Automatic Testing and Benchmarking for Configurable Static Analysis Tools

Austin Mordahl
austin.mordahl@utdallas.edu
University of Texas at Dallas
Richardson, Texas, USA

ABSTRACT

Static analysis is an important tool for detecting bugs in real-world software. The advent of numerous analysis algorithms with their own tradeoffs has led to the proliferation of configurable static analysis tools, but their complex, undertested configuration spaces are obstacles to their widespread adoption. To improve the reliability of these tools, my research focuses on developing new approaches to automatically test and debug them. First, I describe an empirical study that helps to understand the performance and behavior of configurable taint analysis tools for Android. The findings of this study motivate the development of ECSTATIC, a framework for testing and debugging that goes beyond taint analysis to test any configurable static analysis tool. The next steps for this research involve the automatic creation of real-world benchmarks for static analysis with associated ground truths and analysis features.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Automated static analysis**.

KEYWORDS

configurable static analysis, testing, debugging, benchmarking

ACM Reference Format:
Austin Mordahl. 2023. Automatic Testing and Benchmarking for Configurable Static Analysis Tools. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3597926.3605232

1 INTRODUCTION

Software influences nearly every aspect of modern life. As software becomes more complex and ubiquitous, the risk posed by bugs becomes evermore severe. Thus, techniques to detect and fix bugs are critical. Software testing is the most common technique to detect bugs; however, it is not exhaustive, and thus can fail to find bugs associated with rare inputs or execution paths. Static analysis can be a complementary approach to testing, as it attempts to model every possible execution of a program and prove some property about it (such as it being free of privacy leaks or memory errors).

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3605232

The problem of proving non-trivial properties about a program is undecidable [25]. Thus, a static analysis designer must delicately balance the goals of soundness, precision, and termination. The performance of an analysis depends not only on the analysis algorithms used, but also on the features of the program being analyzed. Thus, many static analysis tools come with various configurations, representing various algorithms with their own tradeoffs. An end user can optimize the analysis by configuring the tool with settings appropriate for their target program(s).

However, configurable static analysis faces obstacles to widespread adoption [12, 30]. Configuration options are often implemented in ad-hoc ways that can make their purpose unclear even to domain experts. There may be undocumented relationships between configuration options that make a tool behave in unintuitive ways. Because of the large configuration spaces, such tools are often poorly tested and only evaluated on a single configuration [7, 11, 23, 24, 28]. One reason for this is that it is difficult to come up with new benchmarks that are capable of testing the precision and correctness of an analysis [18].

The goal of my research is to improve the reliability of configurable static analysis tools. Achieving this goal would help push towards widespread adoption of static analysis tools. In addition to producing higher quality analysis tools, this would also result in higher quality software in general as these tools become part of everyday developer workflows. I plan to achieve this goal through a combination of empirical studies, novel theoretical contributions, and engineering efforts. Towards this goal, I identify three significant milestones in the progress of my Ph.D. research: better understand the full behavior of configurable static analysis tools (M1), develop an approach to simplify and automate the task of testing and debugging configurable static analysis tools (M2), and develop an approach to ease the production of new benchmarks that test specific features of static analysis tools (M3).

I first present an empirical study that aims to reevaluate three taint analysis tools for Android which had only been compared on single configurations before (M1; Section 2). This study proposes an explicit encoding of configuration spaces to capture intended behaviors of configuration options. We leverage these relations between configurations to develop new testing and debugging approaches for static analysis tools, that work even without ground truths. These approaches are realized in a tool, ECSTATIC, a reproducible and extensible framework for automatically testing and debugging static analysis (M2; Section 3). Although ECSTATIC can find issues that manifest between configurations, it cannot be used to find bugs that are everpresent or that exist in tools without configurations. Therefore, the final piece of the puzzle is the development of analysis feature specifications that can be used to automatically produce

static analysis benchmarks from real-world programs (M3; Section 4).

