Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe

Lina Qiu
University of British Columbia,
Vancouver, Canada
lqiu@ece.ubc.ca

Yingying Wang
University of British Columbia,
Vancouver, Canada
wyingying@ece.ubc.ca

Julia Rubin
University of British Columbia,
Vancouver, Canada
mjulia@ece.ubc.ca

ABSTRACT

Numerous static analysis techniques have recently been proposed for identifying information flows in mobile applications. These techniques are compared to each other, usually on a set of syntactic benchmarks. Yet, configurations used for such comparisons are rarely described. Our experience shows that tools are often compared under different setup, rendering the comparisons irreproducible and largely inaccurate. In this paper, we provide a large, controlled, and independent comparison of the three most prominent static analysis tools: FlowDroid combined with IccTA, Amandroid, and DroidSafe. We evaluate all tools using common configuration setup and on the same set of benchmark applications. 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.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; **Empirical software validation**;

KEYWORDS

Static analysis, information flow analysis, mobile, empirical studies

ACM Reference Format:
Lina Qiu, Yingying Wang, and Julia Rubin. 2018. Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3213846.3213873

1 INTRODUCTION

Numerous static taint analysis techniques have recently been proposed for identifying information flows in mobile applications [7, 11–13, 19, 24, 27, 37]. The authors of these tools compare them to each other, typically on a set of existing benchmarks for evaluating taint analysis techniques, such as DroidBench [1] and ICC-Bench [2]. Several surveys also offer independent qualitative and quantitative comparisons of the tools [21, 30].

ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3213873

Our experience, however, shows that the tools are often compared under different setups, making the comparison inaccurate. The goal of this work is, therefore, to conduct a controlled and independent study for comparing static taint analysis tools to each other, focusing specifically on the three most-popular open-source tools: FlowDroid combined with IccTA (FlowDroid+IccTA), Amandroid, and DroidSafe. We perform a large-scale experiment evaluating the tools on more than 130 benchmark applications, report on the results of our study, compare them with the results in earlier reports, and discuss 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 benchmarks 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: What are the accuracy and performance of the tools when compared under common configuration setup?
RQ2: Can we reproduce results of earlier experiments?
RQ3: What are the main strengths and weaknesses of each tool?

To answer RQ1, we extracted the configuration setup used in the empirical evaluation of FlowDroid+IccTA [19], Amandroid [37, 38], and DroidSafe [13]. We learned that none of the reports provide detailed information on all of the following: (a) the parameters used for configuring the tools; (b) the list of the sources and sinks used; (c) the exact set of apps (from the DroidBench and ICC-Bench benchmarks) selected for the evaluation.

For example, FlowDroid has 43 configuration parameters and DroidSafe has 57; the exact values of even the most significant of those parameters are not reported. While the DroidSafe authors provided precise information about the set of sources and sinks they used, they did not list the names of the benchmark applications used for evaluation. Both FlowDroid+IccTA and Amandroid listed the names of the benchmark applications; yet, we were unable to map those names to the applications in the current version of the benchmark due to naming inconsistency. We thus 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, and extended them with our own benchmark applications, UBCBench [23], which we developed to extract more accurate and detailed information.

For RQ2, our findings show that the accuracy of the tools in our experiment often differs from the results reported in earlier studies. For instance, the authors of Amandroid report an F-measure of

81% [37] and 96% [38] when executed on the DroidBench benchmark suite; yet, we were unable to reproduce these results, observing an F-measure of 68% for our experimental setup.

We manually analyzed all false positive (FP) and false negative (FN) results reported by each tool (RQ3). In a nutshell, we observed that FlowDroid+IccTA fails to accurately parse and track ICC Intents involving complex string analysis and list management operations; Amandroid does not accurately handle field sensitivity and location-related flows, and also overestimates ICC-related flows; and DroidSafe has problems reporting the correct entry method of the detected leaks and, as also reported by the authors, handling flow sensitivity.

To summarize, this paper makes the following contributions:

- It provides the first in-depth independent analysis that compares static taint analysis tools for Android applications [19, 30, 37, 38] under a similar setup and puts the results in 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.

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, selection of benchmarks, and our definition of expected results. Section 4 presents the results and Section 5 discusses lessons learned. In Section 6 we describe the limitations of our approach and threats to the experiment validity. We finalize the paper with the discussion of related work in Section 7 and conclusions in Section 8.

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 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 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 [34]. For the example in Listing 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 [34]. Specifically, the analysis qualifies the method local variables with the allocation site of the receiver object of the method call. For the example in Listing 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 [28, 34].

A *flow-sensitive* analysis takes the order of statements into account [40]. 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 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.

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

Listing 1: Context and Object Sensitivity

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 [31]. 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 [22]. 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. [8].

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 [15]. Some classic frameworks like Spark [18] and Paddle [39] 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 [16]. Other frameworks, such as Soot [36], remove unrealizable exception edges from the intraprocedural control flow graphs [15].

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 Android execution environment, tools typically either (a) conservatively assume that the return value of all framework methods is tainted if any of the arguments 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 [6].

