) flows from sensitive sources to public sinks (Internet, SMS transmission). In this case, sensitive data is leaked. Taint analyses are most frequently used to detect such leaks: it taints sensitive data at its source, and propagates the taint information through the application (or even a combination of apps), issuing a warning if tainted data reaches a sink. Taint tracking can be performed statically on program code, or dynamically by executing the app and monitoring tainted data.

Android provides an open communication structure between apps. Moreover, when Android apps include third-party libraries, those execute with the same access rights as the app itself. Those features make Android apps particularly vulnerable to attacks targeting private data. Taint-analysis tools can cope with these special features to various extents. The following programming-language features and analysis functionalities are supported by some but not all tools:

Aliasing The same memory location / object may be referenced by different variables. In this situation, one variable is an alias of another, and a taint related to one alias must be carried over to all others.

Static Fields Static fields are declared on a type, not their instance. In particular, their values can be accessed without requiring access to any object reference. Static taint analyses must thus treat static fields differently from instance fields.

Lifecycle and Callbacks Each Android component has its own lifecycle, defining a sequence of invocations to callback functions that the Android framework issues at appropriate lifecycle events. User-interface interactions map to callbacks as well. To model all possible execution sequences of an app, the analysis must take all appropriate callbacks into account, and it can do so with various levels of precision.

Inter-Component Communication A leak may originate in one class and end in another. Additionally, Android allows

[1]Google Play store https://play.google.com/store/apps
[2]https://FoelliX.github.io/ReproDroid/#droidbench

```
Found a flow to sink virtualinvoke $r4.<android.telephony.
    → SmsManager: void sendTextMessage(java.lang.String,
    → java.lang.String,java.lang.String,android.app.
    → PendingIntent,android.app.PendingIntent)>("+49 1234",
    →  null, $r5, null, null), from the following sources:
      - $r5 = virtualinvoke $r3.<android.telephony.
            → TelephonyManager: java.lang.String
            → getDeviceId()>() (in <de.ecspride.
            → MainActivity: void onCreate(android.os.Bundle
            → )>)
```

Listing 1: Excerpt of FlowDroid's result (`DirectLeak1.apk`)

for inter-component or inter-app communication (ICC/IAC) via the instantiation of so-called *intents* and *intent-filters*. For example, to access the device's pre-installed camera app, it is sufficient to dispatch a certain intent. Intents may propagate tainted data from one app or component to another.

Analysis Abstraction and Algorithmics Depending on the exact analysis abstraction and algorithmics, the taint analysis may or may not support different "sensitivities", such as flow, context, path, field, object and/or thread-sensitivity [2]. While generally, the support for more such "sensitivities" may increase precision, reducing the amount of false positives, the positive effects differ. For instance, while some level of object-sensitivity is known to be important for the precise analysis of Java and Android applications [26], thread-sensitivity may well be less important in the case of Android.

Reflection Java's reflection mechanism allows one to invoke methods (or access fields) through dynamically generated strings. An analysis must resolve these strings to reliably detect taint flows through such invocations.

The fact that each Android taint analysis tool supports those features to a different extent makes it important to evaluate them comparatively, as without such comparison it is impossible to tell which features actually matter. Conducting a fair and automatic comparison of tools, however, is complicated by differences in pursued target flows and output formats. A necessary part of every taint analysis is the identification of sources, i.e., statements that extract sensitive information from a certain resource, and sinks, i.e., statements that exfiltrate information out of the app. There exist different definitions of sources and sinks as well as various approaches to determine which statements belong to which set [3, 22]. For instance, some analysis tools consider logging statements as sinks, others do not. Whereas one analysis approach defines that each source and sink has to be protected by at least one permission, another might not consider such constraints.

The second difference concerns the representation of flows between sources and sinks, for which every tool uses its own (usually textual, individually-structured) format. This renders it impossible to automatically compare results produced by different tools without additional effort. For example, the result generated by FlowDroid (see Listing 1), from a structural point of view, has little in common with DidFail's result (see Listing 2). Yet, both listings show the same finding: a flow from `getDeviceId()` (source) to `sendTextMessage(...)` (sink).

```
### 'Sink: <android.telephony.SmsManager: void
    → sendTextMessage(java.lang.String,java.lang.String,
    → java.lang.String,android.app.PendingIntent,android.
    → app.PendingIntent)>': ###
['Src: <android.telephony.TelephonyManager: java.lang.String
    →  getDeviceId()>']
```

Listing 2: Excerpt of DidFail's result (`DirectLeak1.apk`)

```
import android.telephony.TelephonyManager;
/**
 * @testcase_name DirectLeak1
 * @version 0.1
 ...
 * @description Easy testcase: The value of a source is
       → directly sent to a sink
 * @dataflow source -> sink
 * @number_of_leaks 1
 * @challenges -
 */
public class MainActivity extends Activity {
```

Listing 3: Excerpt of DroidBench's app source code (`DirectLeak1.apk`)

2.2 Benchmarks

Most Android app analysis tools are evaluated by means of benchmarks. In this context, a benchmarks is a collection of apps that have certain features. For example, the most frequently used micro-benchmark DroidBench [2] comprises 190 apps. While most Android taint analysis tools are evaluated by applying them to DroidBench, such evaluations are of limited value because DroidBench currently only imprecisely specifies the *ground truth* in each benchmark case, i.e., the correct result of flow analyses. Listing 3 gives an excerpt of the source code of one of DroidBench's apps, namely *DirectLeak1* (the one the prior analysis results referred to). Between the imports and the source code of the main activity of this app a comment is placed. This comment – including the number of leaks – is the only description of the expected analysis results: there is a source directly connected to a sink. Neither do we learn which source and sink this might be, nor how the flow propagates through the app(s). Moreover, the information is not given in a machine-readable format. Looking it up in the source code reveals that the source is a `getDeviceId()` function call, which is used as a parameter of a `sendTextMessage(...)` statement. In this case, a manual inspection is easy and can be done quickly. However, for more challenging benchmark cases or hundreds of cases this task becomes exhausting and error prone. For example, the machine readable information, number of leaks, is sometimes simply wrong (e.g. for DroidBench's *StrongUpdate1* it says 1 but the description and app execution prove that there is no leak). Still, the authors of ICC-Bench [28] adopted the same method to specify their expected results. Hence, automatically comparing actual results against expected ones is neither possible for DroidBench nor for ICC-Bench.

DroidBench and ICC-Bench are micro-benchmarks. Evaluations on real-world apps usually lack a ground truth altogether. Such evaluations usually focus on the 500 to 1000 most downloaded or most popular apps that can be downloaded from Google's PlayStore. DIALDroid-Bench [7] is a benchmark suite comprising real-world apps. Its apps, however, are given without source code, only in the form of .apk-files, and comprise no description of the data leaks they contain.