2 UNDERSTANDING CONFIGURABLE TAINT ANALYSIS

Despite the presence of configuration options in many static analysis tools, evaluations of such tools tend to focus only a single configuration [7, 11, 23, 24, 28]. These evaluations fail to capture the full range of a given tool's behavior, and may present a misleading picture about the relative capabilities of different tools. Thus, we performed the first comprehensive study of configurable static taint analysis tools that considers their behavior throughout many configurations. For this study, we studied two static taint analyzers for Android: FlowDroid [7] and DroidSafe [11]. We chose these two tools because they were extensively configurable and had been compared against each other in prior works [8, 17, 23, 24], all of which only studied a single configuration of each tool.

Our goals in this work were to evaluate these tools through the lens of their full configurability, and to make actionable suggestions for best practices regarding the tuning and testing of these tools. The methodology for achieving these goals consisted of two components. First, a manual investigation of the source code of FlowDroid and DroidSafe, with the goal of understanding the tradeoffs between the individual settings of analysis options, as well as the relation between different analysis options. Second, we adopted techniques from combinatorial interaction testing (CIT) to perform an empirical evaluation of the two tools through the lens of their full configuration space.

As a result of the manual investigation, we identified and encoded two distinct relations in the configuration spaces of these two tools. The first relation is *disablement*, in which one option setting masks the effect of another. For example, FlowDroid has an option, *implicit*, which indicates the extent to which the tool should track implicit flows [14]. Setting this option to *ALL* (i.e., track all implicit flows) *disables* all settings of another option, *codeelimination*, which allows FlowDroid to perform code optimizations that are sound with respect to the detection of explicit flows, but may change implicit data flows. These relationships, intentional or not, existed in both tools and were never documented.

The second relationship comprises two partial orders over an option's settings – these are the *precision* and *soundness* partial orders. For example, an analysis setting of *2-object-sensitivity* would, with regard to precision, precede (i.e., be at least as at precise as) a *1-object-sensitivty* setting. These relations encode the expected effects of a setting relative to another setting on the result set produced by an analysis. Let A and B be two partially-ordered configurations, where two configurations are partially ordered if they differ only in the settings of one option o, such that A's setting of o and B's setting of o have a partial order relationship. If A is more precise than B, then we expect the set of false positives produced by B (denoted $FP(B)$) to be a subset of $FP(A)$ on the same input. Similarly, a soundness partial order encodes the expectation that, if A is more sound than B, the set of true positives produced by A (denoted $TP(A)$) should be a superset of $TP(B)$ on the same input. We explicitly encoded all disablement and partial order relations that we identified and used them to run multiple experiments to test the configurations of FlowDroid and DroidSafe.

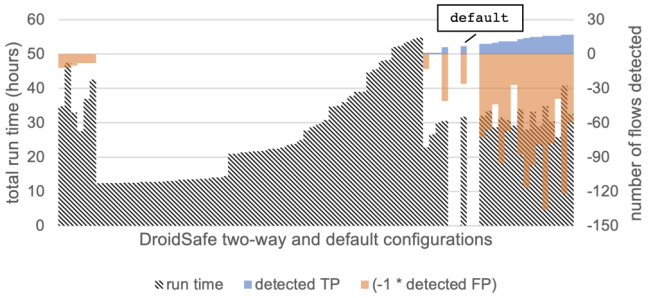

Figure 1: The number of true and false positives produced by each two-way configuration of DroidSafe on FossDroid, plotted alongside performance.

For our study, we collected three benchmarks: DroidBench 3.0 [1, 7, 23], a microbenchmark developed by the creators of FlowDroid and for which the ground truths are known; 30 apps from the FossDroid repository of open-source Android software [2]; and 50 apps from the Google Play app store.

We constructed two distinct configuration samples in order to understand different aspects of the tools. First, to better understand the effects of individual option settings, we generated a sample of configurations that, had at most only a single option difference from the default configuration (*single-option configurations*). Then, to better understand the wide range of possible behaviors caused by option interactions, we used combinatorial testing approaches to generate sample configurations for FlowDroid and DroidSafe. We ran each configuration of both tools on all benchmarks.