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. Static analysis tools model this lifecycle to ensure correct propagation of taint. To extract the set of lifecycle methods and callbacks, 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 [10]. Tools like Epicc [27] and IC3 [26] extract ICC information, enabling ICC-aware information flow analysis. PRIMO [25] 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 [20] 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.

3 STUDY DESIGN

In this section, we describe our method for selecting and configuring the static taint analysis tools for the study. We also list the selected subject applications, sources and sinks that we used, and outline our specification of expected results.

Table 1: Selected Tools

Tool Name	Citations	Version
FlowDroid	547	2017-Oct-10
IccTA	129	
Amandroid	148	2017-May-07
DroidSafe	100	2016-Jun-22

3.1 Tool Selection

Li et al. [21] performed a systematic literature review and identified 38 static taint analysis tools for Android applications. From this list, we selected all open-source tools cited more than 100 times on Google Scholar[1] as of June 2017. The names and citation counts for the selected tools – FlowDroid, IccTA, Amandroid, and DroidSafe, are listed in the first two columns of Table 1. The last column of the table lists the release date of the latest publicly available version of each tool as of October 2017; we used this version in our experiments. 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, referring to it as FlowDroid+IccTA. 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 [7] is a flow-, context-, field-, and object-sensitive static analysis tool for Android applications, which is built on top of Soot [36] and Dexpler [9]. It precisely models the Android lifecycle and handles data propagation via callbacks of UI objects. It also provides a precise model of the most common Android framework methods. For the remaining ones, it applies a conservative strategy, tainting all parameters and return values of methods with at least one tainted parameter. To improve performance, FlowDroid allows users to manually specify packages and methods that they want to include/exclude for analysis [6]. 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 [7], but newer versions of FlowDroid are able to handle implicit flow [6]. Furthermore, FlowDroid conducts inter-procedural data flow analysis and thus cannot handle ICC or IAC.

IccTA [19] extends FlowDroid with the analysis of inter-component communication. It leverages existing ICC extraction tools, specifically, Epicc and IC3, augmenting them with application-level instrumentation of ICC-related methods for extracting precise ICC flows. It also leverages APKCombiner [20], 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 use FlowDroid+IccTA version 2.0 from October 2017[2].

Amandroid [37, 38] 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. Like FlowDroid, Amandroid precisely models a subset

[1] https://scholar.google.com
[2] https://github.com/secure-software-engineering/soot-infoflow-android/tree/FlowDroid_2.0

Table 2: Configuration Decisions of FlowDroid+IccTA, Amandroid, and DroidSafe

Configuration		Selected Configuration	FlowDroid +IccTA		Amandroid		DroidSafe	
			Default	Configurable	Default	Configurable	Default	Configurable
Sensitivty	field	✓	✓	yes	✓	no	✓	no
	context	✓	✓	yes	✓	yes	✓(static method only)	yes
	object	✓	✓	no	✓	no	✓	yes
	flow	✓	✓	yes	✓	no	×	no
	path	×	×	no	×	no	×	no
Implicit flows		×	×	yes	×	no	×	yes
Reflection		✓	×	yes	✓	no	✓	no
Exception		✓	✓	yes	✓	no	✓	yes
ICC		✓	×	yes	✓	no	✓	no
IAC		×	×	yes (ApkCombiner)	✓	no	×	yes
UI elements detection		×	✓	yes	-	no	-	no

of Android framework methods and applies a uniform conservative model for the remaining ones. 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 [37, 38]. In our study, we used version 3.1.1 of Amandroid released on May 07, 2017[3].

DroidSafe [13] implements an object-sensitive and flow-insensitive analysis. It builds a comprehensive Android execution model that contains analysis stubs for most of Android framework and native methods. This allows the tool to precisely track flows through Android APIs. 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 [13], but 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[4].

3.2 Configuration Parameters

Each of the tools provides numerous configuration parameters: FlowDroid+IccTA has 43 parameters, Amandroid has 11, and DroidSafe has 57. Because the results of the analysis largely depend on the tuning of each tool, comparing tools with different configuration setup is not meaningful.

We investigated the tool documentation and configuration setup used in previous studies [13, 19, 37, 38], 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+IccTA, 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.

[3] https://bintray.com/arguslab/maven/argus-saf/3.1.1
[4] https://github.com/MIT-PAC/droidsafe-src

1. Sensitivities. The first five rows of Table 2 list five types of sensitivity-related configurations, namely, field, flow, object, context, 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 (line 4) can only be configured in FlowDroid+IccTA; 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 (line 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. For both FlowDroid+IccTA and Amandroid, we used a context sensitivity depth of five; DroidSafe does not allow to set this parameter.

2. Implicit flows. FlowDroid+IccTA and DroidSafe support implicit flow tracking; this option is disabled by default in both tools. Amandroid does not document whether it supports implicit flow tracking or not. In our communication with the Amandroid authors, they confirmed that the tool cannot handle implicit flow. Therefore, we disabled implicit flow tracking for all tools (line 6).

3. Java-specific features. FlowDroid+IccTA, 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, and we enabled this option in FlowDroid+IccTA as well (line 7).

