Benchmarking Android Data Leak Detection Tools

Bachelor Thesis

Timo Spring
from
Thun, Canton of Bern

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

August 16, 2018

Prof. Dr. Oscar Nierstrasz
Claudio Corrodi
Software Composition Group
Institut für Informatik
University of Bern, Switzerland
Abstract

In 2017, Android hit a global mobile market share of 88% which makes it the most popular mobile platform. Application stores, such as the Google Play Store, are offering millions of mobile applications to consumers, which are installed and updated on a daily basis. However, the security of those applications is a major concern. A thorough security analysis before the publication of each application is time and resource consuming. Hence, platform providers cannot and do not manually vet every application handed in for publication. Consequently, many malicious and vulnerable applications find their way to the app stores and through there to the end users’ devices. Those applications exhibit serious security issues, such as leaking of sensitive information.

During the previous years, researchers proposed a myriad of techniques and tools to detect such issues. There also exist large scale taxonomies classifying such tools into different categories. However, it is unclear how these tools perform compared to each other. Such a comparison is almost infeasible, since most tools are no longer available or cannot be set up any more.

In this work, we review static analysis tools for detecting data leaks in Android applications. Out of 87 tools in the vulnerability detection domain, we are able to obtain 22 tools. We then identify 5 tools in the data leak detection domain and run them. We run them on a given data set with known data leak vulnerabilities and compare their performance. Furthermore, we run the tools on a larger set of real-world applications to assess the prevalence of data leak issues in open-source Android applications.

We propose our own approach—DistillDroid—to compare security analysis tools by normalising their interfaces. This simplifies result reproduction and extension to other security vulnerability domains. In addition, the user experience and usability is highly improved.
## Contents

1 Introduction 1  
  1.1 The State Of Android Security 1  
  1.2 Analysis Tools 3  
  1.3 Research Questions 3  
  1.4 Outlook 4  

2 Related Work 6  
  2.1 Android Security Analysis 6  
  2.2 Data Leak Vulnerabilities 7  
    2.2.1 Sources and Sinks 7  
  2.3 Comparing Software Artefacts 9  

3 Classification and Selection 10  
  3.1 Hypotheses 10  
  3.2 Preliminary Questions 11  
  3.3 Obtaining Tools 11  
  3.4 Selected Tools 14  

4 Benchmark Implementation 18  
  4.1 Benchmarking Concept 18  
  4.2 Benchmarking Java Implementation 19  

5 Experimental Setup 22  
  5.1 Database Selection 22  
  5.2 Tools Configuration 23  
    5.2.1 List of Sources and Sinks 23  
    5.2.2 Original Configurations 24  
    5.2.3 Shared Configurations 25  
    5.2.4 Switching Between Configurations 26  
  5.3 Evaluation Process 26  
    5.3.1 DroidBench Evaluation Process 26
In this chapter, we provide an overview of the current state of Android security. We point out what major security challenges are still prevalent and how platform providers such as Google are trying to tackle them. We briefly present the efforts made by the research community and state the goal of this work. Finally, we formulate the research questions and provide an overview of the rest of this thesis.

### 1.1 The State Of Android Security

There are over 2 billion devices worldwide running on the Android platform. With a market share of 87.7% (Q2, 2017), Android is the most popular mobile platform worldwide. However, since the platform exhibited several security breaches and vulnerabilities during the past few years, it has a poor reputation in the public eye.

As of spring 2018, there are nine major versions of Android in circulation, ranging from Android Gingerbread, released in 2010, to Oreo, released in 2017. Oreo—Google’s latest Android version—is only running on 1.1% of the devices worldwide.

---

CHAPTER 1. INTRODUCTION

all Android devices (February 2018). Together with its predecessor Nougat, they run on about 30% of Android devices, followed by Marshmallow (2015) with 28.1%. Android Lollipop (2014) and KitKat (2013) are installed on 36.6% of all Android devices.

![Pie chart showing Android OS versions in the field (February 2018)](image)

Figure 1.1: Overview of Android OS versions in the field (February 2018)\(^5\). We note that Google does not list Honeycomb and Froyo OS version because their distribution is close to 0%.

Over 70%—or more than 1.4 billion Android devices—are more than two years out of date, posing several challenges for both Google and the Android ecosystem.

A major challenge is fixing security vulnerabilities. Due to the huge fragmentation of Android versions, security patches cannot be rolled out in a straightforward way. They need to be compatible with several Android versions. With its latest OS version—Android Oreo—Google tries to improve the state of security and to decrease fragmentation. Initiatives such as Project Treble\(^6\) and Google Play Protect\(^7\) intend to tackle these challenges.

\(^5\)https://c.mi.com/forum.php?mod=viewthread&tid=759720&aid=1668480&from=album&page=1
\(^6\)https://source.android.com/devices/architecture/treble
\(^7\)https://www.android.com/intl/de_de/play-protect/
1.2 Analysis Tools

Not only Google is making a leap forward to improve the state of Android security and to remove vulnerable and malicious applications from online app stores. The research community has proposed a myriad of techniques and approaches over the past years to detect such applications.

Furthermore, there exist large scale taxonomies aiming to list and classify existing tools and approaches on the Android Security domain [33, 78, 80, 90]. However, these taxonomies lack comparisons of similar tools.

To provide such a comparison, we need to obtain the tools. However, we observe that most tools mentioned in literature are not available to the public anymore. One of the root causes is that the fragmentation forces researchers to update their tools on a frequent basis. We note that this is usually not the case and that most tools are as outdated as the Android version they are built upon.

1.3 Research Questions

The aim of this work is to compare a set of analysis tools in the domain of data leak vulnerabilities for a common set of applications. Unfortunately, during our project, we came to the belief that research papers, even if they present and discuss a particular tool, are rarely accompanied by an artefact for reproducing the results. Thus, a comparison with other tools is not feasible. This yields the following research question:

RQ1 To what degree are security analysis approaches and results in the information disclosure domain in Android applications reproducible?

To answer this question, we review existing approaches proposed by the research community and try to obtain the corresponding tools through multiple channels (e.g., using search engines or by contacting the authors of the papers). Once a tool is obtained, we try to set it up based on the documentation.

We report findings similar to what Amann et al. report in the domain of application programming interface (API) misuse analysis [4]. In their work, the authors ended up evaluating their benchmark on just five tools.

The second part of this paper presents the evaluation and comparison of five static Android security analysis tools on the domain of information disclosure vulnerabilities. To facilitate the comparison and reproduction of results, we present our own implementation of a flexible
benchmark suite that allows others to easily add their tool to the suite. The analysis will then be executed automatically, and the results consolidated with the other tools. Thanks to our implementation, we can run all tools on a common set of applications and gather unified results. This is not only useful for the evaluation, but can also serve as a basis for an analysis front end that provides normalised reports to the user.

Using the benchmark enables investigation into the second research question, which we state as follows.

**RQ2** How do information disclosure analysis tools perform (individually and compared to each other) on a common set of applications?

To answer this question, we run the five obtained artefacts on a common set of applications with known vulnerabilities from a given dataset. Then we report on precision and recall for each tool. We hypothesize that tools with high agreement tend to report the same issues when configured to use the same list of sources and sinks. Thus, we formulate the final research question.

**RQ3** To what extent do reported issues from various approaches overlap?

We perform a pairwise analysis of the tools’ results after running them on the set of applications with known vulnerabilities. Afterwards, we hypothesize which tools tend to report the same findings and evaluate our assumptions on a larger scale with real-world applications.

Answering these questions allows us to identify common weaknesses amongst the tools, which helps us to point out areas in need of further research.

### 1.4 Outlook

The remainder of this paper is organized as follows. Chapter 2 gives an overview of relevant work. This includes an overview of data leak analysis in Android and related work to analysing and comparing tools.

In Chapter 3 we elaborate our course of action for obtaining, reviewing, and classifying analysis tools for data leak detection. We show which tools were initially considered for the benchmark and how the process of elimination yielded a set of just five tools.

We then explain our benchmarking concept and present our own benchmark implementation in Chapter 4. This implementation aims to facilitate the comparison of tools and increase reproducibility.
We proceed in showing our experimental setup used for evaluating the selected tools in Chapter 5, before presenting the obtained results in Chapter 6.

Finally, we state threats to validity in Chapter 7 and conclude the thesis in Chapter 8.
Android security analysis is a popular topic. Over the past few years, the research community has done extensive work in this field. Considerable efforts have been devoted in detecting vulnerabilities in Android applications and in the Android platform itself. In most cases, the approach is evaluated using a prototype tool. Thus, the range of existing related works and approaches is exhaustive.

In this chapter, we give an overview of the works that served as a basis for this thesis. It is organised as follows. First, we cover the papers that provide an overview of the state of the art of Android security analysis in general. Second, we give a brief overview of the data leak vulnerability detection and taint analysis tools that are evaluated in this thesis. However, we do this without going into much details and instead refer to Section 3.4 for a detailed presentation of the selected tools. Finally, we point out some existing benchmarking works on comparing software artefacts, particularly for the security domain.

2.1 Android Security Analysis

There exists a vast amount of published work on the topic of Android security analysis. First, it is crucial for us to get a better understanding on what kinds of vulnerabilities to expect in Android applications. Ghafari et al. present a study on security smells in Android [45]. They list 28 code smells that may
lead to vulnerabilities in applications and show their prevalence in real-world Android applications. The code smells also cover data leak vulnerabilities and confirm that these issues are in fact common among Android applications.

Second, we focus on the state-of-the-art of Android security analysis. Fortunately, there exist excellent, recent taxonomies that cover a wide range of static, dynamic, and hybrid approaches to Android security analysis.

One of the latest and most extensive taxonomies is presented by Sadeghi, and provides a large-scale overview of Android security analysis in general [80]. Their paper cites more than 500 works in the domain of Android security analysis and provides a detailed categorization of approaches and tools in that domain. Among other lists, the paper provides a list of tools with the analysis objective of vulnerability detection [80, p. 12, Table 3].

This list served as a starting point for our review. In fact, we were unable to find additional approaches that were not already covered by this overview. Moreover, we also cross referenced the list with other published taxonomies such as the works by Sufatrio et al. [89] or Reaves et al. [78]. All taxonomies presented contain similar lists of references.

### 2.2 Data Leak Vulnerabilities

In general, data leaks describe the practice to release potentially private or sensitive data. For example, a device’s unique identifier (UID). In this thesis, we are interested in the release of data to unauthorized clients or otherwise to unintended places.

Many approaches, including those used in our evaluation, use taint analysis to determine locations in the code where possibly sensitive data is leaked. In taint analysis, one tries to determine whether sensitive data can flow from a source to a sink.

#### 2.2.1 Sources and Sinks

We define sources and sinks similar to Rasthofer et al. in their work on the SuSi tool [77].

\textit{Definition (Source):} Sources are calls into resource methods returning non-constant values into the application code. (Rasthofer et al. [77])

Resource methods in this context are predefined API calls provided by the Android operating system to access shared resources outside the application’s
address space. Therefore, a source is a location within the application’s source code, where a resource method is called returning a non-constant value from a shared resource.

A typical example for a source is the `getDeviceID()` method returning a unique identifier of the underlying device such as the IMEI (International Mobile Equipment Identity). Such globally unique identifiers can be used for tracking and physical device association. An attacker could for example use the IMEI to perform a remote SIM card rooting\(^1\).

Sinks, on the other hand, are locations in the source code, where sensitive information is leaving the current component. Rasthofer et al. define them as follows.

\[ \text{Definition (Sink): Sinks are calls into resource methods accepting at least one non-constant data value from the application code as parameter, if and only if a new value is written or an existing one is overwritten on the resource.} \]

(Rasthofer et al. [77])

Therefore, a sink is considered when using the Android API calls within the application’s source code to write to shared resources.

A typical example of a sink is the `sendTextMessage(String message)` method with the message text being non-constant. However, a sink is only problematic if the values passed as arguments are actually sensitive information, such as a unique identifier. In the `sendTextMessage(String message)` example above, we could also pass a random String message not containing any sensitive information. In that case, the sink would not be leaking sensitive information and therefore is no data leak.

Taint analysis requires a list of sources and sinks to check possible data flows. SuSi is an approach to automatically generate such lists based on an `android.jar` artefact [77].

The `android.jar` file is a compiled version of the Android SDK. It mainly serves as a compile-time library for 3rd party applications and is available for different API levels. We use the SuSi tool in our benchmark to obtain a neutral configuration and to reduce bias stemming from individual lists shipped with the respective tools.

Employed techniques for taint analysis include, for example, static data-flow and control-flow analysis. Besides approaches in taint analysis, there

\[^1\text{https://threatpost.com/mobile-applications-leak-device-location-data/121392/}\]
exists a wide range of approaches and tools that aim to detect data leak vulnerabilities.

