



UNIVERSITÄT PADERBORN

Die Universität der Informationsgesellschaft

Faculty for Computer Science, Electrical Engineering and Mathematics

Improving Real-World Applicability of Static Taint Analysis

Linghui Luo

Dissertation

Submitted in Partial Fulfillment
of the Requirements for the Degree of
Doktor der Naturwissenschaften (Dr. rer. nat.)

Advisor:
Prof. Dr. Eric Bodden

Paderborn, September 12, 2021

Abstract

Static taint analysis is a program analysis technique that can be used to detect malicious software and a wide range of security vulnerabilities. Although there have been many static taint analysis tools created in both industry and academia, very few are widely used in industry, despite the importance of the problems 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.

The first problem is the lack of real-world benchmarks. 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 for real-world software applications (big and complex programs). This leads to analysis tools that work well on micro benchmarks, but are less effective in finding real-world issues, since real-world applications contain more corner cases. To address this problem, we constructed a real-world malware benchmark suite called TAINTBENCH, with a well-documented baseline ground truth, i.e., verified security issues that should be found. With TAINTBENCH we re-evaluated popular static taint analysis tools for Android applications. Our results show that these tools miss most of the critical issues we identified for the TAINTBENCH suite. We investigated FLOWDROID, which performed best in our evaluation. Our investigation reveals that incomplete call graphs are a major reason why FLOWDROID could not detect many issues. To ameliorate this problem, we created GENCG, which produces a placeholder library that models framework's behaviors to construct sound call graphs. We show that our approach enables the static taint analysis in FLOWDROID to detect more issues in TAINTBENCH and DROIDBENCH without losing the precision. We also demonstrate the application of GENCG on Spring-framework-based web applications to show it is not limited to Android, but is generally applicable for modeling Java framework behaviors.

For the sake of scalability, static taint analysis tools often ignore path conditions and produce only may-results. This leads to the second problem: tools can produce warnings that are either unrealizable or a given user will not care about (e.g., issues exist in code targeting old hardware that are not supported anymore). The third contribution of this dissertation tackles this problem using COVA, which computes path constraints that can be applied for refining the results of any client analysis. Using COVA we conducted a large-scale study on real-world commercial Android applications to gather information about the nature of real-world taint flows. Our study shows how static taint-analysis tools can be improved and how path constraints computed by COVA can be used to eliminate false positives and prioritize warnings in analysis reports.

One more reason that state-of-the-art static taint analysis tools are not widely adopted by software developers, is that they are not integrated into IDEs that are commonly used by developers, making them cumbersome to use. We tackle this issue with a general approach called MAGPIEBRIDGE, which enables analysis providers to bring analyses into IDEs more easily and quickly in comparison to traditional plugin-based IDE integration. We shows its generaliz-

ability by integrating multiple analyses produced both in academia and industry. As the last contribution of this dissertation, we conducted a user study with software developers at Amazon Web Services to understand what specific features are expected by developers for IDE integration of cloud-based Static Application Security Testing (SAST) tools. We implemented MAGPIEBRIDGE-based IDE support for Amazon CodeGuru Reviewer and tested its usability with developers. One of our findings is that developers expect the IDE support for cloud-based SAST tools to behave similarly to the lightweight static analyzers they are familiar with. However, reusing the same visual components for the SAST tool that are also used by lightweight static analyzers created confusion. Developers need more clear visual cues in the IDE to understand the asynchronous nature of cloud-based analyses. Our study also shows that IDE integration significantly encouraged developers to interact more with the SAST tool in comparison to its existing browser-based integration.

Through this dissertation, 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.

Zusammenfassung

Statische Taint-Analyse ist eine Programmanalysetechnik, mit der bösartige Software und eine Vielzahl von Sicherheitslücken aufgespürt werden können. Obwohl sowohl in der Industrie als auch im akademischen Bereich viele statische Taint-Analyse-Werkzeuge entwickelt wurden, werden nur sehr wenige davon in der Industrie eingesetzt, und dies ungeachtet der Bedeutung der Probleme, die diese Werkzeuge erkennen können. Diese Dissertation untersucht die Gründe dafür und konzentriert sich darauf, die Praxistauglichkeit der statischen Taint-Analyse zu verbessern. Sie befasst sich mit drei bestehenden Problemen, die den Einsatz der statischen Taint-Analyse in der Praxis behindern.

Das erste Problem ist das Fehlen von realistischen Benchmarks. Statische Taint-Analyse-Werkzeuge wurden meist an Mikro-Benchmarks (kleine Programme mit künstlich konstruierten Schwachstellen) evaluiert, die oft von den Entwicklern der Analysewerkzeuge entworfen wurden, um die Funktionen der Werkzeuge zu testen, und die nicht repräsentativ für reale Softwareanwendungen (große und komplexe Programme) sind. Dies führt zu Analysewerkzeugen, die zwar in Mikro-Benchmarks gut funktionieren, aber weniger effektiv bei der Suche nach Problemen in der realen Welt sind, da reale Anwendungen mehr Sonderfälle enthalten. Um dieses Problem zu lösen, haben wir in dieser Dissertation eine Benchmark-Suite für reale Malware namens TAINTBENCH mit einer gut dokumentierten Ground-Truth-Basis entwickelt, d. h. verifizierte Sicherheitsprobleme, die gefunden werden sollten. Mit TAINTBENCH haben wir populäre statische Taint-Analyse-Werkzeuge für Android-Anwendungen neu evaluiert. Unsere Ergebnisse zeigen, dass diese Werkzeuge die meisten der kritischen Probleme nicht erkennen, die wir für die TAINTBENCH-Suite identifiziert haben. Wir haben FLOWDROID genauer untersucht, das bei unserer Bewertung am besten abgeschnitten hat. Unsere Untersuchung ergab, dass unvollständige Call-Graphen ein Hauptgrund dafür sind, dass FLOWDROID viele Probleme nicht erkennen konnte. Um dieses Problem zu beheben, haben wir GENCG entwickelt, das eine Platzhalterbibliothek erstellt, die das Verhalten des Frameworks modelliert, um solide Call-Graphen zu erstellen. Wir zeigen, dass unser Ansatz die statische Taint-Analyse in FLOWDROID in die Lage versetzt, mehr Probleme in TAINTBENCH und DROIDBENCH zu erkennen, ohne an Präzision zu verlieren. Wir demonstrieren auch die Anwendung von GENCG auf Spring-Framework-basierte Webanwendungen, um zu zeigen, dass es allgemein für die Modellierung von Java-Framework-Verhaltensweisen anwendbar ist.

Aus Gründen der Skalierbarkeit ignorieren statische Taint-Analyse-Werkzeuge oft Pfadbedingungen und liefern nur “May-Results”. Dies führt zum zweiten Problem: Werkzeuge können Warnungen ausgeben, die entweder nicht realisierbar sind oder die ein bestimmter Benutzer nicht beachten wird (z. B. Probleme in Code, der auf alte Hardware abzielt, die nicht mehr unterstützt wird). Der dritte Beitrag dieser Dissertation befasst sich mit diesem Problem und stellt das Werkzeug COVA vor, das Pfadbeschränkungen berechnet, die zur Verfeinerung der Ergebnisse einer beliebigen Client-Analyse verwendet werden können. Mit COVA haben wir eine groß angelegte Studie mit realen kommerziellen Android-Anwendungen durchgeführt, um Informa-

tionen über die Art der realen Taint-Flows zu sammeln. Unsere Studie zeigt, wie Taint-Analyse-Werkzeuge verbessert werden können und wie die von COVA berechneten Pfadeinschränkungen zur Eliminierung von falsch-positiven Ergebnissen und zur Priorisierung von Warnungen in Analyseergebnissen verwendet werden können.

Ein weiterer Grund dafür, dass statische Taint-Analyse-Werkzeuge, die dem neuesten Stand der Technik entsprechen, von Softwareentwicklern nicht in großem Umfang eingesetzt werden ist, dass sie nicht in IDEs integriert sind, die von Entwicklern üblicherweise verwendet werden, was ihre Verwendung umständlich macht. Wir gehen gegen dieses Problem mit einem allgemeinen Ansatz namens MAGPIEBRIDGE vor, der es Analyseanbietern ermöglicht, Analysen einfacher und schneller in IDEs einzubinden, als dies mit der traditionellen Plugin-basierten IDE-Integration möglich ist. Wir zeigen seine Verallgemeinerbarkeit, indem wir mehrere Analysen integrieren, die sowohl im akademischen Bereich als auch in der Industrie erstellt wurden. Als letzten Beitrag dieser Dissertation haben wir eine Nutzerstudie mit Softwareentwicklern bei Amazon Web Services durchgeführt, um zu verstehen, welche spezifischen Funktionen von den Entwicklern in Bezug auf die IDE-Integration von Cloud-basierten Static Application Security Testing Tools (SAST-Tools) erwartet werden. Wir haben eine MAGPIEBRIDGE-basierte IDE-Unterstützung für Amazon CodeGuru Reviewer implementiert und die Benutzerfreundlichkeit mit Entwicklern getestet. Eine unserer Erkenntnisse ist, dass Entwickler erwarten, dass sich die IDE-Unterstützung für Cloud-basierte SAST-Tools ähnlich verhält wie die leichtgewichtigen statischen Analysewerkzeuge, mit denen sie vertraut sind. Die Wiederverwendung derselben visuellen Komponenten für das SAST-Tool, die auch von leichtgewichtigen statischen Analysewerkzeuge verwendet werden, führte jedoch zu Verwirrung. Die Entwickler benötigen klarere visuelle Hinweise in der IDE, um die asynchrone Natur der Cloud-basierten Analysen zu verstehen. Unsere Studie zeigt auch, dass die IDE-Integration die Entwickler im Vergleich zur bestehenden browserbasierten Integration deutlich mehr zur Interaktion mit dem SAST-Tool ermutigt.

In dieser Dissertation zeigen wir, dass die statische Taint-Analyse für die reale Welt verfeinert werden muss und dass sie verbessert werden kann, indem die oben genannten praktischen Probleme angegangen werden.

Acknowledgement

I would like to specially thank my husband Jonas Manuel who encouraged me to pursue a PhD. He always believes in me and has always been supportive no matter the good and bad days during my PhD journey. I would also like to thank my PhD advisor Eric Bodden, for introducing me to the field of program analysis, for his continued support of my research and generous feedback. I am also very grateful for his support in letting me intern in the industry, which led to fruitful publications. Through my internships I could work with some amazing researchers: Julian Dolby from IBM research, Martin Schäf and Daniel Sanchez from Amazon. Julian Dolby was not just a collaborator, but also became a mentor for me. I am very thankful for his continued support even after our collaboration on the MAGPIEBRIDGE work was completed and all the discussions about program analysis research we had, which were very inspiring and helpful. I also want to thank Martin Schäf for challenging me with my internship project. His support enabled me to interview software developers at Amazon to understand how to build better program analysis tools for them. This experience was extremely valuable and changed my ways of thinking.

I would also like to thank my other coauthors: Johannes Späth, Goran Piskachev, Felix Pauck, Manuel Benz, Ben Hermann, Martin Mory, Ivan Pashchenko, Erik Krogh Kristensen, Nataniel P. Borges Jr. and Fabio Massacci. In particular, thanks to Johannes to be my first collaborator and helping me a lot at the start of my research career. Special thanks to Felix Pauck, Goran Piskachev, Manuel Benz, Ivan Pashchenko, Ben Hermann, Martin Mory and Fabio Massacci for their continued support and advice in the TAINTBENCH work, which spanned over two years. I am also very thankful to all my colleagues and ex-colleagues, especially Goran Piskachev, Martin Mory, Vera Meyer, Jürgen Maniera, Manuel Benz, Andreas Dann and Lisa Nguyen Quang Do. During my PhD, I was also lucky to supervise a few talented students: Markus Schmidt and Christian Brüggemann helped me in the TAINTBENCH work. Fynn Hauptmeier contributed to the extension of COVA. Many thanks to them.

I also want to thank my dissertation committee for their availability for my defense: Eric Bodden, Julian Dolby, Ben Hermann, Juraj Somorovsky and Simon Oberthür. Special thanks to the three reviewers of my dissertation: Eric Bodden, Julian Dolby and Ben Hermann.

This dissertation would not have been possible without the funding from the state of North Rhine-Westphalia in Germany. Special thanks to the coordinator of the research training group NERD.NRW Martin Degeling.

Lastly, I would like to thank my parents for supporting me to study in Germany, my sister for her emotional support and encouragements.

Contents

Acronyms	1
1 Introduction	3
1.1 Problem Statement	3
1.2 Common Benchmarks Are Small and Incomplete	4
1.3 Real-World Issues Often of Limited Interest	5
1.4 Little Adoption by Developers	5
1.5 Outline and Publication Details	6
2 Real-World Malware Benchmarking of Android Taint Analyses	9
2.1 Terminology	10
2.2 Related Work	11
2.2.1 Android Taint Analysis Tools	11
2.2.2 Existing Benchmark Suites	12
2.3 Benchmark Construction Criteria	13
2.4 The TAINTBENCH Framework	15
2.4.1 Part 1—Construction	16
2.4.2 Part 2—Evaluation	17
2.4.3 Part 3—Inspection	19
2.5 Real-World Benchmarking	21
2.5.1 Part 1—Construction of the TAINTBENCH Suite	21
2.5.2 Part 2—Evaluation with the TAINTBENCH Suite	29
2.5.3 Part 3—Inspection of the Analysis Results	36
2.6 Threats to Validity	39
2.7 Conclusion	39
3 GenCG: A General Approach to Modeling Java Framework Behaviors	41
3.1 A Motivating Example	42
3.2 Background	44
3.2.1 Entry Points and Lifecycle Modeling	44
3.2.2 Inter-Component Communication	45
3.2.3 Analysis of Library Methods	45
3.2.4 Construction of Application-only Call Graphs with AVERROES	46
3.3 Existing Problems with AVERROES’s Model	49
3.4 The GENCG Approach	51
3.4.1 Main Improvements	52
3.4.2 Sound and Precise Call Graph	55

3.4.3	Supporting Detection of ICC Leaks	58
3.5	Evaluation of GENCG	61
3.6	Application of GENCG on the Spring Framework	67
3.6.1	Handling Annotated Entry Points	67
3.6.2	Handling Bean Autowiring	69
3.6.3	Implementation Details	71
3.6.4	Evaluation with CGBENCH	71
3.7	Related Work	76
3.8	Limitations and Threats to Validity	77
3.9	Conclusion	77
4	Towards Path-Sensitive Analysis with COVA	79
4.1	A Motivating Example	80
4.2	Non-Distributivity	82
4.3	The Inter-procedural Constraint Analysis in COVA	82
4.3.1	The VASCO Framework	82
4.3.2	Analysis Domain	85
4.3.3	Flow Functions of the Taint Domain	85
4.3.4	Flow Functions of the Constraint Domain	88
4.3.5	Termination	91
4.4	Implementation	91
4.5	Evaluation of COVA	92
4.6	COVA-assisted Qualitative Analysis of Android Taint-Analysis Results	94
4.7	Usage of COVA for Targeted Testing Input Generation	103
4.7.1	Android Testing Frameworks	103
4.7.2	Extended COVA	104
4.8	Threats to Validity	107
4.9	Related Work	107
4.10	Conclusion	108
5	Integrating Static Analyses into IDEs with MagpieBridge	111
5.1	Related Work	113
5.2	Approach	115
5.2.1	The MAGPIEBRIDGE Workflow	116
5.2.2	The MAGPIEBRIDGE System	123
5.3	Integration of Existing Static Tools	127
5.3.1	Diagnostics	127
5.3.2	Code Lenses	128
5.3.3	Hovers	130
5.3.4	Repairs	130
5.4	More Tool Integrations	131
5.5	Conclusion	134
6	IDE Support for Cloud-based SAST Tools	135
6.1	Background	136
6.2	User Interviews	137
6.2.1	Methodology	137
6.2.2	Result of the User Interviews	138
6.3	Prototyping	141
6.4	Second-round Interviews	144

6.5	Usability Testing	145
6.5.1	Methodology	145
6.5.2	Quantitative Analysis	146
6.5.3	Qualitative Analysis	151
6.6	Threats To Validity	154
6.7	Related Work	154
6.8	Conclusion	155
7	Conclusion and Future Work	157
	Bibliography	160
A	Supplementary Material of Chapter 2	181
A.1	Usability Test	181
A.1.1	Participants	181
A.1.2	Study Design	181
A.1.3	Data Collection	183
A.1.4	Results	183
A.2	Figures	185
B	Supplementary Material of Chapter 3	187
B.1	Figures	187
B.2	Tables	190
C	Supplementary Material of Chapter 5	199
C.1	Comparison Between MAGPIEBRIDGE-Based Approach and Plugin-Based Approach	199
C.1.1	Comparison Between MAGPIEBRIDGE-Based CogniCrypt and CogniCrypt Eclipse Plugin	199
C.1.2	Comparison to Other Plugin-Based Approaches	201
D	Supplementary Material of Chapter 6	205
D.1	Script For User Interviews	205
D.2	Codes	206
D.3	Survey For Usability Tests	209

Acronyms

API	Application Programming Interface
APK	Android Application Package
App	Application
CFG	Control-Flow Graph
CG	Call Graph
CHA	Class Hierarchy Analysis
CI/CD	Continuous Integration/Continuous Delivery
DB	DroidBench
DTA	Declared Type Analysis
FN	False Negative
FP	False Positive
GUI	Graphical User Interface
ICFG	Inter-procedural Control-Flow Graph
ICC	Inter-Component Communication
IDE	Integrated Development Environment
I/O	Input/Output
IR	Intermediate Representation
LSP	Language Server Protocol
PAG	Pointer Assignment Graph
RTA	Rapid Type Analysis
SAST	Static Application Security Testing
SASP	Static Analysis Server Protocol
SARIF	Static Analysis Results Interchange Format

SDK Software Development Kit

SMT Satisfiability Modulo Theories

SSA Static Single Assignment

TAF Taint Analysis Benchmark Format

TB TaintBench

TN True Negative

TP True Positive

UI User Interface

VSC Visual Studio Code

VTA Variable Type Analysis

Introduction

Security breaches happen on a daily basis and are a serious threat to enterprises. As reported in IBM’s 2019 Cost of a Data Breach Report, the average total cost of a data breach has reached \$3.91 million [Sec19]. Just in 2019, there were 1473 data breaches in the US and more than 164 million records stolen [Sta20]. In May 2021, the fuel pipeline operator Colonial Pipeline had to pay 75 Bitcoins (nearly \$5 million) to recover its data stolen by a ransomware [MDSK21]. 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 about its usefulness has arisen [Reu21][Ker21]. A significant amount of money could have been saved and damage to company image could have been avoided if such threats had been detected at an early stage. Evidently, this is not always the case in practice, despite a wealth of work on a variety of prevention techniques. In this dissertation, we focus on static program analysis. It analyzes programs without executing them and its application ranges from discovering undefined behaviors (e.g., null pointer dereferences) to detecting security vulnerabilities (e.g., SQL injections) to preventing malware infection (e.g., data theft). Static taint analysis, in particular, is able to detect a wide range of security vulnerabilities and malicious behaviors. 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.