For FlowDroid+IccTA 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 (line 8).

4. Android-specific features. We configured FlowDroid+IccTA to use IC3 model for ICC flow extraction: according to earlier studies, this model has higher accuracy than Epicc [26]. We did not use PRIMO because it relies on statistical similarity between the

analyzed applications, which is absent in the benchmark suites. In addition, by default, the FlowDroid+IccTA 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+IccTA authors, they advised us to disable this functionality using `noiccresultspurify` parameter and evaluate the tool against the standard taint flow semantics. We thus switched the ICC purification off. Amandroid and DroidSafe both implement proprietary ICC handling mechanisms and do not provide any configuration options. We relied on the default behavior of these tools for our study (line 9).

FlowDroid+IccTA can detect inter-application flows when combined with ApkCombiner [20]. DroidSafe can also detect IAC flows, but that option is disabled by default. Amandroid is the only tool that enables inter-app leakage detection by default. As the benchmark suite we used contains only three IAC cases, we focused our study on intra-application leakages and disabled IAC tracking in all tools (line 10).

Finally, FlowDroid+IccTA is able to detect flows from sensitive UI elements, such as password fields. However, Amandroid and DroidSafe do not implement the corresponding feature. We thus disabled it in FlowDroid+IccTA (line 11).

5. Other configuration parameters. Upon reviewing additional configuration parameters, we observed that 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. Most of the remaining FlowDroid+IccTA 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. We left the default values for all these parameters.

3.3 Subject Applications

DroidBench and ICC-Bench are often used for comparing the tools [7, 13, 19, 30, 37, 38]. These benchmarks are publicly available; 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 DroidBench version 2.0 [1], which covers numerous categories of Android analysis problems, including intra-component and inter-component communication, handling of reflection, sensitivity types and more. We excluded 10 applications from this suite that focus on testing configuration options that we disabled, i.e., tracking of implicit flows, sensitive UI elements detection, and inter-application flow detection, arriving at 109 benchmark apps. Excluded benchmarks are: ImplicitFlow1-4, PrivateDataLeak1-2, SendSMS, StartActivityForResult1, Echoer, and VirtualDispatch4.

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 ICC-Bench v2.0 [2]. Our test suite thus consists of 133 applications in total from DroidBench (109) and ICC-Bench (24), covering 11 DroidBench categories and 4 ICC-Bench categories. The full list of the applications we analyzed is available online [23].

Table 3: A list of source and sink methods

	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)

Android SDK Version. ICC-Bench applications require Android version 7.1.1 and above (API level 25). We thus configured both FlowDroid+IccTA and Amandroid to work with this API level. However, the latest API level supported by DroidSafe is 19. We thus did not run DroidSafe 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.

3.4 Sources and Sinks

In previous studies, Arzt et al. [7] and Li et al. [19] used the list of sources and sinks generated by the SuSi project [29]; the authors of Amandroid confirmed that they used the sources and sinks marked in each of the benchmarks [38]. The list of sources and sinks in DroidSafe is not configurable without modifying the source code of the tool; Gordon et al. [13] thus ran the experiments with all sources and sinks hard-coded in the tool (4, 051 sources and 2, 116 sinks in total).

In our experiments, we inspected the headers and comments of all 133 benchmark applications and extracted the sources and sinks used in these applications (5 sources and 11 sinks listed in Table 3). We configured both FlowDroid+IccTA and Amandroid to use this unified list. For DroidSafe, we confirmed that these sources and sinks 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.

3.5 Expected Results

With our unified configuration setup, the expected flows for each benchmark application might deviate from the result specified by the benchmark designers, e.g., because we disabled implicit flows or 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

Table 4: Differences in expected results for DroidBench and ICC-Bench

Category	Benchmarks	# of Expected flows	
		Orig.	Ours
DroidBench			
1. Aliasing	Merge1	0	1
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
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

our configuration setup. To ensure validity, the manual analysis was performed independently and in parallel by two authors of this paper. Any observed differences were discussed in a meeting with all the authors, towards reaching a common resolution. In rare cases a resolution could not be achieved, we contacted the authors of the benchmarks and commonly identified the correct expected result.

A full list of all expected flows is available online [23]. Only in 21 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, `void startActivity(android.content.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, we excluded from our analysis benchmarks that were solely designed to check the features we disabled, e.g., *Implicit1-4* that examine implicit flows tracking.

3.6 Runtime Environment

We ran all analyses on Amazon AWS Ubuntu m4.4xlarge instance, with 16 vCPU, 64GB memory, and 2,000 Mbps bandwidth[5].

4 RESULTS

We now answer the research questions introduced in Section 1.

[5] https://aws.amazon.com/ec2/instance-types/

4.1 RQ1: What is the accuracy and performance of the tools when compared under common configuration setup?

To answer RQ1, we ran FlowDroid+IccTA, Amandroid, and DroidSafe, on the 133 benchmark apps using the configuration setup described in Section 3. Similar to the process of establishing the ground truth 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). The differences in the results of this analysis were, again, 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 $\frac{2 \times Precision \times Recall}{Precision + Recall}$.