We abstain from discussing individual approaches here. Instead, we refer to the work we considered for the evaluation later in this thesis and point the interested reader to the exhaustive, generic taxonomies on Android security analysis [80, 89, 78]. In Section 3.4, we also present the five tools included in this benchmark.

2.3 Comparing Software Artefacts

There are several pieces of relevant work related to analysing and benchmarking software artefacts.

In 2016, Amann et al. have analysed artefacts for detecting API misuse violations [3]. Their approach is similar to ours in that they developed a framework for comparing such tools. Furthermore, the authors first reviewed corresponding literature and obtained artefacts in a similar fashion as we did.

Hoffmann et al. have evaluated several static and dynamic analysis tools with regard to obfuscation [50]. They use synthetic sample applications and automatically obfuscate them before running the analysis. In contrast, we do not provide our own samples; instead, we rely on DroidBench\(^2\), a database of synthetic applications with known vulnerabilities. In addition, we evaluate the artefacts on real-world applications.

DroidBench is a collection of synthetic applications intended for evaluating analysis tools. Because vulnerabilities are documented in these applications, they are well-suited for the qualitative analysis we present in this work. Using this benchmark, not only can we identify true positives easily, but, due to the limited size of the synthetic programs, we can determine false positives as well.

The work by Reaves et al. comes closest to ours. In their study, they use DroidBench to analyse results obtained from Amandroid, AppAudit, DroidSafe, Episc, FlowDroid, MalloDroid, and TaintDroid [78]. They do not, however, cover IC3, COVERT, IccTA, or HornDroid. In contrast to our work, the evaluation is lacking a comparison of tools amongst each other. Furthermore, the authors do not report on the number of vulnerabilities that were detected.

\(^2\)https://github.com/secure-software-engineering/DroidBench
3 Classification and Selection

In this chapter, we state hypotheses about the outcome of our benchmark. Furthermore, we explain how we selected the tools included in this work and what questions and limitations led to our choice. In addition, we present the five selected data leak detection tools.

3.1 Hypotheses

Based on the number of true positives, false positives, true negatives, and false negatives in a tool’s report, we can formulate several hypotheses about the tools and the quality of their results.

\[ \text{H1} \quad \text{As more tools report the same vulnerability, the likelihood of it being a true positive increases.} \]

\[ \text{H2} \quad \text{The fewer tools report a certain vulnerability, the more likely it is a false positive} \]

\[ \text{H3} \quad \text{If a tool reports much more data leaks than all the other tools, then it is likely to report more false positives than the others.} \]

We validate our hypotheses in Section 6.4.
3.2 Preliminary Questions

There are two preliminary questions we had to answer before starting with the selection of the tools.

**Q1** *On which domain of Android security analysis do we want to focus?*

**Q2** *On what level do we want to compare the tools and in what metrics are we interested?*

For the first question, we checked different domains of Android Security analysis. In general, our work is not limited to a single domain and could be easily extended to other domains. However, due to the large number of published works for static approaches and to ensure comparability of the tools, we decided to focus on the domain of data leak vulnerabilities.

To answer the second question, we refer to the literature, where common metrics in the field of software comparison are precision and recall. These metrics allow us to draw conclusions on a tool’s performance and to easily compare them amongst each other. In order to calculate the precision and recall, we require the number of true positives/negatives and false positives/negatives. So, the number of reported vulnerabilities of a tool plays a key role in this benchmarking.

The identification of true/false positives and negatives also has implications on the dataset of applications used in this work, since we need to be able to identify them within the applications source code. The selection of the dataset of applications is further discussed in Section 5.1.

3.3 Obtaining Tools

In this benchmark, we take advantage of two recent taxonomies in the security analysis domain.

Sadeghi *et al.* have collected 517 references to tools and approaches in their taxonomy on security related Android analysis software [80]. In this paper, the authors classify the tools and approaches according to different categories, such as the type of sensitivity or the type of program analysis (static, dynamic, hybrid, formal, machine learning). We use Sadeghi’s work as a starting point for our tool selection. Hence, for the initial selection of tools, we chose the 87 references to vulnerability detection tools presented by the paper [80, p. 12, Table 3].
CHAPTER 3. CLASSIFICATION AND SELECTION

We compare Sadeghi’s list of references with two other taxonomies with mostly overlapping sources [78, 89], and arrived at the following list of artefacts: ASM [49], Addicted [80], Amandroid [97], ApkCombiner [60], AppAudit [100], AppCaulk [82], AppCracker [17], AppFence [51], AppGuard [8], AppProfiler [79], AppRay [91], AppSealer [102], Aquifer [72], AuthDroid [96], Bagheri [11], Bartel [12], Bartsch [13], Bifocals [23], Buhov [16], Buzzer [20], CMA [85], COVERT [9], CRePE [25], CoChecker [28], ComDroid [24], ConDroid [83], ContentScope [54], Cooley [26], Copes [80], CredMiner [107], CryptoLint [34], Dare (Enck) [30], DexDiff [67], DroidAlarm [106], DroidCIA [22], DroidChecker [21], DroidGuard [10], DroidRay [105], DroidSearch [76], FineDroid [103], FlowDroid [6], Gallo [42], Geneialtakis [43], GrabNRun [36], Harehunter [1], HornDroid [18], IC3 (EpiCC) [73], IPCInspection [40], IVDroid [37], IccTA [59], Juxtap [48], KLD [84], Kantola [56], Lintent [15], Lu [63], MalloDroid [35], Matsumoto [65], Mutchler [71], NoFrack [44], NoInjection [55], Onwuzurike [74], PCLeaks [58], PaddyFrog [98], PatchDroid [69], PermCheckTool [95], PermissionFlow [81], Poeplau [75], Pscout [7], QUIRE [31], Ren [52], SADroid [47], SDroid [41], SEFA [99], SMVHunter [88], STAMBA [14], Scoria [93], SecUp [101], Smith [87], Stowaway [39], Supor [53], Tongrini [62], Vecchiatto [94], VetDroid [104], WeChecker [29], Woodpecker [46], Zuo [108]. Note that in cases where the approach or tool has no proper name, we use the first author’s surname instead.

In a next step, we further categorized and filtered the tools. First, we tried to obtain all artefacts by employing the following strategy:

1. Review the paper, look for links or directions on how to obtain the artefact,
2. look for the artefact on the authors’ or research group’s websites,
3. search online with contemporary search engines for the artefact, and
4. contact the authors and inquire whether the tool is available or can be made available to us (at most two requests by email within three weeks).

By reviewing the corresponding paper for each tool, we identified 13 tools that cannot be executed because they are formal models or frameworks only and thus are not suitable to be included in the benchmark. For 14 tools, we could obtain the artefact by reviewing the paper or using contemporary search engines.

\(^1\)http://siis.cse.psu.edu/ded/
For the remaining 61 tools, we have contacted the researchers to request access to their artefacts. Out of those 61 requests, 49 remained unanswered and four researchers refused to give us access due to the commercial usage of their tool.

This results in the following list of 22 available and obtainable tools: Amandroid, APK-Combiner, AppCaulk, AppGuard, Aquifer, ASM, ConDroid, COVERT, CRePE, Dare (Enck), DexDiff, IC3 (Epicc), FlowDroid, Geneiatakis, Grab n’Run, HornDroid, IccTA, Lintent, MalloDroid, NoFrack, PScout, and SCanDroid.

Figure 3.1: Overview of tool categorization. Tools marked with an asterisk are not classified in Sadeghi’s work.

Figure 3.1 provides an overview of the obtainable tools categorized according to the type of security threat, type of approach and scope of the analysis. In order to compare the tools, they need to focus on the same type of vulnerability. As can be seen in Fig. 3.1, the domain of information disclosure or rather data leak vulnerabilities is addressed by most tools with 11 approaches compared to escalation of privileges with just 7 tools. That is why we selected data leak vulnerabilities as the domain for this work.
From the artefacts available, we further filtered out 17 tools for the following reasons:

- **Dynamic or hybrid approach:** In this benchmark, we focus solely on static approaches for comparability reasons. Dynamic or hybrid approaches include *AppCaulk, AppGuard, Aquifer, ASM, ConDroid, CRePE, Geneiatakis, Grab n’Run*.

- **Not in the selected domain:** Tools not in the information disclosure domain include *Amandroid, Lintent, MalloDroid, NoFrack, PScout* and *SCanDroid*.

- **Analysis enabler:** *ApkCombiner* is used to combine several APKs into one APK to enable inter-app communication analysis. *Dare (Enck)* is used to decompile Android applications from the installation image to source code on which other analysis tools can perform. Neither of the two tools are reporting any data leaks and thus are excluded from the benchmarking.

- **Setup failed:** We failed to setup *DexDiff*. This is mainly due to inadequate documentation and lack of support by the responsible researcher.

Out of the initial list of 87 papers and their corresponding artefacts, we could obtain 22 tools. We selected data leak detection as the target domain and therefore selected five tools from these 22 that are suitable for benchmarking.

### 3.4 Selected Tools

In this benchmark, we will focus on the following five tools: *COVERT, FlowDroid, HornDroid, IccTA* and *IC3*. Table 3.1 provides an overview of the five selected tools and their main characteristics.

*FlowDroid* is a taint analysis based on the *Soot* and *Heros* frameworks. The authors report high rates of precision and recall. Especially its call graph construction procedure is popular, since it helps increasing the context-, flow- and object-sensitivity and is therefore used by other tools such as *IccTA, COVERT, or IC3*.

To model application lifecycle, *FlowDroid* uses a list of source and sinks. To generate such a list, the researchers have created the *SuSi* tool [77], which we also applied in this benchmark. It takes an *android.jar* as an input and produces a list of all sources and sinks found using machine learning algorithms. Along with the list of sources and sinks, *FlowDroid* also uses the
## Chapter 3. Classification and Selection

<table>
<thead>
<tr>
<th>Name</th>
<th>COVERT Type</th>
<th>FlowDroid Type</th>
<th>Horn-Droid Type</th>
<th>IccTA Type</th>
<th>IC3 Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>static, formal</td>
<td>static</td>
<td>static, formal</td>
<td>static</td>
<td>static</td>
</tr>
<tr>
<td>Scope</td>
<td>Inter-App</td>
<td>Inter-Component</td>
<td>Inter-Component</td>
<td>Inter-Component</td>
<td>Inter-Component</td>
</tr>
<tr>
<td>Artefact</td>
<td>Manifest, Bytecode</td>
<td>Manifest, Layout, Bytecode (DEX)</td>
<td>Bytecode</td>
<td>Manifest, Layout, Bytecode (DEX)</td>
<td>Manifest, Bytecode</td>
</tr>
<tr>
<td>Sensitivity</td>
<td>context, value</td>
<td>object, context, flow, field</td>
<td>object (p), context, value, flow (p), field</td>
<td>object, context, flow, field</td>
<td>context, flow, field</td>
</tr>
<tr>
<td>Sources and sinks</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 3.1: Overview of main tool characteristics

manifest file, dex files and layout files to perform the analysis. In order to ensure flow-sensitive information flow analysis, it creates a dummy main method emulating each possible interleaving of the callbacks during the application lifecycle.

In addition to static analysis, *HornDroid* is also using a formal approach. It translates the application into so-called Horn clauses, which are used to abstract the program semantics. These Horn clauses are then discharged using an off-the-shelf theorem prover (*Z3*)\(^2\). This formal proof of soundness helps to exclude unreachable program parts. *HornDroid* uses Dalvik VM bytecode as a base artefact. It is value- and context-sensitive, and partially flow-, field-, and object-sensitive. So, it can skip unreachable program parts, approximate runtime values, compute different static approximations upon different method calls, take the order of statements into account, and approximate static fields. *HornDroid* uses a list of sources and sinks supplied by the *DroidSafe* tool. As a known limitation, *HornDroid* does not find any implicit information flows and might produce false positives for heap abstractions, exceptions and inter-app communication.

*IccTA* is an inter-component communication based taint analysis tool. However, the approach is generic and promises to be applicable to any data-

\(^2\)https://github.com/Z3Prover/z3
flow analysis. It reports precise tracks of information flow from source points to sinks. First, IccTA decompiles the Android app in the form of Dalvik bytecode into Jimple using the Dexpler decompiler. Jimple is an internal representation of the Soot framework. IccTA then extracts all ICC methods using Epicc. Then it uses IC3 to parse the URIs in order to support content provider related ICC methods. The target components are retrieved, based on the manifest file and bytecode. It then connects components to enable a data-flow analysis between the different components, on which an intra-component analysis is performed. For that purpose, IccTA uses a modified version of FlowDroid. This yields in a control flow graph of the whole application. The tainted paths or leaks are stored in a database to reuse results. Similarly to FlowDroid, IccTA is a context-, object-, flow-, and field-sensitive analysis.