1.1 Problem Statement

In the past, many static taint analysis tools have been created both in industry and academia, but only few of them have found widespread use in industry. There are two countervailing reasons for this: scalability on the one hand, and good results on the other. Static tools are considered to produce good results if they yield high precision (i.e., a low rate of false positives) and recall (i.e., a low rate of false negatives) in benchmarking. However, producing good results and being scalable at the same time is very challenging in terms of real-world applications (apps) because of both their size and complexity. Many taint analysis tools have assessed their precision and recall on micro benchmark apps, hence optimized to achieve good results on them [PBW18, QWR18]. In contrast to evaluation on micro-benchmark apps, evaluations on real-world apps are uncommon due to the lack of established benchmarks. This leads to three problems that have hampered real-world applicability of these tools:

- **Common benchmarks are small and incomplete:** Micro benchmark apps rarely use the full range of real platforms, leading analysis to miss issues due to incomplete modeling [BGC15].
- **Real-world issues often of limited interest:** Analysis tools ignoring real-world scenarios like code conditioned for platform versions for the sake of scalability [LBP⁺17]. This can produce warnings that a given user will not care about or are even unrealizable, i.e. can never happen at runtime [ARHB15].
- **Little adoption by developers:** Micro benchmark apps tend to be simple, but real programs are sprawling and often include intricate issues. Developers need tool support to understand these difficult real-world issues. Static analysis tools produced in academia have been mostly restricted to automated experiments where the analyses are run as command-line tools [ARF⁺14, OMJ⁺13, LBB⁺15, WROR14], paying little to no attention to usability aspects on the side of developers. There has been less integration of these analyses in IDEs commonly used by developers.

In this dissertation, we tackle these three problems to improve real-world applicability of static taint analysis. We focus on Android in this dissertation, but similar issues arise in other settings [SAP⁺11, WR13, AFK⁺20, JSMB13]. In the following, we introduce our contributions addressing each problem.

1.2 Common Benchmarks Are Small and Incomplete

To address the first problem, the first step is to obtain more realistic benchmarks. As a first contribution, presented in Chapter 2 of this dissertation, we constructed a real-world malware benchmark suite for Android taint analysis. The benchmark suite, called TAINTBENCH, is the first real-world suite in this area with a documented baseline—a subset of the ground truth. It can only be a subset, since the problem (i.e., identifying all security issues in real-world applications) is intractable in general. Along with the suite, we developed a set of tools which allows a faster benchmark-suite construction, a reproducible evaluation of static taint analysis tools on this suite and easier triaging of static findings. Using TAINTBENCH we evaluated popular static taint analysis tools. Our results show that these tools have much lower precision and recall 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 flows), the majority of malicious taint flows in TAINTBENCH remain undetected. We further investigated FLOWDROID, which produced the best result on TAINTBENCH. Our investigation reveals that 35% of the taint flows in TAINTBENCH could not be detected by FLOWDROID due to relevant methods missing from the call graphs. Clearly, one needs to construct better call graphs.

Our next contribution, presented in Chapter 3, is a new way to build a call graph that is more complete, but still tractable to build. Call graphs are major building blocks for interprocedural static analyses, which are challenging to build for modern framework-based applications. To be scalable, most static analysis tools model rather than analyzing frameworks [GKP⁺15, WROR14, ARF⁺14, SAP⁺11]. FLOWDROID, for instance, models the behaviors of the Android framework by constructing a dummy main method which simulates the lifecycle of each Android component and starts the analysis from there. However, this precise modeling is hard to maintain, since every year a new version of Android comes out with new APIs, which introduces new behaviors to the framework. It is also impractical to do so for every framework. Popular frameworks for Java enterprise applications such as Spring make use of annotations

which guides the framework to decide which code should be executed through reflective calls. To be useful, a call graph must somehow overcome these frameworks: analyzing them is impractical, but modeling is too. In this dissertation, we propose a general approach to modeling Java frameworks. Our approach GENCG is not limited to any framework or analysis tool, and therefore, highly reusable. In GENCG, we developed AVERROES-GENCG—an improvement of AVERROES [AL13] that generates a placeholder library for a given Android/Java framework. This generated placeholder library can be used as a replacement of the original framework by popular call graph construction algorithms (e.g., VTA, RTA, Spark [SHR⁺00, LH03]) and further client analyses. The framework behavior is modeled in the placeholder library code and will be reflected in the constructed call graphs. While a generic approximation can be noisy, we show our carefully-constructed one does well. We demonstrate its generalization with two frameworks—Android and Spring. Experiments on Android with a client taint analysis show that our approach produces more complete call graphs than the original analysis. As a result, both precision and recall of the client analysis on TAINTBENCH are improved.

1.3 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 developers don’t care about or even false-positives. 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. This approach is presented in Chapter 4. We implemented a tool called COVA that combines data-flow analysis with SMT solving. 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 taint flows from a large set of real-world commercial Android apps. In this study, we could identify how these real-world taint flows are conditioned on environment settings (e.g., platform versions), user interactions (e.g., button clicks) and I/O operations (e.g., file reading and writing). Such qualitative data about the circumstances under which taint flows may actually occur can be used to guide dynamic approaches which confirm static findings. We further extended COVA to not only compute the path constraint of a specific statement, but also generate concrete user inputs that are required to execute this statement at runtime. Experiments with a small set of apps from F-Droid show the feasibility of using this approach to generate valid user inputs for testing randomly selected statements, which is a step towards dynamic validation of static findings.

1.4 Little Adoption by Developers

As many recent studies show, if static analysis tools do not yield actionable results, or if they do not report them in a way that developers can understand and fit into developers’ workflow, then the tools will not be adopted [JSMB13, CB16, DAL⁺17a]. While static taint analysis tools can help developers find security vulnerabilities in their code, there has been less adoption of such analyses in tools commonly used by developers, i.e., in interactive development environments (IDEs) such as Eclipse, IntelliJ, Android Studio and Visual Studio Code. Although IDEs are

the ideal reporting location wanted by developers for static analysis. Even with the existence of IDE integration, tools like DroidSafe [GKP⁺15], Cheetah [DAL⁺17b] and IBM Security AppScan [IBM07] mostly target one specific IDE, since a substantial engineering effort is involved to integrate a specific analysis for a specific language into a specific IDE. Given that degree of needed effort, the sheer variety of popular tools and potentially-useful analyses makes it impractical to build every combination. To foster a better adoption of these tools by developers, researchers need ways to bring tools into IDEs more easily and quickly. As a fourth contribution of this dissertation, we created a general approach to integrating static analyses into IDEs and Editors—MAGPIEBRIDGE. To show MAGPIEBRIDGE’s generalizability, we integrated a few analyses from academia—FlowDroid [ARF⁺14], CogniCrypt [KNR⁺17] and Ariadne [DSAR18], and two analyses from industry—Facebook Infer [Fac15] and Amazon CodeGuru Reviewer [Ser20] into IDEs. This work is presented in Chapter 5.

As a last contribution presented in Chapter 6 we conducted a multiple-staged user study with software engineers at Amazon Web Services (AWS) to explore how IDE support for a purely cloud-based static analysis, that is typically used in continuous integration (CI) or continuous delivery (CD), should be designed to meet the expectations of developers. In this study, we built a prototype of the IDE support for a cloud-based Static Application Security Testing (SAST) tool—Amazon CodeGuru Reviewer. We evaluated this prototype with 32 software engineers at AWS with a usability test. We share challenges and lessons learned in the exploration that can be beneficial for others who wish to build such IDE support.

1.5 Outline and Publication Details

Most parts of the dissertation are based on publications I authored over the course of my PhD. The list below states the outline of this dissertation and details publications each chapter is based on:

- Chapter 2 presents the TAINTBENCH work on constructing real-world benchmarks and evaluation of Android taint analyses using the constructed benchmark suite. A paper about this work is accepted by the Empirical Software Engineering (EMSE) journal and currently in the publishing process. In addition to some parts of the work that are presented in the paper, an in-depth case study of FLOWDROID’s false negatives is also introduced in this chapter.

Linghui Luo, Felix Pauck, Goran Piskachev, Manuel Benz, Ivan Pashchenko, Martin Mory, Eric Bodden, Ben Hermann, Fabio Massacci. TaintBench: Automatic Real-World Malware Benchmarking of Android Taint Analyses. Empirical Software Engineering (EMSE), 2021 [LPP⁺21].

- Chapter 3 presents the details about the GENCG work on call graph construction and evaluations on both Android and Spring frameworks. An early conception of this work is published in the following short paper and won the second place at the ACM Student Research Competition:

Linghui Luo. A General Approach to Modeling Java Framework Behaviors. The ACM Student Research Competition at the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2021 [Luo21].

- Chapter 4 introduces details of the tool COVA and the qualitative analysis of Android Taint-Analysis results from the following publication:

Linghui Luo, Eric Bodden, Johannes Späth. A Qualitative Analysis of Android Taint-Analysis Results. 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019 [LBS19].

- Chapter 5 presents details about MAGPIEBRIDGE, its development and applications since its first appearance in the following publication:

Linghui Luo, Julian Dolby, Eric Bodden. MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors, 33rd European Conference on Object-Oriented Programming (ECOOP), 2019 [LDB19].

- Chapter 6 introduces the user study we conducted with software engineers at Amazon Web Services published in the following paper:

Linghui Luo, Martin Schäfer, Daniel Sanchez, Eric Bodden. IDE Support for Cloud-Based Static Analyses, the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2021 [LSSB21a]

- Chapter 7 concludes the dissertation and discusses future research directions.

1.5 OUTLINE AND PUBLICATION DETAILS

Real-World Malware Benchmarking of Android Taint Analyses

Many mobile applications are granted to access and process sensitive data such as contact lists or the user’s GPS location, which require protection against abuse. In case of Android, the most popular mobile operating system [Sta19], it is crucial to protect users’ privacy. Hence, in case of most marketplaces, such as Google Play Store, any app trying to enter must be reviewed. Numerous malware detection mechanisms have been developed that can be used for automating the review process [WROR14, ARF⁺14, EGC⁺14, GKP⁺15, GS17, YS17]. Nonetheless, frequent news reports of malware apps bypassing such mechanisms and being accepted by marketplaces show that this process sometimes fails [Son20, Mic20a].

One such malware detection mechanism, static taint analysis, in particular, is able to detect security threats, e. g., data leaks, before they are actually exploited. To show the effectiveness of a new taint analysis prototype, tool authors are expected to evaluate it empirically and compare to previous approaches. Fortunately, there exist a few well-established benchmark suites for this purpose, e. g., DROIDBENCH [ARF⁺14], SECURIBENCH [LL05] and ICC-BENCH [WROR14]. However, all these benchmark suites are sets of micro *benchmark apps*—small programs which are artificially constructed for benchmarking purposes only. Evaluations of Android taint analyses frequently use such micro benchmark apps as representatives of real-world apps [CHN16, BS18, PZ19, ZTD19, WROR14, ARF⁺14].

In contrast to evaluations on micro benchmark suites, evaluations on real-world apps are uncommon and—due to missing or undocumented details—usually not reproducible. For instance, the authors of DROIDSAFE [GKP⁺15] evaluated both DROIDSAFE and FLOWDROID [ARF⁺14] on 24 real-world Android malware apps. However, information about the malicious taint flows in these apps is only documented in form of the types of sources and sinks (e. g., source type is location and sink type is network). Missing details about code locations related to these flows makes it impossible to reproduce the results, which hinders the measurement of research progress. Additionally, documentation of the *ground truth* for a set of real-world apps rarely exists, since the only tools which could be used as oracles to determine these flows are the tools to be evaluated. Thus, the associated evaluation results often come unchecked and only comprise an enumeration of countable findings without guarantees that the findings actually represent feasible taint flows [AKG⁺15, ARF⁺14].

But security threats that plague our world of course happen in real-world apps, so the lack of high-quality benchmarks—publicly available real-world benchmark suites with a well-documented ground truth—hinders progress in vital taint analysis research, even as existing tools seem to work less well in the real world [WWZ⁺20, RM20, LBS19]. The work presented in this chapter addresses this problem by constructing a real-world benchmark suite, specifically

for benchmarking Android taint analysis.

The outline of the remaining sections in this chapter is as follows: we first introduce terminology required to understand our work in Section 2.1 and related work in Section 2.2. We then discuss criteria for real-world benchmark construction in Section 2.3. To ease the benchmark construction, evaluation of Android taint analyses and inspection of the analysis results, we developed the TAINTBENCH framework—a collection of tools. This framework is introduced in Section 2.4. Next, we explain details about how this framework supported us to construct the TAINTBENCH suite, evaluate popular Android taint analyses tools using this suite and inspect shortcomings of the tools in Section 2.5. We present an in-depth case study of FLOWDROID regarding unexpected analysis outcomes and false negatives in Section 2.5.3, which show directions for improvement and further research, motivating the work introduced in Chapter 3 of this dissertation. In the end, we discuss the limitations in Section 2.6 and conclude this chapter in Section 2.7.

2.1 Terminology

Static Taint Analysis Static taint analysis is a type of data-flow analysis that tracks tainted data flows through a program and detects data flows between sources and sinks. For detecting data leaks in Android apps, sources are usually APIs that read sensitive private information (e.g., phone number, contact list, etc.), while sinks are APIs that can send data to untrusted external parties (e.g., HTTP requests), as in the example shown in Listing 2.1. For detecting injection vulnerabilities (e.g., SQL injection, Command injection, etc.), APIs that read untrusted user inputs are usually considered as sources, while APIs that execute a command or access databases are the sinks. When configured with sufficient sources and sinks, static taint analysis tools can detect the majority of SANS/CWE top 25 most dangerous software weaknesses [PDB19] and common malware behaviors (e.g., financial fraud, personal information theft, etc.) [Ras16].

```

1 public class WrehifsdkjsActivity extends Activity {
2     public void run() {
3         Message msg = new Message();
4         String line1Number = ((TelephonyManager)
5             this.getSystemService("phone")).getLine1Number(); // source
6         // ...
7         String result = doPost("https://malicious.com", "t=" + line1Number + "&app=Wrehifsdkjs");
8         // ...
9     }
10    public String doPost(String url, String params) throws Exception {
11        HttpPost method = new HttpPost(url);
12        DefaultHttpClient client = new DefaultHttpClient();
13        StringEntity paramEntity = new StringEntity(params, "UTF-8");
14        paramEntity.setChunked(false);
15        paramEntity.setContentType("application/x-www-form-urlencoded");
16        method.setEntity(paramEntity);
17        HttpResponse response = client.execute(method); // sink
18        // ...
19    }
20 }

```

Listing 2.1: A data leak example taken from the malware app *exprespam* in TAINTBENCH.

Taint Flows The data flows between a pair of source and sink are called taint flows. Multiple data-flow paths might result in the same taint flow as the example in Listing 2.2 shows. In

evaluations, a taint flow is usually counted as detected by a static analysis tool once a connection consisting of one or more data-flow paths between the associated source and sink is found.

```

1 void onCreate(){
2     A a = new A(); B b = new B();
3     a.f = contact.read(); // source
4     if (a.f.startsWith("0"))
5         b.f1 = a.f; // on data-flow path 1
6     else
7         b.f2 = a.f; // on data-flow path 2
8     leak(b); // sink
9 }
```

Listing 2.2: Two data-flow paths result in one taint flow.

```

1 void onCreate() {
2
3     String[] arr = new String[3];
4     arr[0] = "";
5     arr[1] = contact.read(); // source
6     leak(arr[1]); // leak, expected taint flow
7     leak(arr[0]); // no leak, unexpected taint flow
8
9 }
```

Listing 2.3: Expected vs. unexpected taint flow.

Expected vs. Unexpected Taint Flows The ground truth defined in a micro benchmark suite for benchmarking taint analysis is usually a set of expected and unexpected taint flows. *Expected taint flows* are real security issues, and *unexpected taint flows* specify false-positive cases which an imprecise analysis tool might still report. Established benchmark suites often use unexpected cases to assess the precision of a tool (e.g., *ArrayAccess1* in DROIDBENCH). Consider the example in Listing 2.3 from a benchmark app, and assume the ground truth defines an expected taint flow from line 5 to line 6, as well as an unexpected flow from line 5 to line 7. The expected flow in this example specifies a true data leak, while the unexpected flow specifies a case for which tools might produce false-positive warnings (e.g., a tool overapproximates for arrays and taints the whole array once a tainted value is written into an array). Once a taint analysis tool finds the expected taint flow while analyzing this benchmark app, it is counted as a *true positive* (TP). A missed expected taint flow is counted as a *false negative* (FN). Consequently, found and missed unexpected taint flows are counted as *false positives* (FP) and *true negatives* (TN) respectively.

Benchmarking Metrics We call the combination of a benchmark app and an expected/unexpected taint flow in it a *benchmark case*. Based on the countings of TP, TN, FP and FN with regard to the benchmark cases defined in a suite, the benchmarking outcome is then usually evaluated with respect to accuracy in terms of three metrics: precision, recall, and F-measure. These metrics are computed with the following equations:

$$Precision = \frac{TP}{TP + FP} \quad , \quad Recall = \frac{TP}{TP + FN} \quad , \quad F - measure = \frac{2Precision \cdot Recall}{Precision + Recall}$$

Additionally, the analysis time is recorded in most evaluations to argue about a tool’s scalability.

2.2 Related Work

In the area of Android taint analysis, there exist many static [ARF⁺14, WROR14, GKP⁺15, LBB⁺15, BLYW17], dynamic [EGC⁺14], and hybrid [BKKL⁺20, PW19] analysis tools, as well as benchmark suites [ARF⁺14, WROR14, MR17]. We highlight the most prominent static analysis tools and benchmark suites with respect to taint analyses.