As a result of our evaluation, we found that configurations introduce a wide variance in terms of performance, precision, and soundness. For example, Figure 1 shows the results of running DroidSafe's two-way configurations as well as its default configuration on the FossDroid benchmark. As can be seen in the configurations to the right of the default, we found multiple two-way configurations that detected more true positives than the default configuration. We also found that the majority of configurations did not produce any results. Furthermore, we found evidence of bugs in the tools by comparing the expectations encoded by our partial order relations to the actual results. In 21 cases across both tools we found cases where a more precise/sound setting behaved contrary to our expectations. These findings indicate that future evaluations of static analysis tools should consider many configurations rather than just one. They also point to a lack of thorough testing and evaluation of non-default configurations in these tools.

3 TESTING CONFIGURABLE STATIC ANALYSIS

Motivated by the lack of rigorous testing of configurable static analysis tools, we aim to develop a framework to assist practitioners (i.e., analysis designers and users) find and fix bugs in these tools. To do so, we leverage the partial order relationships identified in the previous work to develop novel approaches for (1) finding bugs in configurable static analysis, and (2) reducing input programs to the features that induce these bugs. We then aim to instantiate these ideas in a framework that can be used by practitioners. The goals in

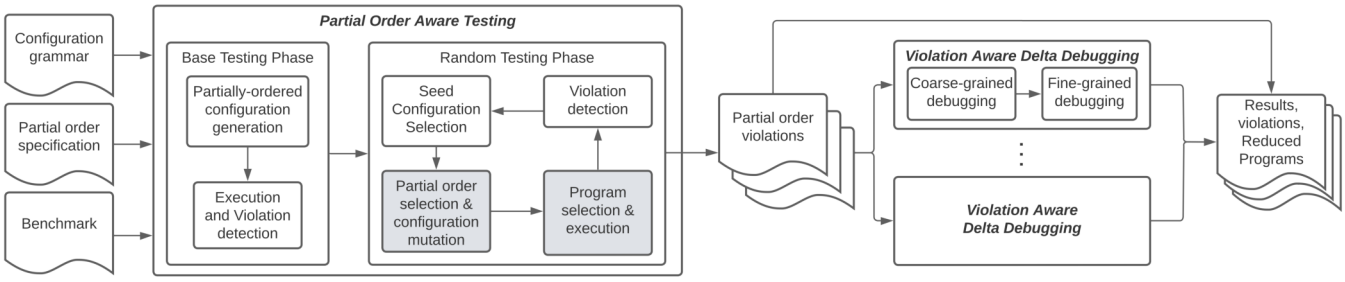

Figure 2: An overview of ECSTATIC.

designing such a framework are to make it easy-to-use and general, such that a user (i.e., an analysis designer or user) could integrate new tools and input programs with ease. We aim to assist the user not only in detecting bugs, but also in the debugging process.

We realize this goal in ECSTATIC, which is a highly customizable approach and accompanying tool. ECSTATIC realizes novel partial order-aware techniques to detect bugs and then uses a new two-staged violation-aware delta debugging approach to reduce input programs to their failure-inducing features. Figure 2 shows an overview of ECSTATIC's approach. Given a partial order specification, a configuration grammar, and a set of input programs as input, ECSTATIC tests programs in two phases. In the *base testing phase*, ECSTATIC generates configurations akin to the single-option configurations described in Section 2 (i.e., a single setting different from the default). It runs each configuration on each program in the input program set, then detects partial order violations. The second phase is *random testing*, which aims to construct configurations that exercise more feature interactions. We randomly select a number of partial orders for which a violation has not yet been detected, then produce a random configuration using the supplied configuration grammar. We mutate this random configuration to produce partially-ordered configurations, then run these configurations on a random subset of input programs. This phase is repeated until a set time limit is hit.

