Improving Real-World Applicability of Static Taint Analysis

Linghui Luo

Promotion: Universität Paderborn, Fakultät für Elektrotechnik, Informatik und Mathematik

Erstgutachter: Prof. Dr. Eric Bodden

Zweitgutachter: JProf. Dr.-Ing. Ben Hermann

Drittgutachter: Dr. Julian Dolby

Datum der Prüfung: 27. Oktober 2021

Veröffentlichung: Universitätsbibliothek Paderborn, DOI: https://doi.org/10.17619/UNIPB/1-1233

Kurzfassung: Security breaches happen on a daily basis and are a serious threat to our society. The average cost of a data breach in 2021 has achieved the highest record in the 17-year history of IBM's Cost of a Data Breach Report, rose from $3.84 million to $4.24 million [1]. In May 2021, the American oil pipeline operator Colonial Pipeline had to pay 75 Bitcoins (nearly $5 million) to recover its stolen data in a ransomware attack that disrupted fuel supplies in the northeastern United States [2]. Since the first release of the German Luca app for contact tracing during the COVID-19 pandemic, a chain of security vulnerabilities has been discovered and a great skepticism has arisen [3][4]. Such incidents do not only cost a significant amount of money and company reputation, but can also be a threat to national security. As reported in 2019, German Chancellor Angela Merkel and hundreds of German politicians were hit by a data hack, with confidential letters, contact information and party memos leaked on Twitter [5]. Ensuring security of software applications is more important than ever in today's fast-paced technological world.

While there exist many techniques to ensure software applications being built securely and cyber resilient, attacks remain a potent threat due to the limited effectiveness and adoption of these techniques. Static taint analysis is a program analysis technique that can be used to prevent a wide range of security vulnerabilities and detect malicious software. As Figure 1 shows, it tracks data flows from sensitive *sources* (e.g., API which reads untrusted user input or private data) to sensitive *sinks* (e.g., API which executes a dangerous function or posts data to the internet). Such data flows are called taint flows. Many well-known issues can be triggered by taint flows, e. g., data theft, SQL injections, cross-site scripting, etc. Although there have been a wealth of static taint analysis tools created in both

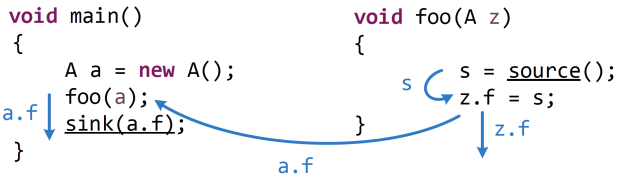

Figure 1: How static taint analysis tracks data from a source to a sink.

industry and academia, very few are widely used in industry, despite the importance of the issues these tools can detect. This dissertation investigates why and focuses on improving the real-world applicability of static taint analysis. It addresses three existing problems that hinder the real-world adoption of static taint analysis.

Problem 1: common benchmarks are small and incomplete. Static taint analysis tools have been mostly evaluated on micro benchmarks (small programs with artificially constructed vulnerabilities), which are often designed by analysis builders to test tool features, and are not representative of real-world software applications (big and complex programs). This leads to analysis tools that work well on micro

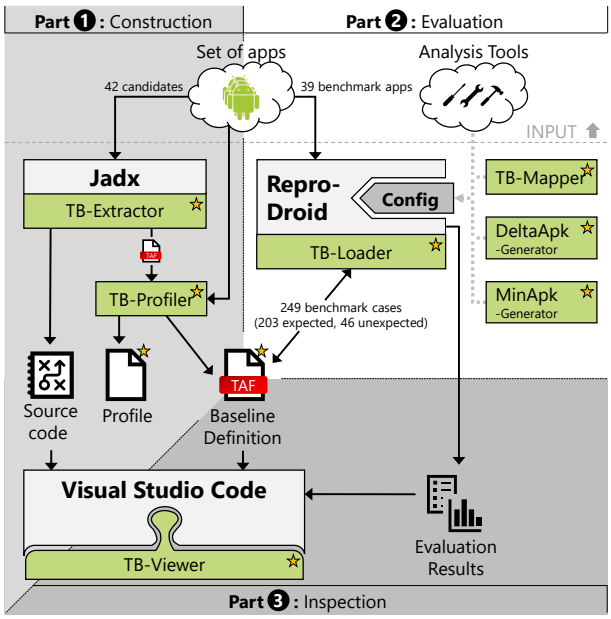

Figure 2: Overview of the TaintBench framework.

