The Impact of Tool Configuration Spaces on the Evaluation of Configurable Taint Analysis for Android

Austin Mordahl
The University of Texas at Dallas
austin.mordahl@utdallas.edu

Shiyi Wei
The University of Texas at Dallas
swei@utdallas.edu

ABSTRACT

The most popular static taint analysis tools for Android allow users to change the underlying analysis algorithms through configuration options. However, the large configuration spaces make it difficult for developers and users alike to understand the full capabilities of these tools, and studies to-date have only focused on individual configurations. In this work, we present the first study that evaluates the configurations in Android taint analysis tools, focusing on the two most popular tools, FlowDroid and DroidSafe. First, we perform a manual code investigation to better understand how configurations are implemented in both tools. We formalize the expected effects of configuration option settings in terms of precision and soundness partial orders which we use to systematically test the configuration space. Second, we create a new dataset of 756 manually classified flows across 18 open-source real-world apps and conduct large-scale experiments on this dataset and micro-benchmarks. We observe that configurations make significant tradeoffs on the performance, precision, and soundness of both tools. The studies to-date would reach different conclusions on the tools' capabilities were they to consider configurations or use real-world datasets. In addition, we study the individual options through a statistical analysis and make actionable recommendations for users to tune the tools to their own ends. Finally, we use the partial orders to test the tool configuration spaces and detect 21 instances where options behaved in unexpected and incorrect ways, demonstrating the need for rigorous testing of configuration spaces.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**;
• **General and reference → Empirical studies**.

ACM Reference Format:
Austin Mordahl and Shiyi Wei. 2021. The Impact of Tool Configuration Spaces on the Evaluation of Configurable Taint Analysis for Android. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3460319.3464823

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464823

1 INTRODUCTION

Static taint analysis is essential for detecting security vulnerabilities in Android apps. However, language features such as reflection [30] and callbacks [46] complicate the task of static analysis. Different algorithms for handling these features present tradeoffs between precision, soundness and performance. For example, several Android taint analysis tools handle lifecycle callbacks through a synthetic main method [10, 44]. One way to construct this method is to use a fixed-point algorithm to include only the reachable callbacks [10]. A less precise but faster solution is to scan for classes implementing callback interfaces and include all of them [1]. A useful tool needs to achieve a "sweet spot" between the competing needs for accuracy and performance. This sweet spot is a moving target, dependent upon target programs, user's preferences, and available resources. Thus, static analysis tools often include configuration options so that tool designers can evaluate different design choices and users can tune configurations according to their needs. For example, FlowDroid has 22 algorithmic options (i.e., options that affect the analysis algorithms), such as *implicit* which enables/disables the tracking of implicit information flows [24].

In the past few years, many algorithms and tools for Android taint analysis have been developed [6, 8, 10–12, 14, 16, 17, 19, 23, 25, 29, 44]. Recently, several empirical studies have compared the most popular tools (FlowDroid [10], DroidSafe [19], Amandroid [44], IccTA [29], DidFail [25], and DIALDroid [12]) and revealed important insights about their behaviors [13, 31, 35, 36]. However, these studies have focused only on one configuration of each tool, often the default. Their insights may not capture the tools' full capabilities because certain behaviors may be exposed only through other configurations. Furthermore, since these evaluations rely heavily on hand-crafted micro-benchmarks with known ground truths, the conclusions may not generalize to real-world programs, which are the intended targets for the tools' use.

In this work, our goals are to (1) evaluate the state-of-the-art static taint analysis tools through the lens of their full configurability on programs that are more representative of those that users want to analyze, and (2) make actionable suggestions for tuning the tools and provide a methodology for testing the implementations of configuration options. Of the six tools listed above, the first three have algorithmic configuration options. We focus on FlowDroid and DroidSafe, as we observed nondeterminism in Amandroid's results that made it unsuitable for this study [33].

To meet our goals, one challenge is that the documentation of configuration options may be ambiguous. We investigate the implementations of these two tools with the goal of better understanding the intended behaviors of their configuration options and their settings. During this investigation, we identify two undocumented relationships in the tools' configuration spaces (Section 2). The first

is *disablement*, wherein one setting of an option can prevent another setting of an option from taking effect. This commonly happens when the code that one setting runs is within a conditional block guarded by a setting of another option. This relationship is detrimental to users, since it is undocumented and can mislead a user into believing that a configuration option does nothing. Second, we identify *precision and soundness partial orders*, which quantify the expected tradeoffs in terms of true and false positives produced by changing an option to a different setting. These partial orders make the configuration space easier to understand and provide a new method to automatically test different configurations for bugs. We identify 158 precision and soundness partial orders across both tools.

The second challenge is that the large configuration spaces make it infeasible to study configurations exhaustively and the lack of representative benchmarks makes it difficult to assess the tool behavior in real-world programs. Because there are no large real-world benchmarks with known ground truths for Android taint analysis, we collected 30 open-source real-world apps from FossDroid [7] and classified the flows detected by the tools, yielding a new dataset of 63 true positive and 693 false positive flows (Section 3). We use combinatorial interaction testing (CIT) techniques to sample the configuration spaces of FlowDroid and DroidSafe. We then run these configurations on the FossDroid dataset along with a broadly used micro-benchmark, DroidBench 3.0 [5], and 50 apps from the Google Play store to evaluate the impact of tool configuration spaces and dataset selection.

Our experiments, enabled by our solutions to these challenges, result in key findings that recontextualize the recent studies (Section 4). We find that *only evaluating a single configuration presents a limited and misleading picture of a tool's capabilities* (RQ1). There are several instances in our data where testing on a single configuration gives the false impression that a tool does not support certain language features (e.g., reflection) or supports them poorly. While tools certainly have limitations, we find that aggregating the results across configurations gives a more complete picture of what a tool is capable of. We also observe that *the results on micro-benchmarks do not generalize to real-world programs* (RQ2). Our FossDroid dataset draws out more exaggerated impacts of configuration options than the micro-benchmark in terms of both performance and precision. This demonstrates the urgent need for a large dataset of real-world programs in which the ground truths are known.

In addition, we perform statistical analysis to understand how the configuration options impact the behaviors of FlowDroid and DroidSafe. We find that *15 out of 45 options across both tools affect performance without an apparent precision hit and 2 options in FlowDroid present significant tradeoffs between performance and precision* (RQ3). This result suggests that users may consider using the faster settings in these 15 options by default and tuning the 2 options related to the IFDS solver [37] and reflection handling. Through an investigation enabled by our precision and soundness partial orders, we observe that *21 options display incorrect behaviors that indicate errors in the tools' documentation and/or implementation* (RQ4). This demonstrates both the utility of partial orders as a testing aid for static analysis tools and the lack of testing of the configuration spaces of the Android taint analysis tools.

Overall, this paper makes the following contributions:

- The first empirical study that focuses on the configurations in Android taint analysis tools. The experiments on three datasets reevaluate the current capabilities of state-of-the-art tools, and the observations shed light on the necessity of evaluation and testing of their configuration spaces.
- A manual investigation of the source code of FlowDroid and DroidSafe that summarizes how configurations are implemented in state-of-the-art tools. As a result of this investigation, we produce soundness and precision partial orders, which formalize the expected behavior of configuration options and can be used to test configuration spaces of static analysis tools.
- A dataset consisting of 63 true positive and 693 false positive flows across 18 applications from the FossDroid open-source repository, which can serve as a starting point for future real-world benchmarks.

We have made the artifacts, including datasets, investigation and experimental results, available for future tool development and evaluation [33].