The first row of Table 5 summarizes the precision, recall, and F-measure obtained in our experiments, which we report separately for DroidBench and ICC-Bench. We also extracted the corresponding numbers from previous experiments [13, 19, 37, 38], and list them in rows 2-5 of Table 5.

For performance evaluation, we measured the execution time of each tool on the DroidBench applications only, as DroidSafe does not run on ICC-Bench. We averaged execution time from five consecutive runs, and further averaged the execution time for all analyzed benchmarks. To ensure a fair comparison, we configured FlowDroid+IccTA and Amandroid to use the same set of sources and sinks that DroidSafe uses[6]. Our results show that all the analyzed tools processed benchmark applications within a relatively short time, with FlowDroid+IccTA being the fastest: 9 seconds, followed by Amandroid: 18 seconds, and DroidSafe: 137 seconds.

4.2 RQ2: Can we reproduce results of earlier experiments?

In our experiments, FlowDroid+IccTA achieves the highest accuracy on the DroidBench benchmarks (F-measure of 85%), followed by DroidSafe (80%[7]) and Amandroid (68%). Yet, Amandroid performs better on ICC-Bench: 92% vs. 87% for FlowDroid+IccTA[8].

Li et al. [19] report a much higher accuracy of 97% for FlowDroid+IccTA in their experiment. The difference stems from our study covering a substantially larger set of benchmarks (133 apps vs. 22 selected ICC-related apps from DroidBench and 9 apps from ICC-Bench). With more benchmarks, we observed more failures, i.e., those related to the intra-component flow detection, tracking

[6] The set of sources and sinks in DroidSafe cannot be configured, as discussed in Section 3.4.
[7] The adjustments we made to arrive at 92% accuracy are discussed in Section 4.3.
[8] We did not run DroidSafe on this benchmark, as discussed in Section 3.3.

Table 5: Comparisons of Precision, Recall, and F-measures

Source	Benchmark		Tool	Precision (%)	Recall (%)	F-measure (%)
Ours	DroidBench (109 apps from v2.0)		FlowDroid+IccTA	90	81	85
			Amandroid	67	69	68
			DroidSafe	76	85	80 (92)
	ICC-Bench (24 apps from v2.0)		FlowDroid+IccTA	100	76	87
			Amandroid	85	100	92
			DroidSafe	-	-	-
Li et al. [19]	DroidBench (22 apps developed by IccTA's authors) and ICC-Bench (9 apps)		FlowDroid+IccTA	97	97	97
			Amandroid	79	52	63
Wei et al. [37]	DroidBench (39 apps)		Amandroid	87	75	81
	ICC-Bench (16 apps)		Amandroid	100	100	100
Wei et al. [38]	DroidBench (18 apps, about ICC)		FlowDroid+IccTA	86	83	85
			Amandroid	96	96	96
			DroidSafe	85	96	90
	ICC-Bench (24 apps)		FlowDroid+IccTA	97	90	93
			Amandroid	97	100	98
			DroidSafe	10	3	5
Gordon et al. [13]	DroidBench	(94 apps from v1.2)	FlowDroid+IccTA	73	81	76
			DroidSafe	88	94	91
		(40 apps developed by DroidSafe authors)	FlowDroid+IccTA	79	35	48
			DroidSafe	100	100	100

Intents through list operations, handling of remote procedure calls (RPC), and more (see Section 4.3 for a detailed discussion of all observed failures).

The results for FlowDroid+IccTA reported by Wei et al. [38] on DroidBench are comparable with ours (85% in both cases). Yet, the reported accuracy on ICC-Bench is slightly higher than the accuracy that we observed in our experiments (93% vs. 87%). The differences are mainly related to the changed *Intent* matching mechanism introduced in the FlowDroid+IccTA version that we used. We also observed several tool crashes when building call graphs involving RPC communications.

Gordon et al. [13] reported a lower accuracy for FlowDroid+IccTA than the results obtained in our experiments (76% for 94 DroidBench apps from v1.2 and 48% for the 40 apps DroidSafe team developed vs. 85% for DroidBench apps in our experiments). In addition to the differences in the experimental setup that include different sets of benchmark apps and source / sink selection, that work evaluated an older version of FlowDroid+IccTA. In fact, the results reported by Gordon et al. in 2015 are consistent with our evaluation of the FlowDroid+IccTA version from 2015-May-31 [23], implying that the accuracy of FlowDroid+IccTA improved over time.

For Amandroid, our results on ICC-Bench, i.e., an F-measure of 92%, are lower but comparable with the reports of Wei et al. in both experiments: 100% and 98% in [37] and [38], respectively. The differences mainly stem from additional apps exposing Amandroid's overestimation of ICC flows and inaccurate handling of field sensitivity. Yet, the results on DroidBench differ substantially: in our study, Amandroid achieved the F-measure of 68%, compared with 81% and 96% in previous reports. As our study covered a much larger set of DroidBench apps (109 in our case vs. 39 and 18 in previous reports), more failures occurred, including those mentioned above. Our results are consistent with 63% accuracy for Amandroid reported by Li et al. [19].