2.2.1 Android Taint Analysis Tools

FLOWDROID [ARF⁺14], AMANDROID [WROR14], ICC_{TA} [LBB⁺15] and DROIDSAFE [GKP⁺15] are the most cited static Android taint analysis tools. AMANDROID, FLOWDROID and ICC_{TA}

use configurable lists of sources and sinks to be considered during analysis. SUSI [RAB14] is a machine-learning approach developed to automatically create such lists by inspecting the Android APIs. More comprehensive or precise lists were produced in more recent research [PDB19]. DROIDSAFE’s list of sources and sinks is hard-coded in its source code, which makes it hard to adapt for real-world apps. While DROIDSAFE and ICCTA are not maintained anymore, FLOWDROID and AMANDROID still appear to receive frequent updates [Ama18, Flo19]. Furthermore, all tools support different features and sensitivities that influence their precision and soundness. FLOWDROID and AMANDROID, for example, are context-, flow-, field-, object-sensitive and lifecycle-aware. Only ICCTA and AMANDROID support the analysis of inter-component communication (ICC). None of the tools is path-sensitive due to scalability drawbacks and their static nature. Table 2.1 shows an overview of the main characteristics of these tools. Evaluations of the abilities of each tool can be found in previous studies [QWR18, PBW18].

Table 2.1: Overview of the main characteristics of relevant static taint analysis tools.

Tool	configurable sources & sinks	actively maintained	ICC	context- sensitive	flow- sensitive	field- sensitive	object- sensitive	path- sensitive	lifecycle- aware
FLOWDROID	✓	✓	✗	✓	✓	✓	✓	✗	✓
AMANDROID	✓	✓	✓	✓	✓	✓	✓	✗	✓
ICCTA	✓	✗	✓	✓	✓	✓	✓	✗	✓
DROIDSAFE	✗	✗	✗	✓*	✗	✓	✓	✗	✓

*: static method only

2.2.2 Existing Benchmark Suites

The most cited and hence most established benchmark suite in this field of research is DROIDBENCH [ARF⁺14]. DROIDBENCH is a collection of artificial apps that forms a micro benchmark suite. Its ground-truth description can be found in code comments in the source code associated with each benchmark app. The up-to-date version 3.0 [Dro16] comprises 190 apps with benchmark cases in 18 different categories related to the features and sensitivities exploited. Subsets, variants and extensions of DROIDBENCH have been used to evaluate certain features or more specialized taint analysis tools [WROR14, BLYW17]. ICC-BENCH [WROR14] comprises benchmark cases to evaluate the abilities of analyses to handle inter-component communication. A recent suite contributed by us¹ is DROIDMACROBENCH [BKKL⁺20]—a collection of 12 real-world commercial Android apps with annotated taint flows reported by FLOWDROID. However, due to the high complexity of commercial apps, we only labeled the taint flows as feasible (i.e., it is possible for data to flow from a given source to a given sink) or infeasible without characterizing or giving details about the flows. For example, it remains unclear whether the tainted data is sensitive. Thus, regarding security aspects, many feasible labeled taint flows might not be real security threats. Moreover, because DROIDMACROBENCH comprises closed-source apps, we could not legally make the suite publicly available as open source. Due to these two limitations, it cannot be used as publicly accessible and comparable proving ground truth for taint analyses.

GHERA [MR17] is a repository of micro benchmark apps sorted into different categories of Android vulnerabilities, sometimes including taint flows. As of January 2021, GHERA contains 8 categories with 60 vulnerabilities where each one contains three apps: a benign, a malicious and a secure one. The ground truth is documented in a text file that hold a natural language description of the vulnerability within the specific app. However, not all of GHERA benchmark apps are suitable for benchmarking taint analysis tools. For example, the apps in the Crypto

¹I am a co-author of the paper and contributed to this suite.

catalog of GHERA contain cryptographic misuses that require tpestate analysis rather than taint analysis. Table 2.2 summarizes DROIDBENCH, DROIDMACROBENCH, and GHERA.

Table 2.2: Overview of the main characteristics of relevant Android benchmark suites.

Benchmark Suite	Real-world apps	Number of apps	Open source	Ground Truth
DROIDBENCH	✗	190	✓	Comments in source code
DROIDMACROBENCH	✓	12	✗	Jimple code labeled as in-/feasible
Ghera	✗	180	✓	README file

Pauck et al. [PBW18] proposed REPRODROID to refine, execute, and evaluate on benchmark suites. Among other suites they refined DROIDBENCH such that each benchmark app now comes with a precisely defined, machine-readable ground truth. We include and extend REPRODROID for enabling automatic evaluations in our TAINTBENCH framework as described in next section.

2.3 Benchmark Construction Criteria

This section describes the criteria we used to construct a real-world malware benchmark suite for Android taint analysis. Since there exists no widely accepted real-world benchmark suite for this purpose, nor criteria for establishing such a suite, we used and recommend the following three criteria for real-world benchmark suite construction:

(I) Ground-Truth Documentation Mitra et al. proposed the “*Well Documented*” benchmark characteristic—benchmarks should be accompanied by relevant documentation [MR17]. In context of our work, each benchmark app should come with a documentation of the expected and unexpected taint flows for benchmarking. Such documentation was provided by DROIDBENCH, however, only in form of code comments that mostly hold natural language descriptions. It lacks information about the exact code locations of the taint flows. Pauck et al. [PBW18] pointed out that such documentation could lead to incorrect evaluation of analysis results. Thus, on top of the Well Documented characteristic, the taint flows of each benchmark app should be documented in a standard machine-readable format such as XML or JSON. It should contain both high-level textual information which describes the purpose of each taint flow (as in DROIDBENCH’s code comments) and exact code locations of the source, the sink, and the intermediate statements of each flow. Furthermore, since different taint analysis tools may use different intermediate representations (IRs) the format must support an encoding of code locations that can be converted into arbitrary IR code locations.

However, to create such a ground truth is difficult in case of real-world apps and impossible in general. This is particularly true for taint flows that both human analysts and tools cannot detect. Thus, *our goal* here is to create a *baseline definition* (i.e., a subset of all expected and unexpected taint flows) for each benchmark app. Regarding expected taint flows, we focus on those flows which are not only *feasible* (i.e., it is possible for data to flow from a given source to a given sink) but also *critical* under security aspects, e.g., leaking sensitive information. Our work aims to serve as a starting point towards a solid real-world benchmark suite for Android taint analysis. The facilities we put in place should allow (and possibly foster) extension and improvement of the suite.

(II) Representativeness Nguyen Quang Do et al. recommended representativeness with respect to the target domain of the evaluated tool or analysis as an important aspect for benchmark selection [DEB16]. In this work, we choose to focus on Android apps *identified*

as malware. There are several reasons for this decision. First, some available Android malware datasets come with descriptions of the malicious behaviors, which are unavailable for open-source datasets (e.g., F-Droid [F-D20]) or commercial applications. Such descriptions accelerate the manual inspection process, for example, when faced with the manual task of separating true from false findings produced by automated tools, since the descriptions provide hints of the malicious behavior. Labels such as malware families or types are insufficient to drive this process. Second, well-known Android malware datasets have often been used in evaluations in scientific papers [WL16, ZZD⁺12, RCJ13, HZT⁺14] including Android taint analysis approaches [HDMD15, YQL⁺16]. Last but not least, Android malware is less likely to cause licensing issues. A benchmark suite must be open-source and publicly available, and thus cannot legally include commercial apps. Because of including closed-source, commercial apps in the DROIDMACROBENCH suite, the authors could not make the suite publicly available [BKKL⁺20]. Representativeness in our work consists of two aspects:

- the expected taint flows in the benchmark suite should be *representative* of taint flows that address static-analysis challenges. These challenges are commonly agreed on by the community such as field-sensitivity or the necessity to model implicit control flows through the application’s lifecycle. A good example is DROIDBENCH, which groups its benchmark cases into 18 categories based on such challenges, e.g., aliasing, callbacks and reflection.
- the benchmark apps should be *representative* of the dataset it is sampled from. Reif et al. provide a tool to generate metrics for Java programs to assess representativeness during benchmark creation [REHM17]. Similarly, we define a set of metrics which are relevant for Android taint analysis benchmarking in this work. Details are introduced in Section 2.5.1.2.

(III) Human-understandable Source Code Source code availability has been widely used in previous benchmark works [BGH⁺06, PRL⁺19, MR17]. Whenever possible, the benchmark suite should provide human-understandable source code (either directly or by decompilation) in addition to compiled executables. This criterion is important for the following three reasons. First, it can help users of the benchmark suite to understand the documented taint flows. Second, it allows the inspection of potential false positives produced by automated tools. Lastly, it enables the community to do source-code level analysis such that the baseline definition can be checked, improved and extended. Considering our focus on malware, source code is naturally hard to come by. We will elaborate how we address this challenge in Section 2.5.1.

2.4 The TAINTBENCH Framework

This section introduces the TAINTBENCH framework, which we developed to

- simplify and speedup real-world benchmark suite construction for Android taint analysis (Part 1—Construction in Section 2.4.1),
- allow automatic evaluation of analysis tools (Part 2—Evaluation in Section 2.4.2),
- and support source code inspection of analysis results (Part 3—Inspection in Section 2.4.3).

Figure 2.1 gives an overview of this framework, which is structured into these three parts.

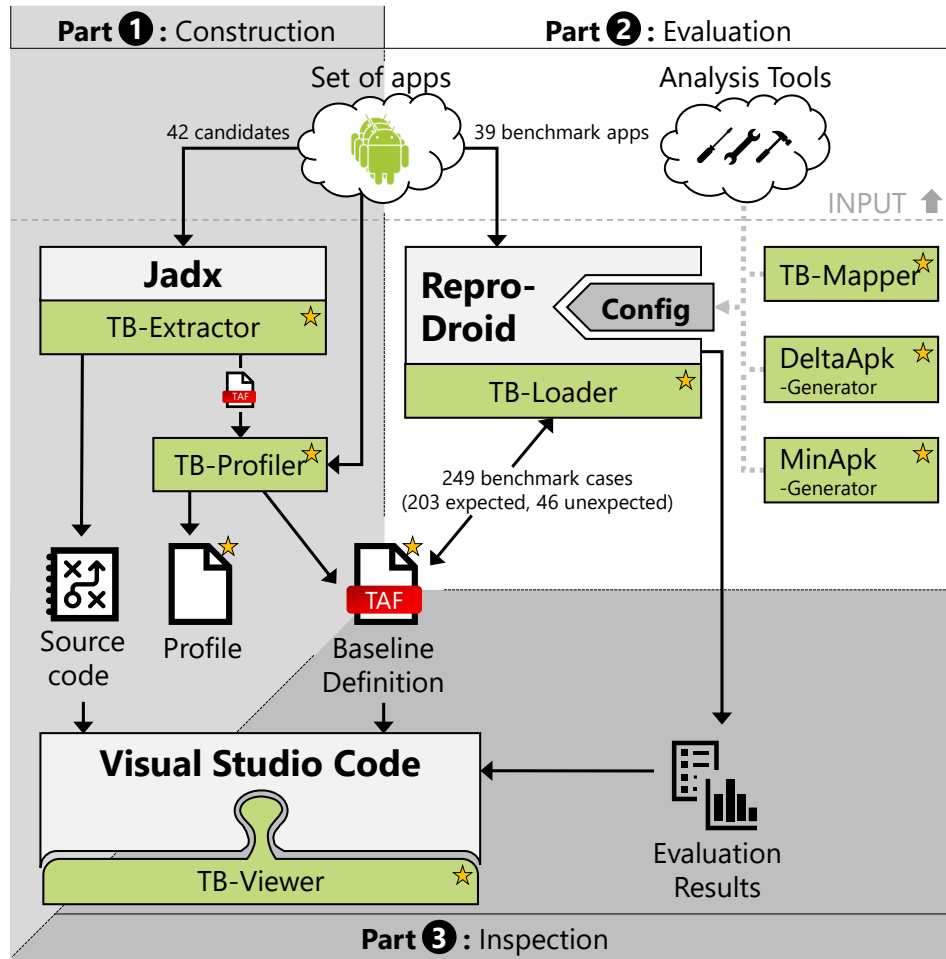


Figure 2.1: Overview of the TAINTBENCH framework. Every box refers to a tool extended or built for this framework. All elements contributed in this work are marked by a ☆-symbol.

We describe the framework using a running example depicted in Figure 2.2. The first class is an Activity component (`MainActivity`) which comprises one source (s_1) and one sink (s_5) in its `onCreate` lifecycle method. The source extracts the device’s id (`getIMEI`) which is considered sensitive data. Once it reaches the sink (`sendTextMessage`), it is leaked via an SMS. The flow from s_1 to the logging statement (s_7) should not be recognized as a leak, since only the value of the not-null check is logged. Class `Foo` contains only one method (`bar`) which contains a second taint flow from s_8 (source) to s_9 (sink).

Assume a human analyst wants to specify a baseline definition contains two expected taint flows (solid green edges) and two unexpected taint flows (solid red edges). There is also an undocumented taint flow (blue dashed edges) that can be potentially reported by analysis tool. We denote these five flows with the following notion:

$$(s_1 \rightarrow s_5), (s_8 \rightarrow s_9), (s_1 \rightarrow s_9), (s_8 \rightarrow s_5), (s_1 \rightarrow s_7)$$

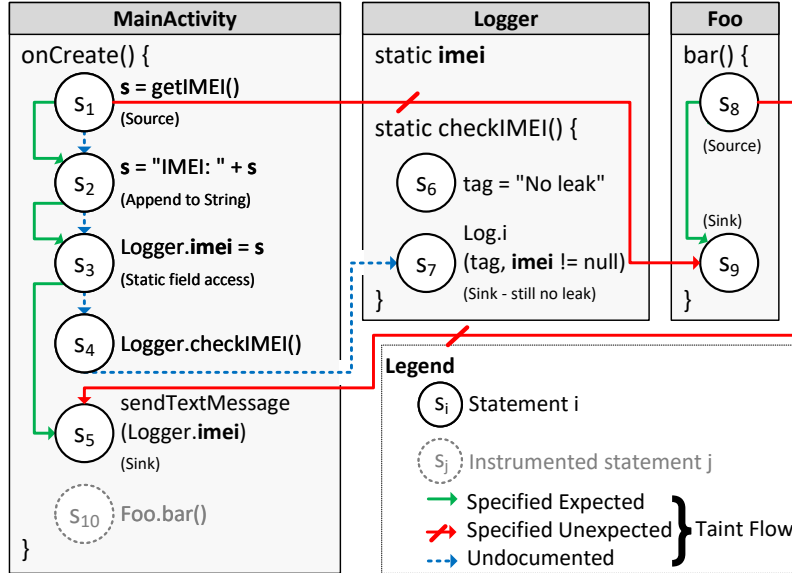


Figure 2.2: Running example (`example.apk`).

2.4.1 Part 1—Construction

In Part 1, two tools come into play. We explain them with the running example. The first tool, the JADX decompiler [DtJd20], allows us to extract source code from Android application package (APK) files. We extended JADX’s GUI by adding the TB-EXTRACTOR. It enables us to manually specify source, sink, and intermediate assignments of taint flows by selecting the relevant source code statements directly in the extended GUI. The extension also allows inspectors to add a high-level description and *attributes* (i.e., special language or framework features) to each taint flow. Once the taint flow specification is done, TB-EXTRACTOR outputs a JSON file which stores the information logged for each taint flow.

The second tool is the TB-PROFILER. It takes both the APK and the JSON file generated by TB-EXTRACTOR and outputs automatically detectable attributes which were missed or incorrectly assigned by the inspectors. For example, if there is a statement on a documented taint flow which involves reflection and the human inspectors forgot to assign the attribute **reflection**, the TB-PROFILER will detect it automatically by checking the API signatures and language features involved in the respective statements.² To avoid documenting false attributes produced by TB-EXTRACTOR, the human inspectors make the final decision if the detected attributes should be assigned or not. The JSON file derived this way can be stored as the *baseline definition* that specifies the taint flows for the associated benchmark app. The TB-PROFILER extracts other static information from the APK, such as the target platform version, a list of used permissions, sensitive API calls, etc., and stores them in a profile file.

²A list of all attributes considered is given in Table 2.6 in Section 2.5.1.2.


```

1      { "findings": [{
2        "ID": 1,
3        "isUnexpected": false,
4        "description": "This malicious flow sends IMEI in a SMS."
5        "source": {
6          "statement": "String s = getIMEI();",
7          "methodName": "onCreate",
8          "className": "MainActivity",
9          "lineNo": 1,
10         "targetName": "getIMEI",
11         "targetNo": 1,
12         "IRs": [{"type": "Jimple",
13         "IRstatement": "$r2 = virtualinvoke ..."}]},
14       "sink": {
15         "statement": "sendMessage(Logger.imei);", ...},
16       "intermediateFlows": [{
17         "ID": 1, "statement": "s = \"IMEI: \" + s;", ...},{
18         "ID": 2, "statement": "Logger.imei = s;", ...}],
19       "attributes": {
20         "staticField": true,
21         "appendToString": true},
22     }, { ... } ]...}

```

Listing 2.4: The TAF-File for the running example.

With our TB-EXTRACTOR extension, one could specify the baseline definition by selecting the relevant source code statements constructing the taint flows (illustrated as solid green edges in Figure 2.2) directly in JADX. To store the baseline definition, a JSON file conforming to our Taint Analysis Benchmark Format (TAF)³ is generated (the output of TB-EXTRACTOR). A shortened version of the TAF-File for the running example is provided in Listing 2.4. Each element of the array `findings` describes one taint flow. It can be either expected or unexpected, indicated by the attribute `isUnexpected`. In the listing, the expected taint flow ($s_1 \rightarrow s_5$) is visible. The second flow ($s_8 \rightarrow s_9$) is hidden in Line 22. An example that shows how source, sink, and intermediate flows are described with code locations, is given in Lines 5-18. If a statement contains multiple function calls, `targetName` and `targetNo` specify which function call is meant. Intermediate flows are assigned with IDs which indicate the order of their appearances in the taint flow. The attributes `staticField` and `appendToString` indicate that the tainted data flows through a statically declared variable (s_3) and is appended to a String (s_2). The `IRs`-array holds the intermediate representations (IR) associated to the statement, such as Jimple [VCG⁺99, LBLH11c]. Jimple is the IR of the analysis framework SOOT on which FLOWDROID is based, and it is supported by REPRODROID. We included Jimple in the baseline, but one could certainly fill this array with IRs from other frameworks. Jimple statements are automatically added by TB-LOADER in Part 2.

2.4.2 Part 2—Evaluation

The harness we provide to evaluate Android taint analysis tools on the TAINTBENCH suite is an extension to REPRODROID [PBW18]. REPRODROID is a configurable open-source benchmark reproduction framework to (i) refine, (ii) execute, and (iii) evaluate analysis tools on benchmark suites.