In summary, while there seems to be a general agreement of evaluating tools on the grounds of public benchmarks, the evaluation itself is imprecise: The ground truth needed to determine the tools' precision and recall is often simply not known. To evaluate tools automatically, one further misses a standardized, machine-readable format for expected as well as detected data flows. In the end, this impedes a reproducible and unbiased comparison of analysis tools. We next explain how ReproDroid overcomes these limitations.

3 APPROACH

The ReproDroid framework supports tool evaluation and comparison by the following three concepts:

- the design of AQL, a *query language* for precisely formulating questions about app properties such as flows,
- the design and implementation of a *query execution system* being able to interface to diverse tools, and
- the design and implementation of a *query wizard* for determining the ground truth in apps and for executing thus specified benchmarks and rating their outcome.

Together, they provide an automatic benchmarking system for app-analysis tools. In the following, we describe all three parts in turn.

3.1 App Analysis Language

The Android App Analysis Query Language (AQL) [20] consists of two main parts, namely AQL-Queries and AQL-Answers. AQL-Queries enable us to ask for Android specific analysis subjects in a general, tool independent way. The grammar defining AQL-Queries currently allows to ask for analysis subjects such as flows, intents, intent-filters and permissions. Considering our running example, we can enumerate all taint flows within the `DirectLeak1.apk` app by composing the following query:

```
Flows IN App('/path/to/DirectLeak1.apk') ?
```

or instead explicitly check the taint flows we expect:

```
                    Flows FROM
            Statement('getDeviceId()')
->Method('onCreate(...)')->Class('MainActivity')
        ->App('/path/to/DirectLeak1.apk')
                        TO
          Statement('sendTextMessage(...)')
->Method('onCreate(...)')->Class('MainActivity')
        ->App('/path/to/DirectLeak1.apk')
                         ?
```

Furthermore, the AQL offers several options to merge and filter queries as well as methods to match intents and intent-filters. Similarly, AQL-Answers are used to represent analysis results in a standardized form. The syntax of AQL-Answers is defined via an XML schema definition. Considering the running example again, we might get an answer that represents the flow from `getDeviceId()` to the `sendTextMessage(...)` statement (cf. Listing 4). In this, each statement comes with a precise description of where it can be found, by naming the method, class and app containing the statement. AQL supports additional syntax to uniquely identify statements and apps, for example using function-call parameters, full Jimple syntax[3] statements or `.apk` file hashes.

```
<answer>
    <flows>
        <flow>
            <reference type="from">
                <statement>... getDeviceId() ...</statement>
                <method>... onCreate(...) ...</method>
                <classname>... MainActivity</classname>
                <app>
                    <file>.../DirectLeak1.apk</file>
                    <hashes>...</hashes>
                </app>
            </reference>
            <reference type="to">
                ...
                sendTextMessage(...)
                ...
            </reference>
        </flow>
    </flows>
</answer>
```

Listing 4: Shortened AQL-Answer (`DirectLeak1.apk`)

3.2 Query Execution System

The AQL-System [32] is our approach to process AQL-Queries and to determine AQL-Answers. In the scope of this paper it is sufficient to observe the AQL-System as a blackbox, which accepts analysis questions encoded in AQL-Queries as input, executes appropriate analysis tools and converts their output into AQL-Answers. To do so, it requires a *configuration* in form of an `.xml` file that describes (a) which tools are avaliable in a certain instance of the AQL-System and how to execute these, (b) which queries can be answered by which tool and (c) how to convert a tool's result into an AQL-Answer. For instance, an AQL-System can be configured to execute FlowDroid in case of intra-app flow questions and IccTA in case of inter-app questions, since FlowDroid does not support ICC/IAC. Considering the running example such an AQL-System recognizes that FlowDroid is available and able to answer the query regarding flows inside one app only. Consequently, FlowDroid is launched by executing the command or script specified in the configuration. Once its computation is finished a tool-specific converter translates the tool's result into an AQL-Answer. We currently have converters covering in particular the six tools of our experiments in Section 4. With this, we can orchestrate the execution of tools and convert their results into the standardized AQL format.

3.3 Benchmark Refinement and Execution

ReproDroid's final component is the Benchmark Refinement and Execution Wizard (Brew) [33]. It is an assistant that can be used to do what the name suggests, first refine and then execute a benchmark. For this, it offers a graphical user interface (GUI) simplifying the handling of different sets of apps and benchmarks and the identification of sources and sinks.

[3] Jimple stands for "Java but simple". It is the primary intermediate language supported by Soot [16, 27].

The process of refining an existing benchmarks suite, i.e., completing it and bringing it into standardized format, consists of three steps:

Case Identification We load the apps of the test suite into the wizard. After loading, each app resembles one benchmark case. We can then restructure the cases by deactivating certain cases or combining cases (e.g., if we are interested in a flow from a source in one app to a sink in another app).

Source and Sink Identification For source and sink identification, the wizard displays all statements that possibly represent a source or a sink inside all apps that have been loaded during the first step. Since the wizard is just an assistant it cannot decide on its own which of these statements are real sources and sinks. However, BREW can preselect all sources and sinks according to a predefined list of sources and sinks as produced by SuSi [22] or PScout [3]. It is up to the user to perform the remaining task of deselecting all unwanted sources and sinks from the preselected ones. It is also possible to combine certain sources and sinks. This might be necessary to unify differently defined sources and sinks. For example, whereas one analysis considers the function call `getLastKnownLocation(...)`, which returns an `Location` object, as source, another analysis only considers the call of `getLongitude()` or `getLatitude()`, called upon such a `Location` object, as source. However, any of these calls refers to the same resources and hence all calls can be interpreted as a single source.

Ground Truth Identification Finally, we have to decide which cases are *positive* and which cases are *negative* benchmark cases. More precisely, there may exist cases where we define a flow that should not be found by an analysis. Determining positive and negative cases remains a manual task which requires inspection of the case.

Considering the running example only the `DirectLeak1.apk` may be loaded as first step. Then, if we choose to automatically mark all sources and sinks according to SUSI, the `getDeviceId()`, `sendTextMessage(...)` but no further statement get marked as source and sink. In the last step this results in one benchmark case, which is correctly and initially always marked as positive case by BREW.