A critical aspect of ECSTATIC is that it elides the static analysis oracle problem by allowing testing on input programs without known ground truths. This allows us to use large, real-world programs to perform testing. We achieve this by pairing every explicitly defined partial order with one or more implicit partial orders; the combination of explicit and implicit partial orders leads to set constraints that can be checked without knowing the classifications of a tool's results. In order to assist users in the debugging of analysis results, for each detected partial order violation, we perform delta debugging [32] to reduce input programs down to failure-inducing features. Our delta debugging approach has two key novelties. First, we use the presence of a partial order violation as the predicate the delta debugger uses to determine whether to accept a change. Still, the specific use case presented by ECSTATIC (i.e., reducing inputs to static analysis tools) presents a unique difficulty in terms of time: compilation and, to an even greater extent, rerunning the static analysis tool can take quite a long time and effective delta debugging requires many iterations, especially for larger programs (e.g., real-world programs). Therefore, the second novelty is the design and implementation of a two-stage delta debugging technique, taking inspiration from Kalhauge and Palsberg's class-dependence

Table 1: Configuration spaces of SOOT, WALA, DOOP, and FlowDroid, and lines of code needed to integrate them into ECSTATIC.

	SOOT	WALA	DOOP	FlowDroid
# Options	20	5	20	22
# Partial Orders	20	26	35	77
Total LoC	90	53	111	156

Table 2: Partial order bugs detected in each tool by dataset. MB is microbenchmark, and RW are real-world. The bar in each cell differentiates bugs detected in base testing (left) and bugs detected only in random testing (right).

	SOOT	WALA	DOOP	FlowDroid	Total
MB	3 | 0	0 | 0	0 | 0	26 | 2	29 | 2
RW	18 | 0	6 | 3	12 | 0	2 | 7	38 | 10
Total	18 | 0	6 | 3	12 | 0	28 | 7	64 | 10

based delta debugging [13]. This algorithm works on a class-level granularity, taking care to only remove sets of classes that form transitive closures on the class-dependence graph (CDG). We first run CDG-based delta debugging on the input program, then run hierarchical delta debugging (HDD) [19] on the reduced program.

ECSTATIC runs every experiment inside a Docker container for portability and consistency, and provides simple interfaces to the user to allow them to run the tool on new tools or input program sets. In order to integrate a new static analysis tool, a user produces a configuration space specification and a configuration grammar. Additionally, a user implements a Dockerfile, as well as writing an implementation of a tool runner and tool reader class, by extending abstract classes provided by ECSTATIC. Table 1 shows the total number of lines of integration code we wrote to implement four static analyzers, SOOT, DOOP, WALA, and FlowDroid. As can be seen, a user needs to write few lines of code in order to integrate new tools. To add new input program sets, the user simply provides ECSTATIC with a script that downloads the source code, as well as an index indicating which programs to analyze.

We used ECSTATIC to test the four aforementioned static analyzers on microbenchmarks and real-world programs. Table 2 shows the number of bugs we were able to find in each tool, broken down by testing phase and dataset. We were able to detect 74 partial order violations across the four tools in total. In SOOT and FlowDroid, we were able to find bugs using the microbenchmark. This is especially significant since DroidBench, the microbenchmark for Android, is used as the test suite for FlowDroid. Most (64) bugs were found by the base testing phase; however, 10 additional bugs were only found

by the random testing phase, indicating that they are predicated upon feature interactions. We have reported 42 of these bugs to developers; in all cases where developers have responded, they have confirmed that we found misbehavior. Our two-phase delta debugger was also effective, able to reduce a real-world program to only 1% of its original size given a 6-hour timeout. These results demonstrates ECSTATIC's bug-finding and debugging utilities.

4 BENCHMARKING CONFIGURABLE STATIC ANALYSIS

To ensure the ongoing quality of static analysis tools, it is necessary to create high quality benchmarks for static analysis. Such benchmarks make evaluations more comparable and provide common understanding on the expected behavior of an analysis.

We define four criteria for high-quality static analysis benchmarks: (1) the benchmark should be categorized by the analysis feature that it is testing. This allows for a comprehensive comparison of the features that different tools support. (2) The benchmark should have ground truths against which an analysis result can be compared. This would allow identification of both precision and soundness issues. (3) The benchmark should be similar to actual programs that an analyzer would be expected to consume in real-world scenarios. (4) The benchmark should be easily extensible with new programs. Microbenchmarks, like CATS [5] and DroidBench, meet criteria (1), (2), and (4), but comprise small, handwritten programs that do not meet (3). Conversely, real-world benchmarks meet criterion (3), but tend to only support either one of (2) or (4) because of the difficulty in determining ground truths for real programs.

