Analyzing Android Taint Analysis Tools: FlowDroid, Amandroid, and DroidSafe

Junbin Zhang, Yingying Wang, Lina Qiu, and Julia Rubin

Abstract—Numerous static taint analysis techniques have recently been proposed for identifying information flows in mobile applications. These techniques are often optimized and evaluated on a set of synthetic benchmarks, which makes the comparison results difficult to generalize. Moreover, the techniques are commonly compared under different configuration setups, rendering the comparisons inaccurate. In this paper, we provide a large, controlled, and independent comparison of the three most prominent static taint analysis tools: FLOWDROID, AMANDROID, and DROIDSAFE. We align the configuration setup for the tools and evaluate them on both a set of common benchmarks and on real applications from the Google Play app store. We further evaluate the effectiveness of additional reflection handling mechanism implemented by DROIDRA, applying it to each of the evaluated tools. We compare the results of our analysis to the results reported in previous studies, identify main reasons for inaccuracy in existing tools, and provide suggestions for future research.

Index Terms—Static analysis, Taint analysis, Mobile applications, Empirical studies, Reproducibility studies.

1 INTRODUCTION

STATIC taint analysis is commonly used for identifying information flows in mobile applications and numerous tools for performing such analysis have being proposed [1]–[8]. The authors of these tools typically tune and evaluate them on a set of common benchmarks, such as *DroidBench* [9] and *ICC-Bench* [10]. Our experience, however, shows that the tools are often compared under different setups, making the comparison inaccurate. The tools are also mostly compared on a set of synthetic benchmarks, making it difficult to predict the performance of tools on real apps.

Some authors [2], [5], [11], [12] evaluate a particular tool on F-Droid [13] and Google Play [14] apps. Others [4], [15]–[18] compare the tools with each other but, as with benchmark apps, do not align the tool setups. Moreover, they mostly focus on reporting the number of *detected* flows and runtime failures, without considering the set of *expected flows*, analyzing false negatives, and reporting reasons leading to false positive and false negative results.

We address these limitations in our work, performing an independent, controlled, and large-scale study comparing the three most-popular open-source static taint analysis tools to each other: FLOWDROID, AMANDROID, and DROIDSAFE. We focus on comparing the performance of the tools in terms of accuracy, execution time, and memory consumption. We configure the tools to use a similar setup and we evaluate the tools on more than 180 benchmark applications and 25 applications from the Google Play store, for which we carefully collected the expected flows (82 flows in total). Furthermore, we augment each tool with a solution supporting better handling of reflective calls, DROIDRA [19], [20], and evaluate its effectiveness on both benchmark and Google Play apps. We compare the results of our study with the results in earlier reports, discuss the strengths and weaknesses of each tool, and the implications of our findings. As our team is not involved in the development of these tools, our analysis is completely independent. Moreover, we make the results of our analysis, the set of subject applications that we used, and our configuration setup available to the community, making our study replicable and reproducible.

Our work aims at answering the following research questions:

RQ1: How well do the tools perform on the benchmark applications?
RQ2: What are the main causes of inaccuracy for the benchmark applications?
RQ3: Do the results on the benchmark applications generalize to the Google Play applications?

To answer **RQ1**, we extracted the configuration options used in the empirical evaluation of FLOWDROID [2], [4], [21], AMANDROID [5], [11], and DROIDSAFE [6]. As most of these reports do not provide detailed information about the version of the tools they used for evaluation, the parameters used for configuring the tools, the list of the used sources and sinks, and the exact set of *DroidBench* and *ICC-Bench* apps used in the evaluation, we performed our own experiments, evaluating each tool with a common configuration setup we have chosen, on the same set of applications, and using the same set of sources and sinks.

We used the *DroidBench* and *ICC-Bench* benchmark suites. Our results show that on these benchmark suites, DROIDSAFE has the highest accuracy, FLOWDROID comes second, and AMANDROID has the lowest accuracy. DROIDRA, originally developed and evaluated with an older version of FLOWDROID in mind, does not help resolve any reflection-related cases for the newer version of FLOWDROID. Interestingly, it helps to improve the accuracy of AMANDROID and DROIDSAFE, albeit slightly.

The accuracy of the tools in our experiment often differs from the results reported in earlier studies. For example, the authors of AMANDROID report an F-measure of 81% [5] and 96% [11] when executed on the *DroidBench* benchmark suite;

• *J. Zhang, Y. Wang, L. Qiu, and J. Rubin are with the Department of Electrical and Computer Engineering, University of British Columbia, Canada.E-mail: {zjbthomas, wyingying, lqiu, mjulia}@ece.ubc.ca*

we were unable to replicate these results, most likely due to a different selection of benchmark apps and tool version, observing an F-measure of 61% for our experimental setup.

For **RQ2**, we manually analyzed the false positive (FP) and false negative (FN) results reported by each tool. In a nutshell, we observed that all tools have major issues in handling reflection, albeit for different reasons. DROIDRA helps resolve some of these issues for AMANDROID and DROIDSAFE but fails to handle complicated reflection-related constructs. In addition, FLOWDROID fails to accurately parse and track ICC Intents involving complex string analysis and list management operations; AMANDROID does not accurately handle Android framework methods, lifecycle and callback methods, and location-related flows.

For **RQ3**, we investigated whether FLOWDROID and AMANDROID are able to identify the flow in two real application scenarios: (a) from the login credentials entered by the user to an Internet transmission operation and (b) from user- or phone-specific sensitive information to an Internet transmission operation. We also experimented with applying DROIDRA. However, we excluded DROIDSAFE from this analysis as the tool authors explicitly state that the tool is not designed to run on Google Play apps [22] and our experiments concur with that statement.

The rationale behind selecting the login case study is that applications have to send user credentials out to the Internet to support the login functionality. For the sensitive information case study, we selected known spyware applications that leak private information to their servers. We use this information transmission as an expected flow in a real-world app and check how well the tools perform for identifying this flow. We also manually inspected all other flows the tools found in each application, identified FP results, and analyzed the reasons behind the tools' FPs and FNs.

We configured both tools to run for three days per each Google Play application and allocated 256 GB of memory for each run. Our results show that FLOWDROID crashes with exceptions in 11 out of 25 applications, mostly when analyzing certain ICC flows and reflective calls. AMANDROID is able to analyze 14 applications but reports a different number of flows in each run. For example, for one of the apps, it reports 0, 3, 28, and 43 flows in four identical runs. We thus could not perform a reliable analysis of false-positive and false-negative results for AMANDROID.

For the 14 applications that FLOWDROID successfully analyzed, it could only detect expected flows in two applications, having difficulties detecting certain callback methods and reflective calls, as Google Play apps rely on callback/reflection mechanisms not covered in benchmarks. DROIDRA does not help improve the accuracy of the tool for reflective calls. Moreover, while FLOWDROID's modeling of Android framework methods is accurate for all methods used in the benchmarks, it does not cover methods used in Google Play apps, leading to both false-positive and false-negative results.

Overall, our results suggest that despite the success on the benchmark apps, the tools cannot be used to reliably analyze flows in real applications. As the failures we observed were not explicitly covered by existing benchmarks, we created a sample app isolating each failure and added it to our own benchmark suite, called *UBCBench* [23]. We intend to contribute this suite to *DroidBench*; we are also working with the authors of FLOWDROID and AMANDROID to fix some of the failures (DROIDSAFE is no longer supported).

Contributions. This paper reports on an extension of our earlier study published at ISSTA'18 [24]. The ISSTA paper compared FLOWDROID, AMANDROID, and DROIDSAFE under the same setup on benchmark apps and reported on the results of our findings. The follow-up work described in this paper performs the same experiments using the latest versions of the tools and benchmark suites. It also includes a new experiment conducted on the Google Play apps, comparing the tools in a realistic use case.

The contributions this paper makes are described below:

- It provides an in-depth independent analysis that compares the available static taint analysis tools for Android applications under a similar setup, on both benchmark and Google Play apps, and puts the results in the context of reports from earlier studies.
- It identifies the main strengths and weaknesses of each tool.
- It provides detailed information about the chosen configuration setup, selected sources and sinks, and applications used for analysis, making the results replicable and reproducible.
- It extends the benchmarks used for comparing Android-specific static taint analysis tools with previously uncovered cases.
- It contributes 25 Google Play apps with manually established expected flows.
- It makes our experimental setup and results publicly available, to support replication and reproducibility [23].

The remainder of this paper is structured as follows: Section 2 provides the necessary background on taint analysis. Section 3 outlines our study design, including the selection and configuration of the tools and subject apps, our definition of expected results, and metrics we used for tool evaluations. Section 4 presents the results of our analysis for the benchmark apps and Section 5 – for the Google Play apps. We outline the limitations of our approach and threats to the experiment validity in Section 6 and discuss the lessons learned in Section 7. We finalize the paper with the discussion of related work in Section 8 and conclusions in Section 9.

2 BACKGROUND: TAINT ANALYSIS

We now introduce the basic concepts of static taint analysis – a popular information flow analysis technique which tracks the flow of sensitive information from a set of sensitive sources to sensitive sinks. In our context, sources define the information we want to protect on a mobile device (e.g., phone number, contacts, location, and unique device identifiers) and sinks define points of unwanted information release (e.g., methods related to the Internet and SMS transmission). If data from a sensitive source reaches a sink, taint tracking identifies the path from the source to the sink as an instance of data leakage.

Taint analysis can be implemented both statically and dynamically; this paper focuses on static taint analysis techniques, i.e., those that track taint propagation by analyzing the code of an Android application without ever running it. Examples of tools implementing static taint analysis

```
1   class A{
2      void sink(String){...}
3   }
4   void foo() {
5      A a1 = new A(); a1.sink("tainted");
6   }
7   void bar() {
8      A a2 = new A(); a2.sink("untainted");
9   }
```

Fig. 1: Context and object sensitivity.

include FLOWDROID, AMANDROID, and DROIDSAFE. The tools mostly differ in design decisions they take for making the analysis accurate and scalable at the same time. Below, we briefly discuss four main dimensions for such decisions.

1. Sensitivities. To handle aliasing and virtual dispatch constructs, typical static analysis for Java programs applies some degree of context, object, and field sensitivity; flow and path sensitivities are used to control the order of statements and their correspondence to branches of the program:

A *context-sensitive* analysis considers the calling context, i.e., a sequence of call sites, when analyzing the target of a method call. Specifically, in a k-call-site-sensitive analysis, the context of a called method includes the current call site of the method and the call sites of the caller methods, up to a pre-defined depth k [25]. For the example in Fig. 1, `foo()` and `bar()` are two different call sites for the method call `sink()`, which are used by a context-sensitive analysis to differentiate the flows to this method.

An *object-sensitive* analysis uses object abstractions, i.e., allocation sites, as context [25]. Specifically, the analysis qualifies the method local variables with the allocation site of the receiver object of the method call. For the example in Fig. 1, object sensitive analysis uses the heap addresses of objects `a1` and `a2` to differentiate the calls to `sink()`. That is, context- and object-sensitive analyses use different program elements – call sites vs. allocation sites – as differentiating contextual information.

A *field-sensitive* analysis distinguishes different fields of the same abstract object, instead of lumping all fields together [25], [26].

A *flow-sensitive* analysis takes the order of statements into account [25]. For example, for a list of statements `x=1; y=x+1; x=2`, a flow-sensitive analysis will be able to determine that `y=2`, whereas a flow-insensitive analysis will conclude that `y=2` or `y=3`.

A *path-sensitive* analysis collects path information which indicates the feasibility of a path. For instance, for a branch condition `x > 0`, the analysis would assume `x > 0` on the target of the branch and `x <= 0` on the fall-through path.

2. Implicit flows. Implicit flows are flows in which sensitive data indirectly impacts the observed output by affecting which branch to take in the control flow [27]. A typical example is: `if taint then output = 1 else output = 0`, where `taint` would affect which branch to take, then further affect the value of the variable `output`. Conceptually, an implicit flow tracker taints all data items that are dependent on the taint from the conditional, hence can result in large numbers of false-positives.

3. Java-specific features. As Android applications are generally written in Java, analysis tools need to handle Java-specific features, such as reflection and exceptions. *Reflection* refers to the ability to dynamically access members and type information of an object, often based on string representations of the member's or type's name. Android developers heavily rely on reflection, e.g., for backward compatibility, generality, and sometimes for hiding sensitive information flows [28]. Accurately resolving reflective calls poses a challenge to the soundness of static analysis. To address this problem, some tools consider all possible resolutions for a reflective call. Others restrict the resolution by type of the variable, its scope, etc. [29].

Exceptions can be thrown by a statement or expression, and then read by the `catch` code block either within the method or up in the call stack. As exceptions can be loaded dynamically and the type of exception determines the code block to execute, precisely handling exceptions is challenging [30]. Some classic frameworks like SPARK [31] and PADDLE [32] use an over-approximation approach to handle exceptions, assigning all exceptions thrown in the program to a single global variable; the variable is then read at the exception catch site. That is, this approach assumes that all possible exceptions are thrown and ignores the information about what exceptions can propagate to a catch site [33]. Other frameworks, such as SOOT [34], remove unrealizable exception edges from the intraprocedural control flow graphs [30].

4. Android-specific features. Even though Android applications are often written in Java, several Android-specific features should be taken into account to achieve accurate results.

Android modeling: To track taint when an Android application interacts with the Android execution environment, tools typically either (a) conservatively assume that the return value of all Android framework methods is tainted if any of the parameters is tainted or (b) precisely model (a subset of) the framework methods. The latter is performed either manually or by automated analysis of Android binary distribution libraries [21].

Application lifecycle: An Android application is composed of four types of components, namely, Activity, Service, Broadcast Receiver, and Content Provider. Each component has its own lifecycle methods, which are called by the Android system to start/stop/resume the component. An application also contains callback methods which are triggered in response to system and user events (e.g., location change and button click). Static analysis tools identify and model lifecycle and callback methods to ensure the correct propagation of taint. To extract the set of such methods, the tools rely on information from the Android Open Source Project and also analyze application code and configuration files, such as manifest and layout XML.

Inter-component communication (ICC): The multi-component model of Android allows different components to communicate with each other and exchange information, usually via *explicit* or *implicit* Intents. The former explicitly define target components to be invoked while the latter rely on Android to find the components which implement the requested functionality. ICC handling is a challenge due to the difficulty to precisely match the invoked components in a static manner [35]. Tools like EPICC [1] and IC3 [36] extract ICC information, enabling ICC-aware information flow analysis. PRIMO [37] overlays a probabilistic model of

ICC on top of these static analysis results, further improving the accuracy of ICC detection.

Inter-app communication (IAC): Similar to ICC, IAC looks into how sensitive data is transferred between components of different applications installed on the same device. Tools such as APKCOMBINER [38] aim at reducing an IAC problem to an ICC problem by combining different applications into a single APK on which existing tools can perform inter-app analysis.

Native code: Android applications can include code written in C and C++ and triggered via Java Native Interface (JNI) [39]. As taint can propagate via such native C and C++ code, a static taint analysis tool needs to be able to track data flow in C or C++ code. AMANDROID is an example of a tool that supports such tracking by relying on the inter-language JN-SAF [40] framework for "stitching" data flows found in Java and native code.

3 STUDY DESIGN

In this section, we describe our method for selecting and configuring the static taint analysis tools for the study (Sections 3.1 and 3.2). We then outline the methodology we used for evaluating the tools, for benchmarks and Google Play apps, separately (Sections 3.3 and 3.4), describing how we selected the subject applications, the set of sources and sinks that we used, and our specification of expected results.

3.1 Tool Selection

Li et al. [41] performed a systematic literature review and identified 38 static taint analysis tools for Android applications. From this list, in our prior work [24], we selected all open-source tools cited more than 100 times on Google Scholar [42] as of June 2017. This paper extends our prior work by analyzing large Google Play applications, in addition to considering newer versions of the benchmark suites. To provide up-to-date results, we opted to use more recent versions of the tools, i.e., those that were available in January 2020. For example, FLOWDROID was integrated with STUBDROID [43] since the earlier version we used, as discussed later in this section.

We further scanned the list of all publicly available taint analysis tools as of December 2020. To this end, we combined the results of the survey by Li et al. [41] with our own systematic literature review covering the timeframe following the survey: January 2016 and December 2020. We identified 181 relevant papers in total, which were independently reviewed by two of the authors. We excluded 17 surveys and comparative studies, like our own one [24]; 78 dynamic or hybrid taint analysis techniques, which are not the focus of this study; 41 papers that utilize existing taint analysis techniques but do not offer new approaches, e.g., [44]; and 4 papers that present extended version of tools that were already in our dataset [45]–[48].

Out of the remaining 41 papers, 15 are designed specifically for IAC, 2 for JavaScript, 5 for native code, and 1 for implicit flow analysis, none of which is the focus of our study. This leaves 18 relevant publications and we contacted the authors of each to inquiry about the availability of the tool and its source code – an artifact we need to investigate reasons for inaccuracies. We identified only four such tools: DROIDRA [19], [20], HORNDROID [49], Ripple [50], and DAPA [51]. Discussions with the authors of HORNDROID [49], Ripple [50], and DAPA [51] confirmed that these tools are no longer maintained and cannot work on our selected applications without significant implementation effort. We thus only included DROIDRA in our study. A detailed discussion about the availability and applicability of all relevant tools is available online [23].

TABLE 1: Selected Tools

Tool	# Citation (Jun. 2017)	# Citation (Dec. 2020)	Version
FLOWDROID	547	1765	2019-Jan-21 (2.7.1)
ICCTA	129	537	
AMANDROID	148	395	2018-Dec-28 (3.2.0)
DROIDSAFE	100	408	2016-Jun-22
DROIDRA	-	117	2021-Feb-3

The names, citation counts, and release dates of the tools used in our experiments – FLOWDROID, ICCTA, AMANDROID, DROIDSAFE, and DROIDRA – are listed in Table 1. Starting from release 2.0, FLOWDROID is integrated with ICCTA and leverages it for processing ICC flows. We thus performed our experiments with the combined version. Experimental data for running FLOWDROID without ICCTA, as well as experiments with older versions of these tools, are available in our online appendix [23]. Next, we briefly describe each of the analyzed tools.