Once the refinement steps have been completed, the benchmark can be executed and evaluated. To do so, BREW determines one AQL-Query and one (expected) AQL-Answer per benchmark case. Hence, to automatically execute and evaluate a benchmark, BREW sends a query to an AQL-System for each benchmark case and checks whether the actual result determined that way matches the expected one. For this purpose, it is checked whether one flow of the expected result matches one flow of the actual result. For example, one analysis may detect a flow from `getLastKnownLocation(...)` to `sendTextMessage(...)` whereas another analysis finds a flow from `getLongitude()` to the same sink. In both cases the expected flow, regarding the accessed resources, has been found. Consequently, one matching flow per benchmark case is sufficient.

To evaluate the outcome of a benchmark execution, BREW counts the number of successful and failed benchmark cases. A case is *successful* if a certain flow that was expected to be found has been

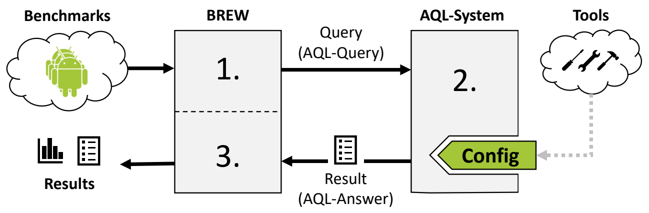


Figure 1: Sketch of the REPRODROID toolchain

found (true positive) or if a flow that was explicitly not expected to be found has not been found (true negative). In contrast, a case *fails* if an expected flow was not found (false negative) or if a not expected flow has falsely been detected (false positive). Based on this information, BREW computes the commonly used metrics precision, recall and F-measure.

Overall, BREW helps and guides its users while refining a benchmark. Refining in this context refers to the process of adding missing information to a benchmark case. Thereby insufficiently described benchmark cases (cf. Listing 3) become usable, i.e., become available in machine-readable format, which allows one to execute and evaluate the benchmark cases automatically.

3.4 The REPRODROID Toolchain

The composition of the three previously described components form our complete framework REPRODROID (see Figure 1): BREW uses the AQL-System which again uses the AQL. In Figure 1, two parts are visualized inside clouds. These symbolize the resources that exist in the community, namely analysis tools and benchmarks. Each gray rectangle represents one of REPRODROID's components. To execute and evaluate a benchmark we first have to choose one and provide it to BREW as input. Two methods are available to do so. Either an already refined benchmark can be loaded, or a new, unrefined set of apps can be refined interactively. After setting up or loading the desired benchmark, BREW will issue one AQL-Query per benchmark case. Then, one query after another arrives at an AQL-System, configured to use a certain set of tools. The AQL-System executes the tools required to answer each query (*2.*) and produces one AQL-Answer per query as output. This answer, in turn, is returned to BREW, which then decides if this actual answer matches the expected one (*3.*). BREW's GUI provides possibilities to compare single results on a textual or graphical level. The latter one depicts all sources and sinks as nodes and connections between them as edges in a graph. All known intermediate statements between each source and sink are also depicted. BREW further allows one to export complete results to an SQL database, e.g. to make them viewable online. In summary, we input a benchmark and one or more analysis tools and receive as output one benchmark result, which includes the expected and actual AQL-Answers of each benchmark case, as well as the calculated values for precision, recall and F-measure.

Considering the running example we could, for instance, input the `DirectLeak1.apk` as a benchmark consisting of only one app and setup an AQL-System to use FLOWDROID. The refinement step performed by means of BREW allows us to identify the expected leak which is consequently available as ground truth or expected AQL-Answer (see Listing 4). Along with that, an AQL-Query is composed to ask whether the expected leak actually exists. The considered

AQL-System takes this query and thereupon executes FlowDroid. Once FlowDroid finishes its computation, the AQL-System converts FlowDroid's result (see Listing 1) into an AQL-Answer (see Listing 4) which is returned to Brew. Since the expected and actual result coincide (also see Listing 4), the benchmark case is evaluated as successful. Considering this tiny benchmark, precision, recall and F-measure would be at its optimal value (1.0).

This toolchain is applicable to different scenarios. For instance, a specialist could refine a benchmark based on his expertise. This refined benchmark can then be executed and used in an evaluation by any type of user such as an analysis tool developer. The toolchain also allows us to refine and execute a benchmark in a distributed and iterative way. Multiple people can iteratively refine a single benchmark. A benchmark can be executed, extended by some additional cases and executed again without rerunning the already executed cases. This can split the burden of refining and executing a benchmark among multiple persons.[4]

4 EXPERIMENTAL SETUP

The design and implementation of ReproDroid for the first time facilitates an accurate comparison of tools. In the following, we re-evaluate six taint analysis tools for Android apps and their promises on a number of micro-benchmarks and real world apps.

4.1 Tool Selection

We start with a description of our tool selection. The tools selected for our evaluation all implement *taint* analyses. This is the most frequently used technique for finding data leaks, and thus provides us with a number of tools to be studied. Furthermore, for the purpose of this study we only consider *static* taint analysis tools. We further consider only approaches that are at least flow-sensitive or context-sensitive, such as to assure that all tools are competitors within the same league.

Table 1: List of Tools

Tool	Version
Amandroid [30]	November 2017 (3.1.2)
DIALDroid [34]	September 2017
DidFail [35]	March 2015
DroidSafe [36]	June 2016 (Final)
FlowDroid [37]	April 2017* (Nightly)
IccTA [38]	February 2016

* Has been updated since then.

Table 1 lists the tools employed in our evaluation along with their release dates and versions (if available). For all tools we generally used the most recent version. Just for FlowDroid it holds that it is so frequently updated that we had to fix a version for our experiments. All six tools we consider have at least ICC and at best IAC capabilities, except for FlowDroid, which computes flows within single components only. Yet, it is an important tool to consider because it is the most widely used static-analysis tool for Android. Considering IAC is interesting because prior evaluations typically showed a deteriorating precision when IAC benchmark cases were involved. We thus seek to specifically measure how well tools perform on more accurate IAC benchmarks.

We briefly describe some characteristics of the tools:

Analysis Engine All tools except Amandroid are based on Soot [16, 27] and operate on Jimple as intermediate language.

Source and Sink Identification The sources and sinks considered by DIALDroid, DidFail, FlowDroid and IccTA are specified by SuSi [22], a machine-learning approach for source and sink detection. DroidSafe employs its own source and sink identification (which is claimed to be even more precise than SuSi's). The list of considered sources and sinks used by Amandroid seems similar although shorter; its origin remains unclear. For our micro-benchmarks, we made sure that the sources and sinks needed for finding flows are identified by all tools.