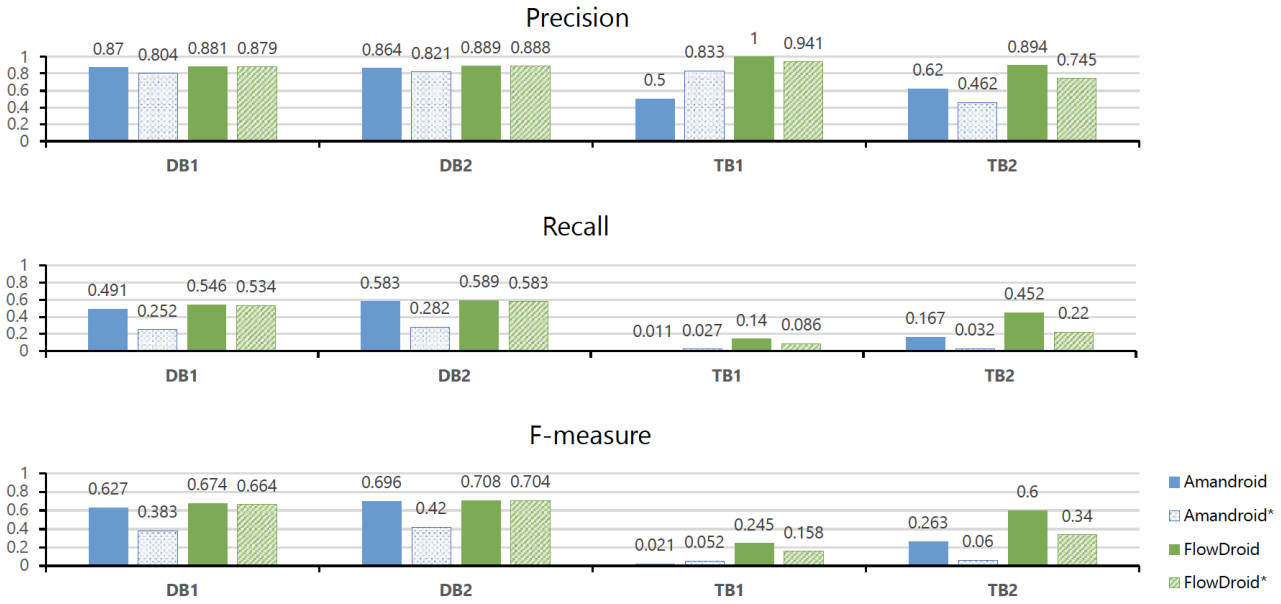

Figure 3: Results of experiment 1 (DB1 & TB1) and 2 (DB2 & TB2). Experiment 1: analysis tools are in default configuration and evaluated on DroidBench (DB1) and TaintBench (TB1). Experiment 2: analysis tools are configured with sources and sinks specified in the ground truth of DroidBench (DB2) and TaintBench (TB2) respectively. * marks update-to-date tool versions.

benchmarks, but are less effective in finding real-world issues. To address this problem, the first step is to construct more realistic benchmarks. As the first contribution of this dissertation, we constructed a realistic malware benchmark suite for Android taint analysis. We focus on Android in this work, but similar issues occur in other settings too [6, 7, 8, 9]. The benchmark suite, called TaintBench, is the first real-world suite in this area with a documented baseline ground truth. So far the ground truth of real-world benchmarks has been rarely documented in past evaluations of static taint analysis tools, although it is needed for reproducing the results.

Our TaintBench benchmark suite contains 39 malware applications with 249 documented benchmark cases as the baseline ground truth. These benchmark cases contain both expected and unexpected cases. Expected cases are verified security issues which should be detected by taint analysis tools, and unexpected cases specify false-positive cases which an imprecise analysis tool might still report. Once a taint analysis tool finds an expected case while analyzing this benchmark app, it is counted as a true positive (TP). A missed expected case is counted as a false negative (FN). Consequently, found and missed unexpected cases are counted as false positives (FP) and true negatives (TN) respectively. Based on the countings of TP, TN, FP and FN with regard to the benchmark cases, accuracy metrics such as precision ($p = \frac{TP}{TP+FP}$), recall ($r = \frac{TP}{TP+FN}$), and F-measure ($f = \frac{2pr}{p+r}$) can be computed. These three metrics are widely used and accepted in evaluations of static analysis tools.