2 INVESTIGATION OF ANALYSIS OPTIONS

The first step we took to better understand the capabilities of configurable tools was to study the implementation of their large configuration spaces. We focused on *algorithmic* options in FlowDroid and DroidSafe to evaluate their impact on precision, soundness and performance. These are options which have an effect on the analysis algorithms themselves, as opposed to other factors like input/output formatting. In total, we studied 22 options in FlowDroid and 23 options in DroidSafe. We performed a manual code investigation with the goals of (1) uncovering undocumented relationships between options, and (2) better understanding the tradeoffs that different settings of an individual option provide. Achieving these goals helps both the user and designer better understand and anticipate the behavior of these tools.

In our manual investigation, we first located the variables in code that were set by configuration options. We then traced the usages of these variables in the tool implementations and identified two types of undocumented relationships that exist between option settings.

Disablement. The first relationship is between the settings of two different options, where setting one option prevents another option's setting from making a difference. We formally define it as follows.

Let $P = ((c_1 \mapsto v_1), (c_2 \mapsto v_2), \ldots, (c_k \mapsto v_k))$ be a configuration of a taint analysis tool, where c_i is an algorithmic configuration option and $v_i \in \mathrm{dom}(c_i)$ is the setting of c_i. We denote the default setting for c_i as v_i^d. Let P_D be the default configuration of the program, such that $P_D = ((c_1 \mapsto v_1^d), (c_2 \mapsto v_2^d), (c_3 \mapsto v_3^d), \ldots)$. Configuration options that are explicitly set are noted in brackets, such that $P_D[c_1 \mapsto v_1] = ((c_1 \mapsto v_1), (c_2 \mapsto v_2^d), (c_3 \mapsto v_3^d), \ldots)$.

Let $F = \{f_1, f_2, \ldots, f_k\}$ be the set of flows produced by a taint analysis. We consider two flows f_1 and f_2 equivalent if (1) they have the same source and sink method signatures, and (2) the source (and sink) of f_1 and the source (and sink) of f_2 are called from the same method. Let $R : P \times A \rightarrow F$ signify running a deterministic analysis tool with configuration P on input A resulting in the set

```
1  //--typesforcontext: use types (instead of alloc sites) for
     object sensitive context elements > 1
2  if(cmd.hasOption("typesforcontext")){ //Config.java:674
3    this.typesForContext = true;
4  }
5  void setSparkPointsToAnalysis() { //SparkPTA.java:786
6    ...
7    opt.put("kobjsens-types-for-context", Boolean.toString(
     Config.v().typesForContext));
8    ...
9  }
```

Figure 1: Excerpts of the implementation and usage of the *typesforcontext* option in DroidSafe.

of flows F (i.e., $R(P, A) = F$). Given two configurations P_i and P_j, $P_i = P_j \iff \forall k, R(P_i, k) = R(P_j, k)$. We therefore say that $(c_1 \mapsto v_1)$ *disables* $(c_2 \mapsto v_2)$ if $\forall i, P_i[c_1 \mapsto v_1, c_2 \mapsto v_2] = P_i[c_1 \mapsto v_1]$.

For example, Figure 1 shows how the *typesforcontext* option (default setting *FALSE*) in DroidSafe is implemented across multiple files. Line 1 shows the command line documentation for this option. Lines 2-4 show how the option is set to true if the *typesforcontext* flag is passed on the command line. Line 7 shows the only place where the *typesforcontext* field is read, which is inside the *setSparkPointsToAnalysis* method. This method is only called if *pta* is set to *SPARK* (i.e., $pta \mapsto SPARK$). Thus, for any $v \neq SPARK$, $pta \mapsto v$ disables $typesforcontext \mapsto TRUE$. In other words, type sensitivity [40] is only supported if the points-to analysis is *SPARK* [27]. If a user were to set another points-to analysis, like *PADDLE* [26], she would not have any indication that the setting of *typesforcontext* is irrelevant.

Another example is that in FlowDroid's configuration space, $implicit \mapsto ALL$, which indicates that FlowDroid tracks implicit flows, disables $codeelimination \mapsto REMOVEALL$, which enables the optimization to eliminate side-effect free code. This means that when FlowDroid analyzes implicit flows, it never removes side-effect free code even if the user explicitly sets the analysis to do so.

In total, we identified 5 and 7 disablement relationships in FlowDroid and DroidSafe, respectively. All these relationships were discovered through code investigation and were not specified in documentation.

Precision and soundness partial orders. While disablement describes relationships between the settings of two different options, we define two partial orders to describe the relationship between settings within an option.

For two settings v_1^1 and v_1^2 of an option c_1, $(c_1 \mapsto v_1^1) \sqsubseteq_S (c_1 \mapsto v_1^2)$ (or simply $v_1^1 \sqsubseteq_S v_1^2$) implies that $P_i[c_1 \mapsto v_1^1]$ will never produce more false negative results than (i.e., is at least as *sound* as) $P_i[c_1 \mapsto v_1^2]$ on the same input I for any P_i. Similarly, $(c_1 \mapsto v_1^1) \sqsubseteq_P (c_1 \mapsto v_1^2)$ (or simply $v_1^1 \sqsubseteq_P v_1^2$) means that $P_i[c_1 \mapsto v_1^1]$ will never produce more false positives than (i.e., is at least as *precise* as) $P_i[c_1 \mapsto v_1^2]$ on the same input I for any P_i. A precision partial order assumes that there is no difference in soundness between the two settings. If there is a difference in soundness, then the more sound configuration may detect more false positives, regardless of any difference in precision. In other words, a precision partial order $v_1^1 \sqsubseteq_P v_1^2$ implies the two soundness partial orders $v_1^1 \sqsubseteq_S v_1^2$ and $v_1^1 \sqsubseteq_S v_1^2$. For example, a DroidSafe configuration with $kobjsens \mapsto 3$ (which controls the level of object sensitivity [32]) should never produce more false positives than the same configuration with $kobjsens \mapsto 2$ since $(kobjsens \mapsto 3) \sqsubseteq_P (kobjsens \mapsto 2)$ and, since precision partial orders imply soundness partial orders, neither configuration should miss true positives that the other detects.

We constructed these partial orders based on our understanding of the intended effects of the settings from the documentation and from our domain expertise in static analysis. In total, we identified 158 partial orders (15 precision and 53 soundness partial orders in DroidSafe, and 16 precision and 74 soundness partial orders in FlowDroid). These counts include the implied soundness partial orders that precision partial orders produce. These partial orders are helpful for understanding the effects of configuration options. Furthermore, they enable more effective testing to uncover issues in configuration option implementation (Section 4.4).

For the tool users to better understand the relationships between analysis options, our released artifacts include configuration space graphs that visualize all disablements and partial orders in FlowDroid and DroidSafe [33].

3 EXPERIMENTAL DESIGN

With a clearer understanding of FlowDroid and DroidSafe's configuration spaces from our manual investigation, we designed our experiments to achieve two goals. First, we aim to recontextualize past studies by showing how considering configuration spaces and evaluating tool quality on real-world programs affects the results of an evaluation. Second, we aim to test and study configuration options in order to better understand their tradeoffs.

Target apps. We performed our experiments with three different datasets. Ideally, we would evaluate these tools on a large real-world benchmark with known ground truths, but such a benchmark does not exist. Thus, we collected two new datasets of real-world programs. First, we selected 30 apps from the FossDroid repository [7]. Since these apps are open-source, we were able to manually classify flows in order to evaluate precision. We initially downloaded the 10 most popular apps from each of the 17 categories on the FossDroid site, but found that DroidSafe could only analyze 30 of these 170 apps because it uses an old version of APKTool [3] that cannot decompile the newer apps. The first three columns in Table 1 show the names, versions and sizes of the apps in our FossDroid dataset.

Second, we collected the top 50 most popular apps from the Google Play app store, downloaded from APKMirror [2]. Without source code, we could not classify the flows produced by running on these apps, but we evaluated the tools' performance from these apps. The Google Play apps range in size from 1.4MB to 105MB, with a median size of 31MB. We only ran FlowDroid on the Google Play dataset because DroidSafe could not analyze any of the apps.