IC3 detects inter-component communication with a special focus on inferring values of complex objects with multiple fields such as intents or URIs. First, it decompiles the Android source code to Java bytecode using Dare. Then it generates an Inter-procedural Control Flow Graph (ICFG) using the FlowDroid call graph construction procedure. This ICFG and the so-called COAL (constant propagation language) specifications describing the structure of the composite objects are passed to a COAL Solver. This COAL Solver uses the Heros IDE to resolve the Inter-procedural Distributive Environment Problems (IDE). In a last step, it performs argument value analyses such as String analysis to determine the values of a function’s argument. IC3 is a context-, flow- and field-sensitive analysis and replaces the previous tool Enck.

COVERT is also a static and formal security analysis. However, its main focus is inter-app communication and escalation of privileges. It tries to find unsafe combinations of apps exploiting these vulnerabilities. In order to also cover the topic of information disclosure, it was extended with a static information flow analysis, performed by FlowDroid. COVERT consists of a model extractor and a formal analyzer. The model extractor takes the applications manifest file and bytecode and specifies a first model of the potential inter-process communications. In a second step, a reachability analysis is performed to determine what permission required functionalities are being exposed by unguarded execution paths. This results in an extended manifest file for each application. Those extended manifest files are transformed into so called Alloy modules using the Alloy language (formal modeling language) and then fed to the formal compositional Alloy Analyzer. The analyzer proofs which combinations of applications are safe and for which combinations the

\(^{3}\text{http://siis.cse.psu.edu/coal/index.html#banner}\)
unguarded execution paths can be exploited. The benefit of *COVERT* is the inter-app communication, which is not possible with most static analysis tools. *COVERT* is a value-, context-and flow-sensitive analysis. *COVERT* also provides a desktop client; however, we did not manage to set it up and execute it. According to the researchers, the desktop client is outdated and not supported any more. Instead, one should only use the command line to perform the analysis.
In this chapter, we explain our benchmarking concept and present our Java tool—DistillDroid—to perform the analysis on all selected tools. Our benchmarking implementation merges all output files and provides a homogeneous overview of the reported vulnerabilities. It allows us to automate the analysis, which is necessary for larger scale analyses where manual inspection is infeasible.

4.1 Benchmarking Concept

This benchmark compares the tools’ performances in practice. To do so, we need to run the analyses on a common dataset of applications. However, the size and structure of the dataset of applications has implications on what information we can deduce from the analysis. A smaller dataset of applications allows for a more thorough analysis with manual validations of the results. A larger dataset will improve the validity of the results, but will make manual investigations of the results much more time consuming.

Synthetic applications with seeded vulnerabilities are useful for determining precision and recall; real-world applications, on the other hand, provide a realistic setting at the expense of not being able to determine those metrics. The two targets complement each other and we use both in our evaluation.

Therefore, this work is built on two pillars. The first pillar is a smaller scale qualitative study, which is done by running the selected tools on a small
scale dataset with known data leak vulnerabilities. Based on the analyses
results, we can draw conclusions on the quality of the results and formulate
hypotheses on how the tools might perform in a bigger scale. In this first
pillar, the focus lies on true/false positives and negatives and related metrics
such as precision and recall.

The second pillar is a large scale quantitative study, for which we run
the tools on a set of real-world applications from the F-Droid app repository.
This gives us insights into the tools’ performance on real-world applications
and helps us to validate the hypotheses. Here, the number of reported vul-
nerabilities and the overlap with other tools play a key role. Moreover, this
analysis provides insights into the general state of real-world applications
concerning data leak vulnerabilities.

4.2 Benchmarking Java Implementation

Besides being useful for summarising the tools’ outputs and thus helping us
to compare the results, there are other potential use cases for DistillDroid.
Especially developers, who want to analyse their applications before publi-
cation can profit from our implementation. The original reports generated
by the tools are often hard to find (e.g., in a nested folder or printed to
standard output and error streams). Furthermore, they are generally cryptic
and over-populated with additional information such as call paths or register
values. An example of such a report produced by the IccTA tool can be seen
in Listing 4.1. It shows the report for a single vulnerability.

Listing 4.1: Example excerpt of console output by IccTA

Found a flow to sink virtualinvoke $r3.<org.cert.sendsms.MainActivity: void startActivityForResult(android.content.Intent,in
t)>(r2, 0) on line 26, from the following sources: - $r6 = v
irtualinvoke $r5.<android.telephony.TelephonyManager: java.lang.String getDeviceId()>() (in <org.cert.sendsms.Button1Liste
ner: void onClick(android.view.View)>) on Path [$r6 = virtual
invoke $r5.<android.telephony.TelephonyManager: java.lang.Str
ing getDeviceId>()(), virtualinvoke $r2.<android.content.Intent putExtra(java.lang.String,java.lan
ge.String)>("secret", $r6), virtualinvoke $r3.<org.cert.sendsm
s.MainActivity: void startActivityForResult(android.content.I
ntent,int)>(r2, 0) - $r4 = virtualinvoke $r3.<org.cert.send
sms.MainActivity: java.lang.Object getSystemService(java.lan
CHAPTER 4. BENCHMARK IMPLEMENTATION

As a result, the exact location of the leaky code is hard to make out within the generated reports. Our wrapper, on the other hand, can present the results in a standardised way, focusing solely on the location of the data leak within the code. Thus, we only present the name of the class and method containing the leaky line of code and the exact sink (method call) that led to the data leak. We abstain from showing additional information such as register values or call paths.

Since our approach can easily be extended with additional analysis tools, it could make a wide range of program analyses more accessible. In addition, there are currently only very few tools covering multiple security domains. Consequently, the end user would need to identify several analysis tools covering several Android security domains to get a complete picture of the security state of the underlying application. In contrast, DistillDroid can be extended with tools covering other security domains and thus simplify the user’s life by providing a single interface. The user will not have to search for available tools and can launch all provided tools with just one click receiving all results in a standardised and understandable way.

Listing 4.2 shows a sample output of our Java tool for the same vulnerability presented in listing 4.1.

Class Name: org.cert.sendsms.Button1Listener
Method Name: onClick(android.view.View) void
Sink Method: startActivityForResult(Intent, int) void
Found by Tool: [iccta, horndroid]

Listing 4.2: Example Java tool output

The benchmarking implementation of DistillDroid consists of runners and parsers. Each tool artefact included in the benchmark must extend and implement these interfaces. A tool runner is a simple Java class that specifies the tools main characteristics such as commands used to run it in the shell,
location of the results files and how to get these results. Furthermore, a tool parser needs to be implemented that parses generated output and creates reports in the form mentioned above (class name, method name, sink). So, to include a new tool in the benchmark, one should follow these steps:

1. Obtain the artefact and adapt the tool to run from the command line (which is already the case for most tools).

2. If the tools do not already create a separate results file, then adapt the command to store the results in such a file.

3. Extend the abstract Tool class by specifying the command used to trigger the analysis and the location of the results file.

4. Extend the abstract Parser class that obtains the relevant data.

5. Provide a path to the application (.apk file) that should be analyzed.

The benchmark is implemented in Java and available from the supplementary website\(^1\). While we do not redistribute the integrated tools’ artefacts, we do provide detailed instructions on how to set them up.

Launching our benchmark tool will sequentially copy and paste the command used to trigger the analysis for each tool to the shell and execute it. Afterwards, all results files are gathered and parsed for class name, method name and sink method. This results in a list of all leaks detected by the tools in a standardised format. However, we want a summarised list, so that if two tools detect the same vulnerability, we do not have several entries in our list. Therefore, we first sort the list and then summarise entries that have the same class name, method name and sink method by adding the tool name to the indicated list of tools that detected the vulnerability. Finally, we store the summarised and standardised list of vulnerabilities in a text file and in a csv (comma separated values) file. If the analysis fails for one of the tools, the others still perform the analysis and the results summary will simply miss the results of the failed tool.

The analysis of real-world applications is time consuming and may take hours or even days to complete. In our benchmark, the user can set a time out for the analysis being aborted after a certain number of minutes and then proceeding with the next tools. If a tool runs longer than the specified timeout, it will be aborted.

\(^{1}\)https://github.com/tiimoS/distilldroid
To benchmark our five selected tools—IceTA, IC3 (Epicc), HornDroid, FlowDroid and COVERT—we have integrated them in DistillDroid, our own Java tool. In this chapter, we describe the experimental setup in terms of specific configurations and database selection used in the evaluation.

5.1 Database Selection

As many others [32, 68, 78, 2, 92, 19, 57, 5, 61, 97, 38, 70] in recent research into Android security analysis, we have selected DroidBench\(^1\) as a test suite for our small scale qualitative study. DroidBench consists of 119 synthetic applications. They specifically target different data leak vulnerabilities with a total of 125 leaks, which are described and indicated directly in the source code. These synthetic applications usually only consist of two to three classes and are therefore very small in size. The leak indication within the source code allows us to review the generated reports and identify the true and false positives and negatives. Based on these metrics, we can calculate precision and recall.

For the large scale quantitative study, we have selected F-Droid\(^2\). F-Droid is an online application store for open-source software. However, due to the long run time of the analyses with larger applications (which often exceeds

\(^1\)[https://github.com/secure-software-engineering/droidbench]
\(^2\)[https://f-droid.org]
three hours, depending on the tool), we only analyse a random sample of 250 applications. Compared to the DroidBench applications, we have no information about possible data leaks, since they are not indicated in the source code.

With these two databases of applications at hand, we perform the analysis as follows:

1. Run all tools on the small DroidBench dataset and manually inspect the applications and results for the number of true and false positives and negatives.

2. Run our benchmarking implementation on the same DroidBench dataset and compare the summarised results with the results from the first run. We evaluate the quality of our parsers and results summarising implementations in Section 7.1.

3. Run the benchmarking implementation on the large F-Droid dataset and check the number of reported vulnerabilities and overlaps among the tools.

5.2 Tools Configuration

For a fair and unbiased comparison, one needs to make sure that the tools are running with similar configurations and are based on similar resources. We have made out the following resources that the selected tools commonly use:

- A list of sources and sinks,
- a list of Android callback methods to simulate the Android lifecycle,
- an Android SDK (android.jar), and
- a specific version of the Apktool\(^3\) used to reverse engineer APK files.

5.2.1 List of Sources and Sinks

As already elaborated in Section 2.2.1, sources and sinks play a key role in data flow analysis. All the tools included in the benchmark, except for IC3, are providing their own list of sources and sinks. These lists vary heavily in length, namely from a few hundred entries to several thousand entries.

\(^3\)https://ibotpeaches.github.io/Apktool/
COVERT is using an additional filter list to parse the sources and sinks list. Mostly, these lists of sources and sinks are based on specific Android OS versions. Hence, one has to provide their own list or use a tool such as SuSi [77] to generate such a list. Listing 5.1 shows two typical entries from a list of sources and sinks.

Listing 5.1: Example sources and sinks list entry

\[\text{<android.telephony.TelephonyManager: java.lang.String getDeviceId(int)> -> } \_\text{SOURCE}\_\]
\[\text{<android.os.Handler: boolean sendMessage(android.os.Message)> -> } \_\text{SINK}\_\]

5.2.2 Original Configurations

Table 5.1 gives an overview of the different tools’ original configurations.

The number of callback methods is similar for all tools. The biggest differences in the configurations is the number of entries in the list of sources and sinks. With over forty thousand entries, HornDroid has by far the most exhaustive list. The list stems from the DroidSafe\(^4\) tool. HornDroid is followed by COVERT with just a bit over twenty five thousand entries. The other tools only have a few hundred entries in their lists of sources and sinks. For IC3 there is no list provided.

The list of Android callback methods used by all tools except IC3 is required to model the Android lifecycle during the analysis. The number of entries is similar for all tools.

FlowDroid and IC3 do not provide an android.jar file for the analysis. Hence, the user will have to provide the path to a valid Android SDK in the command used to trigger the analysis. COVERT and IccTA, on the other hand, are shipped with different versions of the android.jar that are provided together with their artefact. IccTA uses Android 4.3 (Jelly Bean) from 2013. For COVERT, the version of the Android file could not be identified. But based on the year of the paper publication, it cannot be newer than Android 5.1 (Lollipop) from 2015.

COVERT and HornDroid are using the Apktool to decompile the Android applications.

\(^4\)https://github.com/MIT-PAC/droidsafe-src
CHAPTER 5. EXPERIMENTAL SETUP

Table 5.1: Overview of tool configurations. The value "user" indicates that the user has to provide the path to the artefact in the running command.