Finally, for DroidSafe, we could not directly reproduce the results reported in the original experiments (91% on 94 DroidBench apps from v1.2 and 100% on the 40 apps the DroidSafe team developed), arriving at an accuracy of 80%. The differences are mainly due to DroidSafe's strategy for reporting flow sources. Adjusted for flow sources, we arrived at 92% accuracy, consistent with the results reported by the authors. A detailed description of the flow source problem and other failures of the tool is presented in the next section.

To summarize, we were unable to reproduce most accuracy-related results reported in previous experiments. In some cases, the differences can be explained by us using a newer and more accurate version of a tool. However, in the majority of cases, the differences relate to the narrow selection of the benchmark applications, configuration choices, and the considered sets of sources and sinks, leading to the underestimation of failures in previous studies.

With respect to the performance evaluation, our results are consistent with other reports showing that FlowDroid+IccTA is faster than Amandroid, which is, in turn, faster than DroidSafe [13, 19]. We do not perform 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 different configuration selections.

4.3 RQ3: What are the main strengths and weaknesses of each tool?

Each of the DroidBench and ICC-Bench benchmark applications is designed to test a particular aspect of taint analysis, which we

Table 6: FP/FN breakdown by goals, at benchmark level

AppCategory	Tool	# of FP	FP-breakdown	# of FN	FN-breakdown
DroidBench	FlowDroid +IccTA	10	1x DB3.1; 1x DB3.2; 1x DB3.5; 1x DB3.6; 1x DB4.14; 1x DB6.7; 1x DB7.4; 1x DB7.19; 1x DB7.20; 1x DB7.21;	20	1x DB2.8; 1x DB2.10; 1x DB7.8; 1x DB7.11; 1x DB7.13; 1x DB7.14; 1x DB8.2; 1x DB8.3; 1x DB8.6; 1x DB8.8; 2x DB8.16; 1x DB9.5; 1x DB9.11; 1x DB9.14; 1x DB9.17; 1x **UBC1**; 3x UN
	Amandroid	37	1x DB3.1; 1x DB3.2; 1x DB3.5; 1x DB3.6; 4x DB4.11; 4x DB4.12; 1x DB4.14; 14x DB6.1; 1x DB7.4; 1x DB7.19; 1x DB8.2; 3x **UBC3**; 10x **UBC2**; 3x UN	34	1x DB2.7; 1x DB2.8; 1x DB3.3; 1x DB3.4; 1x DB4.4; 1x DB4.5; 2x DB4.13; 1x DB7.5; 1x DB7.8; 1x DB7.10; 1x DB7.14; 2x DB8.16; 1x DB8.17; 1x DB9.5; 1x DB9.6; 1x DB9.7; 1x DB9.8; 1x DB9.11; 1x DB9.17; 1x DB10.3; 1x **UBC1**; 10x **UBC2**; 1x UN
	DroidSafe	28	1x DB3.1; 1x DB3.2; 1x DB3.5; 1x DB3.6; 2x DB4.12; 1x DB4.14; 2x DB6.4; 1x DB6.7; 1x DB7.4; 16x **UBC4**; 1x UN	15	1x DB2.6; 13x **UBC4**; 1x UN
ICC-Bench	FlowDroid +IccTA	0	none	8	1x ICC2.4; 1x ICC2.5; 6x UN
	Amandroid	6	3x DB6.1; 1x ICC2.2; 2x **UBC3**	0	none
	DroidSafe	NA	NA	NA	NA

refer to as the *target criterion* of the benchmark. For example, the *AccessArray1* benchmark application tests whether the analysis distinguishes between different array positions. We mark this criterion with the index of its corresponding benchmark, i.e., DB3.1[9]. A particular benchmark application can evaluate multiple target criteria, e.g., both reflection and location handling. Moreover, we identified four additional failure criteria not covered by existing benchmarks. We marked them UBC1-UBC4 and added the corresponding benchmark applications to UBCBench. In total, we extracted 129 target criteria from our 133 benchmark applications.

Each benchmark application contains none to multiple expected flows. In our suite, there are 142 expected flows – 108 for 109 DroidBench apps and 34 for 24 ICC-Bench apps. Table 6 presents the number of false positive and false negative results reported by each tool, 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.

As mentioned earlier, we observed that a tool can fail on a benchmark for reasons different than the benchmark's target criterion. 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+IccTA and Amandroid both failed for this app, the underlying reason is not that they cannot track the repeating runs of an activity but rather that the tools cannot model `SharedPreferences`. We confirmed this observation with separate test cases which we added to UBCBench.

In 15 cases, we were unable to recover the reason for the failure or the tool crashed on the benchmark with an exception. These cases are indicated by UN in Table 6.

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 main reasons for failures of each tool.

[9]Numbers prefixed with DB and ICC indicate target criteria extracted from DroidBench and ICC-Bench suites, respectively.