Finally, we used the entire DroidBench 3.0 benchmark [5, 35]. It is the latest version of this widely adopted micro-benchmark, and contains ground truths that allow us to measure the tools' accuracy. The benchmark is split into categories, each of which tests whether an analysis tool supports a specific feature, such as implicit flow tracking or field and object sensitivity. Using DroidBench

Table 1: The FossDroid dataset.

App name	Version	LOC	Flow	TP	FP
1010! Klooni	820	5387	0	0	0
Acrylic Paint	17	1274	28	0	28
Alarm Klock	15	3313	226	17	209
AndroSS	17	1405	50	4	42
Apple Flinger	1005006	14404	0	0	0
Budget	44	5984	9	0	7
Budget Watch	29	9836	25	2	23
Calendar Trigger	7	12003	46	0	0
CuprumPDF	4	215	0	0	0
Debian Kit	6	516	21	2	19
Dialer2	17	1868	107	10	97
Emerald Dialer	10	2158	50	7	33
Ensichat	17	338	0	0	0
Flexible Wallpaper	2	291	1	1	0
ForRunners	101030	19870	0	0	0
Free Fall	4	2803	3	1	2
Frozen Bubble	54	18007	50	6	42
Gloomy Dungeons 2	1602221800	121799	0	0	0
Locker	11	654	0	0	0
Mighty Knight	1	13496	0	0	0
OsmAnd+	355	448794	5	0	5
Overchan	54	72605	91	3	82
Polar Clock	10	119	0	0	0
Shana A.W.	10	2170	0	0	0
Tachiyomi	41	39149	5	1	0
Talalarmo	19	1224	1	0	0
Terminal Emulator	72	17429	50	1	40
Tux Paint	923	10837	4	4	0
WiFi Walkie Talkie	14	4542	57	4	52
Workout Log	2	1177	12	0	12
Total			**839**	**63**	**693**

3.0, we compare it to our real-world datasets and replicate past studies' methodologies in a configuration-aware manner. The sizes of DroidBench 3.0 apps range from 8 to 236 lines of code.

Configuration sampling. Because of the large configuration spaces in FlowDroid and DroidSafe, we generated samples of configurations to study. We included all algorithmic options, except for FlowDroid's *nostatics* option because we learned that internally, it simply sets *staticmode* to *NONE*, and so is redundant. For numeric options, we included the default setting and the settings calculated by the default value ± 10%, 20%, 50%, 100%, and 500% that were within the domain of the option. For all other options, we included all settings, except for the *MULTI* and *STUBDROID* settings of the *taintwrapper* option in FlowDroid, which required input files we could not successfully generate.

We created two samples to answer different research questions. First, we generated two-way covering arrays [34] using ACTS 3.2, a combinatorial testing tool from NIST [4]. We chose pairwise sampling over other sampling approaches such as random sampling because it guarantees the sampled configurations contain all two-way combinations of the option settings [15]. It is commonly assumed that most faults are caused by the interaction of only a few features [42]. We believe that this assumption extends to our goal of uncovering interesting behaviors caused by analysis options and their interactions. We sampled 144 *two-way configurations* for FlowDroid and 77 for DroidSafe.

Second, we constructed a set of configurations for each tool that was more suitable for testing individual options. For each setting $v_i^j \mid j \neq d$ of each option c_i, we constructed the configuration $P_D[c_i \mapsto v_i^j]$. We refer to these as *single-option configurations* because they change a single setting of an option from the default. There are 67 single-option configurations for FlowDroid and 41 for DroidSafe.

Experimental environment. All experiments were conducted on a server with 376GB of RAM and 2 Intel Xeon Gold 5218 16-core CPUs @ 2.30GHz running Ubuntu 18.04. We ran each app in DroidBench 3.0 with a 10-minute timeout, and each of the FossDroid and Google Play apps with a 2-hour timeout. The same timeout was used by Pauck et al. [35] for DroidBench 3.0, and our timeout for FossDroid and Google Play is four times longer than their experiments for real-world apps. We used Pauck et al.'s ReproDroid framework to run our experiments [35]. Each experiment was repeated three times and we report the median results for each configuration. In total, our experiments took 33, 940 machine hours to run.

Deduplication and classification. We classified the flows produced on the FossDroid dataset to assess the quality of the configurable tools on real-world programs. First, we post-processed the results to prepare for the classification. We only considered the flows whose sources and sinks are in the application code because we did not always have access to the library code to perform investigations. We also deduplicated the flows per the definition of equivalent flows in Section 2. Second, we performed manual classification. We classified all 423 flows reported by FlowDroid. It was infeasible to manually classify all 1536 distinct flows produced by DroidSafe, so we classified all flows in apps that had 50 or fewer reported flows, and for each of the remaining apps, we randomly selected 50 flows to investigate. This resulted in 416 flows to investigate for DroidSafe.

Classification was done by four student workers over 10 weeks. These students major in computer science and all have Android development experience. In the initial training session, the students were instructed to classify each flow as true, false, or inconclusive. Students were guided to classify a flow as inconclusive if the total time for investigating an individual flow exceeded 20 minutes. Each flow was classified by two students. If two students produced a conflict classifying a flow, one of the authors met with the students and discussed the flow until a consensus was reached. These conflicts happened 101 times. The students also produced justifications for each decision, in either textual format for flows that are relatively short or as graphs tracking dataflow for more complex flows. In total, we classified 63 true positives and 693 false positives. The remaining 83 flows were inconclusive. Columns 4, 5, and 6 in Table 1 show the numbers of investigated flows, classified true positives, and classified false positives for each app, respectively. We have made all the classified results and the justifications available in the released artifacts [33].

Metrics and statistical analysis. For all three datasets, we measured performance as the time in hours a configuration took to analyze the entire dataset. Because ground truths are known in

DroidBench 3.0, we measured the tools' results by F-measure (i.e., harmonic mean of precision and recall). We measured tools' precision (i.e., $\frac{|TPs|}{|TPs| + |FPs|}$) on FossDroid based on our classifications.

To find option settings that have a significant effect on the tool results, we performed linear regression with lasso regularization [43]. Lasso regularization adds an extra term to standard linear regression, selecting a coefficient vector β and a y-intercept β_0 that minimize

$$\frac{1}{2N}\sum_{i=1}^{N}(y_i - \beta_0 - x_i^T\beta)^2 + \lambda\|\beta\|_1$$

where y_i is the target value (i.e., precision, run time, or number of completed apps), x_i is a column vector of the option settings, λ is a hyperparameter, and $\|\|_1$ is the ℓ_1 vector norm. Higher settings of λ put more weight on the regularization term, making the model more aggressive. Adding the regularization term helps prevent overfitting by favoring models that set only a few coefficients to a nonzero value, and thus produces more interpretable models [43]. We experimented with several different feature selection models, and found this model to be the most suitable, as its optimum cross-validated results consistently selected the fewest options. We used R's glmnet package to build these models [18].

Research questions. Our empirical study aims to answer the following research questions:

RQ1: What do evaluations that focus only on a single configuration miss? RQ1 aims to better understand how the lack of attention to configurability has affected the evaluation of static taint analysis tools. We rerun the experiments on DroidBench 3.0 from Pauck et. al's evaluation of six Android taint analysis tools [35], and detail how a study that took configurations into account would come to different conclusions than one that only uses a single configuration.

RQ2: How do tools perform differently on micro-benchmarks and real-world datasets? RQ2 aims to determine whether a synthetic micro-benchmark is sufficient for evaluation, by comparing the behaviors of FlowDroid and DroidSafe's two-way configurations on our three datasets. By quantifying how the tools behave differently on larger apps, we demonstrate the urgency of developing large real-world benchmarks for future evaluations of real-world tools.

