Idea: Benchmarking Android Data Leak Detection Tools

Claudio Corrodi, Timo Spring, Mohammad Ghafari, and Oscar Nierstrasz

Software Composition Group, University of Bern, Switzerland

Abstract Virtual application stores for mobile platforms contain many malign and benign applications that exhibit security issues, such as the leaking of sensitive data. In recent years, researchers have proposed a myriad of techniques and tools to detect such issues automatically. However, it is unclear how these approaches perform compared to each other. The tools are often no longer available, thus comparing different approaches is almost infeasible.
In this work, we propose an approach to execute static analysis tools and collect their output to obtain unified reports in a common format. We review the current state-of-the-art in Android data leak detection tools, and from a list of 87 approaches, of which we were able to obtain and execute five. We compare these using a set of known vulnerabilities and discuss the overall performance of the tools. We further present an approach to compare security analysis tools by normalising their interfaces, which simplifies result reproduction and extension.

Keywords: data leak, Android, benchmarking

1 Introduction

Security of mobile applications is a hot topic in both research and industry. Recent research has suggested that many applications in popular virtual stores, such as Google's Play Store or Apple's App Store, suffer from security vulnerabilities. There exist many approaches to automatically detect data leak issues (*i.e.*, situations where sensitive data may be leaked). Although taxonomies exist that catalogue and categorise such approaches [1,2,3,4], there is a lack of experimental evidence comparing the existing approaches.

In this paper, we tackle the following two research questions.

RQ 1 *To what degree are security analysis approaches and results in the domain of Android data leak detection reproducible?*

Anecdotal evidence from other domains led us to believe that artefacts and tools presented in research papers are rarely available online or by other means. We answer this question by reviewing the state-of-the-art approaches, trying to obtain their artefacts through various means (*e.g.*, using search engines or contacting the authors), and running them to obtain results similar to those originally reported.

RQ 2 *How do data leak analysis tools perform (individually and compared to each other) on a common set of applications?*

We run the artefacts on a set of applications with known vulnerabilities and report on precision and recall for each tool. We present a benchmark suite that allows us to easily run several tools on the same set of target applications and obtain reports in a normalised form.

We report on our findings by performing pairwise analysis on the tools, after running them on a set of synthetic applications with known vulnerabilities.

2 Related work

There exists a wide range of related work. While a thorough discussion goes beyond the scope of this idea paper, we list those most relevant to our research.

Android security analysis literature. Our literature review was mostly focused on works mentioned in the following taxonomies.

Sadeghi *et al.* present a large-scale overview of Android security analysis in general [3]. In particular, we used their categorization of security problems as starting point for our work [3, p. 12, Table 3]. Similarly, Sufatrio *et al.* published a taxonomy on Android analysis tools [5], as did Reaves *et al.* [2]. Gadient *et al.* have studied the prevalence of security issues in Android applications, including data leak vulnerabilities [6,7], and confirm that such issues are in fact common among Android applications.

Comparing software artefacts. In 2016, Amann *et al.* have analysed artefacts for detecting application programming interface (API) misuse violations [8]. Their approach is similar to ours in that they developed a framework for comparing such tools.

DroidBench[1] is a benchmark for evaluating analysis tools. Because vulnerabilities are documented in these applications, they are well-suited for the qualitative analysis we present in this work.

The work by Reaves *et al.* comes closest to ours. In their study, they use *DroidBench* to analyse results obtained from a set of seven Android analysis tools [2]. However, the only one in common with our set is *FlowDroid*. In contrast to our work, the evaluation lacks a comparison of tools amongst each other.

3 Classification and selection of Android analysis tools

The process of reviewing relevant literature, identifying (potential) artefacts, and categorising them is straightforward. In this section, we briefly present the relevant steps.

[1] `https://github.com/secure-software-engineering/DroidBench`

Literature review. We started our process by taking a broad view on security analysis in reviewing related work. We focus on the works mentioned in the recent taxonomy by Sadeghi *et al.* [3]; we further reviewed two additional taxonomies to ensure that we did not miss any relevant approaches [2,5]. Based on those, we obtained and reviewed an initial list of 87 artefacts. Due to space constraints, we do not list them individually here and instead refer the interested reader to the mentioned taxonomies.

Next, 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. search online with contemporary search engines for the artefact, and
3. contact the authors and inquire whether the tool is available or can be made available to us (at most two requests by email).