<table>
<thead>
<tr>
<th>Name</th>
<th>Flow-Droid</th>
<th>Horn-Droid</th>
<th>COVERT</th>
<th>IC3</th>
<th>IccTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sources and sinks</td>
<td>355</td>
<td>40986</td>
<td>26591</td>
<td>N/A</td>
<td>160</td>
</tr>
<tr>
<td>Callbacks</td>
<td>182</td>
<td>183</td>
<td>185</td>
<td>N/A</td>
<td>182</td>
</tr>
<tr>
<td>Android version</td>
<td>user</td>
<td>N/A</td>
<td>≤ 5.1</td>
<td>user</td>
<td>4.3</td>
</tr>
<tr>
<td>Apktool</td>
<td>N/A</td>
<td>2.3.0</td>
<td>1.5.2</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

5.2.3 Shared Configurations

For the benchmark we use Android 6.0 (Marshmallow, API Level 23). Later versions such as Android Nougat or Oreo have caused COVERT to crash during the analysis. To identify sources and sinks for this Android version, we use the SuSi tool. It takes the android.jar as an input and based on machine learning algorithms generates a list of sources and sinks. For Android 6.0 (API Level 23), the list contains 4,050 entries.

In this list, there are entries that show additional tags after the SOURCE indication (see Listing 5.2). These tags have caused crashes with the Horn-Droid tool.

```java
<android.telephony.TelephonyManager: android.telephony.CellLocation getCellLocation()>
android.permission.ACCESS_FINE_LOCATION
android.permission.ACCESS_COARSE_LOCATION -> _SOURCE_|LOCATION_INFORMATION

Listing 5.2: Problematic source entry for HornDroid
```

Therefore, we have built an additional pre-processor for HornDroid that parses the list of sources and sinks and eliminates all these additional tags, resulting in a special list just for HornDroid.

For the list of Android callback methods, there are to the best of our knowledge no tools available to generate such a list. Hence, we have merged the existing lists from FlowDroid, IccTA, COVERT, and HornDroid to get a base list for our shared configurations.

For the Apktool, we are using the latest version 2.3.1.

Configuring the tools in such a way allows us to compare their performance solely based on the approaches and algorithms. However, it is important to note that although it is possible to configure the tools as described, the out-of-the-box experience may differ considerably. End-users who are
unaware of the internals of such tools may thus never be able to properly configure tools and will miss reports. In Section 6.2.3, we evaluate the differences in running the tools with original or with shared configurations.

5.2.4 Switching Between Configurations

In the benchmark, we want to easily change between shared and original configurations. For this purpose, we have created soft links for all shared resources and have created two shell scripts. Before running the benchmark implementation, the user can decide whether to run it in the original or the shared configuration by simply running the corresponding shell script. The scripts will then change the soft links to either point to shared resources or to the resources that the tools were provided with. This allows us to simplify the comparison between shared and original configurations.

5.3 Evaluation Process

The following section presents the evaluation process for the two datasets of applications used in this benchmark.

5.3.1 DroidBench Evaluation Process

For the small scale qualitative analysis on the DroidBench applications, we run our benchmark twice. Once with shared configuration and once with original configuration. After the first test run, we manually review the obtained reports. For each DroidBench application, we proceed as follows:

1. Create a list of all known vulnerabilities for all DroidBench applications by scanning the source code of each application for the DroidBench author’s sink indication.

2. Compare the list of known vulnerabilities with the reports generated by the tools and record which tools found the leak (true positives) and which ones did not (false negative).

3. For each additional report, record the false positives and true negatives.

After processing 125 DroidBench vulnerabilities, we have a complete list of all known vulnerabilities in the DroidBench test suite and for each tool a list of true positives, true negatives, false positives, and false negatives. This allows us to report on each tool’s precision and recall.
CHAPTER 5. EXPERIMENTAL SETUP

Precision expresses the proportion of reported real vulnerabilities amongst all reports, while recall is the ratio between reported vulnerabilities and all existing vulnerabilities. Thus, we can compare the performance of our five tools, which is not feasible by simply reviewing existing literature. Furthermore, we apply McNemar’s Test [66] which can be used since our tools run on the same configurations and the same dataset. McNemar’s Test is a statistical test for determining whether two tools are likely to report similar issues.

In the second test run, we use the initial, original configuration. Then we calculate precision and recall and compare the results with the shared configurations. Thus, conclusions on the effect of the change in configurations can be made.

5.3.2 F-Droid Evaluation Process

For the large scale quantitative analysis, we run the benchmarking implementation on the F-droid dataset. Since we do not identify the true and false positives and negatives, we can only report on the number of reported vulnerabilities and the number of overlaps among the tools. Furthermore, as the applications analysed in this step are real-world applications from the F-Droid app repository, we can draw conclusions on the security of those applications. Finally, we evaluate the hypotheses stated in section Section 3.1.
In this chapter, we report on the qualitative and quantitative evaluations. We compare the tools’ performance in terms of precision, recall and accuracy and interpret the results from McNemar’s test. Furthermore, we investigate the differences of the results when running the tools with shared configurations or with the original out-of-the-box configurations.

In the second part of this chapter, we interpret our findings from the larger scale quantitative analysis and evaluate our hypotheses from Section 3.1.

### 6.1 Small Scale Qualitative Analysis

For the small scale qualitative analysis, we evaluate and interpret the results gathered from the DroidBench dataset analysis. We first look at the results of the analysis with shared configurations. Hence, we focus on the individual tool’s performance and then on the combined performance, where we compare the performance of our benchmark implementation with the individual tools. In the end, we compare the tools pairwise.

In Section 6.2.3, we also present the results for the analysis with original configurations and try to interpret potential differences to the results with shared configurations.
6.1.1 Individual Performance

The manual analysis of the DroidBench test suite resulted in a list of 125 vulnerabilities that analysis tools ought to detect. We ran the five tools—COVERT, FlowDroid, HornDroid, IccTA, and IC3—on the DroidBench dataset and added reported leaks that are not documented in DroidBench (which are potential false positives). This resulted in a list of 269 distinct reports, including the aforementioned 125 real vulnerabilities. We reviewed the list of reports and obtained a Boolean value indicating whether DroidBench considers the finding a vulnerability or not. For each of the 269 reports, we indicate whether a particular tool has reported this vulnerability or not.

This allows us to easily make out the number of true positives (TP) and true negatives (TN), as well as false positives (FP) and false negatives (FN) for each tool.

We consider a reported vulnerability that is not described in DroidBench a false positive. We further manually checked all 144 potential false positives and verified them not to be true positives. We note that the tools only report on potential vulnerabilities and thus do not report negatives explicitly.

Table 6.1 summarises the results.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Flow-Droid</th>
<th>Horn-Droid</th>
<th>COVERT</th>
<th>IC3</th>
<th>IccTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>TP</td>
<td>99</td>
<td>99</td>
<td>8</td>
<td>4</td>
<td>97</td>
</tr>
<tr>
<td>FP</td>
<td>54</td>
<td>87</td>
<td>3</td>
<td>37</td>
<td>59</td>
</tr>
<tr>
<td>TN</td>
<td>90</td>
<td>57</td>
<td>141</td>
<td>107</td>
<td>85</td>
</tr>
<tr>
<td>FN</td>
<td>26</td>
<td>26</td>
<td>117</td>
<td>121</td>
<td>28</td>
</tr>
</tbody>
</table>

Table 6.1: Raw counts of true/false positives/negatives.

As can be seen in Table 6.1, IC3 reported the least true positives and therefore has a poor performance on the DroidBench dataset compared to the other tools. Out of 41 reported data leaks, only 4 reports were actually true positives. The remaining 37 reports were all false positives. Hence, only close to 10% of the reported vulnerabilities were in fact true. This makes it the only tool reporting more false positives than true positives. We further observe that the tool only reports leaks in certain DroidBench data leak categories such as inter-component communication, inter-app communication and emulator detection. Since most of the other tools are reporting leaks for all categories, it becomes clear why IC3 underperforms in comparison.

With just 11 reports, COVERT reported the least findings of all tools. Out of those 11 reports, 8 reports were in fact true positives. With slightly
over 70% of the reported vulnerabilities being true, COVERT is already a strong improvement compared to IC3 in terms of credibility of the results. Similar to IC3, COVERT does not report leaks for all categories of data leak vulnerabilities present in DroidBench. As we will observe later on in this chapter, it only reports data leaks in 2 out of 13 categories. We further note that COVERT and IC3 overall produce far fewer reports than the other tools, even in the categories where all tools are reporting data leaks.

For FlowDroid and IccTA, we observe similarly high numbers of true positives and false positives. Both tools report more than 12 times more true positives than COVERT and IC3. However, they also report more false positives.

HornDroid and FlowDroid report most true positives. Nonetheless, HornDroid also reports most false positives. As a known limitation, HornDroid might produce false positives for heap abstractions, exceptions and inter-app communication, which might be an explanation for the higher number of false positives.

**Precision**

To evaluate the quality of the reports, we calculate the precision, recall, and accuracy of each tool.

The precision of a tool is the ratio of correctly reported vulnerabilities to all reported vulnerabilities \((\frac{TP}{TP + FP})\). Thus, it helps us to answer the question of how many of the reported vulnerabilities are in fact real data leaks.

The closer precision, recall, and accuracy are to 1, the better is the quality of the results. Table 6.2 shows the values for each tool.

<table>
<thead>
<tr>
<th>Metric</th>
<th>FlowDroid</th>
<th>HornDroid</th>
<th>COVERT</th>
<th>IC3</th>
<th>IccTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recall</td>
<td>0.792</td>
<td>0.792</td>
<td>0.064</td>
<td>0.032</td>
<td>0.776</td>
</tr>
<tr>
<td>Precision</td>
<td>0.647</td>
<td>0.532</td>
<td>0.727</td>
<td>0.098</td>
<td>0.622</td>
</tr>
<tr>
<td>Accuracy</td>
<td>0.703</td>
<td>0.580</td>
<td>0.554</td>
<td>0.413</td>
<td>0.677</td>
</tr>
</tbody>
</table>

Table 6.2: Collected metrics for each tool as observed on DroidBench vulnerabilities. Bold values indicate maxima for the respective metric.

In terms of precision, COVERT has the best performance with over 72%. Accordingly, most of its reported vulnerabilities are actually real data leaks. One reason for this good performance can be that COVERT creates a formal
model of the security specifications of each application before starting the analysis. This allows the tool to find critical security properties that ought to be analysed.

FlowDroid and IccTA have a slightly poorer precision with over 62%, but are still on a decent level. One of the reasons for the good performance in terms of precision is their aim for a flow-sensitive analysis. Flow-sensitivity allows the tool to take the order of statements into account during the analysis. Asynchronous callbacks in Android applications is a major challenge for this case. IccTA and FlowDroid are both using control flow graphs of callbacks implemented in the application to make out the program paths of the application. Calls to Android APIs can, however, have multiple callbacks with multiple possible sequences of such callbacks depending on the call site. To handle these cases, both tools create a dummy main method emulating the application lifecycle with all possible callbacks. This is crucial to ensure flow-sensitivity and high precision. However, the construction of this dummy main method is very complex and mis-constructions can lead to missing data leaks. Nonetheless, both tools seem to overcome this difficulty with their dummy main methods.

HornDroid and especially IC3 both show high numbers of false positives compared to the number of true positives. Consequently, both tools have a poor precision. HornDroid is only partially flow-sensitive for register values, callbacks and heap locations. It might produce more false positives due to problems with heap abstractions, exceptions and inter-app communications leading to poorer precision.

Recall

Recall is the ratio between true positives and all reported true positives and false negatives \((TP)/(TP + FN)\). Hence, recall tells us how many of all vulnerabilities labelled as real data leaks are actually discovered by the tool. The recall is of particular interest, since it only includes DroidBench vulnerabilities in the calculation (true positives and false negatives).

In terms of recall, both FlowDroid and HornDroid perform equally well on the dataset with close to 80% recall. IccTA performs similarly, with a recall of slightly over 77%. Again, we state that this is mostly due to the flow-, field-, context-, and object-sensitive nature of the tools allowing them to take different program paths into account and abstract different runtime values.

COVERT and IC3 both underperform with a low recall of just 6.4% and 3.2%. Thus, these two tools miss most of the vulnerabilities present in the dataset. COVERT focuses mainly on inter-app communication and escala-
tion of privileges and IC3 is focussing mainly on abstracting complex field values. Both tools lack the focus on other categories of data leak vulnerabilities and therefore show poorer performance there.

**Accuracy**

Accuracy is the ratio of correctly reported findings to the total number of reported findings \( \frac{(TP + TN)}{(TP + TN + FP + FN)} \), i.e., how many of the reported findings are actually correct.

We observe that both FlowDroid and IccTA show a good accuracy with more than 67%, followed by HornDroid and COVERT with an accuracy of about 55% each.

IC3 underperforms again with a precision of just 9.8% and an accuracy of 41%.

**Overall Performance**

Looking at the overall performance, both COVERT and IC3 underperform and cannot compete with the other tools, with IC3 having the worst performance of the selected tools. COVERT shows the best precision, but the poor recall rate of 6.4% and the accuracy of just over 55% somewhat spoil the effect. Both tools lack a broader focus on other categories of data leak vulnerabilities and especially lack the ability to model the application’s lifecycle and handle Android callback methods.