FLOWDROID [2] is a flow-, context-, field-, and object-sensitive static analysis tool for Android applications, which is built on top of SOOT [34] and DEXPLER [52]. It precisely models the Android lifecycle and handles data propagation via callbacks of UI objects. FLOWDROID can only resolve reflective calls whose arguments are constant strings; it bases its exception handling mechanism on that of SOOT. In the original design, the tool did not support implicit flow tracking [2], but newer versions of FLOWDROID are able to handle implicit flows [21].

For modeling Android framework methods, FLOWDROID relies on STUBDROID [43] which, given a binary distribution of the Android framework, performs taint analysis for the most common framework method and computes method summary that captures the propagation of taint within the method. For methods that cannot be analyzed with STUBDROID, e.g., those containing native code, FLOWDROID applies a conservative strategy, dividing the framework methods into four types: *generation*, *exclude*, *kill*, and *default* [21]. When method parameters (including the receiver object itself) or their fields are tainted, the methods in the *generation* type will have their receiver and the return value, as well as all their fields, tainted; no taint will be propagated for methods of the *exclude* type; all taints will be removed for methods of the *kill* type. If a method is not assigned with any of these types, the *default* rule will apply, propagating the taint from the receiver object and its fields to the method return value and its fields. That is, only for the methods in the *generation* type FLOWDROID will propagate the taint from method parameters and will taint the receiver parameter and its fields; it does not taint any other method parameters in any of the conservative strategies.

IccTA [4] extends FlowDroid with the analysis of inter-component communication. It leverages existing ICC extraction tools, specifically, Epicc [1] and IC3 [36], augmenting them with application-level instrumentation of ICC-related methods for extracting precise ICC flows. It also leverages ApkCombiner [38], which generates an integrated APK for two or more applications. As a result, IccTA builds a complete intra-component, inter-component, and inter-application model. In our experiments, we used FlowDroid version 2.7.1 from January 2019 [53], which is integrated with IccTA and is the latest version of the tool at the time of writing.

Amandroid [5], [11] implements a flow- and context-sensitive intra-component data-flow analysis. On top of an inter-procedural control-flow graph and data-flow graph, Amandroid builds a data-dependency graph for each component and then generates a summary table documenting possible component communication connections. Amandroid precisely models a subset of Android framework methods and applies a conservative strategy for the remaining ones. Interestingly, Amandroid's strategy is different from that of FlowDroid: in addition to the receiver object and the return value of a method, Amandroid also taints all method parameters and their fields. However, it is doing so only when the method receiver or parameters themselves are tainted, not when one of their fields is tainted. Strategies applied by both tools lead to false positive and false negative results, albeit in different cases, as we show in our evaluation.

Based on proprietary inter-component and inter-application flow analysis, Amandroid provides support for both ICC and IAC detection. However, Amandroid has limited capacity to handle exceptions and reflection, and it cannot handle implicit flows [5], [11]. In our study, we used version 3.2.0 of Amandroid released on December 29, 2018 [54]. This is the latest version available today. Yet, it can only handle applications targeting API level 25 and below.

DroidSafe [6] implements an object-sensitive and flow-insensitive analysis. It builds a comprehensive Android execution model that contains analysis stubs for most of the Android framework methods. This allows the tool to precisely track flows through Android APIs. Yet, it limits the analysis to a particular Android version. For reflection, DroidSafe uses string analysis to replace reflective calls with direct calls to the target method, when possible. Yet, the tool does not have fully-sound handling of reflection. It uses a proprietary model to handle ICC and IAC flows. Implicit flows were not supported in the original version of the tool [6], but the latest version of the source code contains an option to enable implicit flows. The DroidSafe authors officially stopped supporting the tool since June 22, 2016. Therefore, in our study, we considered the latest version that was available as of June 2016 [22], which supports API level 19 and below.

DroidRA [19] extends state-of-the-art static analysis tools for Android by providing a more advanced reflection handling mechanism. The main idea behind the tool is to use an inter-procedural, context-sensitive, and flow-sensitive static analysis to generate constraints which are further passed to a constraint solver to determine the targets of reflective calls. The tool then instruments the input application, augmenting each reflective call it can successfully resolve with a direct Java call to the corresponding target. In our experiments, we run all taint analysis tools, i.e., FlowDroid, Amandroid, and DroidSafe, with and without DroidRA. In the latter case, we feed the application instrumented by DroidRA into each taint analysis tool and compare the results with those of the original version. We used the latest version of DroidRA, which was made available by the authors in February 2021, after fixing several bugs discovered by our preliminary analysis.

3.2 Tool Configuration Parameters

Each taint analysis tool provides numerous configuration parameters: FlowDroid has 44 parameters, Amandroid has 11, and DroidSafe has 57; DroidRA is not configurable. Because the results of the analysis largely depend on the tuning of each tool, comparing taint analysis tools under different configuration setup is not meaningful.

We investigated the tool documentation and configuration setup used in previous studies [4]–[6], [11], to align the tools along the main configuration dimensions. We observed that most of the studies do not document the selected configuration parameters, making the comparison inaccurate and irreproducible. We also observed that for some tools, a number of important design decisions are hard-coded and cannot be configured at all, e.g., object-sensitivity of FlowDroid, and that some configuration choices are not documented, e.g., field sensitivity of DroidSafe and Amandroid.

In our attempt to align the tools around the same parameters, we created and ran tests to identify the design choice implemented by each tool and align different tools to apply the same decisions, when possible. Our final set of configuration choices is described below.

1. Sensitivities. The first five rows of Table 2 list five types of sensitivity-related configurations discussed in Section 2, namely, field, context, object, flow, and path sensitivities; we document our decisions in the second column of Table 2. The remaining columns of the table describe the default sensitivity choice implemented by each tool and whether this choice can be configured by the user.

As some tools do not provide an option to enable/disable certain sensitivity types, we had to pick their default, arriving at field-sensitive, object-sensitive, and path-insensitive analysis. Flow sensitivity (row 4) can only be configured in FlowDroid; Amandroid supports flow sensitivity and DroidSafe does not. As such, we were unable to fully align the tools for this configuration option. Moreover, DroidSafe is context-sensitive for static methods only (row 2). We still opted for enabling context sensitivity, even though other tools provide context sensitivity for all rather than only static methods, as this option is central for obtaining accurate analysis results. Both FlowDroid and Amandroid allow to set a context sensitivity depth, but use different default values: five and one, respectively. We followed the default setup of FlowDroid and set the context sensitivity depth to five, to obtain more accurate results; DroidSafe does not allow to set this parameter.

2. Implicit flows. FlowDroid and DroidSafe support implicit flow tracking; this option is disabled by default in both tools. Amandroid does not document whether it

TABLE 2: Configuration Decisions of FLOWDROID, AMANDROID, and DROIDSAFE

Configuration		Selected	FLOWDROID		AMANDROID		DROIDSAFE	
			Default	Configurable?	Default	Configurable?	Default	Configurable?
Sensitivity	Field	✓	✓	yes	✓	no	✓	no
	Context	✓	✓	yes	✓	yes	✓[a]	yes
	Object	✓	✓	no	✓	no	✓	yes
	Flow	✓	✓	yes	✓	no	✗	no
	Path	✗	✗	no	✗	no	✗	no
Implicit flows		✗	✗	yes	✗	no	✗	yes
Java-specific	Reflection	✓	✗	yes	✓	no	✓	no
	Exception	✓	✓	yes	✓	no	✓	yes
Android-specific	ICC	✓	✗	yes	✓	no	✓	no
	IAC	✗	✗	yes[b]	✗	yes	✗	yes
	Native code	✗	✗	no	✓	yes	✗	no
	UI elements detection	✗	✓	yes	✓	no	✗	no

[a] DROIDSAFE's context sensitivity is for static methods only.
[b] FLOWDROID requires APKCOMBINER for IAC analysis.

supports implicit flow tracking or not. In our communication with the AMANDROID authors, they confirmed that the tool cannot handle implicit flows. Therefore, we disabled implicit flow tracking for all tools (row 6).

3. Java-specific features. FLOWDROID, AMANDROID, and DROIDSAFE all explicitly report that they do not have a fully-sound handling of reflections. Moreover, support for resolving reflective calls differs among tools. Yet, handling reflection is enabled in AMANDROID and DROIDSAFE by default; we thus enabled this option in FLOWDROID as well (row 7).

For FLOWDROID and DROIDSAFE, exception handling is enabled by default, though it can be disabled. AMANDROID reports on a limited capability to handle exceptions and provides no configuration parameter regarding exception handling. As exception tracking is an important property of static analysis and all tools enabled this option by default, we proceed with that choice (row 8).

4. Android-specific features. As discussed in Section 3.1, FLOWDROID allows to use STUBDROID to precisely model a subset of Android framework methods and use a conservative strategy to cover the remaining ones. We opted for using this strategy. Android modeling is enabled by default and is not configurable for AMANDROID and DROIDSAFE.

We configured FLOWDROID to use the IC3 model for ICC flow extraction: according to earlier studies, this model has higher accuracy than EPICC [36]. We did not use PRIMO because it relies on the statistical similarity between the analyzed applications, which is absent in the benchmark suites. In addition, by default, the FLOWDROID version that we used applies "purification" of ICC-related flows. This means that the tool deviates from the standard taint analysis semantics and implements additional logic when handling sinks involved in ICC communication, such as `startActivity()`. After running a number of experiments, we confirmed that the purification functionality is not fully implemented and aligned with its documented behavior. In our communication with the FLOWDROID authors, they advised us to disable this functionality using `noiccresultspurify` parameter, making the tool adhere to the standard taint flow semantics[1]. We thus report FLOWDROID results for both switching the ICC purification on (the default option in the version that we analyzed) and off.

AMANDROID and DROIDSAFE also implement (different) proprietary ICC handling mechanisms and do not provide any configuration options. We relied on the default behavior of these tools for our study (row 9). Yet, to make the results reported by different tool comparable with each other, we also recalculate the accuracy of these tools using the standard taint flow semantics, as described in Section 4.1.1.

FLOWDROID can detect inter-application flows when augmented with APKCOMBINER [38]. AMANDROID and DROIDSAFE can also detect IAC flows, but that option is disabled by default. As the benchmark suite we used contains only two IAC cases and we did not consider IAC in our Google Play experiment, we disabled IAC tracking in all tools (row 10).

AMANDROID is the only tool that supports native code analysis. For consistency with the other tools, we disabled this option in AMANDROID (row 11). Finally, both FLOWDROID and AMANDROID are able to detect flows from sensitive UI elements, such as password fields. This option is configurable in FLOWDROID but not in AMANDROID. However, DROIDSAFE does not implement the corresponding feature. We thus disabled sensitive UI detection in FLOWDROID (row 12) and filtered out all flows from sensitive UI elements reported by AMANDROID in our analysis.

5. Other configuration parameters. Upon reviewing additional configuration parameters, we observed that most of the remaining FLOWDROID and DROIDSAFE parameters deal with disabling advanced analysis functionality (which we decided to keep) and provide supportive functions, such as an option to generate *.json* reports in DROIDSAFE. The remaining AMANDROID options control input location in the file system, debug choices, and other parameters that do not affect the core flow detection functionality. We left the default values for all these parameters.

3.3 Evaluation Methodology for Benchmark Apps

We now discuss our selection of benchmark applications, taint sources and sinks we used in our analysis, our definition of expected results, and study metrics and measures.

1. This functionality was removed altogether in subsequent versions of FLOWDROID

3.3.1 Subjects

DroidBench and *ICC-Bench* are the most commonly used benchmarks for comparing the tools [2], [4]–[6], [11], [15], [16]. These benchmarks are publicly available; the FLOWDROID, ICCTA, and DROIDSAFE teams contributed to the *DroidBench* test suite, and AMANDROID team created *ICC-Bench*. As all teams participated in the creation of these benchmarks and also used them for evaluating their tools, this selection does not unintentionally benefit any of the tools.

We chose the latest version of the *DroidBench* benchmark suite, version 3.0 [9], which covers numerous categories of Android analysis problems, including intra-component and inter-component communication, handling of reflection, sensitivity types and more. We used 158 apps from this suite, excluding 28 apps that focus on testing configuration options that we disabled, i.e., tracking of implicit flows, inter-application flow detection, native code analysis, and sensitive UI elements detection. In addition, we excluded three applications related to dynamic class loading, as all three tools explicitly state that they cannot handle that scenario. The exact list of the excluded benchmarks is found in our online appendix [23].

The *ICC-Bench* test suite containing 24 benchmarks was originally developed by the AMANDROID team to test ICC-related capabilities of taint analysis tools. We used the latest version of this suite at the time of writing: version 2.0 [10]. *ICC-Bench* applications use Android API level 25. As DROIDSAFE only supports API level up to 19, we did not run the tool on *ICC-Bench*. All *DroidBench* applications are compatible with the API level 19 and thus DROIDSAFE, as well as other tools, runs on these applications successfully.

Our benchmark test suite thus consists of 182 applications in total from *DroidBench* (158) and *ICC-Bench* (24), covering 13 *DroidBench* categories and four *ICC-Bench* categories. The full list of the applications we analyzed is available online [23].

3.3.2 Sources and Sinks

In earlier work, Arzt et al. [2] and Li et al. [4] used the list of sources and sinks generated by the SUSI project [55]; the authors of AMANDROID confirmed that they used the sources and sinks marked in each of the benchmarks [11]. The list of sources and sinks in DROIDSAFE is not configurable without modifying the source code of the tool; Gordon et al. [6] thus ran their experiments with all sources and sinks hard-coded in the tool (4,051 sources and 2,116 sinks in total).

To use the same sources and sinks for all the tools, we inspected the headers and comments of all 182 benchmark applications and extracted the sources and sinks used in these applications (5 sources and 11 sinks listed in Table 3). We named this list of sources and sinks *SS-Bench*. We configured both FLOWDROID and AMANDROID to use *SS-Bench*. For DROIDSAFE, we confirmed that the sources and sinks from this list are the subset of sources and sinks considered by the tool. Furthermore, for a fair comparison between the tools, we ignored flows related to other sources and sinks, if such flows were reported by DROIDSAFE.

3.3.3 Expected Results

With our selection of sources, sinks, and configuration parameters discussed earlier, the expected flows for each benchmark application might deviate from the result specified by the benchmark designers, e.g., because a source extracted from one benchmark application could affect another application. We thus manually analyzed each of the benchmark applications, extracting all flows expected under our configuration setup. To ensure validity, the manual analysis was performed independently and in parallel by two authors of this paper. The disagreements in manual analysis (for seven out of 182 cases, a disagreement rate of 4%) were discussed in a meeting with all the authors, towards reaching a common resolution.

TABLE 3: List of Source and Sink Methods for Benchmark Apps (*SS-Bench*)

	Signature
Sources	android.telephony.TelephonyManager: java.lang.String getDeviceId()
	android.telephony.TelephonyManager: java.lang.String getSimSerialNumber()
	android.location.Location: double getLatitude()
	android.location.Location: double getLongitude()
	android.telephony.TelephonyManager: java.lang.String getSubscriberId()
Sinks	android.telephony.SmsManager: void sendTextMessage(java.lang.String,java.lang.String,java.lang.String, android.app.PendingIntent,android.app.PendingIntent)
	android.util.Log: int i(java.lang.String,java.lang.String)
	android.util.Log: int e(java.lang.String,java.lang.String)
	android.util.Log: int v(java.lang.String,java.lang.String)
	android.util.Log: int d(java.lang.String,java.lang.String)
	java.lang.ProcessBuilder: java.lang.Process start()
	android.app.Activity: void startActivityForResult(android.content.Intent,int)
	android.app.Activity: void setResult(int,android.content.Intent)
	android.app.Activity: void startActivity(android.content.Intent)
	java.net.URL: java.net.URLConnection openConnection()
	android.content.ContextWrapper: void sendBroadcast(android.content.Intent)

A full list of all expected flows is available online [23]. In 33 cases listed in Table 4, our expected results deviated from those documented in the benchmark itself. For example, in benchmark *ActivityCommunication2*, one additional flow is expected because of an additional sink, `startActivity(Intent)`, which was added as it appeared in multiple benchmarks, e.g., *DroidBench IntentSink2*; in benchmark *IMEI1*, no flows are expected because we disabled implicit flow tracking for all tools. It should be noted that, as discussed in Section 3.3.1, we excluded from our analysis benchmarks that were solely designed to check the features we disabled, e.g., *ImplicitFlow1-6* that examine implicit flows.

3.3.4 Metrics and Measures

Similar to the process of establishing the expected flows for our experiments, two authors of this paper manually inspected the flows identified by each tool, comparing them to the expected results. The goal of this analysis was to identify expected flows detected correctly by a tool: true positives (TP); unexpected flows mistakenly identified by the tool: false-positives (FP); and expected flows missed by a tool: false-negatives (FN). Similar to the process of identifying the

TABLE 4: Differences in Expected Results for *DroidBench* and *ICC-Bench*

Category	Benchmark	# Exp. Flows	
		Orig.	Ours
DroidBench			
1. Aliasing	Merge1	0	1
	StrongUpdate1	1	0
2. Android-specific	ApplicationModeling1	1	0
	PrivateDataLeak3	2	1
4. Callbacks	LocationLeak3	1	2
5. Emulator Detection	IMEI1	2	0
8. Inter Component Communication	ActivityCommunication2	1	2
	ActivityCommunication3	1	2
	ActivityCommunication4	1	2
	ActivityCommunication5	1	2
	ActivityCommunication6	1	2
	ActivityCommunication7	1	2
	ActivityCommunication8	1	2
	BroadcastTaintAndLeak1	1	2
	ComponentNotInManifest1	0	1
	IntentSource1	2	0
	UnresolvableIntent1	2	3
9. Lifecycle	ActivityEventSequence2	1	0
11. Reflection ICC	ActivityCommunication2	1	2
	AllReflection	1	2
	OnlyIntent	1	2
	OnlyIntentReceive	1	2
	OnlySMS	1	2
	OnlyTelephony	1	2
	OnlyTelephony_Dynamic	2	3
	OnlyTelephony_Reverse	1	2
	OnlyTelephony_Substring	1	2
	SharedPreferences1	1	2
ICC-Bench			
1. ICC Handling	icc_explicit_src_nosink	0	1
	icc_explicit_src_sink	1	2
	icc_stateful	3	2
	icc_explicit1	1	2
3. Mixed	icc_rpc_comprehensive	3	2

expected flows, the differences in the results of this manual analysis (for six out of 182 cases, a disagreement rate of 3%) were resolved in a discussion involving all authors of this paper and, when required, the authors of the tools.