Along with the suite, we developed the TaintBench framework, an open source tool chain that supports real-world benchmarking of Android taint analysis tools. As shown in Figure 2, the tools are divided into three parts that enable: (1) faster benchmark suite *construction*, (2) reproducible *evaluation* of static taint analysis tools with this suite, and (3) source code *inspection* of analysis results. Using the TaintBench suite and framework, we conducted 6 experiments with different configurations to evaluate two versions of popular static taint analysis tools—FlowDroid [10] and Amandroid [11]. For comparison, we evaluated the same analysis tools using the micro benchmark suite DroidBench [12], which has been widely used for benchmarking Android taint analyses. Our results in Figure 3 show that these tools have much lower precision, recall and F-measure on real-world malware apps in comparison to micro benchmark apps. Even with an unrealistic configuration of these tools (i. e., tools are configured with sources and sinks used by the malicious taint flows in experiment 2.), the majority of malicious taint flows in TaintBench remain undetected.

We further investigated FlowDroid, which achieved the best result with TaintBench in our evaluation. Our investigation revealed that 35% of malicious taint flows in TaintBench could not be detected by FlowDroid because relevant methods were missing in the call graphs. Call graphs are important building blocks for inter-procedural static analysis, which are difficult to create for modern framework-based appli-

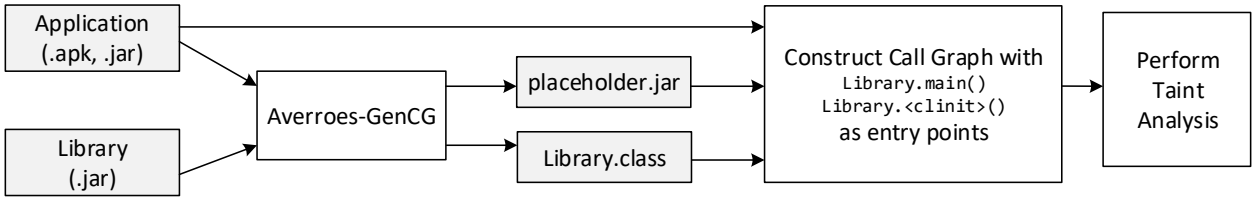

Figure 4: Overview of the GenCG approach.

cations. Modern Java frameworks like Android, Spring and Apache Struts are designed with the Inversion of Control principle, which heavily makes use of the Java reflection API that in turns makes finding reachable code hard. This makes it extremely hard to predict the behaviors of these frameworks by analyzing their code statically. To be scalable, most static analysis tools model the effects of frameworks rather than analyzing them [13, 11, 10, 6]. FlowDroid, for example, models the behavior of the Android framework by constructing a dummy main method that simulates the lifecycle of each Android component and starts the analysis from there. However, this precise modeling is difficult to maintain because every year a new version of Android comes out with new APIs that introduce new behaviors into the framework. Moreover, it is impractical to do this for every framework. FlowDroid has such a hand-crafted model, but we show that it leads to missing code, i.e., incomplete call graphs.

To ameliorate this problem, we created GenCG, a new approach to creating a call graph that is more complete, yet manageable. Our approach GenCG is not limited to any particular framework or analysis tool and is therefore highly reusable. In GenCG, we developed the AverroesGenCG tool—an improvement of Averroes [14], which generates a placeholder library for a given Android/Java framework. This generated placeholder can be used by common call graph construction algorithms (e.g. VTA, RTA, Spark [15, 16]) and taint analysis as a replacement for the original framework. The behavior of the framework is modeled in the placeholder code and reflected in the constructed call graphs. While a general approach can be noisy (produces many false positives), our experiments show that our carefully-constructed one does not sacrifice the precision. We evaluated our approach with both DroidBench and TaintBench. It works especially well on our real-world benchmark suite TaintBench: both the precision (from 0.83 to 0.88) and the recall (from 0.20 to 0.32) of FlowDroid are improved using the call graphs constructed with our approach. On DroidBench, we were not expecting significant difference in the recall, as the false negatives produced by FlowDroid are not mainly caused by incomplete call graphs in these micro benchmark apps. Still the recall is improved, as our approach allowed FlowDroid to detect more taint flows. It produced a few more false positives as we expected for such a general approach. Nevertheless, the precision is only slightly decreased (from 0.87 to 0.82). To show its generalizability, we introduce how our approach can be applied to web applications using the Spring framework. The evaluation with a micro benchmark suite of 42 Spring-based web applications shows promising results.