We thus propose a methodology to automatically produce benchmarks that meet these criteria, through the definition and detection of analysis feature patterns. These patterns represent run-time behavior that can be mapped to a partial analysis result. Consider an analysis tool that claims reflection support (a feature). We can write a dynamic analysis to report calls to Java's `Method.invoke`, along with the run-time parameters (which encode the target method as a string). This detector can be run on real-world programs, and the ground truth can be produced automatically.

The expected contributions of this work include an interface for creating pattern specifications, which will be consumed by software to automatically produce benchmarks. We also plan to contribute an evaluation to demonstrate the utility of our system. To evaluate the system, we plan to perform evaluations similar to other feature-wise evaluations of static analysis tools, in order to demonstrate that our approach can produce benchmarks broken into feature categories that can be used to test static analysis tools. There are various challenges to this direction of work. It will be a challenge to write patterns that balance generality and precision. Furthermore, it will be challenging to write an interface for creating these patterns that is expressive enough to capture various types of runtime behavior.

5 PROGRESS

The results of the work realizing M1 and M2 have been published at ISSTA [21] and ICSE [22], respectively; M3 is my current work. I have three additional relevant publications. Two apply machine learning; one to classify static analysis results [31], and to automatically configurre static analysis tools [15]. The third work concerns the ability of static analysis to find variability bugs in C software product lines [20].

6 RELATED WORK

The work to realize M1 was inspired by other works that evaluated Android taint analysis tools [8, 23, 24]. Each work only evaluates a single configuration, whereas our work tries to understand these tools through the lens of their full configurability. Other studies have studied analysis tools with small configuration spaces [16, 27, 29], while our work focuses on the much larger configuration spaces of Android taint analysis tools.

On testing and debugging static analysis tools, Do *et al.* created Visuflow, a visual debugging environment for FlowDroid [10]. Visuflow presents general information, like the IR and the ICFG, which are useful artifacts for debugging. However, unlike ECSTATIC, it neither performs testing nor finds bugs. We envision Visuflow and ECSTATIC serving complementary roles in helping developers debug static analyses. Andreasen and Møller used a delta debugger, JS Delta [3], and the TAJS inspector [4] to diagnose precision and performance bugs in analysis of jQuery [6]. We similarly apply delta debugging to reduce programs to their failure-inducing inputs, but our approach can debug both precision and soundness issues and is applicable to many analysis tools. Generally, metamorphic testing is commonly used in compiler testing [9]. We are unaware of any other work that applies metamorphic testing to partially ordered configurations in static analysis tools.

The work we outline in Section 4 is related to other works that attempt to benchmark static analysis. Recent work produces new benchmarks for Android taint analysis using fuzzing [26]. Luo *et al.* aim to ease the process of manually inspecting taint analysis results in order to create new malware benchmarks [18]. These works are important first steps toward the vision of automatic benchmarking, but we aim to be more general in our approach, allowing for specification of analysis features that can be detected on programs across multiple languages, hopefully eliding the need for manual investigation.

7 CONCLUSIONS

Configurable static analysis tools have the potential to complement dynamic approaches like testing, but face obstacles to widespread adoption. To help overcome these obstacles, I outline work towards the goal of improving the reliability, maintainability, and usability of configurable static analysis. The first step of this research was an empirical study on two configurable Android taint analysis tools [21]. This study proposed a formalization of relationships within a static analysis tool's configuration space and evaluated the tools through their full configurability. The results of this study motivated ECSTATIC, a tool for automatically performing metamorphic testing on configurable static analysis tools [22]. Next, I will develop an approach that allows the automatic generation of benchmarks with associated analysis features and ground truths.