We then calculated the precision, recall, and F-measure for each tool, as described below.

- *Precision:* the fraction of correctly reported flows out of the total number of reported flows, calculated as $\frac{TP}{TP+FP} \times 100\%$.
- *Recall:* the fraction of correctly reported flows out of the total number of expected flows, calculated as $\frac{TP}{TP+FN} \times 100\%$.
- *F-measure:* the weighted harmonic mean of the *Precision* and *Recall*, calculated as $\frac{2 \times Precision \times Recall}{Precision + Recall}$.

We also measured the execution time and memory consumption of each tool and report the averaged results from five consecutive runs. We ran all experiments on a Ubuntu 16.04 server with a 12-core CPU and limited the RAM allocation for each experiment to 64 GB. We set a two-hour timeout for each benchmark app.

3.4 Evaluation Methodology for Google Play Apps

Identifying expected flows in large and obfuscated Google Play applications, especially when the set of sources and sinks is unknown in advance, is a challenging task. Yet, identifying such flows is necessary to assess the usefulness of the existing techniques in practice. Furthermore, while most existing studies focus on analyzing the scalability of the tools on Google Play samples or analyzing flows detected by at least one existing tool [2], [4], [5], [11], "ground truth" of expected flows is essential for identifying cases missed by all tools. We thus decided to collect the expected flows in Google Play apps manually. To ensure we can perform this task reliably, we focused on two taint analysis application scenarios which contain "hints" helping us detect flows embedded in apps.

For the first scenario, we considered applications that perform user authentication. The goal of taint analysis in this case is to track the flow of the input user credential into internet-related sinks, e.g., to ensure that all passwords are sent over the network encrypted [56]. We selected this case study because to correctly perform user authentication, an app has to send out the credentials to a third-party server, which guarantees the presence of the expected flow. The goal of this case study is thus to check whether the tools can detect these flows and, if not, why not.

For the second case study, we selected Android spyware applications that leak user-sensitive information [57]. The goal of taint analysis in this case is to identify flows between this sensitive information and internet-related sinks – the classic use case of taint analysis. To aid identification of expected flows, we chose spyware samples described in blogs of security analysis companies, focusing on blogs that provide details about the type of the leaked information, which we used to identify sources of sensitive flows.

In the rest of this section, we provide more details about our criteria for selecting subject applications and our choice of sources, sinks, and expected results. We also discuss the metrics we applied and the measures that we performed. For the tool selection, as discussed in Section 1, DROIDSAFE's authors explicitly state that the tool is not designed to work on Google Play application: *"Please do not expect to run DroidSafe on large apps from the Google Play store, and expect DroidSafe to complete and/or give you accurate results."* Moreover, the tool only supports apps with API level up to 19, which are not common at the time of writing. We thus exclude this tool, focusing our analysis of Google Play apps on FLOWDROID and AMANDROID only, with and without DROIDRA.

3.4.1 Subjects

Login scenario. Rows 1-19 in Table 5 outline the login-related apps we selected for our study, while Figure 2 shows an overview of our selection process. We started from top 100 apps from each of the 58 free app categories in the Canadian Google Play Store as of December 2019, arriving at 5,569 apps (some categories contain less than 100 apps). We used Android Asset Packaging Tool (AAPT) [58] to extract the API level (i.e., *targetSdkVersion*) of each app and filtered out apps of API levels that are not supported by the tools. Specifically, as AMANDROID only supports API levels up to 25, we selected apps with API level 25 or below when comparing the tools. This selection resulted in 188 apps.

Next, two authors of this paper used one device each and executed every app for around 20 minutes, to check whether the app contains any authentication functionality

TABLE 5: List of the Selected Google Play Apps, API Version, and the Number of Expected Flows

App ID	App Name	Size (MB)	API Level	# Exp. Flows
1	com.echangecadeaux	3.0	14	1
2	com.rtp.livepass.android	6.4	17	1
3	com.tripadvisor.tripadvisor	5.3	19	1
4	ca.intact.mydrivingdiscount	8.8	21	1
5	com.asiandate	8.4		1
6	ca.passportparking.mobile.passportcanada	14.0	22	1
7	com.aldiko.android	9.5		1
8	com.passportparking.mobile.parkvictoria	14.7		1
9	com.passportparking.mobile.toronto	14.1		1
10	tc.tc.scsm.phonegap	10.0		1
11	com.onetapsolutions.morneau.activity	10.2	23	1
12	net.fieldwire.app	9.8		1
13	com.ackroo.mrgas	5.2	24	15
14	com.airbnb.android	65.4	25	1
15	com.bose.gd.events	15.1		1
16	com.phonehalo.itemtracker	16.7		3
17	com.viagogo.consumer.viagogo.playstore	13.4		1
18	com.yelp.android	20.5		1
19	onxmaps.hunt	11.2		1
20	com.mobistartapp.flashlight	4.4	21	7
21	com.monitor.phone.s0ft.phonemonitor	0.7		12
22	com.mobistartapp.win7imulator	3.9	23	3
23	com.mobistartapp.windows7launcher	4.6	25	11
24	com.tassaly.flappybird	5.7		7
25	ma.coderoute.hzpermispro	5.2		7

and, if so, to validate that the authentication functionality is working. After cross-validating the results, we confirmed reliable authentication mechanism in 51 of the apps. There were no disagreements between the authors in this case.

We then analyzed the bytecode of the identified 51 applications to ensure the flow between username / password and internet-related sinks is fully implemented within the code of the applications the tools are able to analyze. That excluded (1) 7 apps that implement login functionality using WebView – a widget that allows user to input login credentials in a web page; (2) 5 apps built using cross-platform frameworks such as Unity [59] and React Native [60]; and (3) 16 apps that delegate the login functionality to third-party OAuth service providers, such as Google or Facebook, which implement it in a separate application or in a WebView. In all these cases, the expected flows are external to the analyzed apps. We further excluded three apps that were highly obfuscated and we could not identify the authentication type and flow, and one app that was highly similar to a more recent app in our dataset.

Spyware scenario. For this scenario, we crawled malware-related posts from the three mobile security companies consistently ranked on top during the last two years by Gartner's Magic Quadrant for Best Endpoint Security Platforms [61]: Trend Micro [62], Symantec [63], and Sophos [64]. We focused on posts in the last two years, between January 2019 and December 2020, and used the search terms ("Android" | "Google" | "Playstore" | "Play" | "Store") and ("malware" | "malicious" | "malice") and ("spyware" | "stalkerware") to identify relevant posts. This resulted in 44 posts.

However, such automated filtering is rather coarse-grained and the identified posts often contain advertisement, description of non-Android malware, description of non-Java, i.e., ReactNative, malware, etc. Two authors of the paper thus further read the posts and identified four that fit our desired criteria: they describe Android-native spyware from the Google Play market and contain sufficient details about the secrets stolen by malicious apps. The authors analyzing the posts cross-validated their results and all disagreements were resolved in a discussion among all authors of this paper (2 posts, 5% disagreement rate).

We further extracted identifiers of all apps described by each of the selected posts (one post can refer to multiple apps): app, package, and author names. We searched for apps with these identifiers in academic repositories, e.g., AndroZoo [65], and alternative app markets, e.g., APKPure [66] and APKCombo [67]. Our assumption was that these repositories may still contain the malware sample even though it was already removed from the Google Play store following the detection by the security company. We identified eight apps from three posts and further excluded two apps with API level above 25, as AMANDROID cannot analyze these apps.

We verified that the remaining six apps are indeed spyware by using VirusTotal [68] – an online service aggregating results from 60 anti-virus scanners. Specifically, all our apps were marked as spyware by at least 10 scanners from VirusTotal. Figure 3 summarizes our selection process for the spyware scenario.

Our final app selection of Google Play apps consists of 25 apps listed in Table 5 and includes apps such as *Tripadvisor*, *Airbnb*, and *Yelp*. The distribution of apps that we excluded at each stage of our subject selection process is found in the online appendix [23]. In what follows, we refer to the analyzed apps by their app ids listed in the first column of Table 5, e.g., app #1 refers to *com.echangecadeaux*.

The third column of the table shows the size of the app code (.dex file), which ranges from 0.7 MB to 65.4 MB (11.4 MB on average). As a reference, the average size of an app in our initial dataset of 5,569 Google Play apps is 11.6 MB. We thus believe that our app selection is representative. Finally, the fourth column shows the API level of each app and the last column shows the number of expected flows, identified using the methodology described in the next section.

We deliberately focused on apps from the Google Play store in our study because we believe that investigating the applicability of taint analysis techniques to these apps is of primary relevance to the users, security experts, and other researchers. While collecting data for our study, we considered using applications from the F-Droid repository of open-source Android apps [13]. However, only 14 out of

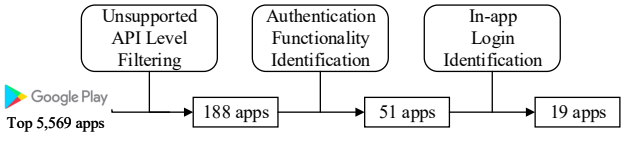

Fig. 2: Login-related apps selection process.

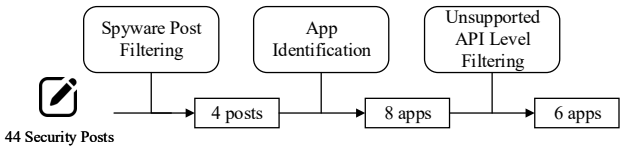

Fig. 3: Spyware selection process.

TABLE 6: List of Source and Sink Methods for Login-Related Google Play Apps (*SS-GPL*)

	Signature	App ID
Sources	android.widget.EditText: android.text.Editable getText()	1-19
Sinks	com.squareup.okhttp.Call: com.squareup.okhttp.Response execute()	17
	cz.msebera.android.httpclient.client.HttpClient: cz.msebera.android.httpclient.HttpResponse execute (cz.msebera.android.httpclient.client.methods.HttpUri-Request)	3
	java.io.OutputStreamWriter: void write(java.lang.String)	5
	java.io.PrintWriter: void write(java.lang.String)	16
	java.net.HttpURLConnection: int getResponseCode()	4, 7, 16
	java.util.zip.GZIPOutputStream: void write(byte[])	2
	okhttp3.Call: okhttp3.Response execute()	12, 14, 15, 19
	okhttp3.Call: void enqueue(okhttp3.Callback)	13
	org.apache.http.client.HttpClient: org.apache.http.HttpResponse execute(org.apache.http.client.methods.HttpUriRequest)	1, 8, 9, 10
	org.apache.http.client.HttpClient: org.apache.http.HttpResponse execute(org.apache.http.client.methods.HttpUriRequest, org.apache.http.protocol.HttpContext)	11
	org.apache.http.impl.client.DefaultHttpClient: org.apache.http.HttpResponse execute(org.apache.http.client.methods.HttpUriRequest)	6, 18

TABLE 7: List of Source and Sink Methods for Spyware Google Play Apps (*SS-GPS*)

	Signature	App ID
Sources	android.content.ClipData$Item: java.lang.CharSequence getText()	20, 23, 24, 25
	android.content.ContentResolver: android.database.Cursor query(android.net.Uri,java.lang.String[],java.lang.String, java.lang.String[],java.lang.String)	21, 23
	android.location.Location: double getLatitude()	20, 21, 23, 24, 25
	android.location.Location: double getLongitude()	20, 21, 23, 24, 25
	android.telephony.TelephonyManager: java.lang.String getDeviceId()	21
	android.telephony.TelephonyManager: java.lang.String getLine1Number()	21
	android.telephony.TelephonyManager: java.lang.String getNetworkCountryIso()	20, 22, 23, 24, 25
	java.util.Locale: java.lang.String getCountry()	23
	java.util.Locale: java.lang.String getLanguage()	20, 22, 23, 24, 25
Sinks	android.telephony.SmsManager: void sendTextMessage(java.lang.String,java.lang.String,java.lang.String, android.app.PendingIntent,android.app.PendingIntent)	21
	java.io.BufferedWriter: void write(java.lang.String)	20, 22, 23, 24, 25
	java.io.OutputStream: void write(byte[])	22

1,631 F-Droid apps are in the top Google Play charts and only two of these apps have API version that satisfies our selection criteria, which was insufficient for our analysis.

3.4.2 Sources, Sinks, and Expected Results

Two authors of this paper performed manual analysis of all the selected apps to identify expected flows. To ensure validity, the authors conducted manual checks independently and cross-validated their findings. They had disagreements in two out of 25 apps (8% disagreement rate); like with the benchmark applications, the disagreements are resolved in a meeting with all the authors. In the end, we were able to identify the flows in all 25 apps: one expected flow in each login-related app and eight, on average, in the spyware apps. We now discuss this process in detail.

Login scenario. We looked for flows from the password field to the internet. We used textual input fields, namely, `EditText.getText()`, as the source for our expected flows and then followed the flow to identify internet-related APIs that send the user's password over the network. We considered them as sinks. While doing our analysis, we noticed that app #16 also writes the password to the log file, for debugging purposes. This is a concerning behavior and we added the corresponding flow and the sink to our expected results. Overall, we identified 21 expected flows: 19 for sending the password over the network (one per app) and 2 for writing it into a log file. The list of the identified sources and sinks is shown in Table 6. We refer to this list as *SS-GPL* and configured the tools to use this list when analyzing login-related apps. The last column of Table 6 lists the ids of apps containing each source and sink.

As source `EditText.getText()` can cover user inputs other than the password, tools may detect additional valid flows which are not login-related. Hence, for each app, two of the authors independently analyzed all flows identified by the tools and categorized them into TPs and FPs. The disagreements (in one app, 5% disagreement rate) were resolved in a discussion with all the authors. We added the identified TP flows to our expected results, which resulted in additional 14 flows in app #13. In total, we identified 35 expected flows, as shown in the fourth column of Table 5. Note that there could be additional login-unrelated flows between sources and sinks in *SS-GPL*, which none of the tools can detect and we are unable to identify manually due to the size and complexity of the considered applications.

Spyware scenario. We analyzed the blogs to extract the description of the user-sensitive information leaked by each app and further mapped it to the corresponding Android API. Specifically, two authors of the paper independently checked each post to extract phrases such as "device ID", "SMS", "location", "contacts", and "call logs". To map these leaked secrets to Android APIs, we relied on the SuSi list [55] that contains 18,045 possible sources in Android projects. Specifically, for each of the apps, we first identified all SuSi sources invoked at least once in an app (634 sources on average, min: 346, max: 727, median: 699) and filtered this list to the type of secrets the app leaks, according to the post. For example, if an app invokes the source API `TelephonyManager.getLine1Number()` and it is reported to leak the phone number, we mark the API as a source for the analysis of the app. Following this process, we identified 11 potential sources on average per app (min: 9, max: 13, median: 11).

We further manually tracked the propagation of information from each invocation site of the potential source API in an app (41 invocation sites on average per app, min: 29, max: 48, median: 43) to the first API on each execution path that sends information out of a device. If found, we marked that API as a sink, added the source to this list of sources for that app, and added the flow to our list of expected flows. At the end of this process, we were able to identify 47 flows in the six analyzed apps in total (8 flows per app on average,

min: 3, max: 12, median: 7). We detected sinks of two kinds: sending secrets over the Internet or in SMS.

The identified sources and sinks for each app are listed in Table 7. As the types of secrets leaked by each app are different, in our analysis, we configured the tools to use the corresponding list of sources and sinks per each app.

In addition, like with the login-related apps, there could be other valid flows from the same sources, which we did not detect in our manual analysis. Hence, similar to login-related case, two of the authors analyzed all flows detected by the tools and categorized them into TPs and FPs. This resulted in no additional flows in any of the apps and there were no disagreements between the authors during this process. The final number of expected flows per app is listed in the fourth column of Table 5.

3.4.3 Metrics and Measures

When detecting expected flows for the Google Play apps, we augmented our manual analysis of flows corresponding to the original login or spyware scenario with additional analysis of flows detected by each to identify other valid TP cases, as discussed in Section 3.4.2. Thus, the categorization into FP and FN results was done alongside with establishing the expected flows. We then calculated precision, recall, and F-measure for each tool; we also measured the execution time and memory consumption of each tool and report on the results averaged over five consecutive runs. All experiments were run on the same Ubuntu 16.04 server with a 20-core CPU; we limited the RAM allocation for each experiment to 256 GB.

We set a 72-hour timeout for each Google Play app. AMANDROID also provides the option to set a maximum running time for each component in an app. To prevent the tool from spending the entire allocated time on one component and being unable to reach the remaining components, we set a per-component limit as 72 hours divided by the number of components declared in an app's manifest file. As the number of components in our selected apps ranges from 7 to 314, the timeout set for each component ranges from 14 to 617 minutes, which is larger than 10 minutes set by the authors of AMANDROID in their experiments [5].

4 Results: Benchmark Apps (RQ1-RQ3)

We now answer RQ1–RQ3 introduced in Section 1.

4.1 Tools Performance (RQ1)

To answer RQ1, we ran FLOWDROID, AMANDROID, and DROIDSAFE, with and without DROIDRA, on the 182 benchmark apps using the configuration setup described in Section 3.2, and measured the accuracy and execution time of each tool.