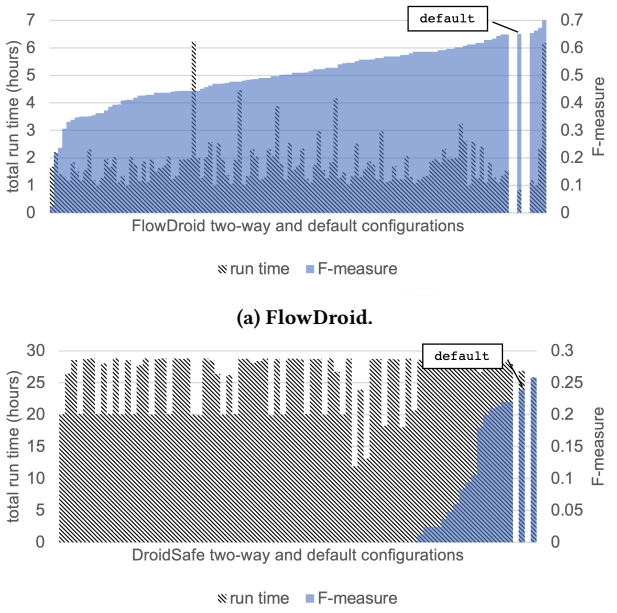
RQ3: Which option settings affect tool behavior? RQ3 aims to determine what effects individual options have on FlowDroid and DroidSafe's behaviors. Using the lasso models and data visualization, we make specific observations and recommendations for how a user could tune the tools.

RQ4: How many options misbehave, according to our partial orders? RQ4 aims to discover how many single options misbehave in terms of their expected behaviors. Our definition of partial orders for precision and soundness in Section 2 enabled this investigation. We show how many partial orders were violated, and present a case study demonstrating the implementation error of one option.

4 EXPERIMENTAL RESULTS

4.1 RQ1: What do evaluations that focus only on a single configuration miss?

We answer RQ1 by rerunning the experiments done by Pauck et al. on DroidBench 3.0 [35]. They evaluated six tools, and only the default configuration of each tool was used. To show the impact of the configurations, we additionally use the 2-way configurations of FlowDroid and DroidSafe. We used the same versions of Amandroid, DIALDroid, DidFail, DroidSafe, and IccTA as Pauck et al. [35], because these tools have not been updated since. We used a later version of FlowDroid [1]. Due to the differences in environment, we could not recreate the F-measures produced by Pauck et al. Thus, rather than comparing to their conclusions, we detail how considering configurations would change an evaluation done on only the default configurations per the results we obtained. Table 2 shows the results of running all six tools on DroidBench 3.0. Figure 2 shows more detail on the results of FlowDroid and DroidSafe's two-way configurations.



(a) FlowDroid.

(b) DroidSafe.

Figure 2: Performance (patterned bars) and F-measure (blue bars) for two-way and default configs on DroidBench 3.0.

Overall, *configurations introduce wide variance in terms of performance and accuracy that paints a different picture of the tools' capabilities.* First, consider FlowDroid. Figure 2a shows that FlowDroid's default configuration is the best-tuned in terms of performance (no other two-way configuration finishes faster), and is well-tuned in terms of F-measure (three configurations achieved a higher F-measure). The default configuration achieved an overall F-measure of 0.65, as shown in the last row of Table 2. The best configuration produced an overall F-measure of 0.71. Even the default configuration of FlowDroid outperforms every other tool in overall F-measure.

However, in terms of the specific features FlowDroid supports, one can be misled by the default, as shown in Table 2. For example, only considering the default, FlowDroid has no support for detecting inter-component flows that incorporate reflection (F-measure of 0 for the category Reflection_ICC in Table 2). The majority of configurations that we ran also achieved an F-measure of 0 for

Table 2: The F-measures of the six tools' results on DroidBench 3.0 organized by the feature categories in the benchmark. FlowDroid and DroidSafe results are each shown in three columns – the default configuration, the "max" category F-measure across any configuration, and the "best" configuration that achieved the highest overall F-measure. Cells in the max or best columns that have a higher F-measure than the default are highlighted in green.

Feature category	Flowdroid			Droidsafe			Amandroid	DidFail	IccTA	DIALDroid
	Default	Max	Best	Default	Max	Best	Default	Default	Default	Default
Aliasing	0.67	0.67	0.00	0.00	0.00	0.00	0.00	0.00	0.67	0.00
AndroidSpecific	0.95	0.95	0.95	0.46	0.46	0.00	0.46	0.43	0.84	0.00
ArraysAndLists	0.67	0.67	0.25	0.86	0.86	0.00	0.86	0.44	0.50	0.00
Callbacks	0.90	0.90	0.79	0.00	0.24	0.00	0.24	0.77	0.90	0.00
DynamicLoading	0.50	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
EmulatorDetection	0.97	0.97	0.97	0.00	0.00	0.00	0.00	0.00	0.93	0.00
FieldAndObjectSensitivity	1.00	1.00	0.44	0.00	0.67	0.00	0.44	0.80	1.00	0.00
GeneralJava	0.81	0.87	0.70	0.55	0.82	0.09	0.70	0.61	0.76	0.00
ImplicitFlows	1.00	1.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
InterAppCommunication	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.31	0.00	0.63
InterComponentCommunication	0.35	0.70	0.70	0.19	0.29	0.27	0.29	0.44	0.17	0.48
Lifecycle	0.83	0.91	0.91	0.34	0.80	0.70	0.80	0.41	0.69	0.00
Native	0.33	0.33	0.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Reflection	0.62	1.00	0.62	0.00	0.36	0.00	0.36	0.20	0.20	0.00
Reflection_ICC	0.00	0.31	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SelfModification	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Threading	1.00	1.00	0.80	0.00	0.00	0.00	0.00	0.67	0.67	0.00
UnreachableCode	1.00	1.00	0.86	0.00	0.00	0.00	0.00	1.00	1.00	0.00
Overall	**0.65**	**-**	**0.71**	**0.24**	**-**	**0.26**	**0.59**	**0.45**	**0.61**	**0.14**

this category. However, six of them achieved a higher F-measure, with a maximum of 0.31. Thus, FlowDroid does have partial support for these flows, making it the only tool in our experiments that supports this feature.[1] The four configurations that achieved this F-measure had the same settings for every option except for *maxcallbacksdepth*.

In the Reflection category, the default configuration shows partial support (F-measure = 0.62), yet the best configuration shows full support (F-measure = 1.00). This makes FlowDroid the only tool to have full support for Reflection in our experiments. The four configurations that achieved an F-measure of 1.00 all turned on the *enablereflection* option and set the *dataflowsolver* option to *FLOWINSENSITIVE*. However, there were 30 other configurations that did the same, which have a Reflection F-measure ranging from 0.36 to 0.94, so it is likely that there are multiple higher-degree interactions that cause this behavior.

Other cases show still partial, but better support than default for some features, like InterComponentCommunication (max: 0.70 vs. default: 0.35, changing FlowDroid from third-best to the best), GeneralJava (max: 0.87 vs. default: 0.81), and Lifecycle (max: 0.91 vs. default: 0.83). This aggregated picture is important, as even the best configuration has drawbacks if taken alone, such as no support for Aliasing, DynamicLoading, and ImplicitFlows. This supports the conclusion that multiple configurations need to be considered to fully understand a configurable tool's behavior.