ICC and IAC Capabilities DidFail, DIALDroid and DroidSafe are the only tools that are shipped with built-in IAC capabilities. The other tools have ICC capabilities only and require a tool called ApkCombiner [17] to lift their analysis to IAC level. ApkCombiner has been developed (along with IccTA) for precisely this purpose. It takes multiple `.apk` files as input and merges them into a single `.apk` file. DroidSafe's built-in IAC capabilities did not show any effect in our experiments, no IAC involving flows were found. Hence, we decided to use DroidSafe in combination with ApkCombiner as well. Note furthermore that DIALDroid is only able to detect inter-component or inter-app taint flows, any intra-component flows are ignored.

A number of other tools (e.g. [4, 9, 11, 14, 19]) would fit into our scope. We shortly comment on and provide reasons why we omitted them in related work (see Section 6).

4.2 Benchmarks

Our experiments are based on three benchmark suites: DroidBench [2], ICC-Bench [28] and DIALDroid-Bench [7]. The first two are well-known micro-benchmark suites which have been used in various evaluations before (usually version 2.0 or 3.0 of DroidBench and version 2.0 of ICC-Bench which we use as well). The third suite, DIALDroid-Bench, is a collection of partially malicious real-world apps downloaded from Google's Playstore and gathered by Bosu et al. [7].

In addition, we have developed 18 apps comprising 21 positive and 6 negative *feature-checking* benchmark cases. A feature-checking benchmark case exploits only one specific feature at a time and can thus be used to explicitly check the handling of a dedicated feature in a tool. This is in contrast to similar cases of DroidBench which often combines two or more features in one case. The feature-checking benchmark cases cover all features listed in Section 2.1.

Since we are particularly interested in ICC and IAC, we have developed three apps to specifically evaluate the precision of *intent-matching* algorithms. Such algorithms play an essential role when inter-component or inter-app flows are analyzed. The analysis has to detect whether a certain intent can be received by a component. If so, the action, category and data attributes of an intent have to match those of a component's intent-filter. To this end, the three apps comprise 2/6/71 positive and 4/3/139 negative cases considering matching action, category and data attributes, respectively.

Together these 21 newly developed apps represent our DroidBench extension. DroidBench together with this extension is refined by means of Brew. As a result, our benchmark suite now

[4]Instructions are available on github: https://github.com/FoelliX/BREW/wiki

contains a collection of 211 apps with (a) their source code and (b) the ground truth for data leaks in the form of AQL-Answers. We made this extended and refined benchmark suite as well as refined versions of the other suites publicly available [40] for other researchers to perform similar experiments.

4.3 Research Questions and Experiments

Our evaluation addresses the following research questions:

RQ1 Do Android app analysis tools keep their promises?
RQ2 How do the tools compare to each other with respect to accuracy?
RQ3 Which tools support large-scale analyses of real-world apps?

We designed specific experiments for every research question.

RQ1. In order to address RQ1, we first need to determine what we consider to be a "promise" of a tool. Looking at the articles introducing tools, two properties of tools play a common role: (1) the supported features and (2) the tool's accuracy, i.e., its precision, recall and F-measure as shown in experiments. Runtime appears to play a minor role: most articles only give vague runtime information. To evaluate the tools with respect to these sorts of promises, we ran the following experiments. First we prepared a benchmark set consisting of (1) the micro-benchmarks from DroidBench and ICC-Bench plus (2) our *feature-checking* benchmark cases, all refined and executed with Brew. To evaluate the six different tools, Brew is launched six times, each time with the respectively configured underlying AQL-System. Each configuration makes the AQL-System use only one tool. The setup of a tool is untouched: only required launch parameters are given and the usable memory is specified. All other options are set to default.

RQ2. As a number of researchers have already carried out a comparison of their own tool with some existing tools, we wanted to see what the outcome of a more refined comparison is. For the comparison, we chose *F-measure* as our means for evaluating accuracy. For each catergory and tool the average value is computed. Basically, for comparison the rule "the larger, the better" can be applied on the achieved value. For evaluation, we again used our refined version of DroidBench. ICC-Bench is not used since we do not want to intermix benchmark cases.

RQ3. Regarding scalability, we seek to evaluate whether the tools are able to deal with (1) large apps (in terms of code size), (2) a large numbers of apps, (3) ICC and IAC and (4) newer Android versions. For (1) we used DIALDroid-Bench, a benchmark suite containing 30 large real-world apps. Again, we employed Brew to help us determine the ground truth for these apps. Due to the size of apps, this is anything but straightforward, as a simple manual inspection of all potential flows is not feasible. We used the following procedure to nonetheless achieve a systematic derivation of a ground truth. First, we created one positive benchmark case for every pair of sources and sinks. This resulted in 841,514 potential positive benchmark cases. Second, we ran all six tools on these cases, ending with a report of 1,007 candidates for privacy leaks. 20 of these potential leaks had been found by two tools. No leak was found by more than three tools, and there were six benchmark cases for which this occurred. For all leaks that had been detected multiple times, we manually investigated the source code of the associated apps. We used the `jadx` [39] decompiler to extract the apps' source code. To inspect the decompiled code, we then used Android Studio. The manual inspection led to the confirmation of 22 positive cases. Four other positive cases had to be rejected, because there exists no data flow between the source (accessing device's location) and the sink (logging e.g. a username). By means of Brew these confirmed and rejected benchmark cases have been stored respectively as positive and negative cases in another refined version of DIALDroid-Bench. We are aware that the determined "ground truth" is most likely incomplete and does not involve all apps of DIALDroid-Bench. However, to our knowledge it represents the first available precise (in terms of flows) ground-truth description for a set of real-world apps. In cooperation with others [21] we plan to extend it in future.

For inspecting the tools' abilities to handle a high number of apps (2), ICC and IAC (3), we used our own intent-matching benchmarks. For checking their applicability to newer Android APIs (4), we ran the tools on different versions of our feature-checking test apps compiled and developed with Android Studio as well as on the DroidBench apps developed with Eclipse.

Table 2: Feature Promises

Tool	Aliasing	Static	Callbacks	Lifecycle	Inter-Procedural	Inter-Class	IAC	ICC (explicit)	ICC (implicit)	Flow-	Context-	Field-	Object-	Path-	ThreadAwareness	Reflection
										Sensitivity						
Amandroid	⊕	⊛	⊛	⊛	⊛	⊛	⊛	⊕	⊕	⊖	⊛	⊕	⊕	–	★	+
DIALDroid							⊖	⊛	⊖†							
DidFail	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯			
DroidSafe	⊕	⊛	⊖	⊛	⊛	⊛	⊕	⊕	⊕	–	⊖	⊕	⊕	–	★	+
FlowDroid	⊕	⊛	⊛	⊛	⊛	⊛	–	–	–	⊛	⊛	⊕	⊛	–	★	–
IccTA	⊕	⊛	⊛	⊛	⊛	⊛	⊛	⊛	⊕	⊛	⊛	⊕	⊛	–	★	–

◯ supported, ★ confirmed, + partially confirmed, – not confirmed, † aborted

4.4 Execution Environment

All experiments were executed on a Debian (Jessie) virtual machine, which has Java 8 (1.8.0_161) installed. It was set up to use two cores of an Intel® Xeon® CPU (E5-2695 v3 @ 2.30GHz) and 32 GB memory whereof 30 GB were assigned to the analysis tool as heap space of the respective Java virtual machine.