FlowDroid+IccTA has 6 failures in the ICC category (DB8.*), which are related to *Intent* tracking through list operations, using complex operations like string manipulations for defining *Intents* and their actions, conservative *Intent* matching mechanism, and `SharedPreferences` problems discussed above. The tool also fails to handle advanced ICC involving URIs and MIME format (2 failures in ICC2.4 and ICC2.5). The remaining failures relate to the intra-component flows: 8 failures are in the General Java category (DB7.*); 4 failures in Lifecycle (DB9.*); 4 in Arrays and Lists (DB3.*). Other, less frequent failures include Callbacks, Field and Object Sensitivity, and Android-Specific categories. We also discovered that the tool failed to propagate taint through `setHint()` and `getHint()` methods of Android widgets (UBC1).

Amandroid has several major reasons for failures: in 17 cases, the tool failed to handle field sensitivity accurately (DB6.1). It also has 13 failures handling Callbacks (DB4.*), 6 failures in Lifecycle (DB9.*), 6 in the Arrays and Lists category (DB3.*), and 6 in General Java (DB7.*). Additional failures are in the Android-specific, Reflection, ICC, and Shared Preferences categories. Similarly to FlowDroid +IccTA, the tool also failed to handle taint propagation through `setHint()` and `getHint()` methods of Android widgets (UBC1).

Amandroid also reported 10 FPs and 10 FNs that stem from handling location-related flows. 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()*. We confirmed this with an additional test case that we added to UBCBench, and mark this failure as UBC2. For the example in Listing 2, Amandroid will not report the expected flow from the specified source `getLatitude()` to the sink `Log.d()` in line 9, but will report a (false-positive) flow from the callback parameter `location` of `onLocationChanged()` (line 7) 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., `location.getTime()`; hence, Amandroid's conservative handling of location sources will result in many FP flows.

Likewise, Amandroid always considers *Intent* parameters of callbacks as sources. For the example in Listing 3, 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 UBC3 in Table 6; it led to 5 FP results.

```
class MainActivity extends Activity {                    1
  LocationManager locManager;                            2
  LocationListener locListener = new                     3
      LocationListener() {
    double lat;                                          4
                                                         5
    @Override                                            6
    void onLocationChanged(Location location) {          7
      lat = location.getLatitude(); // source            8
      Log.d("taints", "Latitude: " + lat); // sink,      9
           leak
      }                                                  10
    };                                                   11
  ...                                                    12
 }                                                       13
```

Listing 2: UBC2 – Location-Related Flow.

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

Listing 3: UBC3 – Callback Intent Handling.

```
class ActivityWithRunnable extends Activity {            1
                                                         2
  @Override                                              3
  void onCreate(Bundle state) {                          4
    ...                                                  5
    TelephonyManager tpm = (TelephonyManager)            6
        getSystemService(TELEPHONY_SERVICE);
    Executors.newCachedThreadPool().execute(new          7
        MyRunnable(tpm.getDeviceId())); // source
}                                                        8
  class MyRunnable implements Runnable {                 9
    String deviceId;                                     10
    MyRunnable(String deviceId) {                        11
      this.deviceId = deviceId; }                        12
    @Override                                            13
    void run() {                                         14
      Log.d("ActivityWithRunnable", deviceId); } //       15
           sink, leak
}}                                                       16
```

Listing 4: UBC4 –Report Correct Entry Method.

DroidSafe has 4 failures related to handling of Arrays and Lists (DB3.*), 3 failures related to handling Callbacks (DB4.*), and 3 failures related to flow insensitivity, which is due to the tool's design (DB6.*). Other, singular failures are in the General Java and Android Specific categories.

In addition, 29 failures are related to handling entry methods: according to the DroidSafe documentation and source code, for each identified flow, the tool reports the location of the entry method, i.e., the method in which the flow source is defined, as well as the source and sink methods. As DroidSafe does not provide any path information for the detected flows, we inspected the output records to determine the correct placement of entry methods, sources, and sinks. We observed that in several cases, the tool produced an output record with correct source and sink methods, but with the entry method pointing to the sink rather than the source (15 cases) or to a different method altogether (1 case). In addition, in 3 cases, the tool produced two reports for an expected flow – one with the correct and another with an incorrect placement of the entry method.

For the example in Listing 4, DroidSafe reports a flow from `getDeviceId()` (line 7) to `Log.d()` (line 15), with `MyRunnable: void run()` being the entry method, which is incorrect. Thus, we categorized this flow as FP, and also considered that the tool missed the correct flow, with `void onCreate()` (line 4) as the entry method. Overall, the handling of entry methods resulted in 16 FPs and 13 FNs (UBC4), significantly lowering the accuracy of the tool. When ignoring the placement of entry methods, 13 of these FP and FN results can be eliminated, which would lead to an overall accuracy of 92% (listed in parenthesis in Table 5).

Summary. Overall, besides a few singular failures, the major issue of FlowDroid+IccTA is that it fails to accurately parse and track ICC Intents involving complex string analysis and list management operations. Amandroid has major issues in handling field sensitivity and location-related flows; it also overestimates ICC-related flows. DroidSafe's main issue is its handling of entry points. Once these major problems are fixed, the tools will have a much higher overall accuracy.