ACKNOWLEDGMENT

This work was partly supported by NSF grants CCF-2047682, CCF-2008905, the NSF graduate research fellowship program, and Eugene McDermott Graduate Fellowship 202006.

REFERENCES

[1] 2021. DroidBench 3.0. https://github.com/FoelliX/ReproDroid.

[2] 2021. FossDroid. https://fossdroid.com.

[3] 2022. JS Delta. https://github.com/wala/jsdelta.

[4] 2022. TAJS. https://github.com/cs-au-dk/TAJS.

[5] 2022. The Call-graph Assessment & Test Suite. https://bitbucket.org/delors/cats/src/master/.

[6] Esben Andreasen and Anders Møller. 2014. Determinacy in Static Analysis for jQuery. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *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

[8] 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 Engineering and Applied Computing (WorldComp)., 46–52.

[9] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1, Article 4 (feb 2020), 36 pages. https://doi.org/10.1145/3363562

[10] Lisa Nguyen Quang Do, Stefan Krüger, Patrick Hill, Karim Ali, and Eric Bodden. 2020. Debugging Static Analysis. *IEEE Transactions on Software Engineering* 46, 7 (2020), 697–709. https://doi.org/10.1109/TSE.2018.2868349

[11] 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.

[12] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. 672–681. https://doi.org/10.1109/ICSE.2013.6606613

[13] Christian Gram Kalhauge and Jens Palsberg. 2019. Binary reduction of dependency graphs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 556–566.

[14] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. Dta++: dynamic taint analysis with targeted control-flow propagation.. In *NDSS*.

[15] Ugur Koc, Austin Mordahl, Shiyi Wei, Jeffrey S Foster, and Adam A Porter. 2021. SATune: a study-driven auto-tuning approach for configurable software verification tools. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 330–342.

[16] 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

[17] 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.

[18] Linghui Luo, Felix Pauck, Goran Piskachev, Manuel Benz, Ivan Pashchenko, Martin Mory, Eric Bodden, Ben Hermann, and Fabio Massacci. 2022. TaintBench: Automatic Real-World Malware Benchmarking of Android Taint Analyses. *Empirical Softw. Engg.* 27, 1 (jan 2022), 41 pages. https://doi.org/10.1007/s10664-021-10013-5

[19] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 142–151. https://doi.org/10.1145/1134285.1134307

[20] Austin Mordahl, Jeho Oh, Ugur Koc, Shiyi Wei, and Paul Gazzillo. 2019. An Empirical Study of Real-World Variability Bugs Detected by Variability-Oblivious Tools. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 50–61. https://doi.org/10.1145/3338906.3338967

[21] 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* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 466–477. https://doi.org/10.1145/3460319.3464823

[22] Austin Mordahl, Zenong Zhang, Dakota Soles, and Shiyi Wei. 2023. ECSTATIC: An Extensible Framework for Testing and Debugging Configurable Static Analysis. In *2023 45th International Conference on Software Engineering (ICSE)*.

[23] 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* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 331–341. https://doi.org/10.1145/3236024.3236029

[24] 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* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 176–186. https://doi.org/10.1145/3213846.3213873

[25] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society* 74, 2 (1953), 358–366.

[26] Stefan Schott and Felix Pauck. 2022. Benchmark Fuzzing for Android Taint Analyses. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 12–23. https://doi.org/10.1109/SCAM55253.2022.00007

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

[28] 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.

[29] 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*. Springer International Publishing, 653–682. https://doi.org/10.1007/978-3-319-89884-1_23

[30] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, You Have given Me Too Many Knobs!: Understanding and Dealing with over-Designed Configuration in System Software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 307–319. https://doi.org/10.1145/2786805.2786852

[31] Sai Yerramreddy, Austin Mordahl, Ugur Koc, Shiyi Wei, Jeffrey S Foster, Marine Carpuat, and Adam A Porter. 2023. An empirical assessment of machine learning approaches for triaging reports of static analysis tools. *Empirical Software Engineering* 28, 2 (2023), 28.

[32] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.

Received 2023-05-24; accepted 2023-06-07