The same conclusion can be drawn if we consider DroidSafe. Like with FlowDroid, Figure 2b shows that the default is well-tuned for DroidBench 3.0, but the only configuration to achieve a better overall F-measure also does so in less time. Considering only the default, one may think that the tool has no support for implicit flows, field and object sensitivity, callbacks, and reflection. As shown in column 6 of Table 2, DroidSafe in fact has full support for implicit flows, and partial support for the other listed features. Furthermore, the support for flows in the GeneralJava and Lifecycle categories can be improved considerably by taking configurations into account. Specifically, DroidSafe's best F-measure for GeneralJava is better than FlowDroid's default.

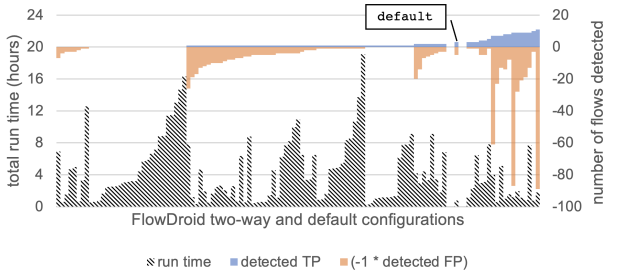
To summarize, only focusing on a single configuration paints an incomplete and incorrect picture of the range of outcomes that tools can present. Had studies to-date focused on configurations, they would have reached different results. We suggest that future evaluations evaluate multiple configurations, rather than only a single one.

4.2 RQ2: How do tools perform differently on micro-benchmarks and real-world datasets?

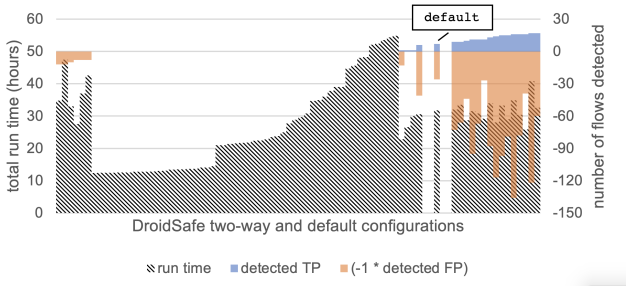
We compare the results across different datasets in terms of both performance and precision/F-measure. We have observed that *real-world datasets draw out more exaggerated impacts of configurations than the micro-benchmark, in terms of both performance and precision.*

First, in terms of performance, the tools' run times show larger variances on FossDroid and Google Play than on DroidBench 3.0. This can be seen in Figures 2-3 for DroidBench 3.0 and FossDroid, and the trend holds for Google Play. The fastest FlowDroid two-way configuration is 7.4x faster than the slowest two-way configuration in DroidBench 3.0. This ratio becomes 16.4x and 37.3x in FossDroid

[1] We use the terms *partial* and *full* support consistent with Pauck et al.'s study [35], in which full support for a feature means the tool achieved an F-measure of 1.00, and partial support means it achieved an F-measure in the range (0.00, 1.00).



(a) FlowDroid.



(b) DroidSafe.

Figure 3: Performance, number of true and false positives for two-way and default configs on FossDroid.

and Google Play, respectively. The variance appears to scale with the size of target app. The median program size for DroidBench 3.0 is 182KB, versus 568KB for FossDroid and 31.3MB for Google Play. This trend also exists for DroidSafe's configurations in Figures 2b and 3b. More importantly, larger programs exaggerate performance issues to such an extent that tools may no longer be scalable. For example, FlowDroid was able to analyze every DroidBench 3.0 app within the timeout (10 minutes), but hit the 2-hour timeout on FossDroid and Google Play in 5% and 24% of runs, respectively.

In terms of quality measurements, in Figure 2a, the F-measure values for all FlowDroid's configurations are greater than 0. This means that each configuration detected at least one true flow in some apps in DroidBench 3.0. However, in Figure 3a, 28% of the two-way configurations were not able to detect any true flows in FossDroid. We investigated these configurations and found that there were no single option settings that were always associated with these configurations. This result suggests that while DroidBench 3.0 is designed to test individual features that a tool supports, real-world apps may require a set of supported features for a tool to be useful. In Figure 3, we can see that in general, DroidSafe finds more true positives in FossDroid than FlowDroid does, with higher precision in some cases. This is a very different conclusion about the relative quality of the two tools compared to the overall F-measure for DroidBench 3.0 shown in Table 2, where FlowDroid clearly performed better.

To summarize, the different results in the real-world datasets and DroidBench 3.0 illustrate the large impacts of program sizes, dataset designs, and language features on the behaviors of tools' configurations. Specifically, the exaggerated impact that real-world datasets have relative to DroidBench 3.0 demonstrate the urgent need to create real-world benchmarks for the evaluation of configurable taint analysis tools. While the full ground truths are not known for the dataset we contribute, it can currently be used as a precision benchmark, and can serve as the starting point for future benchmarks.

Table 3: Analysis options selected by the lasso regularization. Values are colored based on their absolute value relative to other coefficients in that model. Cells with a . indicate that the feature was not selected.

FlowDroid	Precision	Run time	# Completed
λ	0.06	1.32E+06	0.16
aliasalgo	.	1.90E+05	-0.06
codeelimination	.	4.35E+05	-0.23
dataflowsolver	0.05	-1.15E+06	0.14
enablereflection	-0.02	2.58E+06	-0.29
nocallbacks	.	-1.24E+06	0.12
onecomponentatatime	.	6.86E+06	-0.75
onesourceatatime	.	6.62E+06	-0.74
pathalgo	-0.09	-2.89E+06	0.46
pathspecificresults	.	2.16E+06	-0.36
staticmode	.	-7.08E+05	0.10
DroidSafe	**Precision**	**Run time**	**# Completed**
λ	0.09	5.76E+06	1.04
analyzestrings_unfiltered	.	2.69E+06	.
apicalldepth	.	1.47E+04	.
kobjsens	0.02	.	.
limitcontextforcomplex	.	-1.40E+07	0.55
limitcontextforgui	<0.01	.	.
noarrayindex	.	-6.23E+06	.
nofallback	.	7.34E+06	.
nojsa	.	-8.09E+06	.
noscalaropts	.	-2.63E+06	.
nova	.	-1.14E+07	1.04
pta[2]	.	.	-0.02

4.3 RQ3: Which option settings affect tool behavior?

We answer this question by performing linear regression with a lasso regularization term as discussed in Section 3. We kept glmnet's default setting of generating a sequence of 100 λ values. We tried to select values of λ that achieved a Mean Square Error (MSE) near the optimum, but selected fewer features, in order to draw more general conclusions by reducing the risk that we overfit to our data [43]. We observed that the smaller feature sets selected by larger values of λ were typically subsets of the larger sets selected by smaller λ values, so we interpret these options as being good targets for a user to tune. The specific λ of each model along with the associated coefficients are shown in Table 3. Using the statistical models and data visualization, we make the following observations and suggestions for tuning FlowDroid and DroidSafe on real-world apps.

(1) *pathalgo is FlowDroid's most significant option for precision, but behaves in a counter-intuitive way, indicative of a soundness*

[2]Only configurations with *pta* $\mapsto$ *SPARK* produced flows. As such, we did not include *pta* in the regression model for the target precision.

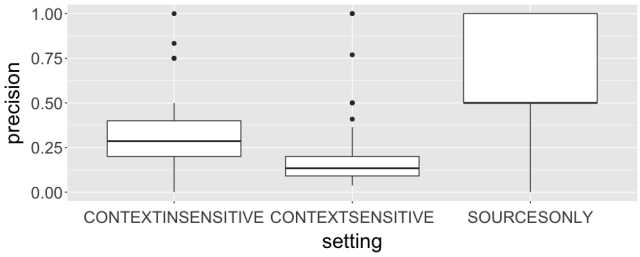
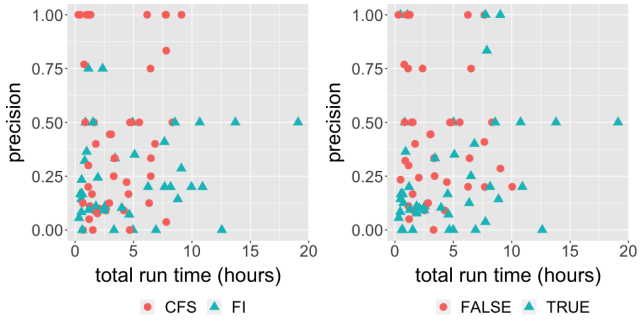


Figure 4: *pathalgo* settings plotted against precision. The top, middle, and bottom of the box are the first quartile, median, and third quartile, respectively. The vertical line is data outside of the interquartile range, and dots are outliers.