Problem 2: real-world issues often of limited interest. Even given a complete enough call graph, analysis precision is still key. In terms of improving precision, various sensitivities are considered by static taint analysis tools. To handle aliasing and virtual dispatch in Java programs, static tools apply context-, object-, and field-sensitivities. While a flow-sensitive analysis takes the order of statements into account, a path-sensitive analysis evaluates branch conditions. Although most static taint analysis tools support multiple sensitivities at the same time, path-sensitivity is usually left out, resulting in warnings that are either unrealizable or that a given user will not care about (e.g., issues exist in code targeting old hardware that is not supported anymore). The third contribution of this dissertation tackles this problem. We designed an approach that computes partial path constraints to enhance results produced by a client analysis. We implemented a tool called COVA that combines data flow analysis with the use of the satisfiability modulo-theories solver Z3 [17]. COVA can be configured to track information one is interested in, e.g., user inputs, I/O operations or system settings, and reasons about the path constraints over such information for each reachable statement in the program.

Using COVA, we conducted a qualitative study of more than 28,000 taint flows from a large number of real-world commercial Android applications. These taint flows were generated by FlowDroid. We classified these taint flows using path constraints computed by COVA. This allowed us to determine how these taint flows depend on environment configuration settings (e.g., platform versions), user interactions (e.g., button usage), and I/O operations (e.g., reading and writing files) as Figure 5 shows: About 1/3 of taint flows are false positives that should be dis-

carded. About 4,000 taint flows are due to the three factors we defined. In addition, 10% of taint flows can be triggered at runtime only by certain user interactions (UI-constrained). Our results suggest that hybrid approaches to dynamic validation of static taint flows should focus on modeling user interactions, but should also consider configuration and I/O to a limited extent. Based on these findings, we extended COVA to not only compute the path constraint of a given instruction, but also to generate concrete user interactions required to execute that instruction at runtime. Experiments with a small sample of applications from F-Droid [18] demonstrate the feasibility of this approach to generate valid user interactions to test randomly selected program instructions. This approach can be used to improve the accuracy of static taint analysis, e.g. only taint flows that can be validated at runtime should be reported to the user.

Problem 3: little adoption by developers. Creating a sound, precise and scalable static analysis tool is unfortunately not enough in practice. Software developers are often faced with the task of learning and applying a large number of technologies in far too short a time. The secure usage of these technologies is a major hurdle. Often, security is not explicitly tested. Although security analysis such as static taint analysis can help, current analysis tools are unfortunately not well adopted by developers. As many recent studies have shown, these tools are not adopted if they do not provide actionable results or if they do not present them in a way that is understandable to developers and integrated into developers' workflow [9, 19, 20].

While static taint analysis tools can help developers find security vulnerabilities in their code, such analysis has been less used in interactive development environments (IDEs) such as Eclipse, IntelliJ, Android Studio, and Visual Studio Code. However, IDEs would be the ideal place for static analysis, and are desired by developers. Even when there is IDE integration, tools such as DroidSafe [13], Cheetah [21], and IBM Security AppScan [22] usually only support a particular IDE, as a significant technical effort is required to integrate a particular analysis for a particular language into a particular IDE. Given this effort, the sheer variety of common tools and potentially useful analyses makes it impractical to develop every combination. To encourage better adoption of these tools by developers, researchers need ways to make tools easier and faster to integrate into IDEs. As the fourth contribution to this dissertation, we have developed a general approach to integrating static analysis into IDEs and editors—MagpieBridge. To show the generalizability of MagpieBridge, we integrated some analyses from academia into IDEs—FlowDroid [10], CogniCrypt [23], and Ariadne [24], and two analysis tools from industry—Facebook Infer [25] and Amazon CodeGuru Reviewer [26].

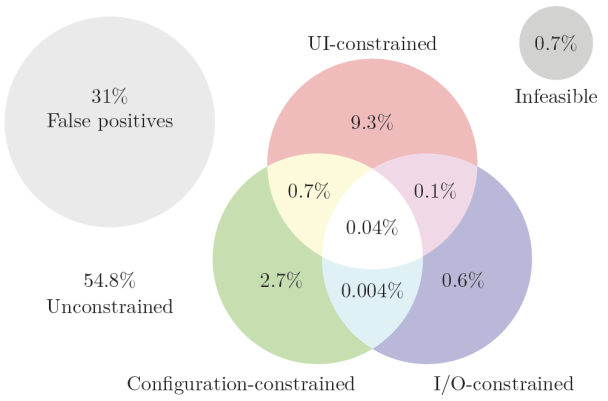

Figure 5: Different types of real-world taint flows.