5 DISCUSSION AND LESSONS LEARNED

The absence of accurate information on tool configuration as well as sources, sinks, and benchmarks used for the analysis, hinders the reproducibility of earlier studies. When each of the analyzed tools was introduced, it reportedly outperformed the others. While we have no doubts about the correctness of experiments reported in earlier studies, we conjecture that the tools were evaluated under different configuration setup and with different sets of sources and sinks. Moreover, many reports focus only on a selected set of benchmarks.

Furthermore, based on our study, we observed that the current ways 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 on 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 Section 4.

Our work contributes to solving several of the issues described above: we separated tangled benchmarks and included individual test cases in UBCBench, augmented the suite with test of different types of sensitivities, 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.

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, 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 and shared the final version of the report with all tool authors to obtain their feedback.

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 results of each tool and comparing it to our ground truth 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 SDK 19 while ICC-Bench applications require SDK level 25, we did not report DroidSafe results on ICC-Bench. This limited the number of benchmarks we tested on DroidSafe. Apart from that, even though the release date of the tools we used varies, all selected tools can handle the benchmarks selected for the evaluation.

7 RELATED WORK

Our discussion of related work focuses on the internal evaluation of the tools we selected for our study and external surveys on Android-specific static taint analysis tools.

Internal Evaluation. Arzt et al. [7] compared the precision and recall of FlowDroid with two commercial tools, IBM AppScan Source [5] and Fortify SCA [4] on DroidBench version 1.0, which contained 39 benchmarks at that time. The authors provided a detailed kit explaining how to reproduce their experiments [3]. Li et al. [19] conducted two comparisons between FlowDroid, FlowDroid+IccTA, DidFail [17], and Amandroid, against 31 ICC related benchmarks from DroidBench and ICC-Bench. This work also compared the execution time of IccTA with FlowDroid and Amandroid on 50 randomly selected Google Play applications. Wei et al. [37] compared Amandroid with FlowDroid and Epicc [27], also focusing on ICC handling ability. The subject programs consisted of 39 applications from DroidBench, 16 applications from ICC-Bench, and another 4 specially designed test cases. Later on, the authors upgraded the tool and conducted new comparisons [38], focusing on 21 DroidBench and 24 ICC-Bench applications related to ICC and IAC. Finally, Gordon et al. [13] compared DroidSafe with FlowDroid+IccTA using 94 applications from DroidBench, 40 additional benchmarks developed by the team and later contributed to DroidBench, and 24 applications from a proprietary APAC benchmark. As mentioned earlier, no detailed configuration setup is provided for these experiments. In our work, we make a particular effort to align the tool configurations and also focus on a much larger set of benchmarks. Furthermore, we provide an independent accuracy assessment as our team was not involved in developing these tool.

External Surveys. Several papers survey Android-specific static analysis techniques for identifying permission and privacy leakage detection [33], robustness against obfuscation [14], and intra- and inter-application communication vulnerabilities [10]. Sufatrio et al. [35] provide a taxonomy of security vulnerabilities in Android. Li et al. [21] and Sadeghi et al. [32] 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. [30] also augment their survey with an empirical experiment comparing the survived 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, understandability of the results, and usefulness of the documentation provided by the tools. They reported on the number of applications that each tool was able to process, without detailed analysis of the accuracy of each tool. Moreover, experiments with different tools were conducted by different auditors and with default configurations of those tools, which hinders the validity of the results. In contrast, our study starts from aligning configurations of the tools and reports not only overall accuracy for each tool but also benchmark-level flow information. Furthermore, we investigate the reasons for all failures and summarize the main weaknesses of each tool. To the best of our knowledge, our study is the first large-scale, in-depth, comparative analysis of the tools under the same setup. Moreover, our configuration and results are publicly available, to allow reproducibility.

8 CONCLUSION

This paper reports on the results of a large-scale, controlled, and independent experiment we conducted to reproduce studies that evaluated most prominent static taint analysis tools for Android applications: FlowDroid+IccTA, Amandroid, and DroidSafe. We aligned the tools along the same configuration setup, used them to analyze 133 benchmark applications, and compared our results to those reported in earlier work. We discussed the identified difference and also inspected all reported false-positive and false-negative flows to identify the main reasons for inaccuracy in each tool. Furthermore, 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, and our expected results available to other researchers. As future work, we plan to contribute our benchmark to FlowDroid. We also plan to extend our experiments to additional taint analysis tools.

ACKNOWLEDGMENTS

We thank the authors of FlowDroid, IccTA, Amandroid, and DroidSafe for answering our questions about tool configurations and for providing constructive suggestions on the setup of our experiments.