4.1.1 Accuracy

The first row of Table 8 ("Our experiment") summarizes the precision, recall, and F-measure obtained in our experiments with benchmarks, which we report separately for *DroidBench* and *ICC-Bench*. Here, we report on all issues observed in the versions of the tools listed in Table 1, including those whose fixes were made available in the later versions of the tools following our discussion with the tools' authors.

```
1 class MainActivity extends Activity {
2   @Override
3   void onCreate(Bundle state) {
4     ...
5     String imei = getDeviceId(); // source
6     Intent i = new Intent(this,
          AnotherActivity.class);
7     i.putExtra("imei", imei);
8     startActivity(i); // sink, leak
9   }
10 }
11 class AnotherActivity extends Activity {
12   @Override
13   void onCreate(Bundle state) {
14     ...
15     Intent i = getIntent();
16     String imei = i.getStringExtra("imei");
17     Log.i("TAG", imei); // sink, leak
18   }
19 }
```

Fig. 4: Flows with ICC-related sinks.

As discussed in Section 3.2, all tools deviate from the standard taint flow semantics and implement additional logic when handling ICC-related flows. By inspecting AMANDROID code we learned that starting from 3.1.2, AMANDROID filters out intra-component flows for ICC-related sinks, e.g., `startActivity(Intent)`, even though the tool can successfully detect them. For the example in Fig. 4, the source in line 5 flows through the sink in line 8 and the sink in line 17. In this case, AMANDROID detects both flows but only reports the second one (line 17), while the standard taint analysis semantics assumes two flows.

Similarly, DROIDSAFE also uses its own proprietary logic to handle ICC flows. Based on DROIDSAFE's documentation and our experiments with multiple examples, we learned that the logic applied by DROIDSAFE is more advanced than that of AMANDROID. Specifically, it only filters intra-component flows for *explicit* Intents; for *implicit* Intents, the tool always reports the intra-component part of ICC flow as well, as components from other applications could also register and receive this Intent.

FLOWDROID is the only tool that allows to explicitly turn the proprietary ICC handling logic on and off. Yet, our experiments and discussions with the authors show that this option is not fully supported. Moreover, its intended implementation differs from the logic proposed by both AMANDROID and DROIDSAFE.

To fairly compare the tools with each other, we re-calculated the accuracy of the tools based on the standard taint flow semantics and report these numbers in Table 8. The original accuracy values for each tool, with its own proprietary handling of ICC flows, are also reported in Table 8 in parenthesis, for reference.

In addition, we found that DROIDSAFE does not report the exact path of the detected flows, making the flow analysis more difficult. Instead, it reports the source and sink methods of the identified flow, as well as the location of the entry method, i.e., the method in which the source is defined,. We used this information to identify the flows and match them to the expected ones. Yet, the tool sometimes reports entry methods incorrectly. For the example in Fig. 5, DROIDSAFE correctly reports a flow from `getDeviceId()` (line 5) to `Log.d()` (line 12), but considers `MyRunnable: void run()` being the entry method, which is incorrect. Overall, there are 17 output record with the correct source and sink methods but with the entry method pointing to the sink rather than

TABLE 8: Benchmark Apps: Comparisons of Precision, Recall, and F-measures

Source	Benchmark Suite	Tool	Precision	Recall	F-measure
1. Our experiment	*DroidBench* (158 apps from v3.0)	FLOWDROID	88	72	79 (33)
		DROIDRA + FLOWDROID	88	72	79 (33)
		AMANDROID	66	56	61 (55)
		DROIDRA + AMANDROID	76	59	66 (60)
		DROIDSAFE	88	89	88 (71)
		DROIDRA + DROIDSAFE	88	90	89 (72)
	ICC-Bench (24 apps from v2.0)	FLOWDROID	100	65	79 (34)
		DROIDRA + FLOWDROID	100	65	79 (34)
		AMANDROID	85	100	92 (67)
		DROIDRA + AMANDROID	85	100	92 (67)
2. Arzt et al. [2] FlowDroid14	*DroidBench* (35 apps from v1.0 developed by FLOWDROID's authors, no implicit flows)	FLOWDROID (no ICCTA)	86	93	89
3. Arzt [21] FlowDroid 2017	*DroidBench* (all 189 apps from v3.0)	FLOWDROID (no ICCTA)	87	84	86
		FLOWDROID	91	84	87
		DROIDSAFE	91	72	80
4. Li et al. [4] ICCTA 2015	22 ICC-related apps developed by ICCTA's authors and *ICC-Bench* (9 apps from v1.0)	FLOWDROID	97	97	97
		AMANDROID	79	52	63
Amandroid 14 5. Wei et al. [5]	*DroidBench* (39 apps from v1.0)	FLOWDROID (no ICCTA)	86	72	78
		AMANDROID	87	75	81
	ICC-Bench (16 apps developed by AMANDROID's authors)	FLOWDROID (no ICCTA)	75	45	56
		AMANDROID	100	100	100
Amandroid 18 6. Wei et al. [11]	*DroidBench* (18 apps from v2.0, ICC-related)	FLOWDROID	86	83	85
		AMANDROID	96	96	96
		DROIDSAFE	85	96	90
	ICC-Bench (24 apps from v2.0)	FLOWDROID	97	90	93
		AMANDROID	97	100	98
		DROIDSAFE	10	3	5
DroidSafe 7. Gordon et al. [6]	*DroidBench* (94 apps from v1.2)	FLOWDROID	73	81	76
		DROIDSAFE	88	94	91
	DroidBench (40 apps developed by DROIDSAFE's authors)	FLOWDROID	79	35	48
		DROIDSAFE	100	100	100
DroidRA/ 8. Li et al. [19], Sun et al. [20]	*DroidBench* (4 reflection-related apps from v2.0) and 9 apps developed by DROIDRA's authors	FLOWDROID	-	8	-
		DROIDRA + FLOWDROID	-	92	-

```
1 class ActivityWithRunnable extends Activity {
2   @Override
3   void onCreate(Bundle state) {
4     ...
5     Executors.newCachedThreadPool().execute(
          new MyRunnable(getDeviceId())); //
          source
6   }
7   class MyRunnable implements Runnable {
8     String deviceId;
9     MyRunnable(String deviceId) { this.
          deviceId = deviceId; }
10    @Override
11    void run() { Log.d("ActivityWithRunnable",
          deviceId); } // sink, leak
12  }
13 }
```

Fig. 5: Placement of entry methods.

the source; in 5 cases, the tool produced duplicated flows, one with the entry method correctly pointing to the source and another – to the sink; in 1 additional case, the entry method pointed to a different method altogether.

As the tool still reports the correct sources and sinks for all these cases, we ignored the placement of entry methods and duplicated flows when calculating the precision, recall, and F-measure for DROIDSAFE, not to disadvantage the tool.

Overall, DROIDSAFE achieves the highest accuracy on *DroidBench* benchmarks (F-measure of 88%, when executed without DROIDRA), followed by FLOWDROID (79%) and AMANDROID (61%). Yet, AMANDROID performs better on *ICC-Bench*: 92% vs. 79% for FLOWDROID. We did not run DROIDSAFE on this benchmark, as discussed in Section 3.3.1. For six benchmark apps, FLOWDROID crashed with exceptions; for one app, DROIDSAFE did not finish after two hours time limit. As all these benchmarks include expected flows, we counted these flows as FNs for the corresponding tools.

DROIDRA slightly improves the performance of AMANDROID and DROIDSAFE on *DroidBench*: for AMANDROID– from 61% to 66%, and for DROIDSAFE– from 88% to 89%. We discuss the reasons for this improvement in Section 4.3. As *ICC-Bench* does not contain reflection cases, the accuracy of the tools on this benchmark suite is the same, with and without DROIDRA.

4.1.2 Execution Time and Memory Consumption

We measured the execution time and memory consumption of each tool on the *DroidBench* applications only, as DROIDSAFE does not run on *ICC-Bench*. To ensure a fair comparison, we configured FLOWDROID and AMANDROID to use the same set of sources and sinks that DROIDSAFE uses, as the set of sources and sinks in DROIDSAFE cannot be configured. We performed five runs of each tool on each benchmark and averaged the measurements from these runs.

Figure 6 shows the execution time of each tool, in seconds, on a logarithmic scale. We excluded from this report one benchmark app for which FLOWDROID crashed and one outlier benchmark app for which DROIDSAFE did not finish after two hours. Our results show that all the analyzed tools processed benchmark applications within a relatively short time, with FLOWDROID being the fastest: 12 seconds on average (min: 8 seconds, max: 53 seconds, median: 11 seconds), followed by AMANDROID: 17 seconds on average (min: 14 seconds, max: 29 seconds, median: 16 seconds), and DROIDSAFE: 139 seconds on average (min: 107 seconds, max: 24.1 minutes, median: 126 seconds). The execution times for each of the tools are largely consistent across all benchmark apps (as can be seen from a narrow interquartile range, i.e., the difference between the 75th and 25th percentiles), with only a few outliers.

Figure 7 shows the memory consumption of each tool, in megabytes (MB). On average, FLOWDROID requires the lowest amount of memory: 432 MB on average (min: 246 MB, max: 5228 MB, median: 393 MB). AMANDROID consumes 2426 MB on average (min: 2362 MB, max: 3677 MB, median: 2417 MB). The high accuracy of DROIDSAFE comes at the expense of its high memory consumption: 6962 MB on average (min: 6137 MB, max: 27488 MB, median: 6536 MB). Yet, none of the tools required the entire 64 GB of RAM that we allocated for benchmark apps in our experiments.

Pre-processing input benchmark apps with DROIDRA takes around 71 seconds on average (min: 60 seconds, max: 186 seconds, median: 65 seconds) and consumes 12801 MB of memory on average (min: 12685 MB, max: 14851 MB, median: 12708 MB). There is no noticeable change in execution times and memory consumption of FLOWDROID, AMANDROID, and DROIDSAFE following that processing. Full performance data for all tools on the benchmark apps is available in our online appendix [23].

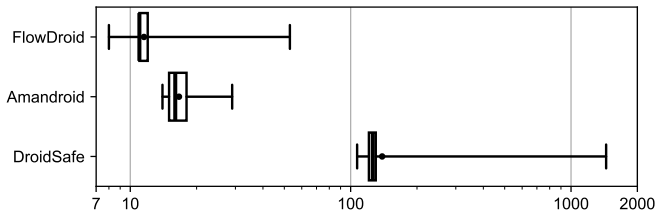

Fig. 6: Execution time (in seconds) on the benchmark apps.

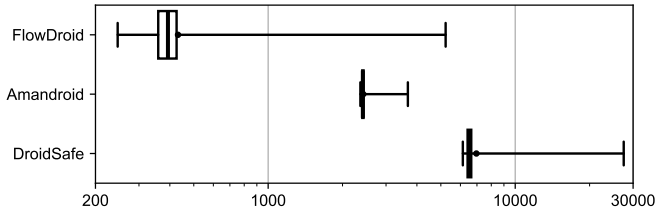

Fig. 7: Memory consumption (in MB) on the benchmark apps.

4.2 Comparison with Earlier Experiments

We extracted the accuracy and execution time reported by previous studies, compared with our results reported in the previous section, and outlined possible reasons for differences in results.

We list the tool accuracy reported by other studies conducted by the tool's authors themselves [2], [4]–[6], [11], [19], [21] in rows 2-8 of Table 8. We noticed that the accuracy observed in our experiments is mostly lower than that reported in earlier studies. For example, Li et al. [4] (row 4) reported a very higher accuracy of 97% for FLOWDROID, compared with 79% on both *DroidBench* and *ICC-Bench* in our study. As our experiments include substantially larger benchmark suites (e.g., 158 *DroidBench* and 24 *ICC-Bench* apps vs. 31 ICC-related apps in Li et al. [4] experiment), we observe more failures and cannot compare our results to other studies directly.

Interestingly, there is also a high variability in the results reported by different studies themselves. This is mainly because each paper only evaluated the tools on a subset of benchmarks, focusing on a specific set of challenges. Even though both Li et al. [4] (row 4) and Wei et al. [11] (row 6) evaluated FLOWDROID on ICC-related benchmark apps, they still used a different subset of apps: the former contributed 22 ICC-related apps and used 9 apps from *ICC-Bench* while the later selected 18 ICC-related apps from *DroidBench* and 24 apps from *ICC-Bench*, most of which were not available in 2015. In fact, the comparison often involves benchmarks contributed by the tools' authors, e.g., [4]–[6], showing a high accuracy for a particular tool on that subset of benchmarks.

Moreover, the results reported by Arzt et al. [2] (row 2) cannot be directly compared with those of Li et al. [4] (row 4), because the latter work introduced the integration of FLOWDROID with ICCTA. Even though both Arzt et al. [2] (row 2) and Wei et al. [5] (row 5) ran FLOWDROID without ICCTA integration, they, again, used a different subset of benchmarks: the former work excluded all benchmarks with implicit flows while the latter rather selected benchmarks related to ICC communication.

The authors of DROIDRA [19], [20] evaluated their tool (row 8) on four reflection-related applications that were available in *DroidBench* v2.0 and nine additional applications developed by the authors themselves, comparing the recall of one of the older FLOWDROID versions (from 2016) and the same version of FLOWDROID extended with DROIDRA. While those experiments substantially improved the accuracy of FLOWDROID for reflection-related cases, we conducted our experiments on a later version of FLOWDROID, which already incorporated handling of these cases "natively". Moreover, our experiments are conducted on *DroidBench* v3.0, which included 15 additional reflection-related cases, most of which cannot be handled by FLOWDROID and DROIDRA+FLOWDROID. We thus did not observe any improvements for FLOWDROID with DROIDRA. Interestingly, the authors of DROIDRA did not combine the tool with AMANDROID and DROIDSAFE, as we did in our experiments;

thus, the effect of the tool on these solutions was unexplored. Yet, we observed that DROIDRA is effective in helping these tools resolve several reflection-related cases.

In addition to the difference in tools, benchmarks, and parameters used to configure the evaluated tools, the set of sources and sinks also differs between the studies (e.g., DROIDSAFE uses a proprietary and non-configurable set of sources and sinks), further hindering the comparison. Our independent and large-scale experiment covering the entire set of benchmark apps under the common setup is necessary to assess and compare the performance of the tools.

Comparison with our earlier work [24]. Our current results are similar to those reported in the earlier version of this paper [24]. The main difference stems from the upgrade of the *DroidBench* benchmark from version 2.0 to version 3.0 used in the current experiments. The *DroidBench* upgrade included 49 new and one modified benchmark apps, which exposed additional weaknesses in all three tools. When excluding the new and modified *DroidBench* benchmarks, FLOWDROID achieves the F-measure of 84% and AMANDROID achieves the F-measure of 65%, which are comparable with 85% and 68%, respectively, reported in the earlier work; DROIDSAFE version did not change, so the F-measure becomes 92%, as before.

Execution Time and Memory Consumption. With respect to the execution time, our results are consistent with other reports showing that FLOWDROID is faster than AMANDROID, which is, in turn, faster than DROIDSAFE [4], [6], [24]. We do not perform a numerical comparison of execution times with other experiments, as these experiments were performed using a different hardware setup, such as CPU and RAM size, and with different configuration selections. We also do not compare the memory consumption to other reports as these reports do not provide such data.

4.3 Causes of Inaccuracy (RQ2)

In this section, we analyze the failures we observed in each of the tools. We start by identifying the *target criterion* for each benchmark – a particular aspect of taint analysis that the benchmark app is designed to test. For example, the *AccessArray1* benchmark app tests whether the analysis distinguishes between different array positions. We mark this criterion with the index of its corresponding benchmark, i.e., DB3.1. Numbers prefixed with DB indicate target criteria extracted from *DroidBench* (126 target criteria in total, grouped into 13 categories); numbers prefixed with ICC indicate target criteria extracted from *ICC-Bench* (18 target criteria, grouped into four categories).

A particular benchmark app can evaluate multiple target criteria, e.g., both reflection and location handling. For example, in benchmark *EventOrdering1*, the designers tested whether the analysis tool is able to take into account different repeating runs of the same activity. At the end of the first run, they stored the tainted variable in `SharedPreferences` and then retrieved it in the next run. While FLOWDROID and AMANDROID both failed for this app, the underlying reason is not that they cannot track the repeating runs of an activity (criterion DB9.16) but rather that the tools cannot model `SharedPreferences` (criterion DB8.16). We distilled the

```
1 public class MainActivity extends Activity {
2   @Override
3   public void onCreate(Bundle instance){
4     ...
5     TelephonyManager manager = ...;
6     Class c = Class.forName("android.telephony
          .TelephonyManager");
7     Method m = c.getMethod("getDeviceId");
8     String imei = (String) m.invoke(manager);
          // source (reflection)
9     Log.v("imei", imei); // sink, leak
10  }
11 }
```

Fig. 8: U-DB10.6 – Sources or sinks invoked using reflection.

`SharedPreferences` failure into a separate test case, which we added to *UBCBench*.

Moreover, we identified ten additional target criteria not explicitly mentioned by the existing benchmarks. We marked them with the "U-" prefix and assigned them the next available id in their corresponding category, e.g., U-DB10.6 for the criterion we added to the Reflection category DB10.*. We used the new criteria to explain some of the failures and also added the corresponding ten benchmark applications to *UBCBench*. In total, we added 19 new benchmarks to *UBCBench*: nine for untangling benchmarks criteria and ten for the new target criteria we identified. We did not use these new benchmarks in our experiments, to focus the analysis on *DroidBench* and *ICC-Bench* only.

Each analyzed benchmark application contains none to multiple expected flows. In our suite, there are 192 expected flows – 158 for the *DroidBench* and 34 for the *ICC-Bench* apps. Table 9 shows the number of false positive and false negative results reported by each tool without the DROIDRA extension, when evaluated against the set of expected flows. We aggregated the results for each tool on a particular benchmark suite by their failing criteria. That is, we inspected each FP and FN flow observed in our experiment to find the reason for the failure and indexed it with the appropriate criterion.