5 EXPERIMENTAL RESULTS

5.1 RQ1: Do Android app analysis tools keep their promises?

Feature promises. Each tool promises to support a certain set of features (see original paper and summary in surveys [18, 25]). On the basis of our feature-checking benchmark cases, we verified which features are supported. Table 2 summarizes the results. Each row stands for one tool; each column represents one feature. The entries describe the promises and their degree of fulfillment: the symbol ★ stands for *full* support, + for *partial* support and – for all feature-checking benchmark cases having been failed. Furthermore, if the symbol is circled (◯), the corresponding feature was promised to be supported. Consequently, a promise violation is denoted as a circled minus symbol (⊖). In the table, five promise violations appear. We shortly describe the reasons for these.

(1) Amandroid failed to detect the correct order of statements causing it to falsely determine a taint flow.

(2) DIALDroid struggled in dealing with implicit intents for ICC and IAC. In case of the test app for implicit ICC, its execution was aborted at the end due to a database error.
(3) DroidSafe failed to handle callbacks correctly. This is surprising, as the paper claims that due to its flow-insensitivity it can handle callbacks more easily. In addition DroidSafe seems to over-approximate in case of the context-sensitivity check, violating a second promise. However, its development has stopped in 2016, and the last supported Android version is 4.4.1. Hence, it is plausible that some features are no longer supported.

For DidFail, we could not check whether it keeps its promises since it cannot handle newer apps (see Table 5), nor apps that do not name its targeted Android API version in its manifest. To do so has become optional along with the establishment of Android Studio as dedicated development platform in 2014.

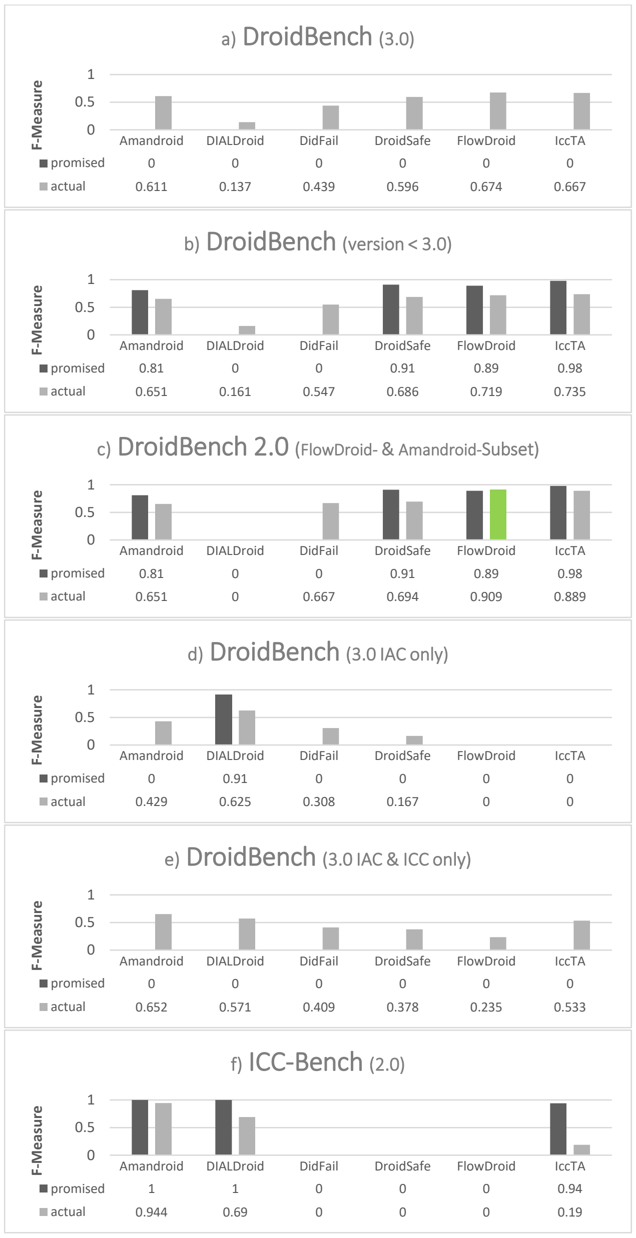
In summary, according to our feature-checking benchmarks, all promises except five are kept and most of the promised features are fully supported.

Accuracy promises. Accuracy is typically evaluated by the metrics 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 the F-measure here. Figure 2 depicts the F-measures for all six tools on different sets of micro-benchmark apps. We used different sets because the promises themselves typically refer to different benchmark suites. The dark bar always represents the *promised* value, the brighter one the *actual* value determined in our experiments.

DroidBench **version 3.0**: Figure 2*a* shows the F-measure values for the current version (3.0) of DroidBench. Since no paper promised anything for the complete 3.0 set, there are no dark (promised) bars shown. All tools have an accuracy of about 60% apart from DIALDroid and DidFail which have less. 60% does not sound to be an inspiring confidence, 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. Furthermore, we find that DIALDroid has a very low value in Figure 2*a* as well as in most of the others. This is because DIALDroid is designed for ICC and IAC cases only (tracking no intra-component flows), and consequently fails in all other cases. DidFail only reached an F-measure of 0.439 for DroidBench 3.0. This is because DidFail is the oldest and fewest updated tool considered.

DroidBench **version < 3.0**: Most tools were proposed when DroidBench 3.0 did not exist, hence an older version of DroidBench was used. The bar chart in Figure 2*b* shows the promised and actual values for DroidBench before version 3.0. On this, no tool achieved its promised value. With a relative deviation of 9% FlowDroid is closest to its promise. The other tools are at about 19% to 25% below the promised value. All tools nevertheless achieved better values for this set than for the 3.0 set.