In contrast, FlowDroid, IccTA, and HornDroid detect most of the real data leaks in the DroidBench dataset with a relatively good accuracy and precision. FlowDroid delivers the best performance, followed shortly by IccTA and then HornDroid, which suffers from the high number of false positives.

We state that these good results are mainly due to the flow-, field, object-, and context-sensitive nature of the tool and their abilities to model the application’s lifecycle and tackle the challenge of asynchronous Android callbacks.

### 6.2 Category Performance

DroidBench presents its synthetic applications in categories reflecting the type of data leak that they target. These categories range from inter-component communication to general Java types and even to implicit flows. Table 6.3 shows the recall for all available categories. This allows us to discuss on which type of data leaks the individual tools perform best and on which they face problems.
CHAPTER 6. RESULTS

<table>
<thead>
<tr>
<th>Category</th>
<th>Flow-Droid</th>
<th>Horn-Droid</th>
<th>COVERT</th>
<th>IC3</th>
<th>IccTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aliasing (0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Android Specific (11)</td>
<td>0.909</td>
<td>0.727</td>
<td>0</td>
<td>0</td>
<td>0.909</td>
</tr>
<tr>
<td>Arrays / Lists (4)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.667</td>
</tr>
<tr>
<td>Callbacks (21)</td>
<td>0.762</td>
<td>0.905</td>
<td>0</td>
<td>0</td>
<td>0.810</td>
</tr>
<tr>
<td>Emulator (23)</td>
<td>0.870</td>
<td>0.957</td>
<td>0</td>
<td>0.043</td>
<td>0.870</td>
</tr>
<tr>
<td>Field-/obj.-sensit. (3)</td>
<td>1</td>
<td>0.667</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>General Java (20)</td>
<td>0.750</td>
<td>0.800</td>
<td>0</td>
<td>0</td>
<td>0.800</td>
</tr>
<tr>
<td>Implicit Flows (9)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Inter-app comm. (8)</td>
<td>0.750</td>
<td>0.500</td>
<td>0.375</td>
<td>0.250</td>
<td>0.750</td>
</tr>
<tr>
<td>ICC (18)</td>
<td>0.944</td>
<td>0.833</td>
<td>0.278</td>
<td>0.056</td>
<td>0.889</td>
</tr>
<tr>
<td>Lifecycle (1)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Reflection (4)</td>
<td>0.750</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.750</td>
</tr>
<tr>
<td>Threading (5)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.600</td>
</tr>
</tbody>
</table>

Table 6.3: Recall for each tool as observed on DroidBench for each category. Bold values indicate maxima for the respective metric and the value in parentheses indicates the number of data leaks present in this category. ICC stands for Inter-component communication.

We can observe that COVERT and IC3 only report vulnerabilities in the field of inter-component and inter-app communication, with the latter also reporting some vulnerabilities for emulator detection. We can see that neither tool reports any leaks for the callback or lifecycle category. As discussed before, this has an impact on the overall precision of the tools. Taking that into account, it becomes apparent why the overall performance is poor compared to the other tools. Even in the two aforementioned categories, the two tools still only detect below 40% of the present data leaks.

Initially, COVERT was built only for detecting escalation of privileges with a special focus on inter-app communication. Later, it was extended with FlowDroid to also use taint analysis. The more surprising is the different behaviour from the standard FlowDroid tool. Even for inter-component and inter-app communication, COVERT recalls much fewer vulnerabilities than its competitor FlowDroid. Therefore, there must be internal algorithms in COVERT influencing and filtering the results.

We further note that none of the tools is able to detect data leaks for implicit information flows. HornDroid mentions this as a known limitation for its analysis. On the other hand, FlowDroid claims to support the detec-
CHAPTER 6. RESULTS

34

The addition of implicit flows\(^1\) comes optionally with the command `-implicit`, which was not used in our tests since we wanted to benchmark the tools in their out-of-the-box configuration. We executed additional test runs for FlowDroid with this additional option but could not observe any changes in the result. COVERT, IC3, and IccTA support implicit intents, but do not mention any additional support for implicit information flows in general.

FlowDroid, HornDroid, and IccTA show good recalls over all categories. All of them report the highest recall in at least four categories each. Nonetheless, HornDroid recalls only 50% of inter-app communication vulnerabilities and IccTA only 66.7% for arrays and lists. Apart from these two dips, the three tools in general recall more than 75% of all present data leaks over all categories.

6.2.1 Combined Performance

The idea of our benchmarking implementation is to facilitate and automate the analysis of several tools, which enhances the reproducibility of our results and improve scalability. However, we are also interested in whether our own implementation can leverage the different base approaches and improve the recall. Hence, we are interested in knowing how many vulnerabilities are reported by at least one tool.

We observe that, when simply aggregating the results of all tools, this is the case for 113 of the 125 real DroidBench data leaks—which means a recall of 90.4%—topping the best individual recall by 11.2%. This is surprisingly high, considering the variety of applications and types of data leak vulnerabilities in DroidBench. Clearly, this comes at the cost of more false positives, since they would also increase in the combined approach.

Nonetheless, a future tool could consider how many of the approaches report the same leak. If there are multiple, possibly fundamentally different approaches that report the same leak, then we could report this leak with a higher confidence than if just a single tool would have reported it. Table 6.4 shows the probability of a true positive, if two specific tools both report the same potential vulnerability. On the diagonal, we can see the individual probability—so, how likely the tool reports a true data leak. For HornDroid, for example, this would mean that 52.9% of all reported vulnerabilities are actually real data leaks.

We note that cases where both tools report a negative are not consid-

\(^1\)https://blogs.uni-paderborn.de/sse/2013/10/01/flowdroid-implicit-flows/
ered, since our DroidBench test suite and the tools themselves focus only on positives.

<table>
<thead>
<tr>
<th></th>
<th>FlowDroid</th>
<th>HornDroid</th>
<th>COVERT</th>
<th>IC3</th>
<th>IccTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlowDroid</td>
<td>0.643</td>
<td>0.731</td>
<td>0.700</td>
<td>0.231</td>
<td>0.631</td>
</tr>
<tr>
<td>HornDroid</td>
<td>0.529</td>
<td>0.700</td>
<td>0.700</td>
<td>0.700</td>
<td>0.700</td>
</tr>
<tr>
<td>COVERT</td>
<td>0.727</td>
<td>0.098</td>
<td>0.700</td>
<td>0.286</td>
<td>0.618</td>
</tr>
<tr>
<td>IC3</td>
<td></td>
<td></td>
<td>0.098</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IccTA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4: Probabilities of a real data leak when two analyses report the same potential vulnerability. The bold value indicates the maxima. Values on the diagonal are the base values of a single tool.

If we look at vulnerabilities that were reported by HornDroid and IccTA, then we can say that 70% of the reported vulnerabilities are real data leaks. In fact, we can leverage the individual probabilities by a certain combination with another tool in nearly all cases. Only for COVERT can we see a slight decrease in the probability when looking at combined results. We also observe that combinations with IC3 are problematic, since its poor performance has a negative impact on the combined performance. Nonetheless, we can increase IC3’s performance when looking at the combined results with FlowDroid or IccTA.

We note that the best combination of tools is FlowDroid and HornDroid. Hence, when both FlowDroid and HornDroid report the same vulnerability, then it is a real data leak with a probability of 73.1%. By including IccTA as well, we observe a real data leak with a probability of 71.9%. However, this comes at the cost of a lower recall. If we only look at the data leaks that all three tools report, then the recall is just 65.6%.

We recognise the importance of still listing all reported vulnerabilities to the user to ensure a high recall whilst classifying the findings depending on the tool combination to indicate the confidence level for each finding.

6.2.2 Pairwise Comparison

Here, we answer the question of how similar every two tools are in the detection behaviour compared to each other. To obtain such a measure expressing similarities, we apply McNemar’s Test [66]. To do so, we obtain the following four numbers for each pair:
• The number of times both tools report correctly (TP+TN)
• The number of times both tools report incorrectly (FP+FN)
• The number of times one tool reports correctly and the other does not
• The number of times the other tool reports correctly and the first one does not

Table 6.5 summarises the collected data.

<table>
<thead>
<tr>
<th></th>
<th>HornDroid</th>
<th>COVERT</th>
<th>IC3</th>
<th>IccTA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>FlowDroid</td>
<td>T</td>
<td>121</td>
<td>68</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>33</td>
<td>45</td>
<td>52</td>
</tr>
<tr>
<td>HornDroid</td>
<td>T</td>
<td>62</td>
<td>92</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>87</td>
<td>26</td>
<td>93</td>
</tr>
<tr>
<td>COVERT</td>
<td>T</td>
<td></td>
<td>104</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td></td>
<td>7</td>
<td>113</td>
</tr>
<tr>
<td>IC3</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.5: Pairwise comparison of tools. For each pair, we list four numbers, denoting the various combinations of correct and incorrect classification. T denotes a true positive or negative, F denotes a false positive or negative. For example, IC3 and IccTA both classify 62 times correctly, while in 120 cases, IC3 classifies incorrectly and IccTA correctly.

We apply McNemar’s Test by determining if there is a statistically significant difference between two tools. This is done by calculating the $\chi^2$ value for each pair of tools:

$$\chi^2 = \frac{(|n_{01} - n_{10}| - 1)^2}{n_{01} + n_{10}}$$

In the formula above, $n_{01}$ and $n_{10}$ indicate the number where one tool reports correctly and the other one does not and vice versa. Under the null hypothesis, the two tools perform equally well. We choose a significance level of $\chi^2_{1,0.01} = 6.635$, which corresponds to a confidence interval of 99%. Higher values indicate a statistically significant difference between the performance of the pair of tools. Lower values indicate that the null hypothesis holds with a probability of at least 99% and that the tools perform similarly.
Table 6.6 summarises the results for McNemar’s Test, with bold values indicating pairs of tools with statistically significant differences in their performance.

<table>
<thead>
<tr>
<th></th>
<th>HornDroid</th>
<th>COVERT</th>
<th>IC3</th>
<th>IccTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlowDroid</td>
<td>9.94</td>
<td>10.56</td>
<td>35.29</td>
<td>2.77</td>
</tr>
<tr>
<td>HornDroid</td>
<td>0.2</td>
<td>8.53</td>
<td>6.01</td>
<td></td>
</tr>
<tr>
<td>COVERT</td>
<td>26.33</td>
<td></td>
<td>6.97</td>
<td></td>
</tr>
<tr>
<td>IC3</td>
<td></td>
<td></td>
<td>28.99</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.6: $\chi^2$-values from McNemar’s test for each pair of tools. Bold values indicate a statistically significant difference.

Based on the results from McNemar’s Test, we note that most pairs of tools show a statistically significant difference in their performance. Only for three pairs of tools can we say that the null hypothesis holds and the tools perform similarly. This is the case for the pairs COVERT and HornDroid, HornDroid and IccTA and last IccTA and FlowDroid.

For the pair, HornDroid and COVERT, we can see that the tools are not statistically significantly different and that the null hypothesis holds. This is surprising, considering the vastly different sets of reported vulnerabilities, especially considering the different number of reported findings.

Even though McNemar’s test is suited to compare classifiers in general, it may not be best suited for this case since it only considers the disagreements, which are evidently close to each other in that case. However, the absolute number of disagreements is very high with 179 (87 + 92). In this case, the test fails to identify the significant differences.

Looking at the presented metrics in this chapter so far, we can postulate that FlowDroid performs best based on precision, recall and accuracy, followed closely by IccTA and HornDroid. Both FlowDroid and HornDroid can be considered statistically significantly similar to IccTA in terms of their performance. Therefore, we would expect a similar behaviour for real-world applications.

COVERT is a precise tool, but lacks a good recall. IC3 cannot compete with the other tools and shows a poor performance among all metrics. Accordingly, it can be considered statistically significantly different to all the tools included in this benchmark.

We showed that our benchmarking implementation can leverage the different base approaches to achieve higher recall. However, this comes at the
cost of more false positives. Nonetheless, we can present combinations of tools that, whilst agreeing on vulnerabilities, increase the confidence in the results and probability of true data leaks. Such a pair is FlowDroid and HornDroid. Therefore, we can overcome the disadvantage of presenting more false positives to the user by ranking the reports depending on the level of agreement between combinations of tools.

### 6.2.3 Differences to Original Configurations

While we based this benchmark mainly on running the tools with similar configurations, we should also consider examining them in their out-of-the-box configuration. In the end, this is the setup that the users unfamiliar with the internals will be using.

In this section, we are interested in the knowing if there is a difference in running the tools with shared or original configurations. Table 6.7 summarises the findings.