³The JSON-schema for the TAF-format can be found at <https://github.com/TaintBench/TaintBench/blob/master/TAF-schema.json>

(i) Refine REPRODROID allows to create benchmark cases via a GUI. First, a set of benchmark apps can be imported. Second, sources and sinks contained in these apps can be selected—manually or automatically by comparison to a given list of source and sink APIs (e.g., the SuSi [RAB14] list). By specifying sources and sinks, taint flows are implicitly specified too. Lastly, REPRODROID allows to categorize these implicitly defined taint flows as expected or unexpected. We adapted REPRODROID to accept our baseline definition as additional input (see TB-LOADER in Figure 2.1). Thereby, the information of the baseline definition are used to automatically select sources and sinks and categorize taint flows as expected or unexpected. Once the benchmark suite is fully setup in REPRODROID, it can be stored. Stored benchmarks can then be loaded to be executed with or without using REPRODROID’s GUI. Our extension can also be used to export tool-specific lists of the defined sources and sinks. Currently, the formats of AMANDROID and FLOWDROID are supported.

For the running example, the expected (green arrow) and unexpected (red arrow) taint flows specified in the baseline definition are:

$$(s_1 \rightarrow s_5), (s_8 \rightarrow s_9), (s_1 \rightarrow s_9), (s_8 \rightarrow s_5) \quad *$$

All taint flows are automatically converted into benchmark cases in REPRODROID.

(ii) Execute When executing an analysis tool on a benchmark suite, REPRODROID creates one AQL-Query per benchmark case. An AQL-Query is understood by REPRODROID to run an analysis tool on a benchmark app and standardize the tool’s result in AQL-format. The configuration of REPRODROID allows us to specify which analysis tools should be ran. By adapting the configuration or transforming the query according to configurable strategies, various queries can be constructed. In our comprehensive experiments we configured REPRODROID to use four analysis tools and six different strategies (see Section 2.5.2).

(iii) Evaluate To evaluate a tool on a benchmark suite, REPRODROID compares the expected and unexpected taint flows constructed on the basis of the baseline definition with the actual result computed per AQL-query.

To evaluate an analysis tool on our running example, let us assume that REPRODROID is configured to run an analysis tool and the tool result contains four flows:

$$(s_1 \rightarrow s_5), (s_8 \rightarrow s_9), (s_1 \rightarrow s_9), (s_1 \rightarrow s_7)$$

REPRODROID’s evaluation only considers the first three flows: the first two are true positives and the third one is a false positive. A flow is evaluated as true positive (resp. false positive) only if it matches a defined expected case (resp. unexpected case) (see * above). The last flow ($s_1 \rightarrow s_7$) is not specified as expected or unexpected case—it is undocumented. For such an undocumented taint flow, manual inspection by a human analyst is required to decide if it should be further documented as an expected or unexpected case. This way 100 additional taint flows were added to TAINTBENCH’s baseline definition during our evaluation introduced in Section 2.5.2. To support manual inspection, the tool TB-VIEWER was created, which will be introduced later in Section 2.4.3.

2.4.2.1 Evaluation-Support Tools

To further support empirical evaluations in the context of the TAINTBENCH framework, REPRODROID was configured with three additional novel tools.

(i) **MINAPK-GENERATOR** To reduce the complexity of TaintBench apps with respect to each benchmark case, we introduce the MINAPK-GENERATOR. The MINAPK-GENERATOR prunes the original APK and generates a *minified APK* for each taint flow defined in the baseline. Considering a taint flow, any part in the code that is not connected to the source, sink, or intermediate flows is removed. This task can be performed more efficiently than slicing the app from source to sink, since the information about intermediate flows is given. Considering the running example in Figure 2.2, the `checkIMEI()` method of class `Logger` is removed because it does not appear in the baseline definition. However, this method would be kept by an ideal forward slicing algorithm starting from s_1 , since the static field `Logger.imei` is used in the method. MINAPK-GENERATOR only keeps the static field of this class. Since lifecycle methods might be removed this way, a new analysis entry point is created. To do so, the component that is launched on app start gets selected. Calls to all methods holding sources are added to one of its lifecycle methods, e.g., a call to `Foo.bar()` (s_{10}) is added to the `onCreate()`-method of `MainActivity` in Figure 2.2. In consequence, it is ensured that the taint flow is reachable in the call graph of the minified APK. The MINAPK-GENERATOR allows one to infer insights about the reason why an actual taint flow may remain undetected by a tool (false negative).

(ii) **DELTAAPK-GENERATOR** The DELTAAPK-GENERATOR automatically generates variants of an input app in which a single predefined taint flow, specified in the baseline definition, is killed. DELTAAPK-GENERATOR can be used to check if the evaluated analysis tool has over-approximated to detect a taint flow. It is used as a preprocessor in REPRODROID. DELTAAPK-GENERATOR kills the flow from the source `getIMEI` by instrumenting an overriding assign statement. It inserts a new assign statement `s = null` directly after the statement s_1 . Hence, the tainted variable `s` is immediately sanitized in the generated *delta APK*. For a tainted variable which has primitive type, DELTAAPK-GENERATOR inserts a statement that assigns a constant value to it. This way all flows are killed from the source. A precise taint analysis tool should report the taint flow ($s_1 \rightarrow s_5$) for `example.apk`, but not for its preprocessed version created by DELTAAPK-GENERATOR. If the taint flow is still detected in the delta APK, it is a false positive.

(iii) **TB-MAPPER** TB-MAPPER lists all the sources and sinks in the baseline definition of `example.apk` and converts the detected sources and sinks into a tool specific format, e.g., a file that comprises a list of sources and sinks used by FLOWDROID.

2.4.3 Part 3—Inspection

The TB-VIEWER is the main component of Part 3, a Visual Studio Code (VSC) [Mic20b] extension using the MagpieBridge framework [LDB19] (will be introduced in Chapter 5). It is used whenever manual inspection is needed. This tool displays specified taint flows directly on the benchmark app’s source code in VSC. It allows us to interactively inspect and compare the baseline definition (Part 1) with the findings of an evaluated analysis tool (Part 2). To do so, TB-VIEWER provides four lists in a tree view as shown for the running example in Figure 2.3:

- (A) a list of expected and unexpected taint flows with data-flow paths that are specified in the baseline definition,
- (B) a list of flows which are reported by an analysis tool during evaluation,
- (C) a list of matched flows,
- (D) a list of unmatched flows.

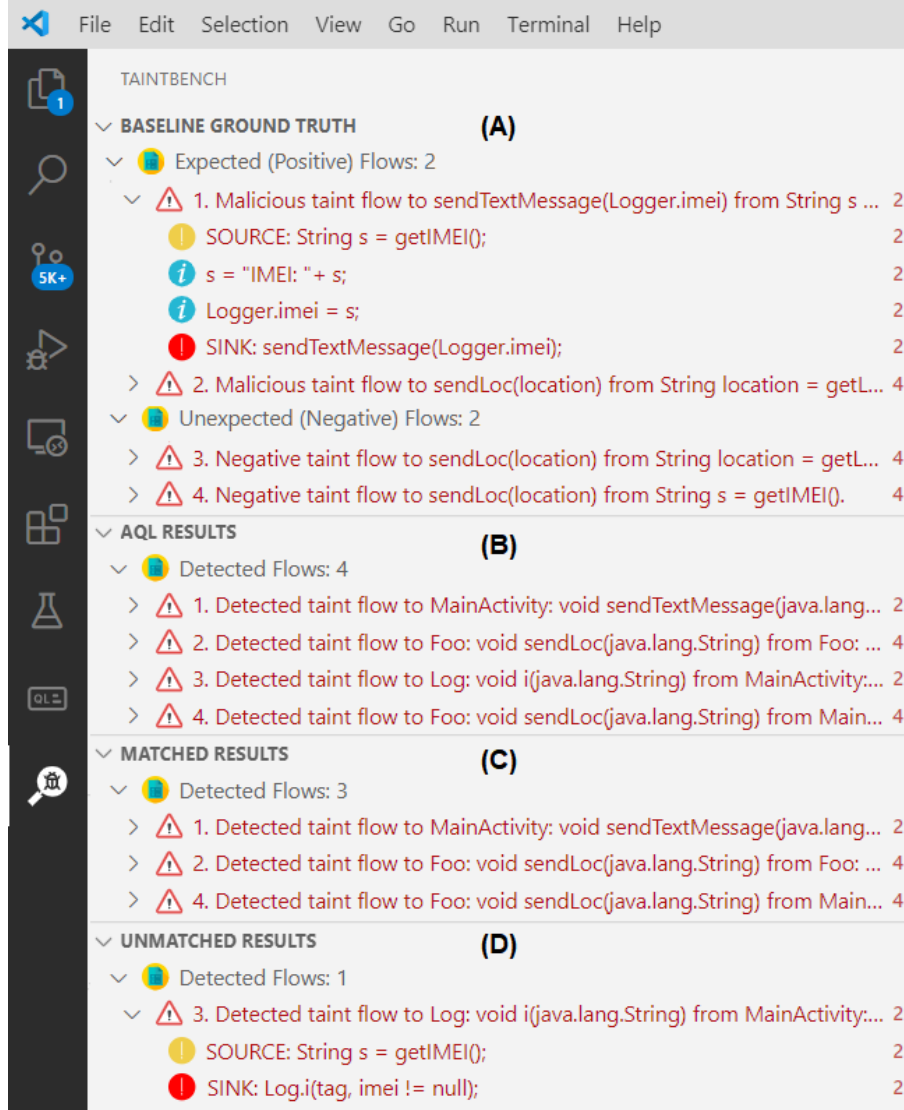


Figure 2.3: Screenshot of TB-VIEWER w.r.t. running example.

List **C** contains all those taint flows of List **A** that are detected during evaluation. The contrary holds for List **D**: it comprises all flows that are reported during evaluation, but do not match any flow in the baseline. These unmatched flows in list **D** cannot be evaluated automatically with REPRODROID by Part 2, which is why TB-VIEWER supports their manual inspection. TB-VIEWER enables the expert to navigate through an application’s source code along visually highlighted taint flows, step-wise from the source to the sink of each taint flow. Considering the running example and the four flows reported by the configured tool, List **A** and **C** would contain the two expected taint flows depicted by solid green edges in Figure 2.2 and an unexpected flow ($s_1 \rightarrow s_9$). List **D** holds one flow: ($s_1 \rightarrow s_7$)—dashed blue edges. Once this latter flow is added to the baseline definition as an unexpected flow, list **D** will be empty.

We have installed TB-VIEWER in the Gitpod online IDE [Git18] for GitHub, thus, all benchmark cases of the TaintBench suite can be viewed in a web browser.⁴ To show that tools we built are usable, we conducted in a small-size usability test. Details about the usability test can be found in Figure A.1.4.

⁴Find the access at <https://taintbench.github.io/taintbenchSuite>

2.5 Real-World Benchmarking

This section introduces real-world benchmarking of popular Android taint analysis tools we conducted. Since the benchmarking was supported by the tools in our TAINTBENCH framework according to the three parts (Section 2.4), we also structure this section into three parts accordingly:

- **Construction:** we introduce the concrete construction of the TAINTBENCH suite—a real-world malware suite for benchmarking Android taint analyses. We evaluate this suite in comparison to DROIDBENCH and the Contagio [Con12] malware dataset to show its representativeness (Part 1 in Section 2.5.1).
- **Evaluation:** we present the evaluation of two prominent taint analysis tools using TAINTBENCH and compare to the evaluation with DROIDBENCH (Part 2 in Section 2.5.2).
- **Inspection:** we present insights from our manual inspection of the analysis results and show directions for future improvement (Part 3 in Section 2.5.3).

2.5.1 Part 1—Construction of the TAINTBENCH Suite

For the suite’s construction, we decided to use available Android malware datasets, since they are very likely to contain malicious taint flows that can and should be detected by Android taint analysis tools. To obtain suitable malware samples to be included in TAINTBENCH, we compared well-known Android malware datasets as shown in Table 2.3. Considering the manual inspection required for identifying the taint flows, we prefer datasets which have more detailed information about malware behaviors, i.e., Contagio [Con12] and AMD [WLR⁺17]. From these two, we then chose the Contagio dataset, since it was updated more recently and its size allows us to qualitatively study all samples. When this thesis is written in 2021, AMD is unfortunately not publicly available anymore.

Table 2.3: Comparison of Android malware datasets.

Dataset	# App	Malware Info	Last Update
Contagio [Con12]*	344	Behavior descriptions	2020
AMD [WLR ⁺ 17]	24,533	Behavior descriptions	2016**
VirusShare [Vir14]	34,265,389	Labels	2019
Drebin [ASH ⁺ 14]	5,560	Labels	2012
Genome [ZJ12]	1,260	Labels	2011**

*: Online source [Con18] (accessed 02/18/2021),

** : Currently unavailable (02/18/2021)

Because original source code is not available for the apps in Contagio, we opted to decompile the Android malware apps. Modern Android decompilation technology allows high-level source code files to be reconstructed successfully in most cases. Decompilation is widely used in reverse engineering and validation of software analysis results for closed-source applications [LBS19, BKKL⁺20]. Another issue was that some applications in the dataset were obfuscated (e.g., class/method/parameter names were renamed to “a”, “bbb”, etc.) such that the decompiled code was very difficult for humans to understand. Considering the difficulty of formulating high-level descriptions for discovered taint flows (as we stated in the documentation criterion) in obfuscated applications and to ease the future validation of the baseline definition by other researchers, we excluded obfuscated applications from our selection. Nonetheless we argue

that our selection is not biased, as we show later in this section our selection is a representative subset of the Contagio dataset.

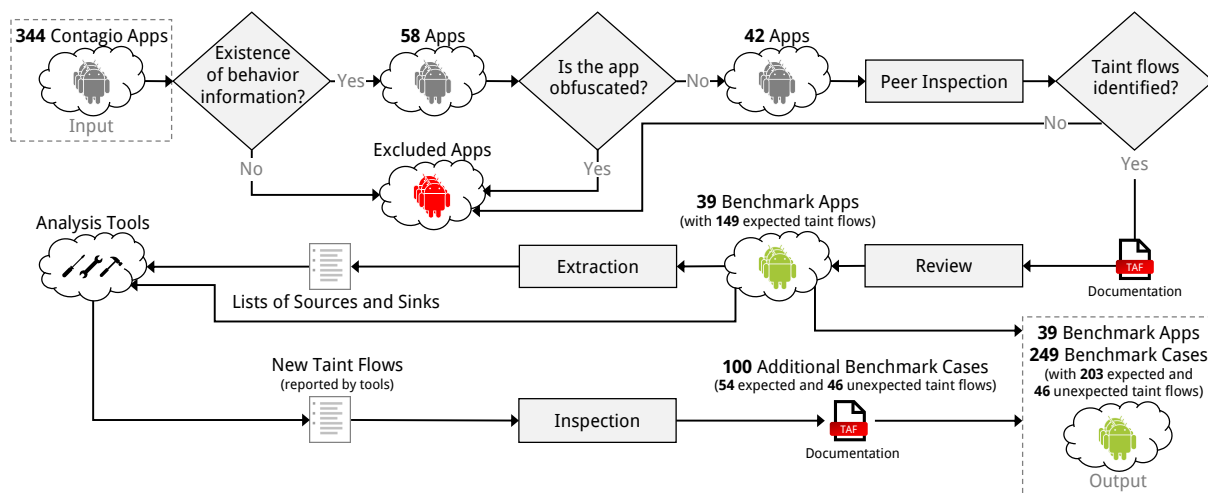


Figure 2.4: The construction process of the TaintBENCH suite. See a full-page version of this figure in Figure A.2.

Figure 2.4 shows our benchmark suite construction process. The Contagio dataset contains 344 apps, but only 58 of them have references to behavior information. From these 58 apps, 42 apps that are not obfuscated became candidates for taint-flow inspection. Initially, we planned to apply existing Android taint analysis tools to the apps and manually check the analysis results, but we quickly gave up on this plan due to the following reasons:

- Too many flows to be checked. The three tools (AMANDROID, DROIDSAFE and FLOWDROID) we initially tried already reported 21,623 flows.
- False negatives remain undetected by tools. We manually inspected a few malware apps. As our inspection reveals, the tools frequently miss critical taint flows which are part of the actual malicious intentions of the malware apps (e.g., leaking banking information), i.e., yield false negatives. Often the sources and sinks that appeared in critical taint flows are not in the tools’ configuration. These false negatives were described in the behavior information written by security experts. Thus, they had to be identified manually.
- The tools also frequently report false positives that prolong code inspection. For the Android malware *fakebank_android_samp*, for instance, FLOWDROID reported 23 taint flows in its default configuration, but only 10 of them are true positives. Moreover, 8 of these 10 only concern the logging of sensitive data using the Android Logging Service, something that is considered secure since Android version 4.1, which protects such logs from being read without authorization [Sie13]. All remaining 13 flows are false positives.

Consequently, we opted for an alternative approach starting with manual inspection. We manually inspected the 42 candidate apps along with their behavior information to identify a set of expected taint flows. For example, if an app’s behavior information like “monitor incoming SMS messages” is included, our inspection starts at sources which read incoming messages.

The inspection for each app was performed by two people, both with background in Android taint analysis research, working together as a pair in front of the same computer. Whenever a taint flow was discovered and confirmed by both inspectors, it was added to the documentation.

After each inspection, a third inspector (a different person) reviewed the documented taint flows. Only the taint flows confirmed by all three inspectors were retained in the final suite. The percentage of agreement between the first two and the third inspector was 96.82%. This resulted in 39 benchmark apps with 149 expected taint flows.

Next, we used an automated tool (see TB-MAPPER in Section 2.4.2.1) to extract sources and sinks from these 149 expected taint flows and used them to configure selected Android taint analysis tools—those which we use during evaluation as well: FLOWDROID and AMANDROID. We then applied the taint analysis tools under this configuration to all benchmark apps. This way 100 taint flows were revealed which have not been documented during our initial manual inspection. Using TB-VIEWER, we manually checked and rated these newly discovered taint flows. Each flow was rated independently by two authors as expected or unexpected. The results were compared and a consensus was established. This resulted in further 100 *additional* benchmark cases—54 expected and 46 unexpected taint flows. For each expected taint flow, we also documented the intermediate steps of one data-flow path as witness. At the end, the TAINTBENCH suite consists of 39 benchmark apps with 203 expected and 46 unexpected taint flows. We developed a few tools, introduced in the next section, to support this process. More details about the evaluation is given in Section 2.5.1.2.

2.5.1.1 The TAINTBENCH Suite

Our TAINTBENCH suite contains 39 benchmark apps with 249 documented benchmark cases in total as shown in Table 2.4. 203 of them are expected taint flows. 149 expected taint flows were

Table 2.4: Summary of the TAINTBENCH benchmark suite.