Nowadays, static taint analysis is mostly implemented in static application security testing (SAST) tools. Many companies offer SAST tools as a service in the cloud. The current solutions for interaction between cloud-based SAST tools and developers are usually web-based. Developers often find it cumbersome to switch back and forth between IDE and web browser when they want to fix the issues detected by these tools in their code. Therefore, we conducted a multi-stage user study with software engineers at Amazon Web Services (AWS) to investigate how IDE support for a purely cloud-based static analysis tool should be designed to meet developer expectations. Based on interviews with developers, we developed a prototype IDE support for the cloud-based SAST tool Amazon CodeGuru Reviewer. We evaluated this prototype with 32 software engineers at AWS in a usability test compared to the existing web-based solution. We found that developers were three times more likely to perform code scans with our IDE prototype than with the web-based solution. They were also able to fix issues identified by Amazon CodeGuru Reviewer faster in the IDE. However, we found that incorporating the tool's results into the IDE did not fully improve developers' workflow. Some developers had difficulty understanding the asynchronous nature of the IDE solution. They would like more, e.g. real-time feedback on analysis progress, fast validation of fixes, etc.

Designing static taint analysis tools for the real-world is challenging, which requires analysis builders to not only find the best trade-off between precision, soundness and scalability, but also design tools with good usability. With respect to these aspects, we show static taint analysis needs to be refined for the real world and that it can be improved by addressing the above mentioned real-world problems. We hope that the presented benchmarks, approaches, their implementations and shared insights in this dissertation can help static taint analysis builders to create better tools and foster the adoption of static taint analyses

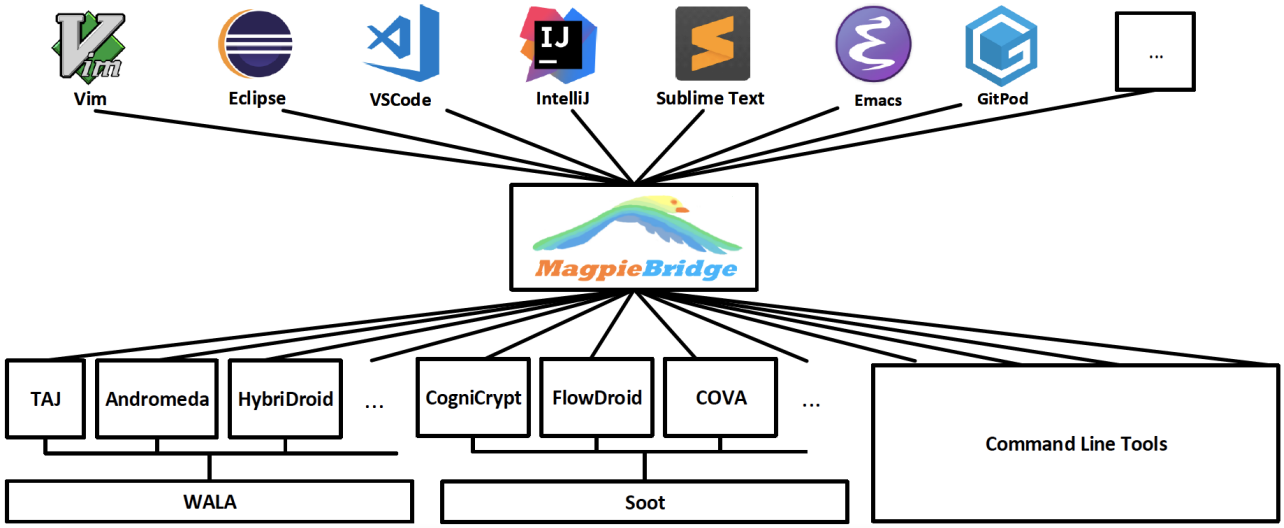

Figure 6: MagpieBridge connects arbitrary static analyses to arbitrary IDEs and editors.

by software developers to build more secure software.

Contributed publicly available artifacts in this dissertation:

- TaintBench:`https://taintbench.github.io`
- COVA:`https://github.com/secure-software-engineering/COVA`
- MagpieBridge:`https://github.com/MagpieBridge/MagpieBridge`

Literatur

[1] Statista, "Annual number of data breaches and exposed records in the united states from 2005 to 2020." `https://www.statista.com/statistics/273550/data-breaches-recorded-in-the-united-states-by-number-of-breaches-and-records-exposed`, 2020. Accessed: 2021-08-03.