In 60 cases, we sent requests to the authors. Of those, a staggering 49 remained unanswered.

We excluded several works for the following reasons. In four cases, the authors refused to give us access, either because their tools are commercially used or discontinued. Thirteen of the artefacts are not tools that can be executed but instead formal models. Nine tools are based on dynamic approaches; in this work, we focus on static ones exclusively. Four tools are not in the same domain, despite having been mentioned to be in the main taxonomy we used as a source [3]. Two additional tools are not relevant because they do not perform analysis on single applications and do not report data leaks. In one case, we failed to set up the tool due to poor documentation.

Selected tools. After eliminating most of the tools as described, we ended up with a set of five tools for our benchmark: *FlowDroid*, *HornDroid*, *IccTA*, *IC3 (Epicc)*, and *COVERT*.

FlowDroid [9] is a taint analysis tool based on the *Soot* and *Heros* frameworks. Its analysis is context-, flow-, and object-sensitive.

HornDroid [10] uses a combination of static and formal analysis.

IccTA [11] is an inter-component communication based taint analysis tool, suited for any data-flow analysis. *IccTA* uses both *Epicc* and *IC3* [12] as part of its analysis.

IC3 (Epicc) [12] detects inter-component communication with a focus on inferring values of complex objects with multiple fields, such as intents or URIs. It uses *Dare* for decompiling, and *FlowDroid* to generate an interprocedural control flow graph.

COVERT [13] is a static and formal security analysis tool. Its main focus is inter-application communication and escalation of privileges. *COVERT* performs value-, context-, and flow-sensitive analysis.

In the remainder of this paper, we present our approach to evaluating and the results obtained from analysing *DroidBench* vulnerabilities with these artefacts.

4 Benchmark Implementation

To easily compare the selected tools, we implemented a Java benchmark suite that allows us to collect results from individual tools. The benchmark then parses each tool's results and creates standardised reports and consolidates results, which allows us to easily review the reports, compare tools, and perform statistical analyses.

The implementation is straightforward. Pairs of runners and parsers correspond to individual tools. Runners handle setting up the tool environment and executing it accordingly, while parsers read and interpret the output for further (consolidated) processing.

To include a tool in the benchmark, one has to do the following. First one needs to set up the tool so that it can be executed from the command-line, and the output is stored on the file system. Second, one has to provide a runner in the benchmark that specifies how the artefact is executed (by implementing a single Java interface that provides information such as where resulting files and logs are stored), and third, provide a parser that creates reports (*i.e.*, objects holding information about a reported data leak).

The benchmark can be executed on a given Android application binary. Then, all relevant tool output files are collected and consolidated reports are generated. For each detected vulnerability, we list the class and method in which the leak happens, and the actual sink where the leak happens, as well as the analysis tools that detected the vulnerability.

An important aspect of implementing such a benchmark is extensibility. With a simple way to add additional tools—by implementing a runner for executing the tool on a given target application and a parser for obtaining the generated output—it is straightforward to obtain fresh results with a set of tools. This allows users to compare several tools on a level playing field. Furthermore, publishing the tools and their corresponding runners and parsers simplifies future evaluation and reproduction of results.

The benchmark and data are available online[2].

5 Experimental setup

To answer the research questions, we use our benchmark to execute the five tools *IccTA*, *IC3 (Epicc)*, *HornDroid*, *FlowDroid*, and *COVERT*.

For a fair comparison, one needs to make sure that the tools use the same lists of sources and sinks. Here, we use the *SuSi* tool [14] to obtain such a list.

We then configure all tools to use, where applicable, the same (i) sources and sinks list, as generated by *SuSi*, (ii) callback list, (iii) android.jar (API level 23), and (iv) apktool.jar (2.3.1).

As many others in recent research into Android security analysis, we use the set of programs provided in *DroidBench* for our analysis. We decided to

[2] `https://github.com/ccorrodi/android_data_leak_detection`

Table 1. Raw counts of true/false positives/negatives.

Metric	*FlowDroid*	*HornDroid*	*COVERT*	*IC3*	*IccTA*
True positive	99	99	8	4	97
False positive	54	87	3	37	59
True negative	90	57	141	107	85
False negative	26	26	117	121	28

Table 2. Collected metrics for each tool as observed on *DroidBench* vulnerabilities. Bold values indicate maxima for the respective metric.