In three cases, we were unable to recover the reasons for the failures. These cases are indicated by UN in Table 9. In addition, DROIDSAFE timed out in one case, which we denoted by TO. FLOWDROID exceptions discussed in Section 4.1.1 are marked as EX. We also identified several bugs related to FLOWDROID's integration with STUBDROID and its handling of ICC flows in Services and Broadcast Receivers. These bugs are marked as BUG1 and BUG2, respectively; we were working with the FLOWDROID authors to fix these bugs and the corresponding fixes will be released in the next version of the tool. The full indexed list of benchmarks, failing criteria for each benchmark and tool, and the *UBCBench* test suite are available online [23]. We now discuss the main reasons for the failures of each tool.

FLOWDROID does not resolve reflective calls when identifying source and sink methods but rather relies on API signatures. For the example in Fig. 8, the `getDeviceId()` source method is invoked reflectively in line 8. FLOWDROID will miss this source method and, thus, will miss the leakage to the sink in line 9. That behavior leads to 13 FN results stemming from six separate benchmarks (out of 10 in the Reflection ICC category, DB11.*). As none of these benchmarks specifically focuses on the issue of sources

TABLE 9: FN/FP Breakdown by Target Criteria for Benchmark Apps

Tool	FN		FP	
	#	Breakdown	#	Breakdown
DroidBench				
FLOWDROID	45	1x DB2.8 ;1x DB7.10 ;1x DB7.13 ;1x DB7.15 ;1x DB7.16 ;1x DB8.2 ; 1x DB8.6 ;1x DB8.8 ;3x DB8.16 ;2x DB8.18 ;1x DB9.3 ;1x DB9.8 ; 1x DB9.14 ;1x DB9.17 ;1x DB9.18 ;2x DB9.19 ;1x DB12.6 ; 1x U-DB7.24 ;13x U-DB10.6 ;6x U-DB10.10 ;2x BUG1 ;1x BUG2 ; 1x EX ;	15	1x DB2.1 ;1x DB3.1 ;1x DB3.2 ;1x DB3.5 ; 1x DB3.8 ;1x DB3.9 ;1x DB4.14 ;1x DB6.7 ; 1x DB7.4 ;1x DB7.21 ;1x DB7.22 ;1x DB7.23 ; 3x DB13.1 ;
AMANDROID	69	1x DB2.7 ;1x DB2.8 ;1x DB2.11 ;1x DB3.6 ;1x DB3.7 ;1x DB3.10 ; 1x DB4.4 ;1x DB4.5 ;1x DB7.5 ;1x DB7.6 ;1x DB7.10 ;1x DB7.12 ; 1x DB7.16 ;3x DB8.16 ;1x DB8.17 ;1x DB9.1 ;1x DB9.5 ;1x DB9.8 ; 5x DB9.9 ;1x DB9.14 ;1x DB9.17 ;1x DB10.2 ;1x DB10.3 ;1x DB10.4 ; 5x DB10.5 ;20x DB11.1 ;1x DB12.6 ;10x U-DB4.15 ;1x U-DB7.24 ; 2x UN ;	45	1x DB3.1 ;1x DB3.2 ;1x DB3.8 ;1x DB3.9 ; 2x DB4.12 ;1x DB4.14 ;2x DB7.4 ;1x DB7.21 ; 1x DB8.2 ;4x DB13.1 ;22x U-DB2.12 ; 7x U-DB4.15 ; 1x U-DB4.16 ;
DROIDSAFE	18	1x DB2.6 ;13x U-DB10.7 ;1x U-DB10.8 ;1x U-DB10.9 ;1x TO ;1x UN ;	20	5x DB1.1 ;1x DB2.1 ;1x DB3.1 ;1x DB3.2 ; 1x DB3.8 ;1x DB3.9 ;2x DB4.12 ;1x DB4.14 ; 2x DB7.4 ;1x DB8.18 ;1x DB9.2 ;3x DB13.1 ;
ICC-Bench				
FLOWDROID	12	1x ICC1.6 ;1x ICC2.4 ;1x ICC2.5 ; 3x BUG2 ;6x EX ;	0	none
AMANDROID	0	none	6	1x ICC2.2 ;1x ICC4.1 ;2x U-DB2.12 ; 2x U-DB4.16 ;

```
1 public class MainActivity extends Activity
    implements View.OnClickListener {
2  @Override
3  protected void onClick(View v) {
4    ...
5    String imei = getDeviceId(); // source
6    ((Button) v).setHint(imei);
7    Log.d("TAG", ((Button) v).getHint().
       toString()); // sink, leak
8  }
9 }
```

Fig. 9: U-DB7.24 – Flows through casted variables.

```
1 public class MainActivity extends Activity
    implements View$OnClickListener {
2  public void onClick(android.view.View) {
3    View $r1;
4    ...
5    String $r3; // tainted by getDeviceId()
6    Button $r4;
7    CharSequence $r5;
8    $r4 = (Button) $r1;
9    virtualinvoke $r4.<Button: void setHint(
       CharSequence)>($r3);
10   $r4 = (Button) $r1;
11   $r5 = virtualinvoke $r4.<Button:
       CharSequence getHint()>();
12   $r3 = interfaceinvoke $r5.<CharSequence:
       String toString()>();
13   staticinvoke <Log: int v(String,String)>("
       TAG", $r3); // sink
14   return;
15 }
16 }
```

Fig. 10: Simplified Jimple code for Fig. 9.

and sinks defined through reflection but rather includes other "complexities" such as string manipulation and ICC flow, we created a new target criterion (U-DB10.6) and the corresponding benchmark in *UBCBench* to pinpoint this issue. Interestingly, AMANDROID also cannot handle this case (as it struggles with reflection in general), but DROIDSAFE is successful in identifying such sources and sinks.

We also discovered that FLOWDROID fails to propagate taint when the involved objects are accessed via casting (U-DB7.24). For the example in Fig. 9, a `View` passed as a parameter to the method is, in fact, a `Button`. Button-specific methods, `setHint()` and `getHint()`, are accessed via casting: in line 6, the hint is set to the tainted `imei` variable and then the value of the hint is leaked in line 7.

To explain the reasons behind this failure, Fig. 10 shows the relevant portion of the Jimple code [69] that corresponds to the bytecode in Fig. 9 (FLOWDROID performs its analysis on Jimple rather than bytecode level). When the tainted value in `$r3` is assigned to a field of `$r4` (via the method `setHint()` in line 9), FLOWDROID triggers backward alias analysis to identify all other variables aliased by `$r4`, as these variables should be tainted as well. The analysis reaches `$r1` (line 8) but fails to backward-propagate the taint, as the assignment in this case is performed via a casting. As a result, when `$r1` is assigned to `$r4` again (line 10), `$r4` is not tainted. Hence, the taint does not reach the sink (line 13), resulting in an FN.

As this issue was, again, discovered in a benchmark whose goal is to test callback handling, we created a separate benchmark, U-DB7.24, to focus specifically on the casting problem and added it to *UBCBench*. AMANDROID also fails on this benchmark albeit for a different reason: it treats casted variables as two completely separate entities; DROIDSAFE manages to handle this case successfully.

Another major reason for FLOWDROID failures is the ICC handling: there are ten failures in the general ICC category (DB8.*) and three in *ICC-Bench* (ICC2.4, ICC2.5, and ICC1.6). These failures are related to Intent tracking through list operations, using complex operations like string manipulations for defining Intents and their actions, and the `SharedPreferences` problems discussed above (DB8.16). The tool also fails to handle advanced ICC involving URIs and MIME format and ICC-related communication between activities.

The remaining, less major failures are from the General Java and Threading categories (eight failures in DB7.* and one in DB12.6), Lifecycle and Callbacks (seven failures in DB9.* and one in DB4.14), Array and Lists (five failures in DB3.*), Field and Object Sensitivity (DB6.7), Unreachable Code (three failures in DB13.*), Android-specific categories (two failures in DB2.*).

AMANDROID reports 24 FPs related to its conservative modeling of Android framework methods: as discussed in Section 3.1, when one of the parameters of

```
1 public class MainActivity extends Activity {
2   @Override
3   public void onCreate(Bundle instance){
4     ...
5     String imei = getDeviceId(); // source
6     int length = Toast.LENGTH_SHORT;
7     Toast.makeText(this, imei, length).show();
8     Log.i("TAG", "" + length); // sink, no
          leak
9   }
10 }
```

Fig. 11: U-DB2.12 – Android framework method modeling (AMANDROID mistakenly propagates taint from one parameter to another).

```
1 class MainActivity extends Activity {
2   LocationManager locManager;
3   LocationListener locListener = new
        LocationListener() {
4     double lat;
5     @Override
6     void onLocationChanged(Location loc) {
7       lat = loc.getLatitude(); // source
8       Log.d("TAG", lat); // sink, leak
9     }
10  };
11  ...
12 }
```

Fig. 12: U-DB4.15 – Location-related flow.

a method is tainted, AMANDROID propagates the taint to other parameters as well. For the example in Fig. 11, the method `Toast.makeText(Context, CharSequence, int)` in line 7 is an Android framework method that presents a pop-up on the screen [70]. All the parameters of this method are independent and there is no data flow between them. However, AMANDROID propagates the taint from the second parameter, `imei`, tainted in line 5, to the third parameter, `length`. As a result, the tool reports an FP flow to the sink in line 8. As this issue is observed in 13 benchmarks aiming at evaluating exception and lifecycle handling, we created a new target criterion and the corresponding benchmark in *UBCBench* focusing solely on this issue (U-DB2.12).

AMANDROID also reports 7 FPs and 10 FNs that stem from handling location-related flows (U-DB4.15). This is because the tool hard-codes the `Location` parameter of the `onLocationChanged()` callback as a source, and does not consider location sources specified by the user like `getLatitude()` and `getLongitude()`. For the example in Fig. 12, AMANDROID will not report the expected flow from the specified source `getLatitude()` to the sink `Log.d()` in line 8, but will report a (false-positive) flow from the callback parameter `loc` of `onLocationChanged()` (line 6) to the sink `Log.d()`. This behavior is troublesome because there are cases where the `Location` object is accessed, but the retrieved data is not sensitive, e.g., `loc.getTime()`; hence, AMANDROID's conservative handling of location sources results in many FP flows. We focused on this issue in the benchmark app U-DB4.15.

Likewise, AMANDROID always considers Intent parameters of callbacks as sources. For the example in Fig. 13, AMANDROID considers the Intent parameter `resultData` as a source, and then propagates a flow to the sink in line 5, which is incorrect. We mark this issue as U-DB4.16 in Table 9; it leads to three FP results.

Another major reason for failures in AMANDROID is its limited handling of reflection, as confirmed by the tool's authors. That results in 28 FNs (DB10.* and DB11.*).

```
1 class MainActivity extends Activity {
2   ...
3   @Override
4   void onActivityResult(int requestCode, int
        resultCode, Intent resultData) {
5     Log.d("TAG", resultData); // sink, no leak
6   }
7 }
```

Fig. 13: U-DB4.16 – Callback Intent handling.

```
1 public class MainActivity extends Activity {
2   @Override
3   protected void onCreate(Bundle
        savedInstanceState) {
4     ...
5     Class c = getClass();
6     Method m = c.getMethod("getImei");
7     m.invoke(this, null);
8   }
9   public String getImei() {
10    String imei = getDeviceId(); // source
11    Log.i("TAG", imei); // sink, leak
12  }
13 }
```

Fig. 14: U-DB10.7 – Reflection via `Object.getClass()` method.

AMANDROID also has ten FNs in Lifecycle category (DB9.*) and five failures in Callbacks (DB4.*). That is mainly because the tool does not model lifecycle methods of the `Application` class and does not correctly handle certain lifecycle methods of Activity and Fragment, e.g., Activity's `onRestoreInstanceState()` and Fragment's `onAttach()`. The tool also fails to track Intents through string manipulation and `SharedPreferences`, leading to seven ICC-related failures (DB8.*, ICC2.2, and ICC4.1).

Similarly to FLOWDROID, the tool also does not handle taint propagation through casted variables (U-DB7.24). Other failures include General Java and Threading (eight failures in DB7.* and one in DB12.6), Arrays and Lists (seven failures in DB3.*), Unreachable Code (four failures in DB13.*), and Android-specific categories (three failures in DB2.*).

DROIDSAFE has 15 FNs in handling reflections. Most of them (13) are due to a singular reason: while the tool can handle the reflective method `Class.forName(String)`, it cannot identify the class retrieved via the method `Object.getClass()` and further fails to resolve reflective calls to methods in the retrieved class. We confirmed this issue with a new benchmark in Fig. 14 and marked it as U-DB10.7. In this example, DROIDSAFE fails to recognize the class `MainActivity` obtained by `getClass()` in line 5. As a result, it misses reflective calls to invoke `getImei()` (lines 6 and 7), and further misses the leakage inside this method in line 11. Interestingly, FLOWDROID can handle this case while DROIDSAFE can handle the reflective calls in Fig. 8 causing FLOWDROID to fail.

The remaining two reflection-related issues in DROIDSAFE are due to (1) mishandling of method overloading, i.e., cases when methods of the same class have identical name but different parameters (we confirmed this issue with a new benchmark, in criterion U-DB10.8) and (2) inability to resolve strings defined in application resource files, when those strings are used as names for reflection targets (we confirmed this issue with a new benchmark in criterion U-DB10.9). FLOWDROID and AMANDROID failed on these benchmarks as well.

TABLE 10: The Effect of DROIDRA for the *DroidBench* Apps

	Reflection-related						Reflection-unrelated					
	FLOWDROID		AMANDROID		DROIDSAFE		FLOWDROID		AMANDROID		DROIDSAFE	
	#	Breakdown	#	Breakdown	#	Breakdown	#	Breakdown	#	Breakdown	#	Breakdown
Added TP	0	none	4	1x DB10.2 1x DB10.4 2x DB10.5	2	2x U-DB10.7	0	none	0	none	0	none
Removed TP	0	none	0	none	0	none	0	none	0	none	0	none
Added FP	0	none	0	none	0	none	0	none	0	none	0	none
Removed FP	0	none	0	none	0	none	0	none	16	16x U-DB2.12	0	none
Total	0		4		2		0		16		0	

DROIDSAFE also has five failures related to flow insensitivity, which is due to the tool's design (DB1.1). Five failures are related to the handling of Callbacks, Lifecycle, and ICC (DB4.*, DB8.18, DB9.2). These are because DROIDSAFE assumes all possible callback orders, even those that cannot occur at runtime. It also conservatively assumes that Intents which cannot be resolved statically, e.g., those that depend on user input, can target all components, leading to FPs. Additional failures are due to the handling of Arrays and Lists (four failures in DB3.*), Unreachable Code (three failures in DB13.*), Android-specific (two failures in DB2.*), and General Java categories (two failures DB7.*).

DROIDRA helps resolve 4 FNs for AMANDROID and 2 FNs for DROIDSAFE, all in the reflection category. In addition, it helps resolve 16 FP results for AMANDROID, albeit not reflection-related. Table 10 summarizes the effect of DROIDRA, in terms of added and removed TP and FP cases for each tool, reported separately for reflection-related and reflection-unrelated categories. We discuss these findings in detail next.

There are 19 reflection-related benchmarks in *DroidBench* v3.0, with 30 flows in total. Out of these, FLOWDROID has 19 FNs, i.e., cannot detect 19 of the 30 flows, AMANDROID – 28, and DROIDSAFE – 15 FN results. For AMANDROID, DROIDRA helps resolve four FNs where the reflective calls contain the name and parameters of the callee methods, which is another method of the apps, e.g., `Method method = getClass().getDeclaredMethod("foo", String.class); method.invoke(this, "bar");`. However, DROIDRA cannot handle the remaining cases where (a) the callee method name and its parameters are constructed dynamically, in app code and (b) the callee method is a method from the Android framework rather than the app itself. We created benchmarks U-DB10.10 and U-DB10.11 to capture each such behavior.

Two of the four FNs that DROIDRA resolves for AMANDROID contain `Object.getClass()` reflective calls. As DROIDSAFE cannot handle such calls (U-DB10.7), DROIDRA helps resolve these two cases for DROIDSAFE as well. The remaining two are handled "natively" by DROIDSAFE. FLOWDROID can handle all these "simple" cases, but fails on reflective calls where the callee method name and its parameters are constructed dynamically (U-DB10.10).

Overall, we observe that DROIDSAFE implements the most robust reflection handling mechanism: the combination of DROIDRA and DROIDSAFE has only 13 FNs on the benchmark suite and we estimate that the tool can easily be augmented to resolve at least the two "simple" `Object.getClass()` reflective calls discussed above.

Interestingly, for AMANDROID, DROIDRA also helped eliminate 16 FP flows from four benchmark apps. All these benchmark apps are related to the framework method modeling (U-DB2.12) and do not contain any reflective calls. Upon further investigation, we discovered that this behavior is actually a side effect of app repackaging performed by DROIDRA: as it relies on SOOT [34] to read and modify the bytecode of the original app and further produce the bytecode of the instrumented app, the application used as the input to a taint-analysis tool differs for the original benchmark app. In the case of AMANDROID, that difference helps eliminate an FP flow related to the AMANDROID conservative modeling of Android framework. In fact, we confirmed that a simple repackaging of the apps with SOOT leads to the same behavior, even without using DROIDRA.

The common pattern among all these benchmarks apps is that the sink is placed inside a loop. Fig. 15 is a simplified version of the benchmark apps that we use to illustrate the issue. In this example, the tainted `imei` variable (line 5) is leaked at the sink `sendTextMessage(...)` in line 9. The sink itself is placed inside a loop running for five iterations. The expected behavior of a taint analysis tool is to report only one flow from line 5 to line 9 (as done by FLOWDROID and DROIDSAFE).