No.	Name	E.	U.	No.	Name	E.	U.
1	backflash	13	11	21	proxy_samp	17	3
2	beita_com_beita_contact	3	0	22	remote_control_smack	17	0
3	cajino_baidu	12	3	23	repane	1	0
4	chat_hook	12	1	24	roidsec	6	0
5	chulia	4	0	25	samsapo	4	1
6	death_ring_materialflow	1	0	26	save_me	25	6
7	dsencrypt_samp	1	0	27	scipix	3	0
8	exprespam	2	0	28	slocker ¹	5	0
9	fakeappstore	3	0	29	sms_google	4	0
10	fakebank ¹	5	0	30	sms_send_locker_qqmagic	6	2
11	fakedaum	2	0	31	smssend_packageInstaller	5	0
12	fakemart	2	0	32	smssilience_fake_vertu	2	2
13	fakeplay	2	0	33	smsstealer_kysn_assassincreed ¹	5	0
14	faketaobao	4	0	34	stels_flashplayer_android_update	3	0
15	godwon_samp	6	0	35	tetus	2	0
16	hummingbad ¹	2	0	36	the_interview_movieshow	1	0
17	jollyserv	1	0	37	threatjapan_uracto	2	0
18	overlay ¹	4	2	38	vibleaker ¹	4	0
19	overlaylocker2 ¹	7	12	39	xbot ¹	3	0
20	phospy	2	3	Σ		203	46

E.: Number of expected taint flows, U.: Number of unexpected taint flows, ¹: Suffix "_android_samp"

Table 2.5: Overview of expected and unexpected taint flows according to source and sink Categories. New categories we added are marked with * when appearing for the first time in the table.

Source Category	Sink Category	E.	U.
ACCOUNT INFORMATION	NETWORK	11	
ACCOUNT INFORMATION	INTENT	1	
ACCOUNT INFORMATION	FILE	2	
ACCOUNT INFORMATION	LOG	1	
ACCOUNT INFORMATION	DATABASE	2	
CONTACT INFORMATION	NETWORK	11	11
CONTACT INFORMATION	INTENT	2	
CONTACT INFORMATION	SMS MMS	1	
CRITICAL FUNCTION *	CRITICAL FUNCTION *	2	
DATABASE	SMS MMS	3	
DATABASE	FILE	24	
DATABASE	NETWORK	19	
DATABASE	DATABASE	2	3
DATABASE	LOG	3	
DATABASE	CRITICAL FUNCTION	1	
FILE	NETWORK	13	2
FILE	FILE	1	1
FILE	CRITICAL FUNCTION	5	
FILE	INTENT	3	
FILE	LOG		1
INTERNET SOURCE *	SMS MMS	2	
INTERNET SOURCE	OTHER STORAGE *	1	
LOCATION INFORMATION	FILE	4	
LOCATION INFORMATION	NETWORK	4	2
NETWORK INFORMATION	EMAIL	1	
NETWORK INFORMATION	LOG	1	1
NETWORK INFORMATION	FILE	1	
NETWORK INFORMATION	NETWORK	9	
NETWORK INFORMATION	SMS MMS		1
OTHER DATA *	NETWORK	1	
OTHER DATA	LOG	7	1
OTHER DATA	CRITICAL FUNCTION		6
OTHER DATA	INTENT		2
SMS MMS	SMS MMS	5	
SMS MMS	NETWORK	11	
SMS MMS	INTENT	6	2
SMS MMS	LOG	1	
SMS MMS	FILE	1	
SMS MMS	CRITICAL FUNCTION	1	
SYSTEM SETTINGS	NETWORK	3	
SYSTEM SETTINGS	CRITICAL FUNCTION	1	1
UNIQUE IDENTIFIER	FILE	1	
UNIQUE IDENTIFIER	NETWORK	25	6
UNIQUE IDENTIFIER	LOG	3	
UNIQUE IDENTIFIER	EMAIL	1	
UNIQUE IDENTIFIER	CRITICAL FUNCTION	3	6
UNIQUE IDENTIFIER	SMS MMS	2	

E.: Number of expected taint flows, U.: Number of unexpected taint flows

discovered by us manually as described in last section. During the evaluation with the benchmark apps, we also inspected taint flows which were reported by both FLOWDROID and AMANDROID manually. Thereby additional 54 expected and 46 unexpected taint flows were added to the suite. We will introduce more details about this in Section 2.5.1.2. Each benchmark app comes with the following assets in its own GitHub repository:

- the APK file,
- the decompiled source code project,
- the baseline definition (TAF-file),
- a profile file about the benchmark app containing statically extracted information including target platform version, permissions, sensitive APIs, behavior description, etc.

All artifacts are publicly available at: <https://TaintBench.github.io>

To give more detail on the taint flows in TAINTBENCH, we classified the flows based on their behaviors according to their source and sink categories as shown in the Table 2.5⁵. We reused the categories defined in the SUSI paper [RAB14] and MUDFLOW paper [AKG⁺15]. We also added new categories such as `INTERNET_SOURCE` and `CRITICAL_FUNCTION`, since our suite includes types of malicious taint flows beyond data leaks, such as Path Traversal (CWE-22), Execution with Unnecessary Privileges (CWE-250) and Use of Potentially Dangerous Function (CWE-676).⁶ The categorization of the sources and sinks was first done by the lead author. To enhance the reliability, the third author checked and discussed the assigned categories with the lead author whenever there were disagreements. Consensus was achieved for the final categorization.

2.5.1.2 Evaluation of the TAINTBENCH Suite

We present our evaluation of the TAINTBENCH suite by answering the following research question: How does TAINTBENCH compare to DROIDBENCH and Contagio?

To answer this question, we evaluate the TAINTBENCH suite with regard to the two aspects of representativeness introduced in Section 2.3: First, we evaluate the expected taint flows in TAINTBENCH and compare them to those in DROIDBENCH in terms of language and framework features involved in the flows. Second, we compare the benchmark apps in TAINTBENCH to the apps in DROIDBENCH and the whole Contagio dataset with a set of metrics.

Comparison of Taint Flows As introduced in the representativeness criterion in Section 2.3, one of our goals for TAINTBENCH is to include taint flows which address different language and framework features (attributes). The numbers of expected taint flows involving different language and framework features are listed in Table 2.6. The attributes of each taint flow are assigned by us and TB-PROFILER as mentioned in Section 2.4. Through these attributes the taint flows can be categorized and also mapped to the majority (11/18) of the DROIDBENCH’s categories as shown in Table 2.6. Categories of DROIDBENCH such as “Android Specific” [Dro16] are not uniquely relatable.

⁵More detailed information of each flow can be https://taintbench.github.io/img/data/Sources_Sinks_Category_Stats.pdf

⁶CWEs can be found at <https://cwe.mitre.org>

Table 2.6: Attributes associated to expected taint flows in TAINTBENCH.

Attribute	Description	DB Category	# Flows
nonStaticField	sensitive values stored in non-static fields	Field and Object Sensitivity	61
staticField	sensitive values stored in static fields	Field and Object Sensitivity	31
reflection	reflection APIs called	Reflection	5
array	sensitive values stored in arrays	Arrays and Lists	39
collections	sensitive values stored in Java collection objects	Arrays and Lists	67
threading	threading mechanisms involved	Threading	80
appendToString	sensitive values appended to Strings	General Java	99
callbacks	callbacks for UI interactions	Callbacks	23
lifecycle	lifecycle methods involved	Lifecycle	104
payload	malicious payload is downloaded at runtime	Dynamic Loading	5
ICC	inter-component communication involved	ICC	49
IAC	inter-app communication involved	IAC	2
implicitFlows*	implicit flows	Implicit Flows	6
pathConstraints ⁺	path conditions must be satisfied	–	74

*Cannot automatically be assigned by TB-EXTRACTOR, ⁺No mapping category in DROIDBENCH (DB).

Moreover, most expected taint flows in TAINTBENCH address multiple (up to 8) features at the same time as shown in the histogram in Figure 2.5. This is not modeled in DROIDBENCH. The majority (175/203) of expected taint flows in TAINTBENCH address multiple features at the same time rather than a single one as designed in DROIDBENCH.

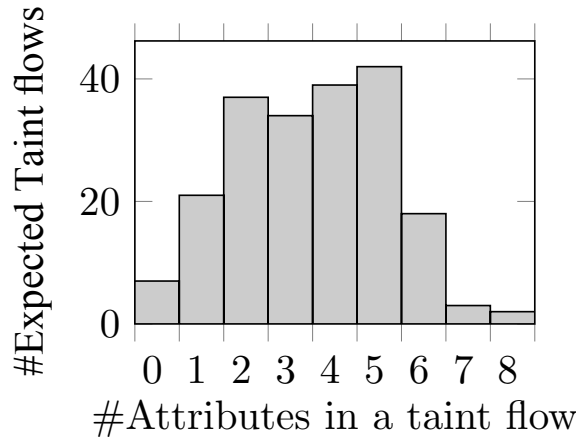


Figure 2.5: Distribution of attributes.

Comparison of Benchmark Apps We compare the benchmark apps in TAINTBENCH to apps in DROIDBENCH as well as to the entire Contagio dataset (from which TAINTBENCH apps are sampled) in three aspects: (i) the usage of sources and sinks, (ii) call-graph and (iii) code complexity. For each aspect, we used a set of metrics for a quantitative evaluation.

Usage of Sources and Sinks: The usage of sources and sinks is a predictor of how many data-flow propagations starting from sources/sinks (depending on forward/backward analysis) are required to capture all possible taint flows. Thus, we quantify the usage of sources and sinks with the following metrics:

- #Sources/Sinks: number of different source/sink APIs appeared in a benchmark app.
- #Usage Sources/Sinks: number of different code locations where source/sink APIs are used in a benchmark app.

To measure these metrics, we first compiled a list of potential source and sink APIs. This list consists of three parts: (i) sources and sinks from SuSi [RAB14] GitHub repository; (ii) sources and sinks detected by the machine-learning approach SWAN [PDB19] when applying it to Android platform jars (API level 3 to 29); (iii) sources and sinks documented in both TAINTBENCH and DROIDBENCH. Based on this list, we computed the values of above metrics for each app.

Table 2.7: Usage of sources and sinks.

	#Sources	#Usage Sources	#Sink	#Usage Sinks
DROIDBENCH				
min	0	0	1	1
max	8	27	13	22
geomean	2.2	2.4	2.8	3.3
TAINTBENCH				
min	4	9	6	13
max	514	4,284	369	3,486
geomean	49	149	38.4	133.2
CONTAGIO				
min	2	2	0	0
max	699	8,044	495	7,870
geomean	55.1	201.6	43.7	182.0

The summarized results are shown in Table 2.7. We report the minimum, the maximum, and the geometric mean⁷ of each suite and the Contagio dataset. We use geometric mean rather than arithmetic mean, because it can dampen the effect of outliers and is more representative of a dataset with the differences of data points varying by multiples of 10. Clearly, TAINTBENCH employs more usages of sources and sinks than DROIDBENCH, since all values of TAINTBENCH are at least six times higher than those of DROIDBENCH. In addition, the geometric means of TAINTBENCH are in the same order of magnitude as for the entire Contagio dataset. Especially regarding the number of different sources and sink APIs that appeared in an app, the difference between TAINTBENCH and Contagio is less than 6. In conclusion, this indicates that a tool must be scalable to handle more data-flow propagations on TAINTBENCH to achieve equally good results as on DROIDBENCH.

Call-graph Complexity: One of the most important tasks for inter-procedural analysis is to construct the call graph. We use AndroGuard [And11] to generate context-insensitive call graphs. To compare fairly, we excluded call graph edges from Android platform APIs to the actual application and edges between Android platform APIs themselves. Based on the resulting sub call graph, we compute:

- Call Graph Size (CG Size): Number of edges in the call graph.
- Maximal Call Chain Length (Max. CCL): Number of edges in the longest acyclic call chain [RKG04, EHMG15]. A call chain is a sequence of call graph edges e_1, \dots, e_k such that the target node of e_i is the same as the source node of e_{i+1} for $1 \leq i < k$ [RKG04]. Our algorithm to compute Max. CCL is to perform a depth first search for each entry point detected by ANDROGUARD. We handle the cycles in the call graph as follows: whenever a

⁷When there are 0s in the dataset, we computed the geomean of all positive numbers.

visited node is seen twice in a call chain, we break the search. For a call chain with cycle $(1, 2), (2, 3), (3, 1), (1, 3), (3, 1), (1, 3), \dots$, the length of the call chain is 2.

- Longest Cycle Size (LC Size): Number of edges in the longest cycle in the call graph.

The call-graph comparison is shown in Table 2.8. The geometric mean indicates, that call graphs in TAINTBENCH are much larger and more complex than in DROIDBENCH. In comparison to the entire Contagio dataset, the call graphs in TAINTBENCH are smaller. However, as shown in Table 2.9, the minimum, maximum and geometric mean of analysis time used by FLOWDROID for an app in TAINTBENCH is almost the same as the time required with respect to the entire Contagio set.

Table 2.8: Call-graph complexity.

	CG Size	Max. CCL	LC Size
DROIDBENCH			
min	11	1	0
max	144	4	0
geomean	27.28	1.4	0
TAINTBENCH			
min	112	1	0
max	83,981	16	31
geomean	1,895.68	4.79	3
CONTAGIO			
min	43	0	0
max	139,298	44	31
geomean	2,599.6	6.3	2.5

While the call graphs in DROIDBENCH have no cycles (recursions), since the values of LC size are all zeros, the call graphs in TAINTBENCH include even large cycles (LC size up to 31). Recursion is considered as an important problem that needs to be solved in a context-sensitive analysis. The absence of it in DROIDBENCH makes it impossible to evaluate the implemented solution for handling recursions. Max. CCL can be seen as an indicator for the choice of context string length (call string length) of a context-sensitive analysis. If the context string length is chosen too small, the analysis can lose precision and soundness. When the length is too big, the analysis may not scale. To build a scalable context-sensitive analysis that produces proper results, the context string length for a best trade-off between precision and scalability may be easy to find for DROIDBENCH with Max. CCL varying from 1 to 4, however, it is much more difficult to find the best fit in TAINTBENCH, since the maximum is up to 16.

Table 2.9: Analysis time(s) measured for FLOWDROID and AMANDROID.

	FLOWDROID		AMANDROID	
	TAINTBENCH	CONTAGIO	TAINTBENCH	CONTAGIO
min	2.5	2.3	16.2	0.6
max	361.6	361.6	762	6,949.3
geomean	8.5	8.2	71.8	113.8

Code Complexity: We compare TAINTBENCH and DROIDBENCH by computing the following Chidamber and Kemerer (CK) metrics [CK94]: Coupling between object classes (CBO),

Table 2.10: Comparing code complexity with CK metrics.

Metrics	CBO	DIT	RFC	WMC	NF	NSF
DROIDBENCH						
min-sum	2	2	4	1	0	0
max-sum	29	17	48	32	27	15
geomean-sum	6.77	5.26	11.01	4.07	5.12	5.99
min-avg	0.2	1.08	0.4	0.09	0	0
max-avg	10	2.67	25	16	11.5	3.25
geomean-avg	2.03	1.57	3.3	1.22	1.09	0.85
TAINTBENCH						
min-sum	11	4	43	14	3	0
max-sum	19,597	5,533	32,661	46,251	10,644	3,354
geomean-sum	653.77	223.8	1,528.58	1,860.69	359.13	145.89
min-avg	2.53	1.25	7.28	3.57	0.75	0
max-avg	7.55	2.45	30	27.64	5.94	2.45
geomean-avg	4.52	1.55	10.56	12.85	2.48	0.92

Depth of Inheritance Tree (DIT), Response for a Class (RFC) and Weighted Method per Class (WMC). These were often used to evaluate software complexity [PRL⁺19, BGH⁺06]. Beside the CK metrics, we also compare number of fields and static fields in the benchmark apps. We used the ck tool [Mau15] on the source code project of each benchmark app to calculate the metrics. The results regarding the CK metrics are listed in Table 2.10. All measurements show that TAINTBENCH benchmark apps are more complex than DROIDBENCH benchmark apps. While this is not surprising, we find it important to compute these numbers for future reference.

2.5.2 Part 2—Evaluation with the TAINTBENCH Suite

In this section, we present our evaluation of Android taint analysis tools on TAINTBENCH and answer the following research questions:

- How effective are taint analysis tools on TAINTBENCH compared to DROIDBENCH?
- What insights can we gain by evaluating analysis tools on TAINTBENCH?

Tool and Benchmark Selection: We evaluated two taint analysis tools, namely AMANDROID and FLOWDROID. These two tools were chosen because they lately scored best in two independent studies [PBW18, QWR18] when evaluated on DROIDBENCH and they are based on distinct analysis frameworks. Two different versions of both tools are employed: (1) the respective up-to-date version, and (2) the version used by Pauck et al. [PBW18] in order to compare our reproduced values to theirs. In the following, we mark the current tool versions by a *-symbol as shown in Table 2.11. TAINTBENCH and DROIDBENCH (3.0) are selected as benchmark suites for all the experiments. Note, because both tools cannot analyze inter-app communication scenarios, the related benchmark cases of DROIDBENCH are not considered in our setup.

Table 2.12: Descriptions of experiments.

Experiment	ID	Description	Comparable
Experiment 1	<i>DB1</i>	Default configuration; evaluated on <i>DB</i>	✓
	<i>TB1</i>	Default configuration; evaluated on <i>TB</i>	✓
Experiment 2	<i>DB2</i>	Sources & sinks w.r.t. <i>DB</i> (Suite-Level)	✓
	<i>TB2</i>	Sources & sinks w.r.t. <i>TB</i> (Suite-Level)	✓
Experiment 3	<i>TB3</i>	Sources & sinks w.r.t. <i>TB</i> (App-Level)	✓
Experiment 4	<i>TB4</i>	Sources & sinks w.r.t. <i>TB</i> (Case-Level)	✓
Experiment 5	<i>TB5</i>	w.r.t. minified apps per <i>TB</i> case	✗
Experiment 6	<i>TB6</i>	w.r.t. delta apps per <i>TB</i> case	✗

Table 2.11: Tools evaluated.

Tool	Version
AMANDROID [Ama17]	November 2017 (3.1.2)
AMANDROID* [Ama18]	December 2018 (3.2.0)
FLOWDROID [Flo17]	April 2017 (Nightly)
FLOWDROID* [Flo19]	January 2019 (2.7.1)

* Up-to-date tool versions.

Evaluation Objectives: In the context of TaintBENCH, we focus on evaluating analysis accuracy in terms of precision, recall and F-measure but less on analysis time.

Execution Environment: The TaintBENCH framework was setup on an Debian (9 – Stretch) virtual machine with two cores of an Intel®Xeon®CPU (E5-2695 v3@2.30GHz), 128 GB memory and Java 8 (Oracle 1.8.0_231) installed. 96 GB memory were reserved for the analysis tools.