<table>
<thead>
<tr>
<th>Metric</th>
<th>FlowDroid</th>
<th>HornDroid</th>
<th>COVERT</th>
<th>IC3</th>
<th>IccTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reports</td>
<td>−0.355</td>
<td>−0.091</td>
<td>+0.364</td>
<td>0</td>
<td>−0.208</td>
</tr>
<tr>
<td>True Positives</td>
<td>−0.356</td>
<td>+0.077</td>
<td>0</td>
<td>0</td>
<td>−0.293</td>
</tr>
<tr>
<td>False Positives</td>
<td>−0.352</td>
<td>−0.271</td>
<td>+0.800</td>
<td>0</td>
<td>−0.097</td>
</tr>
</tbody>
</table>

Table 6.7: Calculated relative change in selected metrics from shared configurations to original configurations. Bold values indicate a positive effect on the quality of results.

We observe that there are indeed differences in running the tools with shared or with original configurations. We note that FlowDroid reports 35.5% less vulnerabilities with the original configurations. However, the ratio of true and false positives remains approximately the same with a slightly negative impact on precision and recall. It misses more reports, but at the same time also reports similarly fewer false positives. With original configurations, FlowDroid’s list of sources and sinks only holds 355 entries as with shared configuration, the list has 4,050 entries. This can explain the higher number of reported leaks with shared configurations, since more data flows between sources and sinks can be analysed.

HornDroid also reports slightly less vulnerabilities. Nonetheless, whilst it reports around 27.1% less false positives, the number of true positives
increases. This has a positive effect on the tool’s recall and precision. Consequently, the tool achieves better results in the original configurations.

COVERT, on the other hand, reports more false positives at a similar number of true positives and therefore makes its performance worse. This again is surprising, since it uses FlowDroid for its taint analysis, which behaved differently.

For IC3, there is no change visible in running it with shared or original configurations. This is mainly because most resources must be provided by the user in the running command. In those cases, we used the same resources as arguments in both runs.

IccTA behaves similarly to FlowDroid with an overall decreasing effect. The number of findings decreases by 20.8%. Precision and recall for IccTA weaken, since the declining number of true positives is relatively bigger than the one of false positives.

In Section 5.2.2 we discussed the highly diverging number of entries in the list of sources and sinks. With shared configurations, the number of entries in the list of sources and sinks increased by factor 11 for FlowDroid and by factor 25 for IccTA. However, we do not observe a similarly strong change for the number of findings or the number of true and false positives. This leads us to the conclusion that both lists of sources and sinks are already pretty complete in terms of potentially leaky sinks and source—in spite of the low number of entries.

On the other hand, the initial list was reduced by 84% for COVERT and for HornDroid by even more than 90%. COVERT reported almost the same number of vulnerabilities with the reduced list, but with more false positives in shared configurations. Surprisingly, HornDroid reported more vulnerabilities with the reduced list, but with a slight decline in true positives and more false positives.

Overall the number sources and sinks does have an impact on the number of findings and the quality of the results. However, a longer list does not necessarily lead to better results, as observed with COVERT. All tools using lists of sources and sinks reacted to a change in the number of entries. Nonetheless, there might be additional internal factors such as special filters, mechanisms or algorithms that boost the effect.

6.3 Large Scale Qualitative Analysis

In this section, we discuss the results for the large scale analysis on applications from the F-Droid store. In total, we have analysed 250 randomly
selected open source applications. Since we cannot identify true and false positives and negatives for every application, the metrics precision, recall, accuracy, and McNemar’s Test cannot be applied. Therefore, we base our evaluations on the following metrics:

- **Number of reported leaks** to compare the tools amongst each other and with the observed behaviour on the *DroidBench* dataset.
- **Number of overlaps** to validate the results from McNemar’s test.
- **Prevalence of sink methods** to give indications on which data leaks are common with real-world applications.
- **Number of timeouts** to draw conclusions on the performance within the given time frame.

Each metric will be discussed in a separate section. We must note here that *HornDroid* had to be excluded from the *F-Droid* analysis. For a sample set of 50 applications, *HornDroid* produced a timeout (set to one hour) for every application. Even with a higher timeout of three hours, *HornDroid* did not finish the analysis a single time. For this reason, we have excluded *HornDroid* from the *F-Droid* analyses, since it does not produce reports within a reasonable amount of time.

### 6.3.1 Number of Reported Leaks

In the *DroidBench* analysis, we have already observed big differences in the number of reported vulnerabilities between the tools. Especially, *COVERT* and *IC3* reported only very few vulnerabilities, whereas *FlowDroid* and *IccTA* produced far more reports. Here, we are interested in whether this trend can also be observed in the *F-Droid* analysis. Table 6.8 shows the results for the number of reported leaks.

<table>
<thead>
<tr>
<th>Metric</th>
<th><em>FlowDroid</em></th>
<th><em>COVERT</em></th>
<th><em>IC3</em></th>
<th><em>IccTA</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaks</td>
<td>1735</td>
<td>1</td>
<td>3353</td>
<td>0</td>
</tr>
<tr>
<td>Average leaks</td>
<td>9.971</td>
<td>0</td>
<td>14.088</td>
<td>0.004</td>
</tr>
</tbody>
</table>

Table 6.8: Number of reported leaks for each tool and average number of leaks per application.

In total, the tools produced 5052 distinct reports. We observe that only *FlowDroid* and *IC3* frequently report vulnerabilities with an average of close
to 10 to 14 reports per application. We note that IC3 reports close to twice as many vulnerabilities as FlowDroid. However, we have little confidence in the reports from IC3 considering its poor performance on the DroidBench test suite. Only close to 10% of all reported leaks were in fact true positives there. We predict a similar behaviour on the F-Droid test suite and suspect that most of IC3’s reports are false positives.

A possible root cause for the high number of false positives of IC3 can be the value-insensitive nature of the analysis. Value-sensitivity allows the analysis to approximate runtime values and skip unreachable program branches [86]. Because IC3 is context-sensitive, the tool can compute different static approximations upon different method calls, resulting in more program branches to be checked during the analysis. However, since the tool is value-insensitive, it cannot skip unreachable program branches, resulting in the detection of non-existent data flows and therefore more false positives. Nonetheless, verifying this is beyond the scope of this work and remains as future work.

FlowDroid, on the other hand, showed a good performance on the DroidBench test suite. We therefore suspect that the tool detected most of the existing vulnerabilities in the real-world applications. Especially, since the tool is both context- and value-sensitive, it does not face similar issues to IC3 and reports less false positives.

COVERT only reports one vulnerability confirming the behaviour observed on DroidBench. The tool focusses on the detection of inter-app communication vulnerabilities and especially on the detection of unsafe combinations of applications that could lead to security exploits. Since we analysed the applications one by one, this could have an impact on internal mechanisms of the tool resulting in fewer reports.

IccTA cannot continue its good performance from the DroidBench test suite and does not report any vulnerabilities. The major issue here is that the tool only completed the analysis for 13 out of 250 applications. For the remaining 237 applications, the analysis was aborted due to internal exceptions or due to timeouts.

Given the poor results of IccTA we cannot confirm the statistically significant similarities observed in DroidBench between FlowDroid and IccTA.

6.3.2 Number of Overlaps

In the DroidBench dataset we observed a lot of overlaps between the tools, especially between FlowDroid and IccTA.

Table 6.9 summarises the results for the F-Droid dataset.
Table 6.9: Number of overlaps for all pair of tools.

<table>
<thead>
<tr>
<th></th>
<th>FlowDroid</th>
<th>COVERT</th>
<th>IC3</th>
<th>IccTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlowDroid</td>
<td>1.735</td>
<td>0</td>
<td>37</td>
<td>0</td>
</tr>
<tr>
<td>COVERT</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IC3</td>
<td>3.353</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IccTA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

We only observe overlaps between FlowDroid and IC3, which can be expected given the low number of reports for the other tools. We note that over all 5,052 reported vulnerabilities, there are only 37 overlaps. So, in only 0.7% of the cases, there is an agreement between the tools. This is very surprising, given the different situation in the DroidBench analysis. There, about 59% of all reported leaks showed overlaps between the tools. For the DroidBench test suite we have calculated the probability of a real data leak when two particular tools report the same vulnerability. For the pair of FlowDroid and IC3 this probability was 23.1%. For the 37 overlaps, this would mean that fewer than 9 reports are in fact true positives.

We can state that the tools’ agreement for real-world applications diverge and the reports are much more heterogeneous than with the synthetic DroidBench applications.

One reason might be that tool developers used DroidBench to evaluate their tools and therefore optimized them to achieve high precision and recall on this dataset. For real-world applications, there are other factors impacting the tools’ performance such as obfuscation or the size of the application. Thus, a good performance on the DroidBench dataset does not necessarily yield good results for real-world applications. This confirms the necessity of evaluations not only on synthetical applications, but also on real-world applications.

6.3.3 Prevalence of Sink Methods

In this section, we are interested in knowing the prevalence of particular data leaks in real-world applications. The analysis on the F-Droid dataset allows us to observe increasing numbers of occurrences of certain sink methods. This could give us an indication what developers should try to avoid whilst developing their application. Out of the 5,043 reports, we can see only 265 distinct sink methods. Table 6.10 shows the five most common sinks.

We note that the five most frequently vulnerabilities together make up
Sink Method                  | Total | Percentage |
--------------------------------|-------|------------|
startActivity(Intent)          | 968   | 0.192      |
getStringExtra(String)         | 337   | 0.067      |
startActivityForResult(Intent, int) | 274   | 0.054      |
getString(String)              | 233   | 0.046      |
startService(Intent)           | 203   | 0.040      |

Table 6.10: Top 5 of most often reported sink methods and absolute as well as relative number of occurrences.

39.9% of all reported leaks. The vulnerability that was reported most is the call to the `startActivity(android.content.Intent)` method. The Intent passed as argument describes an action to be performed and carries necessary data that might be required. An activity presents a single screen within the application. So, the `startActivity(android.content.Intent)` method will start a new instance of an Activity and perform the defined action with the passed data. If this data contains sensitive information, it will leave the current component. In that case, a data leak occurs. This sink method can be categorized as an inter-component communication (ICC) data leak.

Also the methods `startActivityForResult(android.content.Intent, int)` and `startService(android.content.Intent)` pass an Intent to another component and belong to the same category. In fact, such ICC API calls starting another service or activity make up around 28.9% of all reported vulnerabilities. We state that they are the most prevalent type of data leak vulnerabilities in our set of real-world applications.

The security concern with ICC is that we do not have control over what happens to the sensitive information passed to the other component. The Intent message could be intercepted during message passing or the receiving component could for example send out the sensitive information from the Intent to a phone number as an SMS [64]. However, these kinds of sink methods can only be considered data leaks when the Intents actually contain sensitive information. Verifying this is beyond the scope of this work and remains as future work.

The remaining two sink methods in our top five are also related to Intent handling. The `getStringExtra(java.lang.String)` method and `getString(java.lang.String)` method can be used to extract information from the Intent such as the String extra, which could contain for example the device UID. However, given our definition of sink methods and data leaks from Section 2.2.1, we cannot consider reading such data from Intents.
as a sink method leaking sensitive information. Consequently, we suspect all these cases to be false positives. In fact, these method calls were only reported by IC3, where we already commented on the poor confidence level in the results.

We also note the high prevalence of log activities among all reports. In summary, there are 341 log activity sink methods; making these kinds of sink methods the actual second most prevalent of all reports with a total share of 6.8%. An example of such an API call is the Log.d(java.lang.String, java.lang.String) method. Here, a DEBUG message is sent containing a piece of potentially sensitive information as a String text to the log files. Log files are shared resources in Android. Therefore, they could be accessed by other malicious applications to obtain the sensitive information\(^2\). Prior to Android 4.0, any application with the READ_LOGS could access the logs of any other application using the command-line tool LogCat\(^3\). Since Android 4.1, an application can only access its own logs. However, using a PC, one can still obtain log output from other applications\(^4\).

To mitigate this issue, Google advises programmers to limit log usage and to use debug flags or custom Log classes\(^5\).

### 6.3.4 Timeouts

From the 250 analysed applications, there are cases where one or several of the tools was aborted due to time outs or exceptions.

Table 6.11 summarises these cases.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Flow-Droid</th>
<th>COVERT</th>
<th>IC3</th>
<th>IccTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timeouts</td>
<td>76</td>
<td>0</td>
<td>9</td>
<td>118</td>
</tr>
<tr>
<td>Exceptions</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>119</td>
</tr>
<tr>
<td>Completed</td>
<td>174</td>
<td>250</td>
<td>238</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 6.11: Number of timeouts, exceptions and completed analyses for each tool.

\(^2\)https://blogs.uni-paderborn.de/sse/2013/05/17/privacy-threatened-by-logging/
\(^3\)https://developer.android.com/studio/command-line/logcat
\(^4\)https://wiki.sei.cmu.edu/confluence/display/android/DRD04-J.+Do+not+log+sensitive+information
\(^5\)https://developer.android.com/training/articles/security-tips
We observe that COVERT and IC3 nearly always finished the analysis without reaching the timeout or throwing an exception. Therefore, the tools can be considered time efficient. For FlowDroid we can see that in about 30% of the cases a timeout was reached. IccTA produced most timeouts and is therefore time intensive compared to the other tools. In fact, the analysis completed only in 5.3% of all cases for IccTA. In 47.6% of all cases, an exception was thrown such as a RuntimeException. These exceptions must have internal root causes in the IccTA tool. We suspect that larger applications or code obfuscation cause such issues with IccTA. However, verifying this is beyond the scope of this work and remains as future work.