Fig. 16 shows instruction-level code produced from the Java code in Fig. 15 by Android Studio – the official integrated development environment for Google's Android operating system [71]. For simplicity, we show the code in JAWA intermediate representation [72] commonly used by many static analysis techniques, including AMANDROID. This code first sets the variable `v2` to null (line 1) and the variable `v3` – to the tainted value returned by the source API (lines 3-4). The variables `v6` and `v1` are used to check the loop condition; the loop itself is implemented in lines 8-16. The important part are variables `v4` and `v5`, which are set to `v2` (i.e., *null*) and are passed to the sink method, together with the tainted variable `v3` (line 14).

To preserve the loop semantics, AMANDROID unrolls each loop three times. As such, the sink method in line 14 will be executed three times in sequence. In the first iteration, as the variable `v3` is tainted, AMANDROID successfully reports the flow to the sink. However, due to its conservative modeling of Android framework methods, it also propagates the taint from `v3` to other method parameters, including `v2`, `v4`, and `v5`. While the variables `v0` and `v1` are re-assigned in each iteration of the loop (lines 10-11), `v2` keeps the taint and also

TABLE 11: Failures in Benchmark Apps Grouped by Target Criteria

Target Criteria	FLOWDROID	AMANDROID	DROIDSAFE
Flow Sensitivity (DB1.1)	0	0	5
Android-specific (DB2.*, U-DB2.12)	2	27 (11 after repackaging)	2
Arrays and Lists (DB3.*)	5	7	4
Lifecycle and Callbacks (DB4.*, DB9.*, U-DB4.*)	8	35	4
Object Sensitivity (DB6.7)	1	0	0
General Java (DB7.*, U-DB7.*)	9	9	2
ICC (DB8.*, ICC*.*)	11	7	1
Reflection (DB10.*, DB11.*, U-DB10.*)	19	28 (24 with DROIDRA)	15 (13 with DROIDRA)
Threading (DB12.6)	1	1	0
Unreachable Code (DB13.*)	3	4	3
Total	59	118	36

```
1 public class MainActivity extends Activity {
2   @Override
3   protected void onCreate(Bundle
        savedInstanceState) {
4     ...
5     String imei = getDeviceId(); // source
6
7     for (int i = 0; i < 5; i++) {
8       SmsManager sm = SmsManager.getDefault();
9       sm.sendTextMessage("+49 123456", null,
            imei, null, null); // sink, leak
10    }
11  }
12 }
```

Fig. 15: A loop resulting in different instruction-level code for Android Studio and SOOT.

```
1 v2:= 0I;
2 ...
3 call temp:= `getDeviceId`(v7) ... // source
4 v3:= temp @kind object;
5 v6:= 0I;
6 v1:= 5I;
7 /* begin of loop */
8 if v6 >= v1 then goto L18;
9 call temp:= `getDefault`() ...
10 v0:= temp @kind object;
11 v1:= "+49 123456" @kind object;
12 v4:= v2 @kind object;
13 v5:= v2 @kind object;
14 call `sendTextMessage`(v0,v1,v2,v3,v4,v5) ...
       // sink, leak
15 v6:= v6 + 1;
16 goto L6;
17 /* end of loop */
18 return @kind void;
```

Fig. 16: Android Studio version of the example in Fig. 15.

taints `v4` and `v5` in the next iteration of the loop (lines 12-13). As a result, AMANDROID reports three FP flows in line 14 for incorrectly tainted variables `v2`, `v4`, and `v5` each.

Fig. 17 shows the instruction-level code generated by SOOT after repackaging this app. The main difference is in the handling of the loop variables. Here, the variables `v2`, `v4`, and `v5` are re-assigned in each iteration of the loop to variables `v14`, `v15` and `v16`, respectively (lines 15, 17-18), which, in turn, are set to *null*, also inside the loop (lines 10-12). Reassigning the variables in each iteration prevents FP flows detected by AMANDROID in the previous case.

Summary. Table 11 summarizes the criteria where at least one of the tools, with and without DROIDRA, fails. It also lists the total number of failures per criterion for each tool. Overall, the major issue of all the tools is in reflection handling, albeit for different reasons. DROIDRA helps resolve simple reflective calls listing the name and parameters of the callee methods, but is limited in handling more complex cases

```
1 call temp:= `getDeviceId`(v9) ... // source
2 v11:= temp @kind object;
3 v12:= 0I;
4 v6:= 5I;
5 /* begin of loop */
6 if v12 >= v6 then goto L23;
7 call temp:= `getDefault`() ...
8 v13:= temp @kind object;
9 v8:= "+49 123456" @kind object;
10 v14:= 0I;
11 v15:= 0I;
12 v16:= 0I;
13 v0:= v13 @kind object;
14 v1:= v8 @kind object;
15 v2:= v14 @kind object;
16 v3:= v11 @kind object;
17 v4:= v15 @kind object;
18 v5:= v16 @kind object;
19 call `sendTextMessage`(v0,v1,v2,v3,v4,v5) ...
       // sink, leak
20 v12:= v12 + 1;
21 goto L4;
22 /* end of loop */
23 return @kind void;
```

Fig. 17: SOOT version of the example in Fig. 15.

involving dynamic method name and parameter generation, method overloading, and Android framework methods used as callees. Additionally, FLOWDROID fails to accurately parse and track ICC Intents involving complex string analysis and list management operations. AMANDROID has major issues in handling Android framework methods, location-related flows, and lifecycle and callback methods. Once these major problems are fixed, the tools will have a much higher overall accuracy.

5 RESULTS: GOOGLE PLAY APPS (RQ4)

In this section, we answer RQ3 introduced in Section 1: do the results on the benchmark applications generalize to the Google Play apps? We run the tools on 25 apps with API level 25 or lower, as discussed in Section 3.4 and reporting the performance of FLOWDROID and AMANDROID, with and without DROIDRA, on these apps.

5.1 Tools Performance

5.1.1 Accuracy

Table 12 shows the results of app analysis for FLOWDROID: the number of expected and detected flows, as well as the breakdown of reasons for FN and FP results of the tool for each app. FLOWDROID is able to identify flows in only two apps (#13 and #21), listed in the first two rows of the table. The remaining rows show apps with no flows detected. We

TABLE 12: FLOWDROID Results on Google Play Apps ("●" indicates the tool fully analyzed the app and "○" indicates the tool failed to analyze the app)

App ID	# Exp. Flows	# Detected Flows	FN		FP	
			#	Breakdown	#	Breakdown
13	15	17 ● (15 TP, 2 FP)	0		2	2x U-DB2.12
21	12	4 ● (4 TP, 0 FP)	8	8x U-DB2.12	0	
3	1	0 ●	1	1x U-DB4.18	0	
4	1	0 ●	1	1x DB8.16	0	
6	1	0 ●	1	1x U-DB4.17	0	
9	1	0 ●	1	1x U-DB4.17	0	
10	1	0 ●	1	1x U-DB4.17	0	
11	1	0 ●	1	1x U-DB2.12	0	
15	1	0 ●	1	1x U-DB10.12	0	
17	1	0 ●	1	1x U-DB10.12	0	
20	7	0 ●	7	5x DB8.16; 2x U-DB2.12	0	
22	3	0 ●	3	3x U-DB2.12	0	
23	11	0 ●	11	5x DB8.16; 6x U-DB2.12	0	
25	7	0 ●	7	5x DB8.16; 2x U-DB2.12	0	
1, 2, 5, 7, 8, 12, 14, 18, 19	1 (per app)	0 ○	9	9x EX	0	
16	3	0 ○	3	3x EX	0	
24	7	0 ○	7	7x OM	0	
Total	82	21 (19 TP, 2 FP)	63	16x DB8.16; 22x U-DB2.12; 3x U-DB4.17; 1x U-DB4.18; 2x U-DB10.12; 12x EX; 7x OM	2	2x U-DB2.12

mark with ● cases where a tool completed the app analysis and with ○ – cases where the tool failed to analyze an app due to a runtime (EX) or out of memory (OM) exception.

FLOWDROID finished analyzing 14 out of 25 apps, ran out of memory in one app (#24), and produced runtime exceptions in the remaining 10 apps. We identified three cases in which the tool throws exceptions: when handling ICC flows in Services (apps #2, #7, and #14), when handling ICC-related methods with no parameters, e.g., `PendingIntent.send()` (apps #1 and #16), and when calling the underlying SOOT framework to resolve parameters of the reflective `Method.invoke(Object, Object[])` method (apps #5, #8, #12, #18, and #19). We reported these failures to the FLOWDROID authors, discussed possible fixes, and believe that these fixes will be available in the next release of the tool.

For app #13 (login-related), 15 out of 17 flows detected by FLOWDROID are TPs, indicating a high detection accuracy for this app. However, this is the only app where the tool was able to detect the expected login-related flow. For app #21 (spyware), the tool can detect four out of 12 expected flows and has no FP results, leading to a high precision but low recall. We discuss reasons for why FLOWDROID can detect only some of the expected flows in Sections 5.2 and 5.3.

Augmenting FLOWDROID with DROIDRA does not increase the number of identified flows. That is, DROIDRA +FLOWDROID still finds the same flows in apps #13 and #21 only. Moreover, instrumenting apps with DROIDRA resulted in runtime exceptions in six apps: #3, #7, #17, #19, #20, #22. We confirmed with the DROIDRA authors that the exception is due to improper handling of the multi-dex apps.

For AMANDROID, we observed that the number of flows reported by the tool varies across different runs on the same machine, same app, and same configuration setup. For example, on app #11, AMANDROID reported between 0 and 43 flows in different independent isolated runs (specifically, 0, 3, 20, 28, and then 43 flows). As another example, on app #13, the tool reported between 118 and 275 flows in different runs (specifically, 118, 126, 130, and 275 flows). We reached out to the authors of the tool but did not receive a reply explaining this behavior. Due to the flaky nature of the results, we excluded the tool from further analysis, concluding that AMANDROID cannot reliably analyze Google Play apps.

5.1.2 Execution Time and Memory Consumption

To report execution time and memory consumption, similar to the benchmarks, we only consider apps that FLOWDROID analyzed in full and averaged their performance results from five consecutive runs. There are 14 such apps (marked with ● in Table 12). Figures 18 and 19 show the execution time and memory consumption of FLOWDROID for these Google Play apps, respectively. Overall, FLOWDROID spent 2.8 hours per app, on average (min: 1 minute, max: 20.2 hours, median:

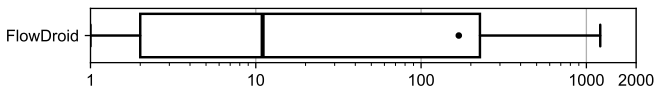

Fig. 18: Execution time (in minutes) of FLOWDROID on the Google Play apps.

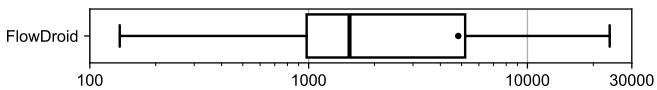

Fig. 19: Memory consumption (in MB) of FLOWDROID on the Google Play apps.

TABLE 13: Android framework methods on FLOWDROID TPs in apps #14 and #22.

App ID	Total TPs	Signature	# TPs	Taint Prop.	STUBDROID?
13	15	android.text.Editable: java.lang.String toString()	7	recv -> ret	✓
		java.lang.String: java.lang.String substring(int,int)	1	recv -> ret	✓
		java.lang.String: java.lang.String replaceAll(java.lang.String,java.lang.String)	1	recv -> ret	✓
		java.lang.String: java.lang.String trim()	6	recv -> ret	✓
		java.lang.String: java.lang.String valueOf(java.lang.Object)	8	param -> ret	✓
		org.json.JSONObject: java.lang.String toString()	15	recv -> ret	✓
		org.json.JSONObject: org.json.JSONObject put(java.lang.String,java.lang.Object)	15	param -> recv	✓
21	4	java.lang.Double: java.lang.String toString()	4	recv -> ret	✓
		java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)	4	param -> recv	✓
		java.lang.StringBuilder: java.lang.String toString()	4	recv -> ret	✓

11 minutes) and consumed 4830 MB of RAM per app, on average (min: 137 MB, max: 23764 MB, median: 1538 MB).

DROIDRA successfully pre-processed 19 out of the overall 26 apps, taking around 19 minutes on average per app (min: 1 minute, max: 2.8 hours, median: 3 minutes). It consumed 27591 MB of RAM per app, on average (min: 21814 MB, max: 37676 MB, median: 27348 MB). As with the benchmark apps, the tools did not require the entire 256 GB of RAM allocated for the Google Play apps in our experiments. Moreover, the execution time and memory consumption of FLOWDROID on the seven apps that it was able to successfully analyze after the DROIDRA pre-processing are similar, albeit slightly higher, than on the original apps. Detailed results of each apps are available online [23].

We did not perform a numerical comparison of performance results with other work due to different hardware setups and configuration selections. Also, our reported performance results are not intended to be statistically representative; we only provide these results for information purposes, to assess the approximate time and memory required to run the tools on Google Play apps.

5.2 Analysis of Correctly Detected Flows

For the app #13, 15 out of 17 flows detected by FLOWDROID are TPs. Overall, all the 15 flows follow a similar pattern: inside a button callback method, a source method `EditText.getText()` is invoked. The retrieved text is converted to a String and added to a JSONObject. Then, the JSONObject is sent out to the Internet via a HTTP request, with the `okhttp3.Call: void enqueue(okhttp3.Callback)` API as the sink.

Fig. 20 shows a slightly shortened example of a TP flow in the app. The callback of the "submit" button is specified in the activity layout XML file (line 3). When this callback method is invoked, it uses the source API method `EditText.getText()` to retrieve the card number specified by the user in the text input widget `cardnumber_input` (line 9). It then creates a new JSON object, adds the card number to it, and converts the object to a string (lines 10-12). This tainted string is used to initialize the body of an HTTP request (lines 13-14). Finally, the request is sent over the Internet via the sink method (line 15), resulting in a leak.

Such flows in app #13 are relatively simple and require the tool to (1) identify callbacks defined in the layout XML file and (2) handle Android framework methods, like those in lines 11-14 of Fig. 20. These challenges are captured

```
1 /* AddCard activity layout XML file */
2 ...
3 <Button android:id="@+id/submit_button"
      android:onClick="submit"/>
4
5 /* Source code */
6 public class AddCard extends
      AppCompatActivity {
7   ...
8   public void submit(View view) {
9     cardnumber = String.valueOf(this.
          cardnumber_input.getText()); // source
10    JSONObject json = new JSONObject();
11    json.put("cardnumber", cardnumber);
12    String jStr = json.toString();
13    RequestBody reqBody = RequestBody.create(
          JSON, jStr);
14    Request request = buildPOSTRequest(reqBody
          , "registration/card_no_password");
15    OkHttpClient().newCall(request).enqueue(
          new RequestCallback()); // sink, leak
16  }
17 }
```

Fig. 20: An example of a TP flow in app #14.

```
1 public class MyClass {
2   void foo(String str) {
3     ...
4     Location location = LocationManager.
          getLastKnownLocation("gps");
5     StringBuilder sb = new StringBuilder();
6     sb.append("Lat : ");
7     sb.append(Double.toString(location.
          getLatitude())); // source
8     str2 = sb.toString();
9     SmsManager.getDefault().sendTextMessage(
          str, null, str2, null, null); // sink
          , leak
10    ...
11  }
12  ...
13 }
```

Fig. 21: An example of a TP flow in app #22.

by benchmarks in the DB4.2 and U-DB2.12 categories, respectively. In this case, they can be successfully addressed by the tool: FLOWDROID can handle callbacks defined in the layout XML file; for the Android framework methods, rows 1-7 in Table 13 list all methods used in the 15 TP flows of this app, as well as the number of flows containing each method, the propagation of the taint inside the method (e.g., between the receiver object, parameters, and the return value), and whether the method is modeled by STUBDROID. The table shows that all the framework methods used in these flows are modeled by STUBDROID and thus FLOWDROID is able to accurately track taint propagation of all these methods.

For app #21, the tool can detect four out of 12 expected flows. These four flows are from location-related source methods, i.e., `getLatitude()` and `getLongitude()`,

```
1 public class MainActivity extends Activity {
2   @Override
3   protected void onCreate(Bundle
        savedInstanceState) {
4     ...
5     pwd = getText().toString(); // source
6
7     HttpPost req = new HttpPost();
8     req.setEntity(new StringEntity(pwd));
9
10    HttpClient httpClient = ...;
11    httpClient.execute(req); // sink, leak
12
13    String uri = new Builder()
14    .appendQueryParameter("password", pwd)
15    .build().getEncodedQuery();
16
17    BufferedWriter writer = ...;
18    writer.write(uri); // sink, leak
19  }
20 }
```

Fig. 22: U-DB2.12 – Android framework method modeling (FLOWDROID does not propagate taint from a parameter to the receiver object and to the return value).

to the sink method `sendTextMessage(...)`, at different call sites. The flows follow the same pattern shown in Fig. 21: a location obtained via a call to a source method is converted to a String and appended to a StringBuilder object (line 7). The StringBuilder is further converted to a String (line 8) and leaked via an SMS through the `SmsManager::sendTextMessage(...)` sink method (line 9).

Like in the case of app #13, the flows are rather simple and the only challenge involved in detecting these flows is to be able to track taint propagation in Android framework methods (U-DB2.12). The framework methods used in these flows are listed in the last three rows of Table 13. Here, again, all methods are modeled by STUBDROID and thus FLOWDROID is able to accurately detect them. Next, we discuss the reasons for FN and FP results.

5.3 Causes of Inaccuracy

The FN and FP columns of Table 12 show the detailed distribution of the false negative and false positive results for the apps FLOWDROID analyzed in full. Similar to the benchmarks, we assign a failing criterion for each FN and FP result: for each failure, we first checked whether the failure occurs due to a reason already captured by a certain target criterion from the benchmark apps. If so, we marked the failure with that target criterion. Otherwise, we created a new target criterion (U-DB4.17, U-DB4.18, U-DB10.12) and also added a corresponding benchmark application to *UBCBench*.