Experiments: Table 2.12 lists all conducted experiments. The first column ID refers to the benchmark suite and experiment number (e.g., TB2 refers to Experiment 2 w.r.t. TaintBENCH). This ID is used throughout the whole section. The second column provides a brief description of each experiment. The last column indicates the comparability of experimental results. Accordingly, the results of all experiments except TB5 and TB6 are comparable with one another.

We first conducted all experiments with the 149 expected taint flows identified by us manually in the benchmark construction phase as described in Section 2.5.1. Afterwards, we manually checked and rated newly discovered taint flows reported by all tools in TB3 and added them as expected and unexpected taint flows into the baseline and re-ran all experiments. In the following, we report the results using the final baseline of the TaintBENCH suite presented in Table 2.4.

2.5.2.1 Experiment 1 (DB1 & TB1):

The tools are executed in their default configuration. Figure 2.6 presents precision, recall and F-measure for DROIDBENCH and TaintBENCH in column DB1 and TB1, respectively.

The results obtained for FLOWDROID and AMANDROID in configuration DB1 are identical to those in the REPRODROID study [PBW18], replicating the results obtained there.

The metrics in Figure 2.6 are calculated based on their formulas introduced in Section 2.1 and Table 2.13, which also shows the expected cases and unexpected cases defined for each

benchmark suite. The evaluation is based on these cases only. Although the baseline definition of TaintBench contains 203 expected and 46 unexpected taint flows, REPRODROID can only reflect 186 expected cases and 35 unexpected cases, since it does not distinguish different flows, when the sources and sinks look exactly the same in Jimple. Jimple statements are not differentiable (by their textual representation) if they (1) occur in the same method of the same class, (2) use variables with the exact same names as well as constants with the same contents, (3) and refer to the same source code line number. As the figure shows, the precision of AMANDROID is dramatically decreased to 50% when evaluated on TaintBench. It only found 4 flows in our baseline and 2 of them are false positives. The precision of AMANDROID* stays almost unchanged, however, this is calculated only from 6 flows. In contrast, the precisions of both FLOWDROID and FLOWDROID* are high (over 90%). However, on TaintBench all tools show a significantly lower recall and F-measure than for DROIDBENCH. In the default configuration most taint flows in TaintBench remain undetected. With 14% (26/186), FLOWDROID’s recall is still the highest.

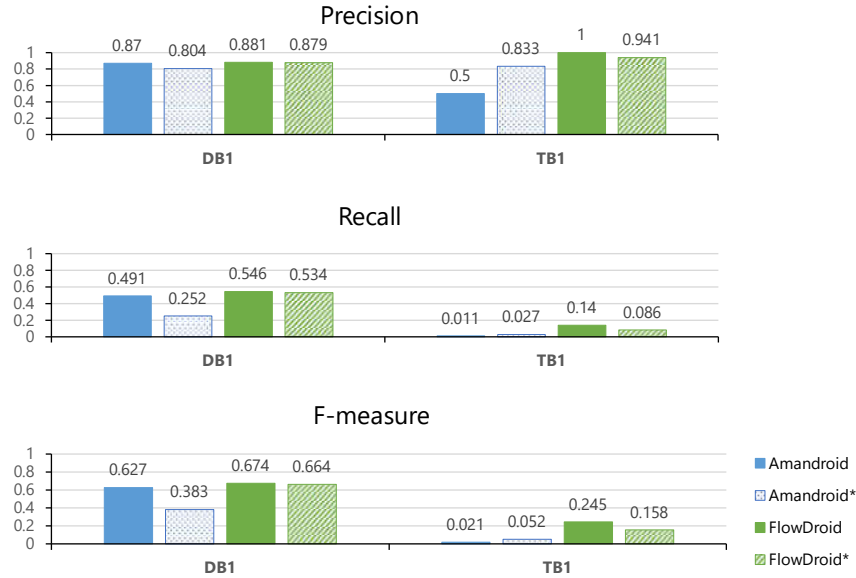


Figure 2.6: Precision, Recall and F-measure of DB1 & TB1.

Table 2.13: Results of DB1 & TB1.

Benchmark	DroidBench		TaintBench	
	Expected	Unexpected	Expected	Unexpected
	163	41	186	35
Tool	DB1		TB1	
	TP	FP	TP	FP
Amandroid	80	12	2	2
Amandroid*	41	10	5	1
FlowDroid	89	12	26	0
FlowDroid*	87	12	16	1

2.5.2.2 Experiment 2 (DB2 & TB2):

To understand if the low recall values for TAINTBENCH in Experiment 1 are mainly caused by the tools' source and sink configurations, we compared the different source and sink sets involved. The results of this comparison are summarized in Table 2.14. Tool names refer to source and sink sets defined in a tool's default configuration. Benchmark suite names refer to the sets occurring in their benchmark cases. While the upper part of the table row-wise shows the intersections of these sets in terms of numbers of sources and sinks, the lower part enumerates sources and sinks contained in one set A but not in another set B . The two rows labeled with TAINTBENCH show that (i) the sets of sources and sinks used by tools and DROIDBENCH have only minor intersections with the set of TAINTBENCH; (ii) TAINTBENCH holds at least 32 different sources and 36 sinks (see column FLOWDROID).

Table 2.14: Intersection and difference of source and sink sets.

$\begin{matrix} B = \\ A = \end{matrix}$	AMANDROID		FLOWDROID		DROIDBENCH		TAINTBENCH	
	Sources	Sinks	Sources	Sinks	Sources	Sinks	Sources	Sinks
Intersection ($A \cap B$)								
AMANDROID	30	42	24	38	4	8	6	4
FLOWDROID	24	38	89	133	7	9	12	8
DROIDBENCH	4	8	7	9	15	23	7	4
TAINTBENCH	6	4	12	8	7	4	44	44
Difference ($A \setminus B$)								
AMANDROID	0	0	6	4	26	34	24	38
FLOWDROID	65	95	0	0	82	124	77	125
DROIDBENCH	11	15	8	14	0	0	8	19
TAINTBENCH	38	40	32	36	37	40	0	0

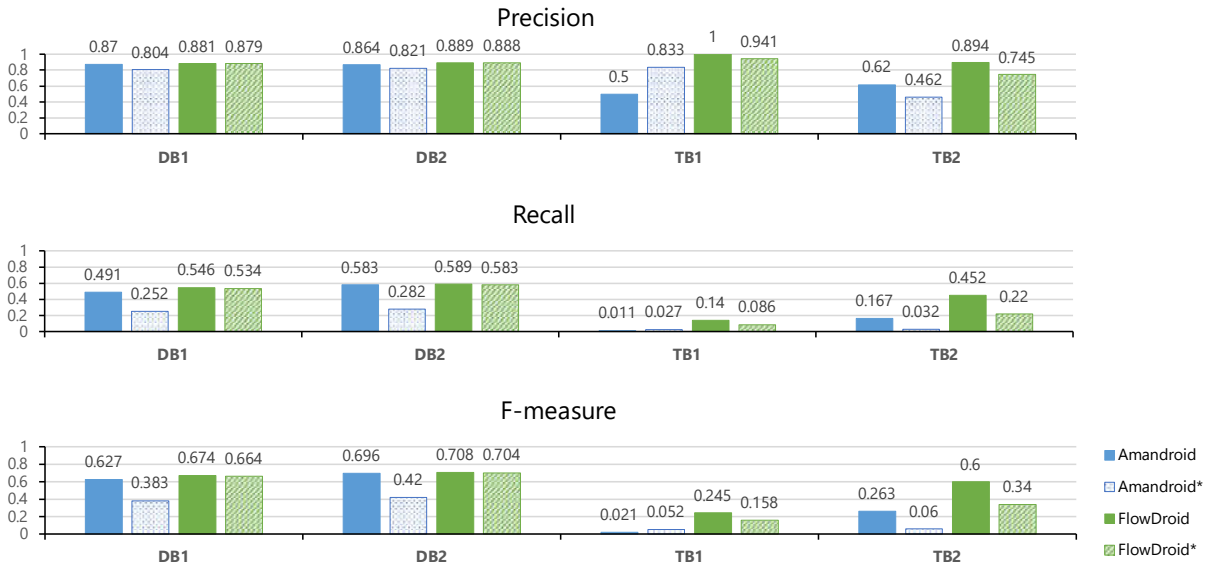


Figure 2.7: Precision, Recall and F-measure of DB1, DB2, TB1, TB2.

In consequence, we generated a list of sources and sinks for each benchmark suite based on the comprised taint flows, using TB-LOADER (see Section 2.4.2). These lists, generated on

the suite-level, are configured to be used by the two tools. As shown in Figure 2.7, when re-configuring sources and sinks this way, the results for DROIDBENCH are affected only slightly (DB1 vs. DB2) but the recall and F-measure values for TAINTBENCH are more than doubled (TB1 vs. TB2). Nonetheless, even under this configuration the tools are still less effective on TAINTBENCH than on DROIDBENCH. Closest is FLOWDROID, which achieves a recall of 45% (84/186) for TAINTBENCH while reaching 59% (96/163) for DROIDBENCH. While FLOWDROID reports 84 true positives, AMANDROID and AMANDROID* detect only 31 and 6 true positives in TAINTBENCH, respectively.

Surprisingly, on TAINTBENCH the current tool versions (AMANDROID* and FLOWDROID*) show a lower recall than their predecessors (AMANDROID and FLOWDROID). This argues for the use of TAINTBENCH also for regression testing.

For FLOWDROID and FLOWDROID*, the difference is small (1 or 2 flows) regarding DROIDBENCH. Considering TAINTBENCH, FLOWDROID* finds only half (41/84) of the true positives that can be found by its old version even under the same source and sink configuration. In addition, FLOWDROID* is less precise, since it reports more false positives than FLOWDROID (14 vs. 10).

Table 2.15: Results of DB2 & TB2.

Benchmark	DroidBench		TaintBench	
	Expected	Unexpected	Expected	Unexpected
	163	41	186	35
Tool	DB2		TB2	
	TP	FP	TP	FP
Amandroid	95	15	31	19
Amandroid*	46	10	6	7
FlowDroid	96	12	84	10
FlowDroid*	95	12	41	14

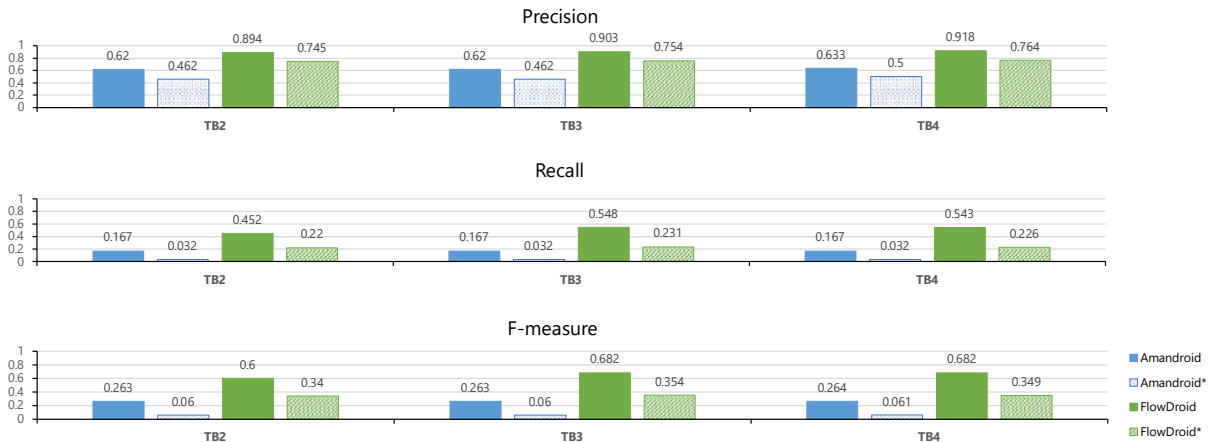


Figure 2.8: Precision, Recall and F-measure of TB2-4.

2.5.2.3 Experiment 3 & 4 (TB3 & TB4):

Since the results of Experiment 2 show that source and sink configurations affect the recall values heavily, we conduct two more experiments in which sources and sinks are configured regarding not just each suite but even each benchmark app (Experiment 3) and each benchmark case (Experiment 4). To this end, we are now using smaller but more precise sets. We expected that the results would be the same compared to Experiment 2, and this also holds for AMANDROID(*) as shown in Figure 2.8.

Surprisingly, FLOWDROID and FLOWDROID* find *more* taint flows in Experiment 3 and 4. The number of true positives increases from 84 to 102 for FLOWDROID and from 41 to 43 for FLOWDROID* as Table 2.16 shows. This indicates that the configuration of “superfluous” sources and sinks, which are actually irrelevant for a specific benchmark app or case, has some shadow effect on the taint computation in FLOWDROID and FLOWDROID*.

Table 2.16: Results of TB2-6.

Benchmark	TaintBench									
	Expected					Unexpected				
	186					35				
Tool	TB2		TB3		TB4		TB5		TB6	
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
Amandroid	31	19	31	19	31	18	29	4	8	42
Amandroid*	6	7	6	7	6	6	7	1	1	12
FlowDroid	84	10	102	11	101	9	59	8	51	62
FlowDroid*	41	14	43	14	42	13	28	5	23	34

2.5.2.4 Experiment 5 (TB5)

With this experiment we seek to test if the call-graph complexity of real-world apps in TaintBench is a cause of some unsatisfactory results. To do so, irrelevant call-graph edges are removed by MINAPK-GENERATOR (see Section 2.4.2.1). This leads to fewer timeouts in all cases in Experiment 5 in comparison to Experiment 2 (cf. Table 2.17; -5 for AMANDROID* and -1 for all other tools). Overall, 27 new true positives are uniquely detected in Experiment 5. 12 of these by AMANDROID and 8 by FLOWDROID. The newer tools are less affected with 3 and 4 newly found true positives in case of AMANDROID* and FLOWDROID*. The fact that Experiment 5 mostly simplifies the call graph indicates that the tools likely miss these flows in the original benchmark apps due to incomplete call graphs.

However, the minified app created by MINAPK-GENERATOR is sometimes unsound (relevant code is removed), causing the tools unable to detect some previously detected true positives. Hence, the results of Experiment 5 look overall worse in Figure 2.8. Currently, the coauthor Felix Pauck is working on improving the MINAPK-GENERATOR.

2.5.2.5 Experiment 6 (TB6)

With this experiment we check if the tools handle data sanitization properly. We use the DELTAAPK-GENERATOR to kill each expected taint flow in the baseline definition (see Section 2.4.2.1). If a previously detected true positive in Experiment 3 is still detected in the respective delta APK (Experiment 6), this flow is a known false positive (TB3 vs. TB6). If the

tools were to fully handle the killing of flows, also known as “strong updates”, the results of both experiments should be identical. This is not the case: both precision and recall decreased (see TB6 in Figure 2.8). To this effect, and because of the adapted interpretation, the results should not be compared to the experiments above.

The tools over-approximate the killing of taint flows, i.e., miss the ability to perform strong updates, which produces a number of false positives on real-world apps.

2.5.2.6 Timeouts and Unsuccessful Exits

The maximal execution time per app is set to 20 minutes during all experiments. On DROIDBENCH (DB1, DB2), neither AMANDROID(*) nor FLOWDROID(*) reach this timeout. On TAINTBENCH, though, AMANDROID* exceeds the timeout on 11 apps. All other tools rarely do so (see Table 2.17). However, for one DROIDBENCH and seven TAINTBENCH apps FLOWDROID* claimed to find no analysis entry point, even though FLOWDROID was able to find those — a clear regression. In case of two other TAINTBENCH apps FLOWDROID* failed its analysis, since it was unable to calculate callbacks. Consequently, on TAINTBENCH FLOWDROID* finds fewer than half of the true positives that can be found by FLOWDROID.

It is alarming that both newer tool versions fail to analyze a striking number of benchmark apps, particularly where earlier analysis versions succeeded. This emphasizes that research progress requires testing on more real-world benchmarks.

Table 2.17: Timeouts, unsuccessful exits and analysis time in experiment 2.

	DROIDBENCH (DB2)				TAINTBENCH (TB2)			
	AD	AD*	FD	FD*	AD	AD*	FD	FD*
Timeouts	0	0	0	0	1	11	1	1
Unsuccessful Exits	0	0	0	1	0	0	0	9
Analysis Time (min)	58	61	20	13	98	41	17	5
↪ incl. Timeouts	58	61	20	13	118	261	37	27

AD: AMANDROID, FD: FLOWDROID

2.5.2.7 Analysis Time

In all scenarios, both versions of FLOWDROID are faster than any version of AMANDROID. With respect to DROIDBENCH, FLOWDROID* is 35%⁸ faster than its predecessor (see Table 2.17). Considering TAINTBENCH, FLOWDROID*’s speed-up is even larger (71%⁹). AMANDROID* is not faster than AMANDROID on DROIDBENCH but 58%¹⁰ faster in case of TAINTBENCH. However, the amount of timeouts thrown by AMANDROID* and unsuccessful exits of FLOWDROID* impede a fair comparison. While new tool versions appear to be faster, the number of timeouts or unsuccessful exits has risen.

⁸35%=(20-13)/20

⁹71%=(17-5)/17

¹⁰58%=(98-41)/98

2.5.2.8 Reproducibility and Continues Benchmarking

The reproducibility of our experiments is guaranteed by REPRODROID. From our experiments, we found out that newer tool versions performed worse than their predecessors when evaluating on TAINTBENCH. To help the tool authors avoid such regressions in the future, we aim to provide a way in which Android taint analysis tools can be evaluated on TAINTBENCH on a continuous basis. We set up GitHub Actions [Git20] for both versions of AMANDROID and FLOWDROID¹¹. Using the TAINTBENCH framework, we were able to configure the evaluation of each tool as an automated workflow of Github Actions. The source and sink configuration of each tool is at app-level as in Experiment 3 (see Table 2.12). The outcome of each workflow includes a benchmark file computed by REPRODROID containing performance metrics (precision, recall, F-measure, analysis time) and raw analysis results of the tool. Each workflow will be triggered on pushes or pull requests to the TAINTBENCH GitHub repository. This way we can easily obtain performance improvements and regressions of newer tool releases evaluated on the newest version of the TAINTBENCH suite in the future.

2.5.3 Part 3—Inspection of the Analysis Results

2.5.3.1 Unexpected Behavior of FLOWDROID*

With the help of TB-VIEWER (see Section 2.4.3) displaying both the baseline findings and analysis results in VS code, we made the following observation: One true-positive flow (A) is detected by FLOWDROID* in Experiment 3, but not in Experiment 4. Instead, in Experiment 4 a *different* true-positive flow (B) is detected¹².

Considering flow (A), we found that FLOWDROID* sometimes does not find a taint flow (*source* \rightarrow *Child.sink*) when *Parent.sink* was *not* declared in the list of sources and sinks, where *Child* is a subclass of the class *Parent*. By intuition the reason seems to be that the flow is only detected when *Parent.sink* is configured as in Experiment 3. Thus, when *Parent.sink* is not configured in the list, the flow to *Child.sink* remains undetected as in Experiment 4.