DroidBench **version 2.0 (subset)**: The results improve if we only take a certain subset of DroidBench 2.0 into account (see Figure 2*c*). This subset has been used in the papers proposing FlowDroid and Amandroid. However, only FlowDroid is able to keep its promise for this subset.



a) DroidBench (3.0)

	Amandroid	DIALDroid	DidFail	DroidSafe	FlowDroid	IccTA
■ promised	0	0	0	0	0	0
▨ actual	0.611	0.137	0.439	0.596	0.674	0.667

b) DroidBench (version < 3.0)

	Amandroid	DIALDroid	DidFail	DroidSafe	FlowDroid	IccTA
■ promised	0.81	0	0	0.91	0.89	0.98
▨ actual	0.651	0.161	0.547	0.686	0.719	0.735

c) DroidBench 2.0 (FlowDroid- & Amandroid-Subset)

	Amandroid	DIALDroid	DidFail	DroidSafe	FlowDroid	IccTA
■ promised	0.81	0	0	0.91	0.89	0.98
▨ actual	0.651	0	0.667	0.694	0.909	0.889

d) DroidBench (3.0 IAC only)

	Amandroid	DIALDroid	DidFail	DroidSafe	FlowDroid	IccTA
■ promised	0	0.91	0	0	0	0
▨ actual	0.429	0.625	0.308	0.167	0	0

e) DroidBench (3.0 IAC & ICC only)

	Amandroid	DIALDroid	DidFail	DroidSafe	FlowDroid	IccTA
■ promised	0	0	0	0	0	0
▨ actual	0.652	0.571	0.409	0.378	0.235	0.533

f) ICC-Bench (2.0)

	Amandroid	DIALDroid	DidFail	DroidSafe	FlowDroid	IccTA
■ promised	1	1	0	0	0	0.94
▨ actual	0.944	0.69	0	0	0	0.19

Figure 2: Accuracy Promises

DroidBench **3.0 IAC only**: Bosu et al. [7] evaluated DIALDroid for all IAC cases of DroidBench 3.0 and claimed to achieve an F-measure of 0.91 (see Figure 2*d*). In our experiments we could only reproduce an F-measure of 0.625 for the same subset. Nonetheless, this is the best value for this subset. All other tools could not even reach 50%, which makes them inappropriate. As expected FlowDroid was not able to detect any inter-app flows. Unfortunately, IccTA was also unable to handle those although it should be as claimed in the associated paper and determined by the feature-checking benchmark. A closer look at the individual results reveals that IccTA could not resolve flows between `setResult(..)` and `onActivityResult(..)` statements. The intra-app parts of the

Table 3: F-Measure Scores

ID	Category	DIALDroid	DidFail	DroidSafe	IccTA	FlowDroid	Amandroid	Ø
1	FieldAndObjectSensitivity	0.000	0.800	0.667	1.000	1.000	1.000	0.745
2	Callbacks	0.000	0.769	0.667	0.897	0.897	0.500	0.622
3	UnreachableCode	0.000	1.000	0.000	0.857	1.000	0.857	0.619
4	AndroidSpecific	0.000	0.429	0.900	0.842	0.900	0.625	0.616
5	GeneralJava	0.000	0.611	0.780	0.762	0.810	0.703	0.611
6	EmulatorDetection	0.000	0.000	0.500	0.966	0.966	0.966	0.566
7	Lifecycle	0.000	0.400	0.933	0.737	0.769	0.545	0.564
8	InterComponentCommunication	0.538	0.452	0.480	0.706	0.348	0.750	0.546
9	Threading	0.000	0.667	0.000	0.667	1.000	0.667	0.500
10	ArraysAndLists	0.000	0.444	0.667	0.500	0.615	0.545	0.462
11	Aliasing	0.000	0.000	0.000	0.667	0.667	0.500	0.306
12	InterAppCommunication	0.625	0.308	0.167	0.000	0.000	0.429	0.255
13	Reflection	0.000	0.095	0.333	0.095	0.095	0.182	0.133
14	DynamicLoading	0.000	0.000	0.000	0.000	0.000	0.500	0.083
15	Native	0.000	0.000	0.000	0.000	0.000	0.333	0.056
16	ImplicitFlows	0.000	0.000	0.000	0.000	0.000	0.000	0.000
17	Reflection_ICC	0.000	0.000	0.000	0.000	0.000	0.000	0.000
18	SelfModification	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	Ø	0.065	0.332	0.339	0.483	0.504	0.506	

leak, however, could be found as well as the connection between the two apps involved. It seems to be a tiny but decisive bug in IccTA's implementation.

DroidBench **3.0 IAC and ICC only**: Once we add all ICC cases of DroidBench, all tools except DIALDroid become more accurate on average (see Figure 2*e*). IccTA improves the most and achieves an F-measure of 0.533. The best value is achieved by Amandroid which overtook DIALDroid for this extended subset. FlowDroid's value is not 0 because some of DroidBench's ICC cases communicate values through static fields or use statements receiving or sending intents as only sources or sinks. FlowDroid can handle both.

ICC-Bench: Finally, we inspected ICC-Bench (see Figure 2*f*). For this micro-benchmark set we have only three promises. Whereas Amandroid almost keeps its promise, DIALDroid lacks 31%. IccTA underperforms the most, because of the reason discussed above.

To conclude, we could not reproduce the accuracy that was claimed in the proposing papers apart from one promise made by Arzt et al. [2] for FlowDroid considering a small set of benchmark cases (see Figure 2*c*). In consequence, the answer to RQ1 is thus that the tools in general keep only parts of their promises.

5.2 RQ2: How do the tools compare to each other with respect to accuracy?

For the comparison of tools wrt. accuracy, we used DroidBench 3.0 (and its specific categories). Table 3 shows all categories of DroidBench 3.0 in its second column. The following six columns show the F-measure of each tool for all benchmark cases in the associated category. The categories supported best are placed at the top of the table. 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.

We find that the first 11 of 18 categories are handled properly by most tools. The F-measure values achieved in Category 12, namely InterAppCommunication, are inadequate. According to the promises made, the tools should be able to analyze inter-app cases. In particular, DIALDroid should achieve a top value here but it does not excel at an F-measure of 0.625. The remaining six categories are insufficiently handled by all tools apart from some special cases. However, this matches our expectations since no tool claimed to be able to handle such cases.

Before we conclude this section, a few remarks regarding the runtime: while most tools needed on average 25 seconds to analyze one app, DroidSafe required more than 200 and FlowDroid less than 10 seconds on average. In addition, DroidSafe timed out in 25 cases by exceeding a maximal execution time of 10 minutes. This also happened in case of DidFail and IccTA but only once.

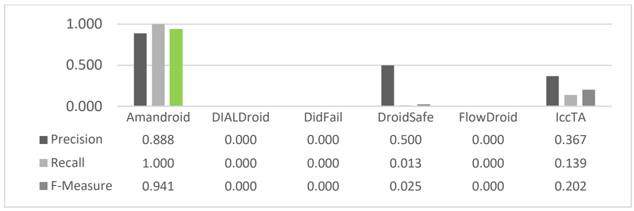
In summary, we see that there is no single "best" tool. Every tool has at least one other tool performing better in at least one category. Overall (see sums in the final row), Amandroid (0.506) and FlowDroid (0.504) score best wrt. accuracy.