FLOWDROID missed 22 flows in six apps, #11, #20, #21, #22, #23, and #25, due to the incorrect modeling of Android framework methods. We mark this failure with the U-DB2.12 target criterion, as in the benchmark apps. Fig. 22 shows a snippet of code producing such FN result: in this case, an Android framework method `HttpPost.setEntity(HttpEntity)` is used to set the content of an HTTP request (line 8). It is parameterized with an `StringEntity` object, which is tainted because the `pwd` field is tainted (line 5). As the framework method is not supported by STUBDROID and is not explicitly classified in any of the conservative framework method modeling strategies described in Section 3.1,

```
1 public class MainActivity extends Activity {
2   String pwd = "";
3   @Override
4   protected void onCreate(Bundle
        savedInstanceState) {
5     ...
6     this.pwd = getText().toString(); // source
7     String name = this.getString(R.string.
          app_name);
8
9     PrintWriter writer = ...;
10    writer.write(name); // sink, no leak
11  }
12 }
```

Fig. 23: U-DB2.12 – Android framework method modeling (FLOWDROID mistakenly propagates taint from a parameter's field to the return value).

FLOWDROID applies the *default* strategy: when a receiver object or one of its fields is tainted, the taint is propagated to the method return value and its fields. As the default strategy does not propagate the taint from the method parameter to the receiver object `req`, the tool misses the flow to the `HttpClient.execute(...)` sink (line 11). This led to FLOWDROID missing one flow in app #11.

Similarly, in line 14, Android framework method `Uri$Builder.appendQueryParameter(String,String)` is used to add a query string to an URI. The return value `uri` should be tainted as it contains the tainted `pwd` string. Again, FLOWDROID applies the *default* strategy on this method, where the taint is not propagated from the method parameters to the return value. As a result, the tool also misses the flow to the `BufferWriter.write(...)` sink (line 18). This led to FLOWDROID missing the remaining 21 flows in apps #20, #21, #22, #23, and #25. One way to fix these failures is to assign the *generation* type to both methods, which would propagate the taint from the method parameter to the receiver object `req` (line 8) and the return value `uri` (line 13).

Another case of framework method misclassification, this time leading to two FP results (both in app #13) is exemplified in Fig. 23. Here, `Context.getString(int)` is an Android framework method that allows developers to retrieve a string resource, e.g., the app name (line 7). The `pwd` field of the method's receiver object, `this`, is tainted (line 6). However, this method is, again, not supported by STUBDROID as it contains native code and is not explicitly classified in any of the conservative framework method modeling strategies. Applying the *default* strategy in this case propagates the taint to the method's return value, which is further assigned to the variable `name` (line 7). The tainted `name` variable flows to the sink `PrintWriter.write(...)` (line 10). Yet, this is an FP result as the return string of the `Context.getString(int)` is not, in fact, affected by the tainted value. To fix the failure, this method should have been assigned to the *exclude* type, which would not propagate the taint.

While FLOWDROID has no failure of this kind on the benchmark apps, Google Play apps contain Android framework methods not covered by the benchmarks, which expose additional weaknesses and the need to tune FLOWDROID to better represent the entire range of Android framework methods. Moreover, even though propagating the taint from a method parameter to the method receiver (like in Figure 22) can be solved by FLOWDROID's *generation* conservative modeling strategy, none of the tool's strategies

```
1 public class MainActivity extends Activity {
2   int id = R.layout.activity_main;
3   @Override
4   protected void onCreate(Bundle
        savedInstanceState) {
5     ...
6     setContentView(id);
7   }
8   // Callback defined in a layout file
9   public void buttonClick(View view){
10    String pwd = getText().toString(); //
          source
11    PrintWriter writer = ...;
12    writer.write(pwd); // sink, leak
13  }
14 }
```

Fig. 24: U-DB4.17 – Missing callbacks.

supports propagating taint between explicit method parameters. Such behavior is required, for example, for the `android.database.DatabaseUtils::appendValueToSql(StringBuilder sql, Object value)` framework method. FLOWDROID framework modeling strategy needs to be refined to support this case; interestingly, such modeling pattern is supported by AMANDROID. That is, for framework methods that are not accurately modeled by the tools, FLOWDROID and AMANDROID apply different types of conservative strategies, both with their own strengths and weaknesses.

Besides framework-modeling-related failures, three additional FNs, in apps #6, #9, and #10, relate to callback method analysis. Fig. 24 shows a simplified version of the code leading to the failure marked as U-DB4.17. In this code snippet, `R.layout.activity_main` is a unique identifier of a file defining the UI elements and their corresponding callback methods for an activity. The identifier is assigned to a variable `id` (line 2); that variable is further used in `onCreate(...)` method to set the layout for the activity (line 6). The layout file defines a callback method `buttonClick()`, which is called when a button is clicked. The implementation of this method contains a flow from the source `pwd` (line 10) to the sink `PrintWriter.write(...)` (line 12).

To detect the expected flow, FLOWDROID needs to detect the callback method first. It does that by identifying and parsing all layout resource files passed as a parameter to `setContentView(int)`. However, we observed that FLOWDROID can only handle layout files when the value of the parameter is a primitive integer rather than a variable. Unlike in this app, all benchmarks and several Google Play apps use `R.layout.activity_main`, which is a *final static* variable that is replaced by its integer value at compile time, directly as an input to the `setContentView(int)` method. FLOWDROID does not perform any analysis on the app to retrieve the value of the `id` variable and thus cannot detect the layout file. As a result, it misses the callback and its related flows.

Moreover, while FLOWDROID models the lifecycle methods of fragments – reusable UI parts defined within an Activity – it does not collect callbacks dynamically registered in fragments' lifecycle methods. For example, the `onClick()` callback method in Fig. 25 (line 9) is dynamically registered in the fragment's `onCreateView(...)` lifecycle method (line 6). FLOWDROID is unable to collect this callback method and considers it as unreachable. As a result, it cannot identify the flow inside this callback method: from the source `getText()` (line 10) to the sink `PrintWriter.write(...)` (line 12).

```
1 public class MyFragment extends Fragment
      implements View.OnClickListener {
2   @Override
3   public View onCreateView(LayoutInflater
        inflater, ViewGroup container, Bundle
        savedinstanceState) {
4     ...
5     Button button = ...;
6     button.setOnClickListener(this);
7   }
8   @Override
9   public void onClick(View v) {
10    String pwd = getText().toString(); //
          source
11    PrintWriter writer = ...;
12    writer.write(pwd); // sink, leak
13  }
14 }
```

Fig. 25: U-DB4.18 – Callback methods registered in fragments.

```
1 public class MyClass {
2   public MyClass() {
3     String pwd = getText().toString(); //
          source
4     PrintWriter writer = ...;
5     writer.write(pwd); // sink, leak
6   }
7 }
8 public class MainActivity extends Activity {
9   @Override
10  protected void onCreate(Bundle
        savedInstanceState) {
11    ...
12    Class<?> cls = Class.forName("MyClass");
13    Constructor cstr = cls.getConstructor();
14    MyClass obj = (MyClass) cstr.newInstance(
          this);
15  }
16 }
```

Fig. 26: U-DB10.12 – Reflective invocation of class constructors.

We marked this issue as U-DB4.18 in Table 12, which leads to one FN (in app #3). This is a serious limitation: while approximately 90% of the current Google Play apps contain at least one fragment [73], *DroidBench* only contains two benchmark apps (*FragmentLifecycle1* and *FragmentLifecycle2*) focusing on fragment lifecycle method modeling.

Another source of failures leading to two FNs, in apps #15 and #17, is due to using reflective calls to invoke class constructors. Fig. 26 shows an example where class `MyClass` (lines 1-7) is instantiated via reflection (lines 12-14). Because FLOWDROID cannot handle the reflective method `Class.getConstructor()`, it cannot reach the constructor method and misses the flow inside that method. We mark this issue as U-DB10.12.

We identified this issue while analyzing FN results in applications using a framework called *ButterKnife* [74], which simplifies the code for registering callbacks of UI elements. *ButterKnife* automatically generates a helper class for each Activity and relies on reflective calls similar to the one in Fig. 26 to invoke the constructor of the helper class and to perform callback registrations. As *ButterKnife* is used by many Google Play applications, lack of support for reflective constructors hinders the applicability of FLOWDROID in practice.

Finally, 16 additional FNs for FLOWDROID, in apps #4, #20, #23, and #25, are due to a reason we already observed in benchmarks: lack of support for `SharedPreferences` (DB8.16).

```
1 public class MyClass {
2   public MyClass() {
3     String pwd = getText().toString(); //
          source
4     PrintWriter writer = ...;
5     writer.write(pwd); // sink, leak
6   }
7 }
8 public class MainActivity extends Activity {
9   @Override
10  protected void onCreate(Bundle
        savedInstanceState) {
11    ...
12    Constructor cstr = findConstructor();
13    MyClass obj = (MyClass) cstr.newInstance(
          this); // Missed by DroidRA
14  }
15
16  public Constructor findConstructor() {
17    Class<?> cls = Class.forName("MyClass");
18    Constructor cstr = cls.getConstructor();
19    ...
20    return cstr;
21  }
22 }
```

Fig. 27: U-DB10.12 – Class constructors returned by a method.

DroidRA does not help improve FlowDroid's accuracy for two reflection-related FNs (in apps #15 and #17), which are caused by using the reflective methods `Class.getConstructor()` and `Constructor.newInstance()`. Interestingly, when calls to these methods are located within the same host method, like in our example in Fig. 26, DroidRA can successfully identify the source and target of reflective calls. However, in apps #15 and #17, calls to these reflective methods are located in two *different* host methods, like in the example in Fig. 27. Here, `Class.getConstructor()` is called inside the method `findConstructor()` in line 18. This method returns the Constructor object, which is further assigned to the variable `cstr` in line 12 of the method `onCreate()`. The `cstr` variable is then used in `Constructor.newInstance()` in line 13 of `onCreate()`. In this case, DroidRA fails to replace the reflection call to `MyClass()` invoked in line 13 with a direct call. As a result, FlowDroid still misses the tainted flow in lines 3-5. We add an additional benchmark to *UBCBench* to cover this case.

Summary. Our experiments identified several major issues not covered by existing benchmarks, such as flakiness of Amandroid on several large-scale apps. For FlowDroid, main issues include inaccurate handling of Android framework methods, callbacks, fragments, and reflective calls. DroidRA did not help resolve missed reflection cases due to its limited support for reflective invocation of class constructors. In addition, the applicability of existing tools for real-world applications is hindered by a large number of runtime exceptions, timeouts, and out-of-memory errors. That is, the high accuracy of the tools on the benchmark apps does not translate to a high accuracy on Google Play apps; our experience shows that none of the tools can be used to reliably detect flows in Google Play apps.

6 Limitations and Threats to Validity

The main threat to the validity of our results stems from the manual analysis we performed to identify the expected results for each benchmark and Google Play app, the FP and FN flows in each tool, and the causes of these flows. We mitigated this threat in three ways: first, two authors of this paper performed the analysis independently and then cross-checked each other's results. Second, we constructed and ran our own test cases, to confirm each hypothesis for a possible cause of a FP/FN result. We also reached out to the authors of each tool to resolve cases where a definite conclusion could not be reached. Finally, we shared a draft of this report with all tool authors to obtain their feedback and addressed all feedback we receive at the time of writing.

As the set of sources and sinks in DroidSafe is not configurable without code modifications, we ignored flows detected by DroidSafe when these flows did not involve sources and sinks in our set. While this part of the process was automated, we could have mistakenly missed important flows. Again, we mitigated this threat by independently inspecting the results by at least two authors of the paper. Yet, we acknowledge that running the tool with an extended set of sources and sinks could alter its behavior.

Finally, as DroidSafe does not support applications above API level 19 and *ICC-Bench* applications require API level 25, we did not report DroidSafe results on *ICC-Bench*. Moreover, limited API support of Amandroid constrained the range of Google Play apps we tested the tool on. As we only run each tool on apps it can support, we believe the experiment is valid. Yet, performing our analysis on apps with API level < 25 is a limitation of our approach. As the issues that we identified, e.g., the limited support of reflection and framework modeling, are not specific to apps with these API levels, they are likely to occur in apps with higher API levels as well. We thus believe our results generalize.

7 Discussion and Lessons Learned

The absence of accurate information on tool configuration as well as sources, sinks, and benchmarks apps used for the analysis, hinders the replicability of earlier studies. When each of the analyzed tools was introduced, it reportedly outperformed the others. That is because the tools were evaluated under different configuration setups, with different sets of sources and sinks, and only on a selected subset of benchmarks. To further add to the confusion, as our experience analyzing ICC communication shows, the tools also have different interpretations of Android-specific flow semantics. Comparing the tools to each other under the same setup helps identify their strengths and weaknesses, e.g., in Android framework method modeling approaches implemented by FlowDroid vs. Amandroid.

Our experience evaluating the tools on Google Play apps shows that the high accuracy on the benchmarks does not necessarily translate to a high accuracy on Google Play apps. While the tools are tuned to perform well on the benchmark apps, Amandroid largely fails to produce reliable results on real apps and FlowDroid misses cases not covered by the benchmarks, e.g., certain types of reflective calls, callbacks, and Android framework methods. Our study helped identify several such cases. We conclude that evaluations of the tools on real applications, with a detailed analysis of each failure, as we did in Section 5.3, is a difficult yet productive direction for future work, which will help improve tools' applicability in practice.

Furthermore, based on our study, we observed that the current way of measuring tools accuracy is sub-optimal: as multiple FP and FN flows might be caused by the same underlying reason, just counting FP and FN flows can produce false impressions with respect to the real tool accuracy. In fact, such counting might produce accuracy metrics that look artificially low, even though the tool outperforms others in many aspects. To mitigate this problem, we suggest to (a) simplify the benchmarks and ensure they focus, as much as possible, on one particular aspect of the analysis and (b) investigate reasons behind each failure, as we did in Sections 4.3 and 5.3.

In our work, we separated tangled benchmarks and extracted new benchmarks representing issues found in the analyzed Google Play apps. We included newly identified benchmarks in *UBCBench* and augmented this suite with tests of different types of sensitivities. We clearly articulated potential reasons for failures in each of the benchmarks and made our setup and experimental data publicly available for others to build on.

We hope our work will benefit tool builders, as it highlights some of the main factors impeding the successful adoption of the tools in practice: scalability, handling of callbacks and reflections, and accurate support of Android framework methods. We also hope our work will help tool users to better interpret the results of analysis and tune the tools for their needs.

8 Related work

Our discussion of related work focuses on the (a) internal evaluation of the tools we selected for our study, (b) external evaluation of the tools, and (c) surveys on Android static taint analysis tools.

Internal tool evaluation. We start by discussing the evaluation of the tools performed by their authors. Arzt et al. [2] compared the precision and recall of FlowDroid with two commercial tools, IBM AppScan Source [75] and Fortify SCA [76] on *DroidBench* version 1.0, which contained 39 benchmarks at that time. They provided a detailed kit explaining how to reproduce their experiments [77]. The authors also ran FlowDroid (without IccTA) on 500 most popular Google Play applications and 1,000 known malware apps from the VirusShare dataset [78]. They concluded that FlowDroid was able to detect leakages of sensitive information into log files. However, the authors did not report on the number of apps the tool successfully analyzed and the number of flows the tool reported per app. They also did not classify all flows into FP and FN results.

Li et al. [4] conducted a comparison between FlowDroid without IccTA, FlowDroid with IccTA, DidFail [79], and Amandroid on 22 ICC-related benchmarks the authors developed and nine benchmarks from *ICC-Bench*. The authors also ran FlowDroid (with IccTA) on 15,000 randomly selected Google Play applications and showed that the tool reported 2,395 ICC-related leakages in 337 apps. Like in the previous study, the authors did not further check the validity of the reported flows. When comparing the execution time of FlowDroid (with IccTA) and Amandroid on 50 randomly selected Google Play applications, the authors showed that FlowDroid was faster than Amandroid, which concurs with our findings.

Wei et al. [5] compared Amandroid with Epicc [1] and FlowDroid, also focusing on ICC handling ability. The subject programs consisted of 39 applications from *DroidBench*, 16 applications from *ICC-Bench*, and another four proprietary test cases. Later, the same authors upgraded the tool and conducted a new comparison [11], focusing on 18 *DroidBench* and 24 *ICC-Bench* applications related to ICC and IAC. In these two reports, the authors also evaluated Amandroid on 753 and 2,300 Google Play apps, respectively, showing that the tool is able to detect data leakages in practice. Again, they did not provide an in-depth analysis of the validity of the flows reported by the tool.

Gordon et al. [6] compared DroidSafe with FlowDroid using 94 applications from *DroidBench*, 40 additional benchmarks developed and contributed to *DroidBench* by the team, and 24 applications from a proprietary benchmark. The authors did not run DroidSafe on any Google Play apps, stating the tool is not designed for this purpose.

Finally, Li et al. [19] and Sun et al. [20] compared FlowDroid with and without DroidRA on 4 reflection-related apps from *DroidBench* version 2.0 and 9 apps developed by the authors. The authors also evaluated the tools on 100 real-world applications, but did not analyze the correctness of the detected flows. Our work extends this evaluation to consider the combination of DroidRA with Amandroid and DroidSafe as well; we also perform a detailed analysis of the detection results.

A detailed comparison with all these works is presented in Section 4.2. In a nutshell, executing the tools on different subsets of benchmark applications and with different versions of the tools makes the results incomparable. Moreover, the detailed tool configuration setup used in these experiments is rarely described. Experiments with Google Play apps focus on evaluating the scalability rather than the accuracy of the tools. In our work, we make a particular effort to align the tool configurations and also focus on a larger common set of benchmarks. As our team was not involved in developing these tools, we provide an independent assessment of their accuracy. Furthermore, we perform an in-depth analysis to investigate the reasons leading to FP and FN results on both the benchmark and Google Play apps for each tool.