Moreover, in case of (B) there are two flows (*source*₁ \rightarrow *sink*₁) and (*source*₂ \rightarrow *sink*₁) with the same sink but FLOWDROID* reports only one of them in Experiment 3. However, the internal analysis of FLOWDROID* is actually capable of finding both flows in Experiment 4, namely, the new true positive (B) is detected. After a closer investigation, we found out that when more sources and sinks than the source and sink of the expected taint flow are configured for FLOWDROID* (Experiment 3) one sink overshadows the other. The order of the relevant sources and sinks appear on two parallel paths in the inter-procedural control-flow graph (ICFG). These path can be illustrated as follows:

*path*₁: *source*₁ \rightarrow *sink*₂ \rightarrow *sink*₁

*path*₂: *source*₂ \rightarrow *sink*₁

In Experiment 3, two flows are found: (*source*₁ \rightarrow *sink*₂), (*source*₂ \rightarrow *sink*₁). However, the expected one (*source*₁ \rightarrow *sink*₁) remains undetected which is not the case in Experiment 4. Because *sink*₁ appears later than *sink*₂ in *path*₁, we think that FLOWDROID* stops the propagation of taints from *source*₁ when the taints reach *sink*₂. We reported our findings to the tool maintainers.

¹¹More information can be found on <https://taintbench.github.io/ci>

¹²A: Flow with ID=1 in *overlay_android_samp*. B: Flow with ID=7 in *cajino_baidu*.

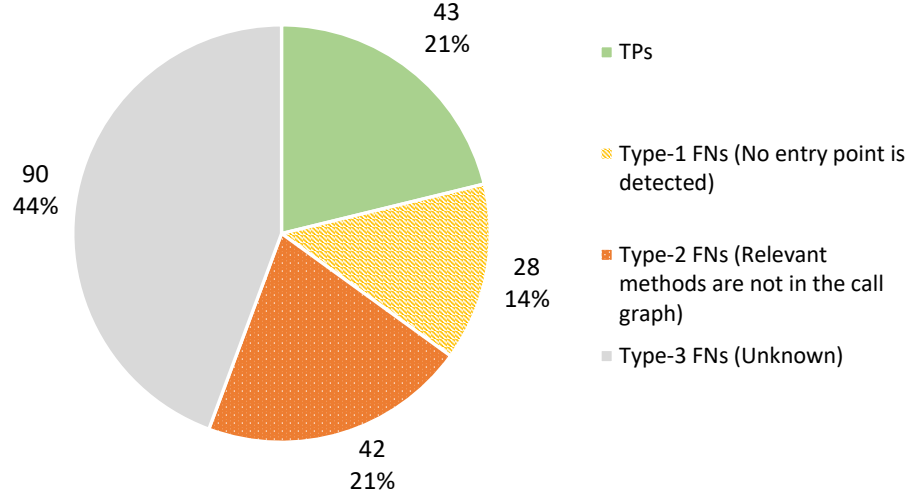


Figure 2.9: Overview of true positives (TPs) and false negatives (FNs) from the evaluation of FLOWDROID on TAINTBENCH. In TAINTBENCH, there are 203 expected taint flows, FLOWDROID only detected 43 (21%) of them.

2.5.3.2 Case Study of FLOWDROID’s False Negatives

To understand why Android taint analysis tools failed to detect taint flows in the TAINTBENCH suite, we further investigated FLOWDROID, since it has a better recall than AMANDROID. The version we used is 2.7.1, since its source code is still available and this allows us to debug into it. Although the nightly build from April 2017 listed in Table 2.11 achieved a better recall on TAINTBENCH, we could not find its source code. In the following, when we talk about FLOWDROID, we mean the version 2.7.1. Note that the case study on FLOWDROID’s false negatives presented in this section was not included in our journal paper [LPP⁺21].

After case studies of a few undetected taint flows, we noticed that FLOWDROID failed to even find the entry points of the respective benchmark apps of these flows. Thus, we extended FLOWDROID to capture the call graphs used in its taint analysis. The extension dumps a serialized call graph in JSON-format for each analyzed app. We ran the extended FLOWDROID on TAINTBENCH. The source sink configuration is at app-level as in the experiment TB3 in Table 2.12. We compared the methods appearing on the documented taint flows in TAINTBENCH to the edges in the call graph, and we found out that 70 out of 203 expected taint flows could not be detected, since relevant methods are not present in the call graphs.

Having incomplete call graphs is at least one reason why FLOWDROID failed to detect these flows. Figure 2.9 shows the proportions of true positives and false negatives from the evaluation of FLOWDROID on TAINTBENCH. Among the 70 false negatives due to incomplete call graphs, FLOWDROID could not even detect the entry points for 28 of them, thus no dummy main method was constructed at all. We call these false negatives the *Type-1* false negatives (Type-1 FNs). For the remaining 42, some relevant methods for these flows are missing in the call graph. We

Table 2.18: Type-1 and Type-2 false negatives in TaintBench.

(a) Type-1 False Negatives		
Benchmark App	No. of False Negatives	Flow ID
chulia	4	1, 2, 3, 4
fakeappstore	3	1, 2, 3
fakemart	2	1, 2
godwon_samp	6	1, 2, 3, 4, 5, 6
samsapo	4	1, 2, 3, 4
slocker_android_samp	5	1, 2, 3, 4, 5
sms_google	4	1, 2, 3, 4
Σ	28	
(b) Type-2 False Negatives		
Benchmark App	No. of False Negatives	Flow ID
chat_hook	1	12
fakedaum	1	1
fakeplay	2	1, 2
hummingbad_android_samp	2	1, 2
overlaylocker2_android_samp	6	1, 2, 3, 4, 5, 6
remote_control_smack	17	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17
repane	1	1
save_me	1	23
scipix	2	1, 2
smsstealer_kysn_assassincreeed_android_samp	2	4, 5
the_interview_movieshow	1	1
vibleaker_android_samp	4	1, 2, 3, 4
xbot_android_samp	2	1, 2
Σ	42	

call these 42 false negatives *Type-2* false negatives (Type-2 FNs). The respective benchmark apps of Type-1 and Type-2 false negatives are listed in Table 2.18. We denote the other 90 false negatives as *Type-3* false negatives (Type-3 FNs) and the reasons why these flows were not detected by FLOWDROID are still unknown.

Cause of Type-1 FNs To understand why FLOWDROID could not detect any entry point in the benchmark apps of Type-1 FNs, we debugged into FLOWDROID as it analyzed these apps. We found out that when FLOWDROID parses the `AndroidManifest.xml` file to search entry point classes, it uses the method `ProcessManifest.checkAndAddComponent()` to validate whether an entry point class is in system namespaces such as `android`, `com.google`, etc. The validation method `SystemClassHandler.isClassInSystemPackage` is shown in Listing 2.5, which simply compares the prefix of the class name to a few hard-coded namespaces. Although Google Play forbids the namespaces `com.example` and `com.android` [Goo21], these namespaces are not forbidden in the build process of Android apps. Malware apps often use such namespaces to bypass security analysis tools. For example, the benchmark app `godwon_samp` uses `android.sms.core` as its package name, which is absolutely legal. There has been some discussions under this GitHub issue¹³ regarding this problem. The main author of FLOWDROID Steven Arzt explained “the general idea behind the heuristic was to save analysis time and memory by not looking into components that have not been implemented by the app developer, but that are rather part of the Android framework or some commonly used library.” However, this heuristic can be leveraged easily by malware apps, as we have shown. Thus, we propose to consider all components as entry point classes if no entry point class could be detected using the current heuristic, since it is likely that the analyzed app is malware. A pull request¹⁴ regarding this change has been merged into the develop branch of FLOWDROID.

¹³<https://github.com/secure-software-engineering/FlowDroid/issues/171>

¹⁴<https://github.com/secure-software-engineering/FlowDroid/pull/329>

```

1 public static boolean isClassInSystemPackage(String className) {
2     return className.startsWith("android.") || className.startsWith("java.")
3     || className.startsWith("javax.") || className.startsWith("sun.")
4     || className.startsWith("org.omg.") || className.startsWith("org.w3c.dom.")
5     || className.startsWith("com.google.") || className.startsWith("com.android.");
6 }

```

Listing 2.5: The validation method `SystemClassHandler.isClassInSystemPackage` checks if a class is in system packages.

2.6 Threats to Validity

The *external* validity of the TAINTBENCH suite is threatened by the fact that we were forced to exclude obfuscated applications. Also, while all taint specifications have been checked multiple times by at least three authors, there is the potential threat that the baseline definition nonetheless misses some actual taint flows (expected cases).

The *internal* validity of the TAINTBENCH suite is impeded by the time that has passed since the malware apps were created. These malware apps do not target the latest Android API level. Nevertheless, we were able to install and execute almost all of the benchmark apps¹⁵ on a Nexus 4 Android emulator with API level 25. Anyway many apps composed in the TAINTBENCH suite are not functional anymore, since they have been communicating with public servers which are not accessible anymore. Most likely the servers were actively taken down when the apps were identified as malware by researchers and field experts. Because of this, only parts of TAINTBENCH can be used to benchmark dynamic taint analysis tools. This in turn embodies another reason, why dynamic tools could not be used to verify our baseline definition.

Note, this work focuses on the construction and evaluation of our novel benchmark suite; an in-depth investigation of reasons for our findings in the tools’ implementations exceeds the scope of this work. During our experiments we noticed that FLOWDROID’s results are non-deterministic (two runs of FLOWDROID(*) with the same inputs produce different results). This is a known issue first mentioned by Benz et al. [BKKL⁺20], which has not yet been fixed. This issue is hardly notable in DROIDBENCH experiments but well recognizable when analyzing larger apps as in TAINTBENCH.

Regarding our experiments using TAINTBENCH, we are aware of the internal threat caused by the measurement used in REPRODROID. If there are two source or sink statements at different positions in the source code (e.g., `s=source()` at line 5 and `s=source()` at line 10 in a method `foo()`), which look identical in Jimple, REPRODROID cannot distinguish them due to missing exact code locations in the results produced by the taint analysis tools. Thus, if one of them is detected, REPRODROID regards both flows as detected. In consequence, the actual recall on TAINTBENCH might be lower than observed. This issue could be mitigated if the analysis tools were to include unique statement identifiers e.g., line numbers in their results. We added this functionality to FLOWDROID for future releases. The related pull request is already merged.¹⁶

2.7 Conclusion

In this chapter, we first proposed a catalog of criteria for constructing real-world benchmark suites for Android taint analysis. We introduced the TAINTBENCH framework, which allows tool-assisted benchmark suite construction, evaluation and inspection. Using this framework,

¹⁵Except the app `cajino_baidu`

¹⁶<https://github.com/secure-software-engineering/FlowDroid/pull/222>

we constructed, based on the criteria, the first real-world malware benchmark suite with a documented baseline: TAINTBENCH (39 malware apps with 203 expected and 46 unexpected taint flows documented in a machine-readable format). We compared TAINTBENCH to DROIDBENCH with respect to various aspects. Evaluation of current and previously evaluated versions of FLOWDROID and AMANDROID using TAINTBENCH reveals insights that could not be gained with the micro benchmark DROIDBENCH. The associated experiments revealed surprising facts about taint analysis tools:

- Android taint analysis tools have difficulties in detecting real-world taint flows in malware apps, yielding very low recall.
- For AMANDROID the situation is particularly bad: the latest version detects almost no taint flow.
- While FLOWDROID shows better recall, a configuration using superfluous sources and sinks that are actually irrelevant for specific taint flows has a shadow effect on its taint computation, causing it to miss some actual flows.
- For both AMANDROID and FLOWDROID, new tool releases are less accurate than their predecessors: as we have shown in Experiments 2, 3 and 4, new releases have lower precision, recall and F-measure.
- Our in-depth investigation on FLOWDROID reveals that many false negatives were due to incomplete call graphs.

Incomplete call graphs as one main cause of the false negatives motivated us to design a new call graph construction approach that aims to produce more complete call graphs. This work will be introduced in the next chapter.

GenCG: A General Approach to Modeling Java Framework Behaviors

In the previous chapter, we introduced the benchmark suite TAINTBENCH for real-world Android taint analysis benchmarking. Our evaluation of popular static analysis tools using TAINTBENCH has shown that these tools have shortcomings in detecting real-world issues. Our investigation on FLOWDROID reveals that incomplete call graphs are one major reason why FLOWDROID failed to detect malicious taint flows in TAINTBENCH. Call graphs are major building blocks for static analyses. To be scalable, most static analysis tools including FLOWDROID choose to construct application-only call graphs and carefully model the behavior of frameworks. However, such carefully crafted models often produce incomplete call graphs, and are impractical to do for every framework, since that would require careful study of each framework’s documentation and even the source code. This chapter addresses this problem and answers the fundamental research question: *How can one construct sound application-only call graphs without precisely modeling the Java frameworks?* Especially, the constructed call graphs should allow a precise client taint analysis effectively finding real-world issues.

In this chapter, we present GENCG, a general approach to modeling Java framework behavior. 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 we created shows promising results.

The outline of the chapter is as follows: Section 3.1 motivates the work with an example of malicious taint flow from TAINTBENCH which both FLOWDROID and AMANDROID failed to report. Necessary background is introduced in Section 3.2. In Section 3.3, we discuss problems of an existing approach on which our work is based. Section 3.4 explains how our approach addresses these problems. Section 3.5 presents the application of our approach on the Android framework and its evaluation with a client taint analysis. Section 3.6 discusses the application of our approach to the Spring framework. We compare our work to existing work in Section 3.7 and

discuss the limitations in Section 3.8. Section 3.9 concludes the chapter. Most work described in this chapter is unpublished. An early conception of the work is published in my ESEC/FSE paper for the ACM Student Research Competition [Luo21] and won the second place.

3.1 A Motivating Example

Listing 3.1 shows a data leak from the `fakedaum` app in our TAINTBENCH suite. Two Android components are involved in this leak: an activity `MainActivity` (line 1-15) and a service `TaskService` (line 24-38). This leak leverages the Inter-Component Communications (ICC) to exchange sensitive data between the activity and the service.

The activity `MainActivity` reads user’s last known location in the lifecycle method `onStart()` (line 6) and stores it into the field `msg` (line 7). Later in the lifecycle method `onPause()` of the activity, the `msg` containing the location is passed to an intent that starts the service `TaskService` (line 12-13).

The service `TaskService` uses a handler `PushMessageHandler` to perform asynchronous tasks. The `msg` containing the location is read from the intent and is encapsulated in a `Message` object for the handler (line 31-32). This `Message` object is sent to the handler through the `sendMessage(Message)` method (line 33) and is processed later in the `handleMessage(Message)` method called by the Android framework. In the `handleMessage(Message)` method, the last known location is leaked to a malicious server (line 53).

To detect this leak, a taint analysis tool needs to:

1. generate a sound call graph that captures all calling relationships on the data-flow path of this leak;
2. handle the ICC-link between the activity and the service;
3. analyze (i. e., analyze the code of the callee) or understand (i. e., does not analyze the code of the callee, but considers the effect of the call) library calls that can generate or kill taints.

The first requirement is crucial, because missing call edges will break the taint propagation of the analysis and result in false negatives. The essential subgraph of the actual call graph of the motivating example is illustrated in Figure 3.1. Because the call graph itself does not encode the order of the calls, we numerate the nodes in this graph to indicate the order when the call happens at runtime. As we can see in Figure 3.1, some of the edges in the call graph are coming from the Android framework. Taint analysis tools typically choose to create models of the Android framework rather than analyzing the framework code itself. This makes the completeness of the model very important. In the motivating example, a handler `PushMessageHandler` is used to schedule tasks to be executed on a separate thread than the main UI thread. The `sendMessage` method at line 33 enqueues a message containing sensitive data that will be processed by the `PushMessageHandler.handleMessage()` method. The call to this `handleMessage` method from the Android framework is not introduced by FLOWDROID’s model of Android. In Figure 3.1, the red dashed edges denote missing calls caused by this incomplete modeling. In the following section, we introduce how FLOWDROID models the Android framework and other necessary background.

```

1 public class MainActivity extends Activity {
2     private Msg msg = new Msg();
3     public void onCreate(Bundle savedInstanceState) {...}
4     public void onStart() {
5         //...
6         Location loc = lm.getLastKnownLocation("network"); // 1. source: read last known location
7         this.msg.setContent(loc.toString()); // 2. store location to field msg
8     }
9     public void onPause() {
10        super.onPause();
11        Intent intent = new Intent(this, TaskService.class);
12        intent.putExtra("data", this.msg); // 3. store location to intent
13        startService(intent); // 4. start TaskService
14    }
15 }
16 public class Msg {
17     private String c;
18     public void setContent(String str){
19         this.c = str;
20     }
21 }
22 public class TaskService extends Service {
23     private LooperThread looperThread;
24     private Handler handler;
25     public void onCreate() {
26         //...
27         looperThread.start();
28     }
29     public int onStartCommand(Intent intent, int flags, int startId) {
30         handler = looperThread.handler;
31         Msg m = (Msg) intent.getSerializableExtra("data"); // 5. get location from the intent.
32         Message msg = handler.obtainMessage(1000, m); // 6. encapsulate location into a Message object
33         handler.sendMessage(msg); // 7. send the msg with location to a handler
34         return super.onStartCommand(intent, flags, startId);
35     }
36 }
37 public class LooperThread extends Thread {
38     private Looper looper;
39     private Context context;
40     protected Handler handler;
41     public void run() {
42         //...
43         handler = new PushMessageHandler(context, looper);
44     }
45 }
46 public class PushMessageHandler extends Handler {
47     String url = "http://103.30.7.178/upMsg.htm";
48     public void handleMessage(Message msg) {
49         try {
50             List<NameValuePair> pars = new ArrayList<>();
51             pars.add(new BasicNameValuePair("loc", new Gson().toJson(msg.obj)));
52             httpPost.setEntity(new UrlEncodedFormEntity(pars, "UTF-8"));
53             httpClient.execute(httpPost); // 8 sink: send the last known location to a malicious url.
54         } catch (...) {...}
55     }
56 }

```

Listing 3.1: The motivating example: in `MainActivity`, the last known location is stored in the intent that starts the background service `TaskService`. Once the `TaskService` is started, the last known location is read from the intent and leaked to a malicious server through the handler `PushMessageHandler`.