Table 4: DIALDroid-Bench results

Tool	Number of successfully analyzed apps	Analysis-time per app (minutes)	Number of true / false positives
Amandroid	21	8	1 / 0
DIALDroid	20	10	0 / 4
DidFail	27	9	21 / 4
DroidSafe	2	5	0 / 0
FlowDroid	18	2	22 / 0
IccTA	18	4	6 / 0

5.3 RQ3: Which tools support large-scale analyses of real-world apps?

Table 4 (second column) shows how many real-world apps each tool was able to analyze without exceeding the maximal execution time of 30 minutes. The third column shows the average execution time of each tool, the last column how many of expected results could be matched by each tool. There is no tool that was able to analyze all apps, and apart from FlowDroid every tool on average required more than two minutes to analyze an app. The results for the newly defined ground truth (last column) reveal that FlowDroid currently seems to be the best choice to reliably deal with large apps, since all other tools missed some confirmed leaks or falsely detected rejected ones. In addition, in case of Amandroid, DIALDroid, DroidSafe and IccTA most of the 22 confirmed leaks remain undetected.



	Amandroid	DIALDroid	DidFail	DroidSafe	FlowDroid	IccTA
■ Precision	0.888	0.000	0.000	0.500	0.000	0.367
■ Recall	1.000	0.000	0.000	0.013	0.000	0.139
■ F-Measure	0.941	0.000	0.000	0.025	0.000	0.202

Figure 3: Intent-Matching: Precision, Recall, F-Measure

Table 5: Up-to-date Status

Tool	Eclipse	AndroidStudio	
	API ≤ 19	API 19	API 26
Amandroid	✓	✓	–
DIALDroid	✓	✓	✓
DidFail	✓	–†	–†
DroidSafe	✓	✓	–†
FlowDroid	✓	✓	✓
IccTA	✓	✓	–†
ApkCombiner	✓	✓	–†

✓ supported, – fails, † crashes

The intent-matching benchmark results further support this conclusion (see Figure 3). Apart from Amandroid, no tool is able to accurately match the action, category or data field. Finally, we investigated whether the tools are able to handle newer Android versions. Table 5 shows the outcomes. Amandroid fails to perform a proper analysis, however without crashing. DroidSafe, IccTA and ApkCombiner all crash while analyzing apps built for an API above 19, which is supported by the majority (82.3%) of Android devices.[5] A common cause is a tool-dependency on the `Apktool` [31]. Old versions of it fail to decompile newer Android apps. The same happens to the ApkCombiner. Thereby Amandroid, DroidSafe, FlowDroid and IccTA lose their ability to analyze inter-app scenarios. As discussed before, DidFail fails to analyze newer apps developed with Android Studio. In summary, the answer to RQ3 is: each tool in our scope still has shortcomings when it comes to analyzing real-world apps.

5.4 Threats to Validity

The main threat to the validity of our experiments arises from the manual, though tool-assisted, definition of the expected results. However, currently we see no way around it because the tools that could potentially be used to derive the ground truth are at the same time the tools we want to evaluate. Thus we cannot rely on a single tool to generate the ground truth. Moreover, ReproDroid allows us to refine the expected result definitions multiple times and thereby to achieve precise results. Similarly, also other experts can use ReproDroid to define and refine their own benchmarks.

In all our experiments the tools have been executed in their default configuration. Only the available memory has been changed according to our system's setup. Some tools may produce different results when executed with specific launch parameters. These results may be more accurate or less, computed faster or slower, and might thus change the outcome of our experiments. For example, FlowDroid has an option to activate the tracking of implicit flows. Consequently, its F-measure value would have been greater than 0 in category 15 (see Table 3). We restricted our experiments to the default configuration nonetheless because this is the one which a non-expert software developer is likely going to use.

Another threat to validity are the metrics precision, recall and F-measure that we and others frequently used to measure for accuracy. For some feature-restricted parts of e.g. DroidBench, these metrics are misleading. The Aliasing subset, for example, comprises four benchmark cases, three of which are negative cases. A tool capable of correctly answering these three negative cases will still have a precision, recall and F-measure value of 0 if it just fails the single positive case.

Finally, our implementation of the AQL-System may contain bugs. In particular, the converter used to translate tool-specific answers into AQL-Answers must work as intended in order to produce correct and meaningful results. We extensively tested the converter and fixed all errors. Due to the imprecise format of some tools' results, sources or sinks are sometimes (more specifically, for DidFail) not uniquely identifiable while converting it into the AQL format. Therefore, the converter over-approximates, i.e. all candidates are taken into account as source or sink respectively. Considering our experiments on real-world apps, Brew similarly over-approximated during the identification of sources and sinks in case of Amandroid, DidFail and DroidSafe. Thus, method calls are matched by method names without considering the parameters given as input. Such aspects have already been taken into consideration.

6 RELATED WORK

Providing an overview of analysis tools for Android apps is the topic of three recent surveys [18, 23, 25]. These works collect and summarize tools and their functionality as outlined in research papers. They provide no systematic experimentation to assess and compare tools, in particular not with respect to their promises.

A thorough comparison of Android analysis tools has so far been difficult due to the lack of precisely defined benchmarks. This situation is different in other (analysis) contexts. Competitions like the annual Competition on Software Verification (SV-COMP [5]), the SAT-solving competition[6] or the Hardware Model Checking Competition (HMCC [6]) provide well-defined benchmarks in different categories with precisely fixed outcomes. Often, they do not just require participating tools to give yes/no answers, but in addition to provide witnesses or proofs of their results. With the AQL, we already have a format for witnesses of taint flows available.

Finally, there are more tools which potentially fit in our scope. Apposcopy [11], WeChecker [9] and Separ [4] should fit perfectly, however are not publicly available. SCanDroid [19] is publicly available and fits into our scope as well, nonetheless it is largely outdated and cannot produce results for any considered (micro) benchmark. DroidInfer [14] employs an interesting type-based approach. However, in this particular case it requires a lot of effort to build a converter to extract the determined taint flows, because of its uncommon result structure. Additionally, the tool seemed not ready for competitive comparison since its execution fails for most micro-benchmark cases. Thus, we decided to omit the tool. Other available tools do not fit into our scope due to their result representation even though they inspect privacy leaks. For example, HornDroid [8] determines sinks and provably shows that these can be reached by taint flows. It fails, however, to name sources, which is why we cannot determine which specific flow is found.