Metric	*FlowDroid*	*HornDroid*	*COVERT*	*IC3*	*IccTA*
Accuracy	**0.703**	0.580	0.554	0.413	0.677
Precision	0.647	0.532	**0.727**	0.098	0.622
Recall	**0.792**	**0.792**	0.064	0.032	0.776

select this benchmark because it specifically targets data leak vulnerabilities, and because individual vulnerabilities are described in text, thus providing us with a ground truth. This enables us to review the reports generated by our tool, and identify reports as true or false positives or negatives.

After running the benchmark with the normalised configuration on all *DroidBench* applications, two authors reviewed the obtained reports. For each *DroidBench* application, we proceed as follows. First, we record the vulnerabilities specified in the code (*i.e.*, the vulnerabilities stated by the *DroidBench* authors). Second, for each detected leak that corresponds to a *DroidBench* vulnerability, we record the tools that detect it (true positives), and those that do not (false positives). Third, for each additional report, we record the tools that report it (false positive), and those that do not (true negative). After processing 125 *DroidBench* vulnerabilities, we can report on each tool's precision and recall.

Because we have evaluated the tools using the same configurations and on the same dataset, we can also perform pairwise comparisons. We do this using McNemar's Test [15], which is a statistical test for determining whether, based on our observations, two tools are likely to report the same issues.

6 Results

Executing the analyses results in 269 distinct reports from the tools. We reviewed all reports manually and determined for each report whether it matches a vulnerability described in *DroidBench* and which tools are reporting it.

Table 1 summarises our results. What is striking is the poor performance of *IC3*, which reports 41 data leaks, of which only 4 are true positives. Furthermore, we note that *COVERT* and *IC3* only produce 11 and 41 reports respectively, a far cry from what the other tools report.

Table 3. χ^2-values from McNemar's test for each pair of tools. Bold indicates a statistically significant difference.

	HornDroid	***COVERT***	***IC3***	***IccTA***
FlowDroid	**9.94**	**10.56**	**35.29**	2.77
HornDroid		0.2	**8.53**	6.01
COVERT			**26.33**	**6.97**
IC3				**28.99**

Table 2 shows accuracy, precision, and recall for each tool. Here, recall is of particular interest, as it only includes *DroidBench* vulnerabilities in the calculation. We observe that both *FlowDroid* and *HornDroid* perform equally well on the dataset, reporting almost 80 % of the vulnerabilities. However, as we will see below, the sets of reported issues do not match completely.

Both *IC3* and *COVERT* distinguish themselves from the other artefacts in that they report very few leaks. Unfortunately, this does not result in better accuracy or recall. The precision of *COVERT*, however, is high; 8 out of 11 reports are true positives.

It is worth to note that 113 of the 125 *DroidBench* vulnerabilities—or 90.4 %—are reported by at least one tool. We think that this is surprisingly high, considering the wide variety of applications and vulnerability characteristics.

Next, we investigate how the performances of two tools relate to each other. We apply McNemar's Test [15] to obtain a measure that expresses similarities between two detectors.

A χ^2 value above $\chi^2_{1,0.01} = 6.635$ (which corresponds to a confidence interval of 99 %) indicates that there is a statistically significant difference between the performances of the two classifiers. Otherwise, the null hypothesis (*i.e.*, that two tools perform equally well) holds with a probability of at least 99 %.

Table 3 summarises the findings. Bold values indicate places where a statistically significant difference between two tools has been observed.

Most pairs of tools report different sets of leaks with a statistically significant difference. As McNemar's Test exclusively considers the cases where tools disagree, this means that for each pair, one tool is wrong more frequently than the other. However, according to the underlying raw data, there is no tool that performs clearly better than all others.

FlowDroid is less often wrong than any other tool. However, it is important to note that *FlowDroid* and *IccTA* perform very similarly; they disagree in only 13 cases. This is reflected in Table 3, as the corresponding χ^2 value indicates that there is no statistically significant difference in performance of the tools.

Similarly, the χ^2 value between *HornDroid* and *IccTA* is below the threshold.

As an odd occurrence, *COVERT* and *HornDroid* also exhibit a very low χ^2 value. This happens even though the two tools report vastly different sets, as evident in Table 1. In this case, McNemar's test may not be best suited, as it only considers the difference of the disagreeing reports, which, in this case, is

very low. Nevertheless, the test is well-suited to compare classifiers in general, and, using additional data, constitutes a valuable metric in our analysis.

7 Threats to validity

The benchmark we implemented may contain bugs that directly influence the results. To mitigate this threat, we implemented unit tests during development, and manually verified the generated output on a regular basis.