6.4 Hypotheses Evaluation

In this section, we evaluate the hypotheses from Section 3.1.

**H1** The more tools report the same vulnerability, the more likely it is a true positive.
In Section 6.2.1 we showed that when a particular combination of tools report the same vulnerability, then the probability of it being a true data leak increase. The best combination is the pair FlowDroid and HornDroid.

**H2** The fewer tools report a certain vulnerability, the more likely is a false positive.
This can also be observed in Section 6.2.1. However, it depends on the underlying tool reporting the vulnerability. For tools with a poor performance such as IC3 it is certainly true. There are also cases where a tool covers edge cases ignored by other tools. Therefore, we cannot say that this hypothesis is true in general.

**H3** If a tool reports much more data leaks than all the other tools, then it is likely to report more false positives than the others.
This is the case for IC3 on the F-Droid data set. However, verifying the reports from being true or false positives is beyond the scope of this work and remains as future work.
In this chapter, we point out threats that might impact the validity of the results. We validate our benchmarking implementation and assess impacts of other threats.

### 7.1 Benchmarking Implementation Validation

Our implementation may contain bugs that directly influence the results. Possible areas of bugs that could impact the results are during the analysis triggering, parsing of the results, or writing to the summary files. Especially parsing the results files from the different tools is not trivial, since the results usually do not have a standardised format and differ in the amount of information they show. To mitigate this threat, we implemented unit tests during the development to ensure that produced results are correctly parsed. Furthermore, we verified the generated output manually by comparing the results from the individual tools with our summarised reports and with the manually gathered data from *DroidBench*.

Our validation shows that about 90% of all *DroidBench* reports were correctly parsed for class name, method name, and sink method. In the remaining reports, one of three issues impacts the validity of the reports:

1. **FlowDroid reports are missing information.** In some cases, *FlowDroid* shows the class and method name of the source instead of the ones where the sink occurs. In fact, this is an issue of *FlowDroid* itself.
CHAPTER 7. THREATS TO VALIDITY

Hence, we are only able to identify it by manually checking within the application source code whether the indicated class and method contain the reported sink method. However, our parser cannot automatically detect if the reported class and method name belong to the sink or the source. Since we manually analysed the DroidBench reports, we were able to identify these cases and correct them.

Consequently, this issue can only impact the F-Droid results in that some of the FlowDroid reports show the wrong class or method name, but the correct sink method. In the DroidBench dataset, we observed this FlowDroid issue in 13 out of 141 reports; so, in about 9.2% of all FlowDroid reports.

2. IccTA reports are parsed for the wrong class and method name. IccTA vulnerability descriptions are extensive. Depending on the vulnerability, they can contain additional information such as special register values or path descriptions. Class and method name indicating where the sink method occurs can therefore change the location within the description. We have identified seven different possible locations and created the corresponding unit tests for these cases. We have invested a considerable amount of work to optimise the IccTA parser and are now able to identify the correct class and method in six out of these seven test cases.

In addition, we manually verified our parser on the DroidBench dataset. It correctly parsed 64% of all IccTA reports. In 52 cases, the parser was returning the correct sink, but showed an issue with the class or method name. In 42 cases, the class name was correct, but the method name was incorrect. In 8 cases, both—i.e., class name and method name—were incorrect. We note that the reported sink method was in all cases correctly parsed.

To mitigate this threat, a future project could modify the original IccTA tool to return the class name, method name and sink method using API calls. With this approach, parsing the results file would not be required anymore.

3. COVERT is not reporting the sink method. This tool only reports the class and method name where the sink occurs, missing the sink method name in the report. However, COVERT indicates the sink type such as SEND_SMS. Based on these sink types, we could manually identify if a sink method matching the sink type was present in the reported class and method. Nonetheless, our benchmarking tool
cannot do this automatically. Consequently, the \textit{F-Droid} analysis never shows an overlap of a \textit{COVERT} report with another tool’s report, since it misses the sink method that would be required to identify such an overlap. For the \textit{DroidBench} analysis however, we were able to overcome this issue by manually checking whether the sink type can be referenced with a sink method in the indicated class and method.

7.2 \textbf{DroidBench Validation}

We assume that the synthetic \textit{DroidBench} applications do not contain other vulnerabilities than the ones that are indicated by the authors of this test suite. By manually analysing potential false positives reported by the tools, we could mitigate this threat. In fact, none of the reported false positives turned out to be a true positive. Hence, there was no vulnerability missing in the test suite.

However, it is still possible that our manual check misclassified such true or false positives due to lack of expertise. Due to the small size of the programs in DroidBench, we are, however, confident that these vulnerabilities would have been detected by the community, the original authors, or us.

\textit{DroidBench} and \textit{FlowDroid} originate from the same research group. Hence, a selection bias that favours \textit{FlowDroid} is possible. Nonetheless, other tools such as \textit{HornDroid} or \textit{IccTA} also used \textit{DroidBench} for their evaluation. To mitigate this threat, other sets of applications with known vulnerabilities could be used in the future such as the one provided by Mitra \textit{et al.} [68]. Creating and maintaining such a set is out of scope for this thesis and remains as future work.

7.3 \textbf{Misconfiguration}

We have used the tools included in this benchmark as distributed. Furthermore, we have followed the provided instructions to set up and run the tools. Nonetheless, it is possible that we have misconfigured some of them. We mitigate this threat by only making minimal changes to shared configurations (as described in Chapter 5).

Our choice to normalise configurations and run the tools with shared configurations may impact the results. We mitigate this threat by also running the tools in the original configurations and by pointing out differences in the results (Section 6.2.3). Nevertheless, we argue that the threat is minimal and that using the same configuration for the tools is a sensible choice.
8

Conclusions and Future Work

8.1 Summary

In this work, we investigate to what degree tools assessing data leak vulnerabilities in Android applications are available, and how they perform in practice compared to each other. We report the process of elimination during the tool selection process, and how, from an initial list of 87 vulnerability detection tools, we arrive at 5 available tools focusing on data leak vulnerability detection.

We present our Java implementation of an automated benchmark to automatically execute the tools and to consolidate the results. To benchmark the tools, we evaluate them in a small scale qualitative study on the DroidBench dataset with synthetical applications with known data leak vulnerabilities. In addition, we perform a large scale quantitative study on the F-Droid dataset of real-world applications.

We then present and discuss the results, where we report on several aspects of the tools. Within the context of the small scale qualitative analysis, we discuss values such as precision, recall and accuracy for each tool. We further perform a pairwise comparison of the tools using the McNemar’s Test [66]. Moreover, we evaluate differences stemming from running the tools with shared or with original configurations.

For the large scale quantitative analysis, we use real-world applications. Using real applications makes it considerably harder to determine precision
and recall of the tools; we would need to manually determine whether a report is a false positive, which is a challenging task, even for experts in the domain. Hence, we discuss values such as number of reported vulnerabilities or the number of overlaps. In addition, we evaluate the prevalence of certain sink methods in real-world applications.

We show that our benchmarking implementation can leverage different base approaches to achieve higher recall rates. It improves the state of the art in that it provides an analysis of five tools in the same domain. To ensure a fair comparison, we configured the tools to use the same configurations for the analysis and apply them to the same targets.

8.2 Conclusions

In this work, we arrive at the following main conclusions:

• **Tool availability is poor.**
  Even though there exist hundreds of tools in the domain of Android Security, we observe that only few of them are available for a specific sub-domain such as data leak vulnerability detection. We further note that the obtainable tools are often out of date and face issues when being used with newer Android versions.

• **There are huge differences in how the tools perform in practice.**
  We observe that, even though tools show similar structures on paper, they perform quite differently when being compared in practice. Consequently, we see considerable differences in key metrics such as precision, recall or accuracy.

• **Different lists of sources and sinks have an impact on recall and precision.**
  The tools’ original configurations and especially the number of entries in the list of sources and sinks differ tremendously. However, a long list is no guarantee for detecting more vulnerabilities as observed with FlowDroid, IccTA, and HornDroid. Changing the list of entries impacts both precision and recall for each tool differently.

• **Tool evaluation on real-world applications is important.**
  A good performance on the DroidBench test suite with synthetic applications is no guarantee that the tools perform equally well for real-world applications. HornDroid and IccTA show a good performance
on the DroidBench test suite, but face major issues with real-world applications such as timeouts or exceptions. We note the importance in evaluating tools not only on synthetical applications with known vulnerabilities, but also on real-world applications. Hence, large applications or obfuscation as seen in such applications can impact the tools performance considerably and should be considered during the tool’s evaluation.

- **FlowDroid has the best performance.**
  In comparison to the other tools FlowDroid shows the best overall performance. It detects most vulnerabilities on the DroidBench test suite with a good precision. It is the only tool reporting credible results for real-world applications and can in most cases complete its analysis within the time limit of one hour.

- **Leveraging different base approaches can lead to higher recall.**
  We showed that our own benchmarking implementation can leverage the different base approaches to achieve higher recall. However, it comes at the cost of more false positives. We propose to rank reports depending on the level of agreement between the tools to overcome this threat.

- **The most prevalent sinks in real-world applications are log activities.**
  The analysis on real-world applications shows that log activities such as `Log.i` make up a considerable amount of all reported vulnerabilities. A future work could further investigate the usage of log activities in applications and to what extent they pose a security threat.

### 8.3 Future Work

Future work could include adding additional tools to the benchmark, either in the same domain or in other domains such as escalation of privileges. Nowadays, a developer wanting to evaluate the overall security of an application is required to use several tools from different security domains. By including other domains in the benchmark, we could overcome this issue and deliver an overall security evaluation for an application in just a few steps. Furthermore, the benchmarking implementation could be extended to show the confidence level in each finding, depending on which tools reported it and if there are overlaps between the tools.
Case studies on larger applications such as Facebook or Candy Crush could be considered to further evaluate the tools’ performance on larger applications.

Future efforts should validate HornDroid for real-world applications with higher timeouts. This would allow us to evaluate whether HornDroid does in general not work for such applications or if it is just very time intensive.

Finally, another direction we could explore is to include the benchmark suite in development tools. Here, the idea is to run analyses on demand, for example, on the program that is currently open in an integrated development environment (IDE). We then can provide the reports directly to the programmer, who can benefit from data from multiple tools, while having to deal with a single, easy to understand interface and report format.
First, I would like to thank my thesis advisor Claudio Corrodi of the Software Composition Group at the University of Bern. His door was always open for questions, feedback, and active support, which steered the project in the right direction whenever needed. I highly appreciated the open exchange with him on the topic. Furthermore, his passionate participation and interest helped the project to thrive considerably and motivated me even more.

Thanks to some extra efforts on all sides, we were able to submit an additional paper on the topic of ”Benchmarking Android Data Leak Detection Tools” for the International Symposium on Engineering Secure Software and Systems (ESSoS 18)\(^1\), which was accepted as an idea paper [27].

I would also like to thank the experts who were involved in the validation either of the aforementioned research paper or this thesis here. Special thanks go to Dr. Mohammed Ghafari for his valuable inputs during the project.

Furthermore, I would also like to acknowledge Prof. Oscar Nierstrasz of the University of Bern as the second reader of this thesis. I am grateful for his valuable comments and inputs on the thesis.

Finally, I must express my very profound gratitude to my family. They provided me with unfailing support and continuous encouragement throughout my years of study as well as during this project. This accomplishment would not have been possible without them. Thank you!

Anleitung zu wissenschaftlichen Arbeiten

The “Anleitung zu wissenschaftlichen Arbeiten” consists of:

- The ESSoS ’18 paper “Idea: Benchmarking Android Data Leak Detection Tools” (C. Corrodi, T. Spring, M. Ghafari, O. Nierstrasz) [27], and

- the instructions to setting up the benchmark in the following pages.
A.1 Tool Setup

Our benchmarking implementation is available on GitHub\(^1\). This manual will lead you through the project and tool setup and explains how to use our benchmarking implementation to reproduce the results.

The benchmark can be set up in four steps:

1. Setup static analysis tools,
2. run `gradle shadowJar` to generate build,
3. Adapt the properties file and change the paths to the tools,
4. Run the tool with `java -jar benchmarking.jar`

We start with the setup of the tools included in the benchmark—`COVERT`, `FlowDroid`, `HornDroid`, `IC3`, and `IccTA`.

The GitHub page contains the following folders that we used for running the analysis. One can either use our proposed folder structure or adapt the properties file with the modified paths to the required artefacts.