7 CONCLUSION

In this paper, we reported on the results of a reproducibility study on static taint analysis tools for Android apps. To support our own as well as similar studies, we developed a framework for inferring data leaks in test apps and for automatically running tools on benchmark sets. With the help of this framework, we assembled precise benchmark suites and re-evaluated six existing tools on them. In the evaluation, we in particular studied the handling of specific features, the accuracy of tools and their relation to the promised values. The results indicate that studies and benchmarks like ours are indeed needed to provide a solid ground for a fair and unbiased comparison of tools.

ACKNOWLEDGMENTS

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

[5]https://developer.android.com/about/dashboards (01/03/2018)

[6]http://satcompetition.org

REFERENCES

[1] Maqsood Ahmad, Valerio Costamagna, Bruno Crispo, and Francesco Bergadano. 2017. TeICC: targeted execution of inter-component communications in Android. In *SAC, Marrakech, Morocco, 2017*, Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng (Eds.). ACM, 1747–1752.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI, Edinburgh, United Kingdom, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 259–269.

[3] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: analyzing the Android permission specification. In *CCS, Raleigh, USA, 2012*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM, 217–228.

[4] Hamid Bagheri, Alireza Sadeghi, Reyhaneh Jabbarvand Behrouz, and Sam Malek. 2016. Practical, Formal Synthesis and Automatic Enforcement of Security Policies for Android. In *DSN, Toulouse, France, 2016*. IEEE Computer Society, 514–525.

[5] Dirk Beyer. 2017. Software Verification with Validation of Results - (Report on SV-COMP 2017). In *TACAS (ETAPS), Uppsala, Sweden, 2017 (LNCS)*, Axel Legay and Tiziana Margaria (Eds.), Vol. 10206. 331–349.

[6] Armin Biere, Tom van Dijk, and Keijo Heljanko. 2017. Hardware model checking competition 2017. In *FMCAD, Vienna, Austria, 2017*, Daryl Stewart and Georg Weissenbacher (Eds.). IEEE, 9.

[7] Amiangshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. 2017. Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications. In *AsiaCCS, Abu Dhabi, United Arab Emirates, 2017*, Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi (Eds.). ACM, 71–85.

[8] Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. 2016. HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving. In *EuroS&P, Saarbrücken, Germany, 2016*. IEEE, 47–62.

[9] Xingmin Cui, Jingxuan Wang, Lucas Chi Kwong Hui, Zhongwei Xie, Tian Zeng, and Siu-Ming Yiu. 2015. WeChecker: efficient and precise detection of privilege escalation vulnerabilities in Android apps. In *WiSec, New York, USA, 2015*. ACM, 25:1–25:12.

[10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick D. McDaniel, and Anmol Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI, Vancouver, Canada, 2010*, Remzi H. Arpaci-Dusseau and Brad Chen (Eds.). USENIX Association, 393–407.

[11] Yu Feng, Isil Dillig, Saswat Anand, and Alex Aiken. 2014. Apposcopy: automated detection of Android malware (invited talk). In *DeMobile, Hong Kong, China, 2014*, Aharon Abadi, Rafael Prikladnicki, and Yael Dubinsky (Eds.). ACM, 13–14.

[12] Gartner. 2017. Gartner Says Worldwide Sales of Smartphones Grew 9 Percent in First Quarter of 2017. https://www.gartner.com/newsroom/id/3725117.

[13] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *NDSS, San Diego, USA, 2015*. The Internet Society.

[14] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and precise taint analysis for Android. In *ISSTA, Baltimore, USA, 2015*, Michal Young and Tao Xie (Eds.). ACM, 106–117.

[15] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. Android taint flow analysis for app sets. In *SOAP, Edinburgh, UK, 2014*, Steven Arzt and Raúl A. Santelices (Eds.). ACM, 5:1–5:6.

[16] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS)*. http://www.bodden.de/pubs/lblh11soot.pdf

[17] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick D. McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *ICSE, Florence, Italy, 2015*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 280–291.

[18] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information & Software Technology* 88 (2017), 67–95.

[19] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. 2009. SCanDroid: Automated security certification of Android applications. *Technical report, University of Maryland* (2009).

[20] Felix Pauck. 2017. *Cooperative static analysis of Android applications*. Master's thesis. Paderborn University, Germany.

[21] Lina Qiu, Yingying Wang, and Julia Rubin. 2018. Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *ISSTA, Amsterdam, Netherlands, 2018*.

[22] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *NDSS, San Diego, USA, 2014*. The Internet Society.

[23] Bradley Reaves, Jasmine 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. 2016. *droid: Assessment and Evaluation of Android Application Analysis Tools. *ACM Comput. Surv.* 49, 3 (2016), 55:1–55:30.

[24] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.

[25] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. 2017. A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software. *IEEE Trans. Software Eng.* 43, 6 (2017), 492–530.

[26] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *POPL, Austin, USA, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 17–30.

[27] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *CASCON, 1999, Mississauga, Canada*, Stephen A. MacKay and J. Howard Johnson (Eds.). IBM, 13.

[28] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *SIGSAC, Scottsdale, USA, 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, 1329–1341.

[29] Daojuan Zhang, Rui Wang, Zimin Lin, Dianjie Guo, and Xiaochun Cao. 2016. IacDroid: Preventing Inter-App Communication capability leaks in Android. In *ISCC, Messina, Italy, 2016*. IEEE Computer Society, 443–449.

[30] 2017. Amandroid. Retrieved 03/08/2018 from https://bintray.com/arguslab/maven/argus-saf/3.1.2

[31] 2017. ApkTool. Retrieved 03/08/2018 from https://ibotpeaches.github.io/Apktool/

[32] 2018. AQL-System. Retrieved 08/12/2018 from https://FoelliX.github.io/AQL-System

[33] 2018. BREW. Retrieved 08/12/2018 from https://FoelliX.github.io/BREW

[34] 2017. DIALDroid. Retrieved 03/08/2018 from https://github.com/dialdroid-android/DIALDroid

[35] 2015. DidFail. Retrieved 03/08/2018 from https://www.cert.org/secure-coding/tools/didfail.cfm

[36] 2016. DroidSafe. Retrieved 03/08/2018 from https://mit-pac.github.io/droidsafe-src/

[37] 2017. FlowDroid. Retrieved 03/08/2018 from https://github.com/secure-software-engineering/soot-infoflow-android/wiki

[38] 2016. IccTA. Retrieved 03/08/2018 from https://sites.google.com/site/icctawebpage/source-and-usage

[39] 2017. jadx. Retrieved 03/08/2018 from https://github.com/skylot/jadx

[40] 2018. ReproDroid. Retrieved 08/12/2018 from https://FoelliX.github.io/ReproDroid