External tool evaluation. The closest to ours is the work by Pauck et al. [16], which proposed a framework for automatically evaluating reports produced by Android taint analysis tools and used the framework for comparing the accuracy and execution time of six static taint analyzers (including FlowDroid, Amandroid, and DroidSafe). The authors evaluated the tools on *DroidBench* version 3.0, *ICC-Bench* version 2.0, and the *DIALDroid-Bench* [80] benchmark consisting of 30 apps collected from Google Play in June 2015. For the *DIALDroid-Bench* apps, the authors manually analyzed all flows reported by at least two tools and determined whether these flows are TP or FP results; they identified 22 TP and 4 FP flows, which formed the baseline for their experiment. Unlike us, the authors ran each tool with its default configuration. They also did not analyze the reasons for failures in details, untangle the benchmarks, or propose additional new benchmarks. Moreover, in our

work, we manually identified the expected analysis result, which allowed us to include many expected flows that are missed by all tools. Our analysis thus identified new failures missed by earlier work, such as those related to modeling Android framework methods, handling reflective constructor calls, callbacks in fragments, and more.

Bonett et al. [17] proposed a framework that leverages mutation analysis to discover undocumented flaws of Android static taint analysis tools. The authors used the framework to generate and inject 2,026 mutants (i.e., data leakages) to seven open-source Android apps, showing that DROIDSAFE can detect the highest number of leakages, followed by FLOWDROID and AMANDROID. The authors further showed that all tools miss flows due to inaccurate handling of callback methods and FLOWDROID and AMANDROID further have issues related to multi-threading. Our work differs as we focused on real rather than injected leakages, which allowed us to identify issues such as unsupported reflective constructor calls used by the *ButterKnife* framework.

Rodriguez and Kouwe [18] ran FLOWDROID, AMANDROID, and DROIDSAFE on 48 benign and 48 malicious apps, randomly selected from AndroZoo [65] – a repository that crawls apps from Google Play and alternative markets. The authors observed a large number of crashes and timeouts when running the tools, looked into the reasons for such crashes, and concluded that AMANDROID and DROIDSAFE have de-compilation issues causing these tools to miss Android APIs. Yet, some of these failures are expected, as the tools do not support the newest Android API levels. In our work, we ran the tools only on apps with supported API levels.

Luo et al. [12] aimed at providing insights into what can be done to increase the precision of taint analysis. They proposed a tool, named COVA, to compute partial path constraints that capture the circumstances under which taint flows may actually occur in practice. The authors further used the tool to conduct a qualitative study on taint flows reported by FLOWDROID on Google Play apps. They showed that many flows occur only under specific conditions, e.g., environment settings, user interaction, I/O. Unlike our work, the authors did not conduct a comparative analysis evaluating multiple tools, focusing on the recommendations of including such path constraints in future taint analysis approaches.

Tiwari et al. [81] compared the accuracy of FLOWDROID (with and without ICCTA), AMANDROID, DIDFAIL [79], IBM AppScan Source [75], DIALDROID [80], and the authors' own tool, IIFA, on a large set of ICC- and IAC-related benchmarks. The authors also ran the tools on 90 Google Play apps. They reported that IIFA is the only tool that supports IAC analysis of apps with an API level above 19, while other tools either fail or crash. Our analysis does not focus on ICC/IAC flows in particular. Like Tiwari et al., we also observe limited applicability of the tools for Google Play apps. However, unlike Tiwari et al., we manually built the set of expected flows, which allowed us to identify reasons for undetected and incorrectly detected flows.

Surveys. Several papers survey Android-specific static analysis techniques for identifying permission and privacy leakage detection [82], robustness against obfuscation [83], and intra- and inter-application communication vulnerabilities [35]. Sufatrio et al. [84] provide a taxonomy of security vulnerabilities in Android. Li et al. [41] and Sadeghi et al. [85] perform large-scale systematic literature reviews on static analysis tools for Android. They also propose a taxonomy, classifying Android security assessment mechanisms and research approaches.

Apart from the conceptual surveys discussed above, Reaves et al. [15] also augment their survey with an empirical experiment comparing the surveyed static analysis tools. This work evaluated usability, performance, and precision of the tools for *DroidBench* applications, mobile bank applications, and the top 10 most popular financial applications in Google Play. The primary emphasis of this experiment is on investigating the usability and accessibility of the tools, the understandability of the results, and the usefulness of the documentation provided by the tools. They reported on the number of applications that each tool was able to process, without a detailed analysis of the accuracy of each tool. Moreover, experiments with different tools were conducted using the default configurations of those tools, which hinders the validity of the results. In contrast, our study starts by aligning configurations of the tools and reports not only the overall accuracy for each tool but also the reasons for all failures, summarizing the main weaknesses of each tool. To the best of our knowledge, our study is the first large-scale, in-depth, comparative analysis of these tools under the same setup. Moreover, our configuration and results are publicly available, to allow reproducibility.

9 CONCLUSION

This paper reports on the results of a large-scale, controlled, and independent experiment we conducted to evaluate the most prominent static taint analysis tools for Android applications: FLOWDROID, AMANDROID, and DROIDSAFE. We also augmented each tool with the additional reflection handling mechanism implemented by DROIDRA and evaluated its effectiveness. We aligned the tools along the same configuration setup, used them to analyze 182 benchmark applications, compared our results to those reported in earlier work, and discussed the identified differences. We also extracted expected flows from 25 Google Play applications and used FLOWDROID and AMANDROID to analyze these applications as well. We observed that AMANDROID cannot reliably analyze large real applications, reporting flaky results. For FLOWDROID, we inspected all reported false-positive and false-negative flows to identify the main reasons for inaccuracy. Furthermore, we inspected true-positive flows, to identify the successful analysis cases and to discuss the difference between them and the failing scenarios. Finally, we identified several deficiencies in existing benchmarks, such as missing checks and dependent checks that are encoded in a single benchmark app.

Our work emphasizes the importance of providing detailed information about an experimental setup, which is required to ensure the correctness and reproducibility of reported comparisons. We make our entire configuration setup, including the tool configuration parameters, the set of sources and sinks, the precise version of each benchmark application used, the list of selected Google Play applications, and our expected results available to other researchers.

ACKNOWLEDGMENTS

We thank the authors of FLOWDROID, ICCTA, AMANDROID, DROIDSAFE, and DROIDRA for answering our questions about tool configurations and for providing constructive suggestions on the setup of our experiments. We also thank the anonymous reviewers for their constructive feedback, which helped us to improve the manuscript.

REFERENCES

[1] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in Android: An essential step towards holistic security analysis," in *Proc. of USENIX Security'13*, 2013, pp. 543–558.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proc. of PLDI'14*, 2014, pp. 259–269.

[3] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual API dependency graphs," in *Proc. of CCS'14*, 2014, pp. 1105–1116.

[4] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting inter-component privacy leaks in Android apps," in *Proc. of ICSE'15*, 2015.

[5] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proc. of CCS'14*, 2014, pp. 1329–1341.

[6] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of Android applications in DroidSafe," in *Proc. of NDSS'15*, 2015.

[7] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, "What the app is that? deception and countermeasures in the Android user interface," in *Proc. of S&P'15*, 2015, pp. 931–948.

[8] Y. Fratantonio, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna, "CLAPP: Characterizing loops in Android applications," in *Proc. of ESEC/FSE'15*, 2015, pp. 687–697.

[9] *DroidBench* benchmark suite, https://github.com/secure-software-engineering/DroidBench/tree/develop, 2017.

[10] *ICC-Bench* benchmark suite, https://github.com/fgwei/ICC-Bench, 2017.

[11] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," *ACM Transactions on Privacy and Security*, 2018.

[12] L. Luo, E. Bodden, and J. Späth, "A qualitative analysis of Android taint-analysis results," in *Proc. of ASE'19*, 2019.

[13] F-Droid, https://f-droid.org/, 2020.

[14] Google Play, https://play.google.com/store, 2020.

[15] B. Reaves, J. Bowers, S. A. Gorski III, O. Anise, R. Bobhate, R. Cho, H. Das, S. Hussain, H. Karachiwala, N. Scaife, B. Wright, K. Butler, W. Enck, and P. Traynor, "*droid: Assessment and evaluation of Android application analysis tools," *ACM Computing Surveys*, vol. 49, no. 3, p. 55, 2016.

[16] F. Pauck, E. Bodden, and H. Wehrheim, "Do Android taint analysis tools keep their promises?" in *Proc. of ESEC/FSE'18*, 2018, pp. 331–341.

[17] R. Bonett, K. Kafle, K. Moran, A. Nadkarni, and D. Poshyvanyk, "Discovering flaws in security-focused static analysis tools for Android using systematic mutation," in *Proc. of USENIX Security'18*, 2018, pp. 1263–1280.

[18] S. A. Rodriguez and E. v. d. Kouwe, "Meizodon: Security benchmarking framework for static Android malware detectors," in *Proc. of CECC'19*, 2019.

[19] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "DroidRA: taming reflection to support whole-program analysis of Android apps," in *Proc. of ISSTA'16*, 2016, pp. 318–329.

[20] X. Sun, L. Li, T. F. Bissyandé, J. Klein, D. Octeau, and J. Grundy, "Taming Reflection: An Essential Step Toward Whole-Program Analysis of Android Apps," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, 2021.

[21] S. Arzt, "Static data flow analysis for Android applications," Ph.D. dissertation, Darmstadt University of Technology, Germany, 2017.

[22] DroidSafe Android static analysis source repository, https://mit-pac.github.io/droidsafe-src/, 2016.

[23] J. Zhang, L. Qiu, Y. Wang, and J. Rubin, "Supplementary materials," https://resess.github.io/PaperAppendices/StaticTaint/index.html, 2020.

[24] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe," in *Proc. of ISSTA'18*, 2018, pp. 176–186.

[25] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Foundations and Trends in Programming Languages*, vol. 2, no. 1, pp. 1–69, 2015.

[26] D. J. Pearce, P. H. Kelly, and C. Hankin, "Efficient field-sensitive pointer analysis of C," *ACM Transactions on Programming Languages and Systems*, vol. 30, no. 1, 2007.

[27] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, p. 236–243, 1976.

[28] L. Li, J. Gao, M. Hurier, P. Kong, T. F. Bissyandé, A. Bartel, J. Klein, and Y. Le Traon, "AndroZoo++: Collecting millions of Android apps and their metadata for the research community," *The Computing Research Repository*, vol. abs/1709.05281, 2017.

[29] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst, "Static analysis of implicit control flow: Resolving Java reflection and Android intents (T)," in *Proc. of ASE'15*, 2015, pp. 669–679.

[30] J. Jorgensen, "Improving the precision and correctness of exception analysis in Soot," McGill University, Canada, Tech. Rep. 2003-3, 2003.

[31] O. Lhoták and L. Hendren, "Scaling Java points-to analysis using SPARK," in *Proc. of CC'03*, 2003, pp. 153–169.

[32] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, "Using datalog with binary decision diagrams for program analysis," in *Proc. of APLAS'05*, 2005, pp. 97–118.

[33] G. Kastrinis and Y. Smaragdakis, "Efficient and effective handling of exceptions in Java points-to analysis," in *Proc. of CC'13*, 2013, pp. 41–60.

[34] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Proc. of CASCON'99*, 1999.

[35] S. Bhandari, W. B. Jaballah, V. Jain, V. Laxmi, A. Zemmari, M. S. Gaur, M. Mosbah, and M. Conti, "Android inter-app communication threats and detection techniques," *Computers & Security*, vol. 70, pp. 392–421, 2017.

[36] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to Android inter-component communication analysis," in *Proc. of ICSE'15*, 2015, pp. 77–88.

[37] D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein, and Y. Le Traon, "Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis," in *Proc. POPL'16*, 2016, pp. 469–484.

[38] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, and Y. Le Traon, "ApkCombiner: Combining multiple Android apps to support inter-app analysis," in *Proc. of SEC'15*, 2015, pp. 513–527.

[39] JNI tips, https://developer.android.com/training/articles/perf-jni, 2020.

[40] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "JN-SAF: Precise and efficient NDK/JNI-aware inter-language static analysis framework for security vetting of Android applications with native code," in *Proc. of CCS'18*, 2018, pp. 1137–1150.

[41] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. Le Traon, "Static analysis of Android apps: A systematic literature review," *Information & Software Technology*, vol. 88, pp. 67–95, 2017.

[42] Google Scholar, https://scholar.google.com, 2020.

[43] S. Arzt and E. Bodden, "StubDroid: Automatic inference of precise date-flow summaries for the Android framework," in *Proc. of ICSE'16*, 2016, pp. 725–735.

[44] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *Proc. of S&P'16*, 2016, pp. 377–396.

[45] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Reflection-aware static analysis of Android apps," in *Proc. of ASE'16*, 2016, pp. 756–761.

[46] Y. Li, J. Ouyang, K. Ma, S. Guo, and B. Mao, "Data flow analysis on Android platform with fragment lifecycle modeling and callbacks," *EAI Endorsed Transactions on Security and Safety*, vol. 4, 2017.

[47] S. Calzavara, I. Grishchenko, A. Koutsos, and M. Maffei, "A sound flow-sensitive heap abstraction for the static analysis of android applications," in *Proc. of CSF'17*, 2017, pp. 22–36.

[48] J. Zhang, C. Tian, and Z. Duan, "An efficient approach for taint analysis of android applications," *Computers & Security*, 2020.

[49] "HornDroid: Practical and sound static analysis of android applications by SMT solving," in *Proc. of EuroS&P'16*, 2016, pp. 47–62.

[50] Y. Zhang, Y. Li, T. Tan, and J. Xue, "Ripple: Reflection analysis for android apps in incomplete information environments," *Software: Practice and Experience*, pp. 1419–1437, 2018.

[51] G. Barbon, A. Cortesi, P. Ferrara, and E. Steffinlongo, "DAPA: Degradation-aware privacy analysis of android apps," in *Proc. of STM'16*, 2016, pp. 32–46.

[52] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, 2012, pp. 27–38.

[53] FlowDroid 2.7.1, https://github.com/secure-software-engineering/FlowDroid/releases/tag/v2.7.1, 2019.

[54] Argus-saf 3.2.0, https://bintray.com/arguslab/maven/argus-saf/3.2.0, 2018.

[55] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing Android sources and sinks," in *Proc. of NDSS'14*, 2014.

[56] S. Fahl, M. Harbach, T. Muders, Baumgärtner, B. Freisleben, and M. Smith, "Why Eve and Mallory love Android: An analysis of Android SSL (in)security," in *Proc. of CCS'12*, 2012, pp. 50–61.

[57] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. of S&P'12*, 2012, pp. 95–109.

[58] AAPT2, https://developer.android.com/studio/command-line/aapt2, 2020.

[59] Unity manual - Android, https://docs.unity3d.com/Manual/android.html, 2020.

[60] React Native, http://reactnative.dev/, 2020.

[61] E. Ouellet and I. McShane and A. Litan and P. Bhajanka, "Magic quadrant for endpoint protection platforms," https://www.gartner.com/doc/3848470/magicquadrant-endpoint-protection-platforms, 2018.

[62] Trend Micro, https://www.trendmicro.com/en_us/research.html, 2021.

[63] Symantec Enterprise Blogs, https://symantec-enterprise-blogs.security.com/blogs/, 2021.

[64] Naked Security, https://nakedsecurity.sophos.com/, 2021.

[65] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, "AndroZoo: Collecting Millions of Android Apps for the Research Community," in *Proc. of MSR'16*, 2016.

[66] APKPure, https://apkpure.com, 2020.

[67] APKCombo, https://apkcombo.com/en-ca/, 2020.

[68] VirusTotal, https://www.virustotal.com/home, 2020.

[69] R. Vallee-Rai and L. Hendren, "Jimple: Simplifying Java bytecode for analyses and transformations," McGill University, Canada, Tech. Rep. 1998-4, 1998.

[70] Toast overviews, https://developer.android.com/guide/topics/ui/notifiers/toasts, 2020.

[71] Android Studio, https://developer.android.com/studio, 2021.

[72] Jawa Language, http://pag.arguslab.org/jawa-language, 2021.

[73] D. Lai and J. Rubin, "Goal-driven exploration for Android applications," in *Proc. of ASE'19*, 2019.

[74] Butter Knife, https://github.com/JakeWharton/butterknife, 2020.

[75] IBM AppScan Source, https://www.ibm.com/us-en/marketplace/ibm-appscan-source, 2017.

[76] Fortify SCA, https://software.microfocus.com/en-us/solutions/enterprise-security, 2017.

[77] PLDI'14 artifact evaluation, https://github.com/secure-software-engineering/soot-infoflow-android/wiki/PLDI'14-Artifact-Evaluation, 2017.

[78] Virus Share, https://virusshare.com, 2020.

[79] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proc. of SOAP'14*, 2014, pp. 1–6.

[80] A. Bosu, F. Liu, D. D. Yao, and G. Wang, "Collusive data leak and more: Large-scale threat analysis of inter-app communications," in *Proc. of ASIA CCS'17*, 2017, pp. 71–85.

[81] A. Tiwari, S. Groß, and C. Hammer, "IIFA: Modular inter-app intent information flow analysis of android applications," in *Proc. of SecureComm'19*, 2019, pp. 335–349.

[82] S. Schmeelk, J. Yang, and A. V. Aho, "Android malware static analysis techniques," in *Proc. of CISR'15*, 2015, pp. 5:1–5:8.

[83] J. Hoffmann, T. Rytilahti, D. Maiorca, M. Winandy, G. Giacinto, and T. Holz, "Evaluating analysis tools for Android apps: Status quo and robustness against obfuscation," in *Proc. of CODASPY'16*, 2016, pp. 139–141.

[84] Sufatrio, D. J. J. Tan, T.-W. Chua, and V. L. L. Thing, "Securing Android: A survey, taxonomy, and challenges," *ACM Computing Surveys*, vol. 47, no. 4, pp. 58:1–58:45, 2015.

[85] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of Android software," *IEEE Transactions on Software Engineering*, vol. 43, no. 6, pp. 492–530, 2017.