We propose the following folders:

- **src**
  Contains the source code to our own benchmarking implementation to run all tools and collect and summarise the results.

- **tools**
  For each tool included in the benchmark, there exists a separate folder. The tools’ artefacts and resources will need to be placed there. During the analysis, the tools will create results files in those folders that will be used by our benchmarking implementation.

  The tools folder also contains a `commonConfig` folder containing all shared resources used for the analysis. To run the tools with shared configurations, you have to provide the following resources in the `commonConfig` folder:

  - **android.jar** We used API level 23 for this work. There are, however, good collections for different API levels available online\(^2\).

\(^1\)[https://github.com/tiimoS/distilldroid]
\(^2\)[https://github.com/Sable/android-platforms]
– *SourcesAndSinks.txt* Text file containing the list of sources and sinks to be checked for data flows during the analysis. We recommend to use the Susi³ tool to obtain such a list based on the selected *android.jar*.

– *AndroidCallbacks.txt* Text file containing the list of callback methods. We recommend to use an existing list for example the one provided by FlowDroid.

– *apktool.jar* We recommend to use the latest version of the Apktool⁴.

• **results**

The summarised and grouped findings from our benchmarking implementation will be stored here; including a summary text file report, a summary csv file report and a csv file showing which tools timed out and which ones completed the analysis.

• **apksToTest**

All applications that should be analysed need to be put in this folder.

### A.1.1 **COVERT** Setup

Clone the benchmarking project from GitHub and navigate to the tools/covert folder inside the project. Then follow the steps below to set up **COVERT**.

1. **Obtain **COVERT**

Note that the folder already contains certain files and folders. These should not be changed for the benchmark to work. In order for **COVERT** to run, you need to obtain the following artefacts and store them in the covert folder. Make sure to rename the files as indicated or adapt the running command later on.

First, you have to obtain the **COVERT** back-end⁵. Then unpack the folder and extract the following files and folders to the covert folder in the benchmarking project directory.

- *covert.bat*
- *covert.sh*

---

³[https://blogs.uni-paderborn.de/sse/tools/susi/](https://blogs.uni-paderborn.de/sse/tools/susi/)
⁴[https://ibotpeaches.github.io/Apktool/](https://ibotpeaches.github.io/Apktool/)
⁵[https://www.ics.uci.edu/~seal/projects/covert/](https://www.ics.uci.edu/~seal/projects/covert/)
APPENDIX A.

- **appRepo** (folder)
- **resources** (folder)

Copy and paste the following resources into the *configCustom* folder inside the *covert* directory. We denote the root directory of the project as `~`:

- `~/resources/AndroidPlatforms/android-8/android.jar`
- `~/resources/Covert/resources/apktool/apktool.jar`
- `~/resources/FlowDroid/resources/AndroidCallbacks.txt`
- `~/resources/FlowDroid/resources/SourcesAndSinks.txt`

These are the configurations that we will change to run the tool with shared configurations. However, we copy the artefacts to the *custom-Config* folder to allow the user to easily switch between original and shared configurations.

2. **Running Command**

Go to the *covert* directory and make sure that all required files and artefacts are present. Copy a sample application into the *app_repo* folder. Then run the following command to start the analysis.

```
./covert.sh <APPLICATION_NAME>.apk
```

Instead of a single application, you could also analyse multiple applications with *COVERT*. To do so, you have to create an additional folder inside the *app_repo* folder. Instead of the application name, you then pass the name of the newly created folder in the running command.

Make sure to run the tool on a sample application and verify that the results are as expected. We recommend to use the SendSMS.apk provided by *DroidBench* for a test run. *COVERT* should detect a data leak for the `sendTextMessage()` method. The source code for the application is available on *DroidBench*.

3. **Results**

The results of the analysis are located in the *app_repo* folder. A new folder with the same name as the analysed application should have been created during the analysis. This folder contains the results file in a `.xml` file.

---

6https://github.com/secure-software-engineering/DroidBench
FlowDroid Setup

Clone the benchmarking project from GitHub and navigate to the tools/flowDroid folder. Then follow the steps below to set up the FlowDroid tool.

1. Obtain FlowDroid

   Note that the folder already contains certain files and folders. These should not be changed for the benchmark to work. In order for FlowDroid to run, you need to obtain the following artefacts and store them in the flowDroid folder. Make sure to rename the files as indicated or adapt the running command later on.

   The following additional libraries are required:

   - soot-trunk.jar
   - soot-infoflow.jar
   - soot-infoflow-android.jar
   - slf4j-simple-1.7.5.jar (libraries for logging)
   - slf4j-simple-1.7.5.jar (libraries for logging)
   - axml-2.0.jar (Android XML parser library)
   - android.jar (Android SDK): For the analysis we use API Level 23

   Furthermore, you need to obtain the following configuration files and store them in the same folder as the artefacts above: Make sure to also copy and paste the SourcesAndSinks.txt and AndroidCallbacks.txt file in the configCustom folder.

   - EasyTaintWrapperSource.txt (taint wrapper)
   - AndroidCallbacks.txt (Android callbacks)
   - SourcesAndSinks.txt (sources and sinks)

   Make sure to also copy and paste the SourcesAndSinks.txt and AndroidCallbacks.txt file in the configCustom folder. This is required to later create soft links that allow switching between running the tools with original or shared configurations.

2. Running Command

   Go to the flowDroid directory and make sure that all required files and artefacts are present. Then run the following command to start the analysis.
java -Xmx4g -cp soot-trunk.jar:soot-infodeflow.jar:soot-infodefloy-android.jar:slf4j-api-1.7.5.jar:slf4j-simple-1.7.5.jar:axml-2.0.jar soot.jimple.infoflow.android.TestApps.Test <PATH_TO_APPLICATION> ./android.jar > flowdroid_results.txt

For the benchmark, we put applications that should be analysed in the apksToTest folder. The corresponding folder path would be ../../apksToTest/<APPLICATION_NAME>.apk.

Make sure to run the tool on a sample application and verify that the results are as expected. We recommend to use the SharedPreferences1.apk provided by DroidBench for a test run. FlowDroid should detect all data leaks as indicated in the applications source code that is available on DroidBench.

3. Results The results of the analysis are stored in the flowdroid_results.txt file. They also get printed to the console.

**HornDroid Setup**

For HornDroid the setup is straightforward and well documented on its project page.

1. **Obtain HornDroid**

   First, clone the HornDroid GitHub project\(^7\) and build it using mvn clean package. A new folder target is created. Move the content from this folder to the horndroid folder inside the benchmarking directory.

   In addition, copy and paste the following files to the horndroid/config-\(\text{Custom}\) folder:

   - apktool.jar
   - Callbacks.txt
   - SourcesAndSinks.txt

2. **Running Command** Go to the horndroid directory and make sure that all required files and artefacts are present. Then run the following command to start the analysis.

\(^7\)https://github.com/ylya/horndroid.git
java -jar fshorndroid-0.0.1.jar ./apktool.jar <PATH_TO_APPLICATION>

Make sure to run the tool on an sample application and verify that the results are as expected. We recommend to use the SharedPreferences1.apk provided by DroidBench for a test run. HornDroid should detect all data leaks as indicated in the applications source code that is available on DroidBench.

3. Results The results of the analysis are located in the OUTPUT.report folder as a JSON file. They are also printed to the console.

IC3 Setup
The setup of IC3 also requires the installation of the Dare tool. It is used to decompile Android applications from the installation image to source code on which IC3 can perform.

1. Obtain IC3
We start by setting up the Dare tool. First, go to the Dare installation page\(^8\) and download the latest version. After the download, unzip the installation package and create a folder named output. Move the whole content of the installation package to the tools_helper/dare folder.

Now, you can start with the setup of the IC3 tool. Begin by cloning the project from GitHub\(^9\). Then go the IC3 directory and build the tool with the following command:

```
git clone https://github.com/siis/ic3
cd ic3
mvn clean package -P standalone
```

Move the content from the newly created target folder to the ic3 folder inside the benchmarking directory. There are already two folders present—dareOutput and ic3output. These folders are used to store the results of Dare and IC3 respectively.

2. Running Command Before starting the analysis, we have to run the Dare tool on the application to decompile it. To do so, run the following command from within the Dare directory:

\(^8\)http://siis.cse.psu.edu/dare/installation.html
\(^9\)https://github.com/siis/ic3
Afterwards, the folder `~/ic3/dareOutput` should contain the decompiled files. Now we are all set to run the IC3 tool with the following command from within the `ic3` folder:

```
./runIC3.sh <PATH_TO_APPLICATION>
```

Make sure to run the tool on an sample application and verify that the results are as expected. We recommend to use the `StartActivityResult1.apk` provided by DroidBench for a test run. IC3 should detect one data leaks at the `startActivityForResult()` method.

3. Results The results of the analysis are located in the `ic3output` folder as a text file.

### IccTA Setup

The setup of IccTA is the most time intensive one. It requires to build the tool yourself and set up a mySQL database to store the intermediate results. Furthermore, it uses several frameworks that you can build yourself. However, there are already built versions available online that we use for this benchmark.

1. **Obtain IccTA** Begin by importing all the following projects to Eclipse or another IDE:
   
   - `jasmin`
   - `heros`
   - `soot`
   - `soot-infoflow`
   - `soot-infoflow-android`
   - `soot-infoflow-android-iccta`

   Then change the build path of `soot-infoflow-android-iccta` to include the above projects and then build each project.

   Afterwards, create a new folder called `output_iccta` inside the `soot-infoflow-android-iccta` directory.
2. **Database Setup** To store intermediate results during the analysis, *IccTA* uses a *mySQL* database. Start *mySQL* and create the following database:

```bash
mysql -u root -p -e 'create database cc?;
mysql -u root -p cc < res/schema;
```

As a database name, you have to use *cc* since it is hardcoded in the *IC3* tool provided. When you are done with importing the *IccTA* schema, you have to adapt the database properties of the tool. The following files need to be updated with the correct username and password to the database:

- `~/iccProvider/ic3/runIC3.sh`
- `~/iccProvider/ic3/runIC3Wrapper.sh`
- `~/res/jdbc.xml`
- `~/res/iccta.properties`
- `~/src/soot/jimple/infoflow/android/iccta/util/Constants`

3. **Running Command** Before starting the analysis, we have to run the *IC3* tool on the application. To do so, run the following command from within the `~/iccProvider/ic3` directory:

```bash
./runIC3.sh <PATH_TO_APPLICATION>
```

Now, go to the *release* directory and run the following command to start *IccTA*:

```bash
java -jar IccTA.jar <PATH_TO_APPLICATION> <PATH_TO_ANDROID_SDK> newline -iccProvider ../iccProvider/ic3
```

Make sure to run the tool on an sample application and verify that the results are as expected. We recommend to use the *SharedPreferences1.apk* provided by *DroidBench* for a test run. *IccTA* should detect all data leaks as indicated in the applications source code that is available on *DroidBench*.

4. **Results** The results of the analysis are located in the *output_iccta* folder as a text file.
A.2 Benchmarking Setup

In order for our benchmarking implementation to work, you have to make sure that all tools are working as explained in Appendix A.1.

A.2.1 Build benchmark

After setting up the tools, you have to build the benchmark as a jar file containing all required dependencies. This can be done using our gradle script. Note that these are only dependencies such as to apache commons or junit and not to the tools themselves.

Run the following command to build the benchmark:

```
gradle shadowJar
```

This generates an executable jar file containing all relevant dependencies.

A.2.2 Adapt Configurations

In case you have set up the tools in the same folder structure as presented in this manual, then you can skip this section. In case you have changed the location or name of certain artefacts, then you can easily adapt the paths used to these benchmarks in our data_leak_detection.properties.defaults file.

Furthermore, you can exclude certain tools from the benchmark by simply changing the enabled flag in the properties file to false.

A.2.3 Run Benchmark

Put the APK file of the application you would like to analyse in the apksToTest folder. Then run the following command to start the benchmark with the enabled tools:

```
java -jar benchmarking.jar
```

The results of the analysis can be found in the results folder. It contains the individual tools’ results as well as the summarised reports. The timedOut folder contains a list of all applications that were analysed with an indication for each tool if the analysis was completed (indicated as 0), interrupted with and exception (indicated as 1), or timed out (indicated as 2).
A.3 Concluding remarks

By following this manual, the benchmark can be set up and presented results reproduced. The majority of work stems from setting up the tools; we do not ship them with our benchmark. Once the tools are up and running, running the benchmark is trivial.
List of Figures

1.1 Overview of Android OS versions in the field (February 2018) 2

3.1 Overview of tool categorization. Tools marked with an asterisk are not classified in Sadeghi’s work. 14


[64] Sam Malek, Hamid Bagheri, and Alireza Sadeghi. “Automated detection and mitigation of inter-application security vulnerabilities in Android (invited talk).” In: DeMobile@SIGSOFT FSE. 2014.