(a) dataflowsolver. **(b) enablereflection.**

Figure 5: The tradeoffs presented by settings of *enablereflection* and *dataflowsolver* in terms of precision and run time.

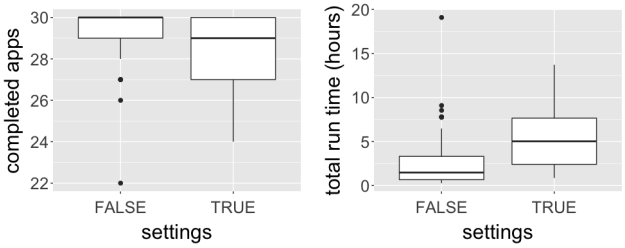


Figure 6: The settings of *onecomponentatatime* plotted against total run times and completed apps.

issue in its implementation. Figure 4 shows the settings of FlowDroid's *pathalgo* option and the range of precision values they achieved on the FossDroid benchmark. This setting controls the computation of the path between source and sink. *SOURCESONLY* is a "very fast context-insensitive implementation that only finds source-to-sink connections, but no paths" [1]. *CONTEXTINSENSITIVE* and *CONTEXTSENSITIVE* both build the taint paths, in a context-insensitive and context-sensitive manner, respectively. We determined two partial orders from reading the documentation: CONTEXTSENSITIVE $\sqsubseteq_P$ *SOURCESONLY* and *CONTEXTSENSITIVE* $\sqsubseteq_P$ *CONTEXTINSENSITIVE*. As shown in the figure, the result ended up being the opposite. Upon investigation of our data, we found this misbehavior was because of a soundness issue, such that configurations with *pathalgo* $\mapsto$ *SOURCESONLY* were able to achieve very high precision but they tended to find very few flows (and likewise for configurations with *pathalgo* $\mapsto$ *CONTEXTINSENSITIVE*). We explore partial order violations in more detail in Section 4.4.

(2) *FlowDroid's dataflowsolver and enablereflection options show important tradeoffs between precision and performance.* As an example, Figure 5 shows the tradeoff between precision and total run time with regard to settings of *dataflowsolver* (i.e., the IFDS solver) and *enablereflection*. The *CONTEXTFLOWSENSITIVE* (*CFS*) setting in Figure 5a is associated with better precision than the *FLOWINSENSITIVE* (*FI*) settings. We also see that the *CFS* setting appears to be more stable with regard to the run time, as configurations with *FI* selected demonstrate a higher run time variance. In Figure 5b, it is interesting that disabling reflection handling in FlowDroid (a less sound setting) resulted in higher precision. This is because discovering more true positives by a more sound configuration can come at the cost of producing more false positives. Thus, a user analyzing different target programs should consider tuning these options to balance the soundness, precision, and performance.

(3) *Several options across both tools show clear performance effects without having a significant effect on precision.* For example, Figure 6 shows that *onecomponentatatime* in FlowDroid (which controls whether the tool analyzes all components together or analyzes them one-by-one) is associated with a higher run time and fewer completed tasks. Furthermore, as shown in Table 3, this option was selected for total run time and completed apps, but not precision. Thus, a user should leave this option disabled, along with disabling *onesourceatatime*, disabling *pathspecificresults*, increasing the precision of *staticmode* (setting it to either *CONTEXTFLOWINSENSITIVE* or *CONTEXTFLOWSENSITIVE*), enabling *nocallbacks*, setting *codeelimination* to *NONE* and *aliasalgo* to *NONE*. Note that the last three settings are not the default settings of FlowDroid.

For DroidSafe, a similar trend as *onecomponentatatime* is observed in *limitcontextforcomplex*. This option, which limits objects with a points-to set of size > 100 to a heap context sensitivity of 1, is associated with lower run time if enabled. Similarly, *nova*, which disables DroidSafe's string analysis, also leads to better run time. As neither of these options appear to negatively affect precision, a user may be able to improve their performance by enabling both of these options (they are disabled by default). We also recommend the user leave *analyzestrings_unfiltered* and *nofallback* disabled, set *apicalldepth* to a low number, enable *noarrayindex*, enable *nojsa*, and enable *noscalaropts*.

(4) *No DroidSafe options strongly affect precision individually.* Table 3 shows that two options, *kobjsens* and *limitcontextforgui*, were selected by the lasso model, but neither of them have large coefficients. Figure 7 shows the relation between *kobjsens* settings (the option that determines the level of object sensitivity in DroidSafe) and precision. The lack of a strong effect is shown in the figure, where *kobjsens* settings less than 18 do not appear to be associated with a significant difference in precision. *kobjsens* $\mapsto$ 18 is associated with an increase in precision – this is a similar situation to that of Figure 4, where these configurations typically found only a single true positive, again indicating some soundness issue with higher

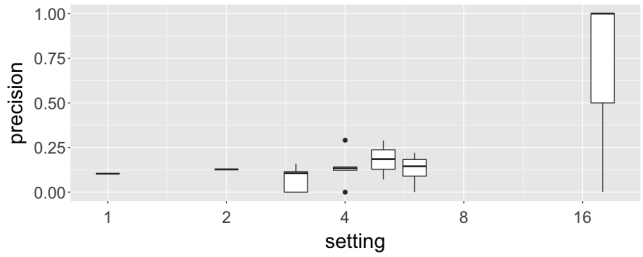


Figure 7: The settings of *kobjsens* plotted against precision.

```
1   Button button1= (Button) findViewById(R.id.button1);
2   button1.setOnClickListener(new View.OnClickListener() {
3     @Override
4     public void onClick(View v) {...}}
```

Figure 8: Extracted code from DroidBench 3.0's *Button2* app.

```
1   for (SootClass sc : Scene.v().getApplicationClasses()) {
2     if (sc.isConcrete()) {
3       for (SootMethod sm : sc.getMethods()) {
4         if (sm.isConcrete()) {
5           analyzeMethodForCallbackRegistrations(null, sm);
6           // ... }}}}
```

Figure 9: Callback search code of FlowDroid's *FAST* setting.

levels of *kobjsens*. Although no single options strongly affected precision, Figure 3b does show large variance in precision values across configurations. This indicates that there may be option interactions that are significant. We did not test these interactions because our data were not large enough.

To summarize, we made three important findings. First, two FlowDroid options (*dataflowsolver* and *enablereflection*) present interesting tradeoffs, and a user should consider tuning them for their analysis. Second, several options across both tools make important impacts on performance without an apparent effect on precision, so users should consider using the faster option by default. Finally, some options across both tools exhibit unexpected behavior indicative of an implementation bug. These are discussed next as violations of partial orders.

4.4 RQ4: How many options misbehave?

To answer RQ4, we compared the different settings of each option to each other in terms of the number of false positives and known false negatives from the DroidBench 3.0 and FossDroid data. For each precision or soundness partial order, we checked whether it was satisfied or violated in the single-option configurations. For example, the precision partial order $v_1^1 \sqsubseteq_P v_1^2$ is violated if $P_D[c_1 \mapsto v_1^1]$ produces more false positives than $P_D[c_1 \mapsto v_1^2]$, or either configuration misses true positives that the other detects because precision partial orders imply soundness partial orders.