There may be vulnerabilities in the synthetic applications of *DroidBench* that are not reported as such. This may influence precision and recall of the tools. To mitigate this thread, we manually reviewed potential false positives (without finding any true positive vulnerabilities not documented by *DroidBench*).

Both *DroidBench* and *FlowDroid* originate from the same research group, so it may be possible that there is a selection bias that favours *FlowDroid*.

It is possible that we have made mistakes in configuring some of the tools that we tested. We mitigate this threat by only making minimal changes to a tool's configuration. Whenever possible, we use the tools as distributed.

Finally, the different publication years suggest that the original authors likely did not work with the same target Android version. Our choice to normalise configurations, in particular using the same sources, sinks, and Android version, may thus influence the results. 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

In this paper, we investigate to what degree static tools that assess data leaks in Android application domain are available, and how they work in practice. We report the progress from an initial list of 87 tools and approaches and describe the elimination process, after which we arrive at five tools that are suitable for our analysis.

We present a benchmark suite that easily allows us to consolidate results from the tools. To ensure a fair comparison, we configure the tools so that they use the same configurations. Furthermore, we apply them to the same targets, and avoid cherry-picking particular *DroidBench* vulnerabilities.

We observe that most tools suffer from a high amount of false positives and negatives. When we compare pairs of tools, they show, with few exceptions, statistically significant differences in their performances.

In our future work, we plan to use our benchmark suite with real-world applications and not just synthetic ones. We plan to also study the root causes of poor performance in the tools.

Acknowledgements. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Analysis" (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018). We also thank CHOOSE, the Swiss Group for Original and Outside-the-box Software Engineering of the Swiss Informatics Society, for its financial contribution to the presentation of this paper.

References

1. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. ACM computing surveys (CSUR) **44**(2) (2008) 6:1–6:42
2. Reaves, B., Bowers, J., Gorski III, S.A., Anise, O., Bobhate, R., Cho, R., Das, H., Hussain, S., Karachiwala, H., Scaife, N., Wright, B., Butler, K., Enck, W., Traynor, P.: *Droid: Assessment and evaluation of Android application analysis tools. ACM Comput. Surv. **49**(3) (2016) 55:1–55:30
3. Sadeghi, A., Bagheri, H., Garcia, J., Malek, S.: A taxonomy and qualitative comparison of program analysis techniques for security assessment of Android software. IEEE Transactions on Software Engineering **43**(6) (2017) 492–530
4. Tam, K., Feizollah, A., Anuar, N.B., Salleh, R., Cavallaro, L.: The evolution of Android malware and Android analysis techniques. ACM Comput. Surv. **49**(4) (2017) 76:1–76:41
5. Sufatrio, Tan, D.J.J., Chua, T.W., Thing, V.L.L.: Securing Android: A survey, taxonomy, and challenges. **47**(4) 58:1–58:45
6. Gadient, P.: Security in Android applications. Masters thesis, University of Bern (August 2017)
7. Ghafari, M., Gadient, P., Nierstrasz, O.: Security smells in Android. In: 17th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM). (September 2017) 121–130
8. Amann, S., Nadi, S., Nguyen, H.A., Nguyen, T.N., Mezini, M.: MUBench: A benchmark for API-misuse detectors. In: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR). (2016) 464–467
9. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. SIGPLAN Not. **49**(6) (June 2014) 259–269
10. Calzavara, S., Grishchenko, I., Maffei, M.: Horndroid: Practical and sound static analysis of Android applications by SMT solving. In: 2016 IEEE European Symposium on Security and Privacy (EuroS P). (March 2016) 47–62
11. Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Traon, Y.L., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P.: IccTA: Detecting inter-component privacy leaks in Android apps. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Volume 1. 280–291
12. Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., Le Traon, Y.: Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis (2013)
13. Bagheri, H., Sadeghi, A., Garcia, J., Malek, S.: Covert: Compositional analysis of Android inter-app permission leakage. IEEE Transactions on Software Engineering **41**(9) (September 2015) 866–886
14. Bu, W., Xue, M., Xu, L., Zhou, Y., Tang, Z., Xie, T.: When program analysis meets mobile security: An industrial study of misusing Android internet sockets. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2017, ACM (2017) 842–847
15. McNemar, Q.: Note on the sampling error of the difference between correlated proportions or percentages. Psychometrika **12**(2) (June 1947) 153–157