[2] N. P. Michael D. Shear and C. Krauss, "Colonial pipeline paid roughly 5 million in ransom to hackers." `https://www.nytimes.com/2021/05/13/us/politics/biden-colonial-pipeline-ransomware.html`, 2021. Accessed: 2021-08-03.

[3] M. Reuter, "Schon wieder desaströse sicherheitslücke in luca app." `https://netzpolitik.org/2021/it-sicherheit-schon-wieder-desastroese-sicherheitsluecke-in-luca-app`, 2021. Accessed: 2021-09-21.

[4] J. Kersten, "Luca-app – mehr kosten als nutzen?." `https://www.daserste.de/information/wirtschaft-boerse/plusminus/sendung/sr/sendung-vom-09-06-2021-luca-app-100.html`, 2021. Accessed: 2021-09-21.

[5] D. Merchan, "German politicians' personal information leaked on twitter in mass cyberattack." `https://abcnews.go.com/International/german-politicians-personal-information-leaked-twitter-mass-cyberattack/story?id=60160048`, 2019. Accessed: 2022-01-31.

[6] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg, "F4F: taint analysis of framework-based web applications," in *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011* (C. V. Lopes and K. Fisher, eds.), pp. 1053–1068, ACM, 2011.

[7] S. Wei and B. G. Ryder, "Practical blended taint analysis for javascript," in *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013* (M. Pezzè and M. Harman, eds.), pp. 336–346, ACM, 2013.

[8] A. Antoniadis, N. Filippakis, P. Krishnan, R. Ramesh, N. Allen, and Y. Smaragdakis, "Static analysis of java enterprise applications: frameworks and caches, the elephants in the room," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020* (A. F. Donaldson and E. Torlak, eds.), pp. 794–807, ACM, 2020.

[9] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *35th*

International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013 (D. Notkin, B. H. C. Cheng, and K. Pohl, eds.), pp. 672–681, IEEE Computer Society, 2013.

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of PLDI*, ACM, 2014.

[11] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of CCS*, ACM, 2014.

[12] DroidBench 3-0. `https://github.com/secure-software-engineering/DroidBench/tree/develop`, September 2016. Accessed: 2021-08-03.

[13] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe," in *Proceedings of the 22nd NDSS*, The Internet Society, 2015.

[14] K. Ali and O. Lhoták, "Averroes: Whole-program analysis without the whole program," in *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings* (G. Castagna, ed.), vol. 7920 of *Lecture Notes in Computer Science*, pp. 378–400, Springer, 2013.

[15] V. Sundaresan, L. J. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for java," in *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000* (M. B. Rosson and D. Lea, eds.), pp. 264–280, ACM, 2000.

[16] O. Lhoták and L. J. Hendren, "Scaling java points-to analysis using SPARK," in *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings* (G. Hedin, ed.), vol. 2622 of *Lecture Notes in Computer Science*, pp. 153–169, Springer, 2003.

[17] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pp. 337–340, 2008.

[18] F-Droid. `https://F-Droid.org`, 2020. Accessed: 2021-08-03.

[19] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, (New York, NY, USA), p. 332–343, Association for Computing Machinery, 2016.

[20] L. N. Q. Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. R. Murphy-Hill, "Cheetah: just-in-time taint analysis for android apps," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pp. 39–42, 2017.

[21] L. N. Q. Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. R. Murphy-Hill, "Just-in-time static analysis," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017* (T. Bultan and K. Sen, eds.), pp. 307–317, ACM, 2017.

[22] IBM, "Appscan." `https://www.hcltechsw.com/appscan`, 2007. Accessed: 2021-08-03.

[23] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, *et al.*, "Cognicrypt: supporting developers in using cryptography," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 931–936, IEEE Press, 2017.

[24] J. Dolby, A. Shinnar, A. Allain, and J. Reinen, "Ariadne: analysis for machine learning programs," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018* (J. Gottschlich and A. Cheung, eds.), pp. 1–10, ACM, 2018.

[25] Facebook, "Facebook infer." `https://fbinfer.com`, 2015. Accessed: 2021-08-03.

[26] A. W. Services, "Amazon codeguru reviewer." `https://aws.amazon.com/codeguru`, 2020. Accessed: 2021-08-03.