We observed that *while most partial orders are preserved in the experimental data, there are 36 unique violations of partial orders concerning 21 options across both tools.* In total, we found 13 violations in DroidSafe (including 3 in DroidBench 3.0) and 23 violations in FlowDroid (including 14 in DroidBench 3.0). Of the 23 violations in FlowDroid, 5 were precision violations and 18 were soundness violations. Of DroidSafe's 13 violations, 3 were precision violations and 10 were soundness violations. We only count a violation if neither setting timed out and if the defined partial order was violated. Furthermore, although partial orders themselves are transitive, we do not record violations that occur between two settings that are only associated transitively in order to avoid duplicate results. Finally, we only count numeric options' partial orders once even if violations occur between multiple settings.

We believe these violations are caused by issues in the tool implementation or the misinterpretation of the intended behavior of an option. We have recorded all the violated partial orders in the release artifacts [9]. We reached out to the developers of the tools regarding these violations, and none indicated that they were aware of these issues.[3]

We investigated the root causes of some of these violations, and spend the rest of this section giving a case study of one violation between the *DEFAULT* and *FAST* settings of FlowDroid's *callback-analyzer* option. According to FlowDroid's documentation, the *DEFAULT* option only collects reachable callbacks in the call graph, while *FAST* includes all callbacks. Based on this, we identified the partial order *DEFAULT* $\sqsubseteq_P$ *FAST* while both settings are sound. However, in our results, we identified a violation, in which *FAST* was less sound than *DEFAULT*.

Upon further investigation of the implementation of FlowDroid, we found that this misbehavior was caused by the *FAST* callback analyzer not associating anonymous callback methods with the lifecycle classes in which they are defined. A common paradigm to define callback handlers is to create them by overriding methods within anonymous classes. As shown in Figure 8, a button's *onClick* method is specified through an anonymous subclass of *View.OnClickListener()*. We found that the *FAST* callback analyzer cannot detect flows in which (1) the sink is within one of these callback definitions, and (2) tainted data propagates through an attribute of the outer class (in this case, *Button2*).

We identified the root cause of this issue as the way the *FAST* callback analyzer scans for classes. As shown in Figure 9, the *FAST* callback analyzer simply searches through the methods of each class for callback registrations. On line 5, it passes *null* as the first parameter to *analyzeMethodForCallbackRegistrations*. This parameter is the lifecycle class, which in the case of the *onClick* callback on line 4 of Figure 8 would be the outer class (*Button2*). Since the *FAST* analyzer does not provide this information on the class hierarchy, the analysis cannot detect that data from a source elsewhere in *Button2* can flow into this callback. The *DEFAULT* setting's implementation uses a worklist algorithm that does retain this class hierarchy information, so it can track data from the outer class into the *onClick* method.

In summary, our definition of partial orders allowed us to systematically test the configuration spaces of FlowDroid and DroidSafe and identify violations that indicate implementation errors. Our case study demonstrates a specific case of misbehavior that we were only able to identify by testing multiple configurations with

[3] A developer of DroidSafe did reply, theorizing that different configurations may cause a change in internal analysis heuristics that might explain the unexpected results.

these partial orders. The number of partial order violations across FlowDroid and DroidSafe point to poor testing of non-default configurations. We suggest that future configurable tool evaluations take advantage of the partial orders and our experimental setup to test tools' correctness.

5 THREATS TO VALIDITY

There are several potential threats to the validity of our study. First, the datasets may not be representative of programs that tool users are interested in. Our mitigation is to use a well-known micro-benchmark and popular apps from two real-world app repositories. Second, we conducted three repetitions of the experiments, and thus performance variation may not be fully accounted for. While more repetitions would add greater statistical assurance, each trial takes an average of 505 machine days to run. We did observe variations in performance, but they were small and did not affect the broader trends. We computed the run time variance of the three trials of a configuration as (max-min)/median. The median variance across all configurations is 10%. Third, the two-way configuration samples may not capture all interactions in configuration options. Research indicates that most behaviors are caused by the interactions of a small number of options [42]. Because we ran every two-way interaction, we believe we captured most of the tools' behavior. Fourth, our manual classification may produce incorrect results. We mitigated this risk assigning each flow to two students, and resolving conflicts through unanimous decisions. Finally, our defined partial orders may be incorrect because of the ambiguity in tools' documentation. We tried to minimize this risk by applying our domain expertise and checking the tools' implementations.

6 RELATED WORK

To the best of our knowledge, we present the first empirical study that focuses on the configurations of Android taint analysis tools. Our work is related to (1) evaluations and comparative studies of Android taint analysis tools, (2) studies of configurations in Java static analysis tools, and (3) software product lines (SPLs) research that models performance of configurations.

Evaluations of Android taint analysis tools. Pauck et al. [35] evaluated whether six Android taint analysis tools fulfill the feature promises of their default configurations across different datasets. They contributed ReproDroid, which allows one to execute taint analysis tools and view their results in the unified AQL-Answer format. Qiu et al. [36] investigated the configuration options in FlowDroid, DroidSafe, and Amandroid and tuned them so that the tools supported the same language features. They then performed their comparative evaluation on these configurations. Boxler and Walcott [13] evaluated FlowDroid, IccTA, and DroidSafe on DroidBench 3.0 and a selection of apps from F-Droid in order to compare their quality. These studies have provided important insights and benchmarks for the literature to better assess the state-of-the-art of Android taint analysis, but all evaluate single configurations and rely heavily on micro-benchmarks. We believe our study demonstrates the pitfalls of using either of these methodologies.

Luo et al. [31] performed a qualitative evaluation of FlowDroid on 1022 real-world applications, and developed COVA, a static analyzer that identifies the conditions under which a tainted flow would occur. The authors also contributed a synthetic benchmark of 92 applications in order to evaluate COVA. ICC-Bench, released with Amandroid by Wei et al. [44] is a micro-benchmark containing 24 apps that test whether Android taint analysis tools can detect inter-app leaks. Bosu et al. [12] developed DIALDroid-Bench with DIALDroid, which consists of 30 real-world apps that contain known inter-app communication vulnerabilities. However, we could not use this benchmark because its source code is unavailable, and the data leaks are undocumented [35]. Similar to the past comparative studies [13, 35, 36], the evaluations of these tools do not focus on the tradeoffs presented by the algorithmic configurations.

Studies of configurable Java static analysis tools. Smaragdakis et al. [40] investigated the performance/precision tradeoffs of various context-sensitive analyses, including call-site, object, and type sensitivities. Lhoták and Hendren [28] evaluated tradeoffs among different abstractions of heap allocation sites in a points-to analysis. Wei et al. [45] evaluated an abstract interpretation tool with five configuration options. They studied 216 configurations and found these options in the numeric and heap domains presented tradeoffs to analysis precision and performance. Our work studies the significantly larger configuration spaces in Android taint analysis tools.

Performance modeling in SPLs. Software product lines research has modeled the performance impact of configurations in other software, e.g., [20–22, 38, 39, 47]. *CART* [20], *DECART* [21], and Zhang et al.'s Fourier transformation-based approach [47] only work on binary options, so they are not suitable to predict performance of our taint analysis tools which use binary, numeric, and enumerated options. SPL Conqueror [39] computes optimal variants of a configurable piece of software based on a user-defined non-functional property. In later work, Siegmund et al. augmented SQL Conqueror with the ability to produce performance-influence models, which quantify the impact of configuration options on the user's performance metric [38]. Ha and Zhang contributed DeepPerf, an approach to predict the performance of configurable software systems based on deep neural networks [22]. While these tools could have generated performance models that would take the place of our regression models, we decided that lasso regression was more suitable as it is easy to understand and limits the regression to select only a few significant options. That said, these tools are good candidates to integrate into future work that tries to automatically tune taint analysis tools based on a user's needs.