REFERENCES

[1] 2017. DroidBench Benchmark Suite. https://github.com/secure-software-engineering/DroidBench.
[2] 2017. ICC-Bench Benchmark Suite. https://github.com/fgwei/ICC-Bench.
[3] 2017. PLDI'14 Artifact Evaluation. https://github.com/secure-software-engineering/soot-infoflow-android/wiki/PLDI'14-Artifact-Evaluation.
[4] 2017. Fortify SCA. https://software.microfocus.com/en-us/solutions/enterprise-security.
[5] 2017. IBM AppScan Source. https://www.ibm.com/us-en/marketplace/ibm-appscan-source.
[6] Steven Arzt. 2017. *Static Data Flow Analysis for Android Applications.* Ph.D. Dissertation. Darmstadt University of Technology, Germany.
[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. of PLDI'14.* 259–269.
[8] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D. Ernst. 2015. Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents (T). In *Proc. of ASE'15.* 669–679.
[9] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. 2012. 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.* 27–38.
[10] Shweta Bhandari, Wafa Ben Jaballah, Vineeta Jain, Vijay Laxmi, Akka Zemmari, Manoj Singh Gaur, Mohamed Mosbah, and Mauro Conti. 2017. Android Inter-app Communication Threats and Detection Techniques. *Computers & Security* 70 (2017), 392–421.
[11] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. What the App is That? Deception and Countermeasures in the Android User Interface. In *Proc. of S&P 2015.* 931–948.
[12] Yanick Fratantonio, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2015. CLAPP: Characterizing Loops in Android Applications. In *Proc. of ESEC/FSE 2015.* 687–697.
[13] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *Proc. of NDSS'15.*
[14] Johannes Hoffmann, Teemu Rytilahti, Davide Maiorca, Marcel Winandy, Giorgio Giacinto, and Thorsten Holz. 2016. Evaluating Analysis Tools for Android Apps: Status Quo and Robustness Against Obfuscation. In *Proc. of CODASPY'16.* 139–141.
[15] John Jorgensen. 2003. *Improving the Precision and Correctness of Exception Analysis in Soot.* Technical Report 2003-3. McGill University, Canada.
[16] George Kastrinis and Yannis Smaragdakis. 2013. Efficient and Effective Handling Of Exceptions in Java Points-to Analysis. In *Proc. of CC'13.* 41–60.
[17] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. Android Taint Flow Analysis for App Sets. In *Proc. of PLDI Workshop on Software Analysis (SOAP'15).* 1–6.
[18] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Proc. of CC'03.* 153–169.
[19] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting Inter-component Privacy Leaks in Android Apps. In *Proc. of ICSE'15.*
[20] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis. In *Proc. of SEC'15.* 513–527.
[21] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static Analysis of Android Apps: A Systematic Literature Review. *Information & Software Technology* 88 (2017), 67–95.
[22] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. 2017. AndroZoo++: Collecting Millions of Android Apps and Their Metadata for the Research Community. *The Computing Research Repository* abs/1709.05281 (2017).
[23] Julia Rubin Lina Qiu, Yingying Wang. 2018. Supplementary Materials. https://resess.github.io/PaperAppendices/ISSTA2018.html.
[24] Zhang Mu, Duan Yue, Yin Heng, and Zhao Zhiruo. 2014. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proc. of CCS'14.* 1105–1116.
[25] Damien Octeau, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. 2016. Combining Static Analysis with Probabilistic Models to Enable Market-scale Android Inter-component Analysis. In *Proc. POPL'16.* 469–484.
[26] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *Proc. of ICSE'15.* 77–88.
[27] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective Inter-Component Communication Mapping in Android: An Essential Step Towards Holistic Security Analysis. In *Proc. of USENIX Security 2013.* 543–558.
[28] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. 2007. Efficient Field-sensitive Pointer Analysis of C. *ACM Transactions on Programming Languages and Systems* 30, 1 (2007).
[29] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *Proc. of NDSS'14.*
[30] Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, et al. 2016. *droid: Assessment and Evaluation of Android Application Analysis Tools. *Comput. Surveys* 49, 3 (2016), 55.
[31] Alejandro Russo, Andrei Sabelfeld, and Keqin Li. 2010. Implicit Flows in Malicious and Nonmalicious Code. *Logics and Languages for Reliability and Security* 25 (2010), 301–322.
[32] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. 2017. A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software. *IEEE Transactions on Software Engineering* 43, 6 (2017), 492–530.
[33] Suzanna Schmeelk and Junfeng Yang andAlfred V. Aho. 2015. Android Malware Static Analysis Techniques. In *Proc. of CISR'15.* 5:1–5:8.
[34] Yannis Smaragdakis, George Balatsouras, et al. 2015. Pointer Analysis. *Foundations and Trends® in Programming Languages* 2, 1 (2015), 1–69.
[35] Sufatrio, Darell J. J. Tan, Tong-Wei Chua, and Vrizlynn L. L. Thing. 2015. Securing Android: A Survey, Taxonomy, and Challenges. *Comput. Surveys* 47, 4 (2015), 58:1–58:45.
[36] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proc. of CASCON'99.*
[37] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proc. of CCS'14.* 1329–1341.
[38] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2017. *Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps.* Technical Report 2017-4. University of South Florida, USA.
[39] John Whaley, Dzintars Avots, Michael Carbin, and Monica S Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Proc. of APLAS'05.* 97–118.
[40] Wikipedia. 2017. Data-flow Analysis: Sensitivities. https://en.wikipedia.org/wiki/Data-flow_analysis.