7 CONCLUSIONS

In this paper, we present the first systematic study on the large configuration spaces of Android taint analysis tools. Our large-scale study illustrates how two configurable Android taint analysis tools, FlowDroid and DroidSafe, behave under different configurations. We ran both tools on open-source real-world applications and contributed a dataset of 756 manually classified flows to assist in future evaluation of Android taint analysis tools. Our experiments on sampled configurations demonstrate that methodologies that only study a single configuration underestimate tool behavior and provide a misleading picture of tool capabilities. We also found that real-world datasets exaggerate the effects of configurations in a

way micro-benchmarks do not, motivating the need for large real-world benchmarks to fully evaluate Android taint analysis tools. Regarding individual configuration options, our statistical analysis identified opportunities for users to tune the default configurations to improve analysis results. Finally, our definition of partial orders exposed 22 options that violate expected behavior with regard to precision and soundness, which indicate the lack of systematic testing of configuration options and the complicated tradeoffs that configuration options can present.

The results of this study suggest several directions for future work. First, we will develop automated approaches to assist the tool users to effectively tune the analysis in the large configuration spaces. Second, we believe there is a pressing need for well-designed real-world benchmark suites, possibly using our dataset as a starting point, to assess the quality of Android taint analysis tools. Third, we plan to take advantage of the insights gained from partial orders to explore new methodologies and develop infrastructures to more systematically test configurable static analysis tools.

ACKNOWLEDGMENTS

This work is partly supported by NSF grants CCF-1816951 and CCF-2047682, the NSF graduate research fellowship program, and the McDermott fellowship program. We would also like to thank Felix Pauck for his help on setting up ReproDroid.

REFERENCES

[1] 2019. FlowDroid. https://github.com/secure-software-engineering/FlowDroid. 72734bd629dfae2aacaf6e6973abfe73d035c979.
[2] 2020. APKMirror. https://www.apkmirror.com. Accessed 2020-02-10.
[3] 2021. Apktool. https://ibotpeaches.github.io/Apktool/.
[4] 2021. Automated Combinatorial Testing for Software (ACTS). https://www.nist.gov/programs-projects/automated-combinatorial-testing-software-acts.
[5] 2021. DroidBench 3.0. https://github.com/FoelliX/ReproDroid.
[6] 2021. Fortify Static Code Analyzer. https://www.microfocus.com/en-us/solutions/application-security.
[7] 2021. FossDroid. https://fossdroid.com.
[8] 2021. HCL AppScan on Cloud. https://www.hcltechsw.com/wps/portal/products/appscan/home.
[9] Anonymous. 2021. *Rethinking Android Taint Analysis Evaluation: A Study on the Impact of Tool Configuration Spaces.* https://doi.org/10.5281/zenodo.4474536
[10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299
[11] Hamid Bagheri, Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. 2016. Practical, formal synthesis and automatic enforcement of security policies for android. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 514–525.
[12] Amiangshu Bosu, Fang Liu, Danfeng Yao, and Gang Wang. 2017. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 71–85.
[13] Dan Boxler and Kristen R Walcott. 2018. Static Taint Analysis Tools to Detect Information Flows. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*. The Steering Committee of The World Congress in Computer Science, Computer …, 46–52.
[14] Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. 2016. HornDroid: Practical and sound static analysis of Android applications by SMT solving. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 47–62.
[15] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. 2003. Constructing Test Suites for Interaction Testing. In *Proceedings of the 25th International Conference on Software Engineering* (Portland, Oregon) *(ICSE '03)*. IEEE Computer Society, USA, 38–48.
[16] Xingmin Cui, Jingxuan Wang, Lucas CK Hui, Zhongwei Xie, Tian Zeng, and Siu-Ming Yiu. 2015. Wechecker: efficient and precise detection of privilege escalation vulnerabilities in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. 1–12.
[17] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 576–587.
[18] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. 2010. Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of Statistical Software* 33, 1 (2010), 1–22. http://www.jstatsoft.org/v33/i01/
[19] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information flow analysis of android applications in droidsafe.. In *NDSS*, Vol. 15. 110.
[20] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wąsowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 301–311.
[21] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. 2018. Data-efficient performance learning for configurable systems. *Empirical Software Engineering* 23, 3 (2018), 1826–1867.
[22] H. Ha and H. Zhang. 2019. DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1095–1106. https://doi.org/10.1109/ICSE.2019.00113
[23] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 106–117.
[24] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. 2008. Implicit Flows: Can't Live with 'Em, Can't Live without 'Em. In *Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings (Lecture Notes in Computer Science)*, R. Sekar and Arun K. Pujari (Eds.), Vol. 5352. Springer, 56–70. https://doi.org/10.1007/978-3-540-89862-7_4
[25] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. 1–6.
[26] Ondrej Lhoták. 2006. *Program analysis using binary decision diagrams*. Vol. 68.
[27] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction* (Warsaw, Poland) *(CC'03)*. Springer-Verlag, Berlin, Heidelberg, 153–169.
[28] Ondřej Lhoták and Laurie Hendren. 2008. Evaluating the Benefits of Context-Sensitive Points-to Analysis Using a BDD-Based Implementation. *ACM Trans. Softw. Eng. Methodol.* 18, 1, Article 3 (Oct. 2008), 53 pages. https://doi.org/10.1145/1391984.1391987
[29] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 280–291.
[30] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. 2016. DroidRA: Taming Reflection to Support Whole-Program Analysis of Android Apps *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 318–329. https://doi.org/10.1145/2931037.2931044
[31] L. Luo, E. Bodden, and J. Späth. 2019. A Qualitative Analysis of Android Taint-Analysis Results. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 102–114.
[32] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (Jan. 2005), 1–41. https://doi.org/10.1145/1044834.1044835
[33] Austin Mordahl and Shiyi Wei. 2021. *The Impact of Tool Configuration Spaces on the Evaluation of Configurable Taint Analysis for Android.* https://doi.org/10.5281/zenodo.4729325
[34] Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *ACM Comput. Surv.* 43, 2, Article 11 (Feb. 2011), 29 pages. https://doi.org/10.1145/1883612.1883618
[35] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do android taint analysis tools keep their promises?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 331–341.
[36] Lina Qiu, Yingying Wang, and Julia Rubin. 2018. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 176–186.
[37] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462
[38] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence models for highly configurable systems. In *Proceedings of*

the 2015 10th Joint Meeting on Foundations of Software Engineering. 284–294.

[39] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPL Conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal* 20, 3 (2012), 487–517.

[40] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. *SIGPLAN Not.* 46, 1 (Jan. 2011), 17–30. https://doi.org/10.1145/1925844.1926390

[41] Ole Tange. 2020. *GNU Parallel 20200522 ('Kraftwerk')*. https://doi.org/10.5281/zenodo.3841377 GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them.

[42] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1, Article 6 (June 2014), 45 pages. https://doi.org/10.1145/2580950

[43] Robert Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)* 58, 1 (1996), 267–288.

[44] Fengguo Wei, Sankardas Roy, and Xinming Ou. 2014. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 1329–1341.

[45] Shiyi Wei, Piotr Mardziel, Andrew Ruef, Jeffrey S. Foster, and Michael Hicks. 2018. Evaluating Design Tradeoffs in Numeric Static Analysis for Java. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 653–682. https://doi.org/10.1007/978-3-319-89884-1_23

[46] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static Control-Flow Analysis of User-Driven Callbacks in Android Applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 89–99.

[47] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki. 2015. Performance Prediction of Configurable Software Systems by Fourier Learning (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 365–373. https://doi.org/10.1109/ASE.2015.15