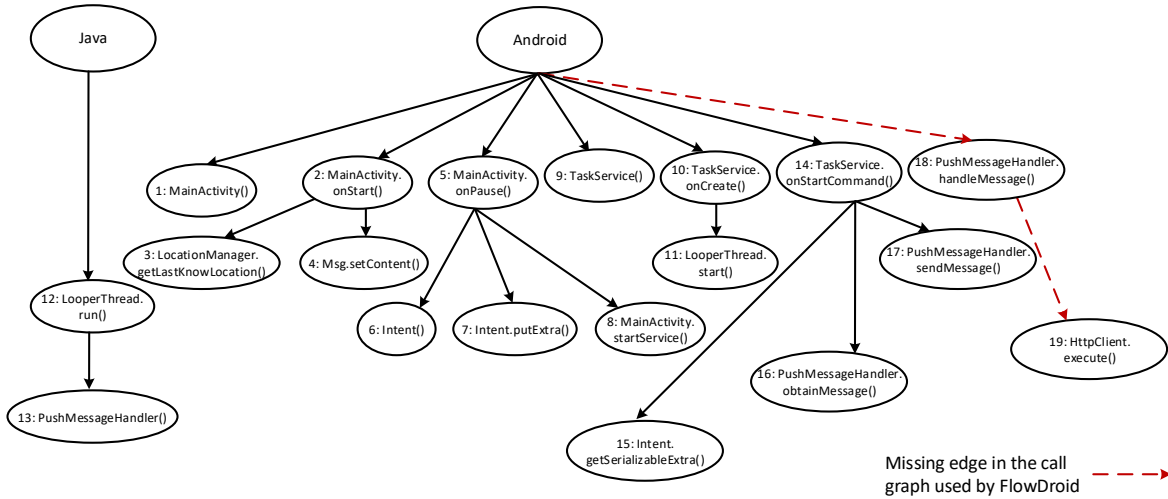


Figure 3.1: Essential subgraph of the actual call graph of the motivating example. See a full-page version of this figure in Figure B.1.

3.2 Background

In this section, we first introduce how existing approaches model the features of the Android framework, such as callbacks, component lifecycle and inter-component communication. Second, we explain how FLOWDROID uses summaries to analyze library methods with the motivating example. Lastly, we introduce AVERROES [AL13], which inspired our work.

3.2.1 Entry Points and Lifecycle Modeling

FLOWDROID models the lifecycle of Android components and creates a `dummyMainMethod` that creates objects for each component class such as `Activity`, `Service`, `BroadcastReceiver`, etc. and calls its lifecycle methods according to the documentation of the Android framework. The red edges in Figure 3.1 are caused by this model being incomplete. The `dummyMainMethod` generated by FLOWDROID for the motivating example is depicted in Figure 3.2. It calls a generated `DummyMainClass.dummyMainActivity()` method that uses opaque predicates p (predicates that will not be evaluated statically) to model the control flows in `MainActivity`’s lifecycle. It also adds UI callbacks that are defined in the app’s layout XML files to this modeled lifecycle. Although this approach precisely models possible transitions in the Android lifecycle, it often misses callback methods if the corresponding library class is not in the list `AndroidCallbacks.txt` used by FLOWDROID.

FLOWDROID also patches some commonly used library classes such as `java.lang.Thread` and `android.os.Handler` with mock implementations. This produces fake call-graph edges such as from `Thread.start()` to `Thread.run`. Unfortunately, the list of to be patched methods is also incomplete, resulting missing edge to the `handleMessage` method in the motivating example. This precise modeling of the framework behavior is hard to keep up to date with the development of the Android framework with its frequent releases. Moreover, this kind of approaches is limited to the modeled framework and often tailored for one specific analysis tool. In the domain of Java enterprise applications, popular frameworks like Java EE, Spring and Apache Struts are rarely modeled or supported by static analysis frameworks, as pointed by Antoniadis et al. [AFK⁺20]. We need a framework-independent and reusable approach for generating sound call graphs.

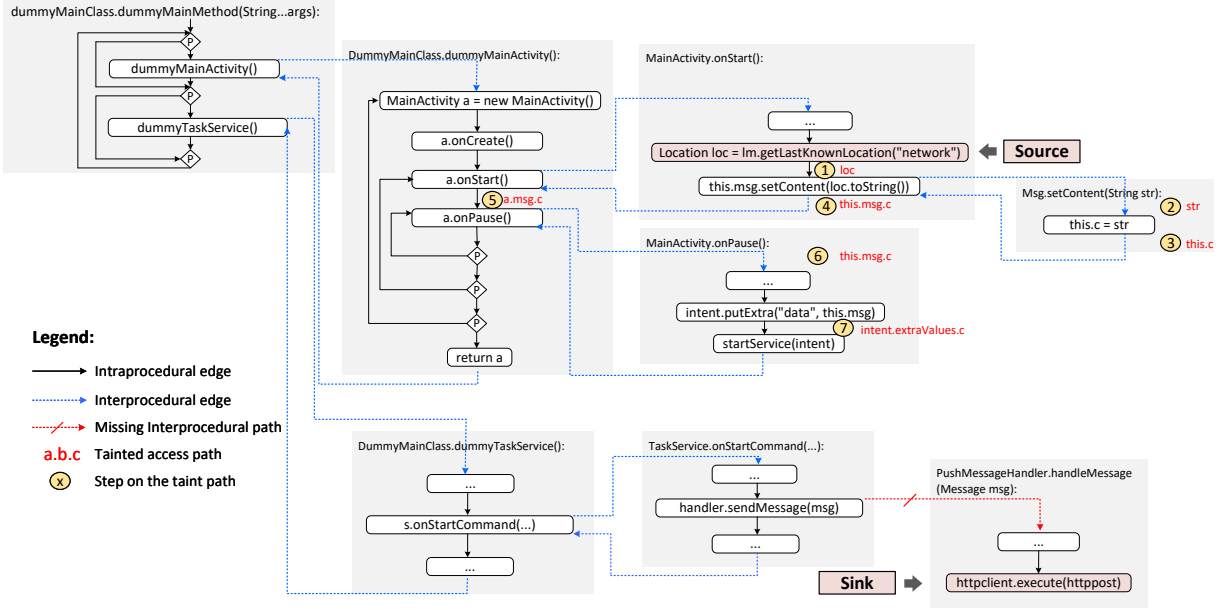


Figure 3.2: FLOWDROID’s taint analysis on generated interprocedural CFG for the motivating example. See a full-page version of this figure in Figure B.2.

3.2.2 Inter-Component Communication

The Android system runs every app in a distinct sandbox and supports multiple ways for apps and app components to communicate. *Intents* are message objects one can use to facilitate communications between app components. The data leak in our motivating example uses an *intent* to exchange data between the activity and service. As pointed out by Li et al. in their ICCTA paper [LBB⁺15], malware apps often use this mechanism to bypass analysis tools. FLOWDROID is not able to detect such ICC-leaks. ICCTA filled the gap by extracting the ICC links with the tool IC3 [OLD⁺15] and connecting the components in the Jimple (see Section 2.4.1) representation of the app. Although this approach enables data-flow analysis between different Android components, it is still tedious and requires manual effort for maintenance. However, both ICCTA and IC3 have not been actively maintained in the past years. For the *fakedaum* app of the motivating example, IC3 crashed as we used it to extract the ICC links. Since ICCTA is built on top of FLOWDROID, it inherits the limitation of FLOWDROID. As the authors also mentioned in their paper, ICCTA only considers the common ICC methods, thus its modeling of the inter-component communication is also incomplete.

3.2.3 Analysis of Library Methods

Modern software applications, including Android apps, heavily rely on libraries. For a taint analysis tool to be precise, it must consider the effect of libraries (the Android framework as well as the third-party libraries) rather than just analyzing the application code. To be scalable, FLOWDROID doesn’t analyze the library code, but precisely models the taint propagation through a subset of library APIs. Its *EasyTaintWrapper* component provides hard-coded models for the most commonly used APIs from the Java collection classes and the String class. Such an approach misses less-used APIs and requires large manual effort. Thus, STUBDROID was proposed as an alternative in FLOWDROID to handle library APIs [AB16]. STUBDROID automatically analyzes library APIs and infers a taint propagation model for each analyzed API. The

```

1 class A1 extends L1{
2     //This method can be invoked by L2.bar.
3     @Override
4     public void foo(){...}
5 }
6 class A2{
7     public void doSth(){
8         A1 a = new A1();
9         L2 lib = new L2();
10        //pass an object of A1 to the library.
11        lib.bar(a);
12    }
13 }

```

(a) Application Code

```

1 public abstract class L1{
2
3     abstract void foo(){...}
4
5 }
6 public class L2{
7
8     void bar(L1 cls){
9         //cls points to the A1 object created in A2.
10        cls.foo();
11    }
12
13 }

```

(b) Library Code

Figure 3.3: A case limited by the *method calls* constraint.

resulting data-flow mappings of the library APIs are stored into summary files. Listing 3.2 shows the summary of the `putExtra` method from the `android.content.Intent` class in XML-format. These generated summary files can be used later in the taint analysis when corresponding library APIs are used in the application. For example, at step ⑦ in Figure 3.2, FLOWDROID uses the summary from Listing 3.2 (line 9-14) and generates a new taint `intent.extraValues.c` from the tainted parameter `this.msg.c`. STUBDROID was applied on the Android SDK (version 4.3) and Oracle Java SDK (version 1.8). Although it generated summaries for most important APIs in both Android and Java SDK and saves manual work, it still misses some APIs, including the factory method `obtainMessage` from the `android.os.Handler` used in the motivating example (see line 32 in Listing 3.1).

```

1 <method id="android.content.Intent.putExtra(java.lang.String,java.io.Serializable)">
2     <flows>
3         <flow isAlias="false" typeChecking="false">
4             <from sourceSinkType="Parameter" ParameterIndex="0" />
5             <to sourceSinkType="Field"
6                 AccessPath="[android.content.Intent: java.lang.String[] extraKeys]"
7                 AccessPathTypes="[java.lang.String[]]" /> />
8         </flow>
9         <flow isAlias="false" typeChecking="false">
10            <from sourceSinkType="Parameter" ParameterIndex="1" />
11            <to sourceSinkType="Field" BaseType="android.content.Intent"
12                AccessPath="[android.content.Intent: java.lang.Object[] extraValues]"
13                AccessPathTypes="[java.lang.Object[]]" /> />
14        </flow>
15        <flow isAlias="true" typeChecking="false">
16            <from sourceSinkType="Field" />
17            <to sourceSinkType="Return" />
18        </flow>
19    </flows>
20 </method>

```

Listing 3.2: Taint summary for `Intent.putExtra`.

3.2.4 Construction of Application-only Call Graphs with AVERROES

AVERROES [AL13] is a general approach to construct sound and reasonably precise call graphs for Java programs that only applies a shallow analysis to the concrete library code: it only examines the constant pool of each class. Based on collected references from the constant pool, AVERROES builds a class hierarchy of both application and library classes. This class hierarchy is then used to generate a placeholder library that approximates the possible behaviors of an original library. An existing whole-program call graph construction framework can use the placeholder library as replacement of the original library to construct a sound and precise application call graph.

The placeholder library is generated based on the *separate compilation assumption*. This assumption states that all of the library classes are compiled in the absence of the application classes. As a result, this limits what the library classes can interact with the application classes, e. g., what kinds of objects the library can hold references to, which methods in the application code can be called by the library under which circumstance. Eight constraints that followed from this assumption were introduced with AVERROES. The most relevant ones for our work are the two constraints *local variables* and *method calls*. In the following, we explain them by focusing on how library code can interact with the application code and omit how library code can interact with itself.

The *local variables* constraint says that the local variables in the library can point to objects of application classes that are (1) instantiated by the application and then passed into the library due to inter-procedural assignments (2) stored in fields accessible by the library code, (3) or whose runtime type is a subtype of `java.lang.Throwable`. Based on this constraint AVERROES uses a field called `libraryPointsTo` to abstract all the local variables in the library.

The *method calls* constraint limits the methods in the application code that can be invoked by the library: a method that is non-static and overrides method of a library class and the library holds reference to an object of the method’s declaring class or its subclass. This constraint basically says that there should be a subtyping relation between the application class and the library class, as the example in Figure 3.3 shows. In this example, the reference to an object of an application class is held by the library as it is passed as an argument of a call to the library method `L2.bar`. This is the case (1) limited by the *local variables* constraint.

Based on those constraints from the separate compilation assumption, AVERROES generates a placeholder jar that models what the library could potentially do. Although AVERROES was targeting pure Java applications, the separate compilation assumption is also applicable for Android applications. We adapted AVERROES to analyze Android apps and generate a placeholder library for the Android framework (i. e., `android.jar`). Since AVERROES only needs to build a class hierarchy, the `android.jar` with stub methods distributed in the Android SDK is sufficient for this purpose. In the following, we explain with the motivating example how AVERROES works.

The `doItAll()` method The key concept in AVERROES that simulates most of the library behaviors is the `doItAll()` method in the `Library` class generated by AVERROES. This method implements library behaviors such as class instantiation and library callbacks. The library callbacks part is based on the subtyping relationship between the application classes and the library classes described in the *method calls* constraint we introduced above. Listing 3.3 shows the `doItAll()` method (line 13-41) for our motivating example generated by AVERROES. In line 16-23, the code simulates the behavior that the library can instantiate each concrete library or application class via reflection. Following the method calls constraint, code in line 24-38 calls all methods of library classes (classes from `android.jar`) that are overridden in application classes (classes from the `apk`). AVERROES also generates code (hidden in line 39) for simulating writing objects into any array element that is pointed to by the library and exceptions that can be thrown by the library.

Library Points To The `android.jar` distributed by Google in the Android SDK contains only stub methods, which only raise `NotImplementedExceptions`. The actual implementation is in devices that run the Android operating system. In this case, the actual library code is usually not available for call graph construction. Without analyzing the statements in the library code, a sound call graph construction algorithm should assume that the unanalyzed code could do anything—creating objects for any type, calling any method, assigning any value to any field,

etc. Yet a call graph constructed based on this conservative assumption is too imprecise. In fact, the library can only call methods on objects that it has a reference to. In AVERROES, the public field `libraryPointsTo` is for storing such information in the placeholder library. The field `libraryPointsTo` of type `java.lang.Object` in class `AbstractLibrary` class is an abstraction that represents all local variables in the original library code. It points to every object that could be assigned to a local variable in the library code. Intuitively, the library can hold the reference of an object due to two cases: (1) the object is created by the library (2) the reference of the object is passed as an argument for a call to a library method in the application code.

```

1 public class averroes.Library extends averroes.AbstractLibrary{
2     public static void <clinit>(){
3         averroes.Library r0;
4         averroes.AbstractLibrary $r1;
5         r0 = new averroes.Library;
6         specialinvoke r0.<averroes.Library: void <init>()>();
7         $r1 = <averroes.AbstractLibrary: averroes.AbstractLibrary instance>;
8         $r1.<averroes.AbstractLibrary: java.lang.Object libraryPointsTo> = r0;
9         <averroes.AbstractLibrary: averroes.AbstractLibrary instance> = r0;
10        return;
11    }
12
13    public void doItAll(){
14        r0 := @this: averroes.Library;
15        r1 = <averroes.AbstractLibrary: averroes.AbstractLibrary instance>;
16        // class instantiation
17        r2 = new example.MainActivity;
18        specialinvoke r1.<example.MainActivity: void <init>()>();
19        r1.<averroes.AbstractLibrary: java.lang.Object libraryPointsTo> = r2;
20        r3 = new example.TaskService;
21        specialinvoke r3.<example.TaskService: void <init>()>();
22        r1.<averroes.AbstractLibrary: java.lang.Object libraryPointsTo> = r3;
23        //...
24        // library callbacks
25        r4 = r1.<averroes.AbstractLibrary: java.lang.Object libraryPointsTo>;
26        r5 = (android.os.Handler) r4;
27        r6 = r1.<averroes.AbstractLibrary: java.lang.Object libraryPointsTo>;
28        r7 = (android.os.Message) r6;
29        virtualinvoke r5.<android.os.Handler: void handleMessage(android.os.Message)>(r7);
30        r8 = r1.<averroes.AbstractLibrary: java.lang.Object libraryPointsTo>;
31        r9 = (android.app.Service) r8;
32        virtualinvoke r9.<android.app.Service: void onCreate()>();
33        r10 = r1.<averroes.AbstractLibrary: java.lang.Object libraryPointsTo>;
34        r11 = (android.app.Service) r10;
35        r12 = r1.<averroes.AbstractLibrary: java.lang.Object libraryPointsTo>;
36        r13= (android.content.Intent) r12;
37        virtualinvoke r11.<android.app.Service: int onStartCommand(android.content.Intent,int,int)>(r13, 1,
38            1);
39        //...
40        // array element writes and exception handling
41    }
42 }

```

Listing 3.3: The `Library.doItAll()` method generated by AVERROES.

The first case is modeled in the `doItAll()` method, as shown in line 19 and line 22 in Listing 3.3, variables holding the references of newly created objects are assigned to `libraryPointsTo`. The second case is modeled in the placeholder method AVERROES generates for each referenced library method. Every placeholder method contains parameter assignments that assign param-

eters to the `libraryPointsTo` field as shown in line 9 in Listing 3.4 for the library method `Handler.obtainMessage()`. Line 11 models the fact that the library also points to the current object (the `this` reference) on which the method is called on. Moreover, every library method can potentially call other methods, create objects, etc. These side effects are modeled by calling the `doItAll()` method of class `AbstractLibrary` as shown in line 12 in Listing 3.4.

```

1 public final android.os.Message obtainMessage(int, java.lang.Object){
2     averroes.AbstractLibrary r0;
3     android.os.Message r1, r3;
4     android.os.Handler r2;
5     r2 := @this: android.os.Handler;
6     r1 := @parameter0: android.os.Object;
7     r0 = <averroes.AbstractLibrary: averroes.AbstractLibrary instance>;
8     // parameter assignment
9     r0.<averroes.AbstractLibrary: java.lang.Object libraryPointsTo> = r1;
10    // assign the current object
11    r0.<averroes.AbstractLibrary: java.lang.Object libraryPointsTo> = r2;
12    virtualinvoke r0.<averroes.AbstractLibrary: void doItAll()>();
13    r3 = (android.os.Message) r0.<averroes.AbstractLibrary: java.lang.Object libraryPointsTo>;
14    return r3;
15 }
```

Listing 3.4: The placeholder Jimple method body generated by AVERROES for the library method `Handler.obtainMessage`.

3.3 Existing Problems with AVERROES’s Model

Although the placeholder library generated by AVERROES can be used to construct a sound and precise application-only call graph for normal Java programs, one could not directly use it for analyzing Android apps or Web apps. We are particularly interested in taint analysis, below we list the problems that limit applying AVERROES directly for a client taint analysis:

Problem 1. AVERROES uses a single `libraryPointsTo` field to represent all objects that the library holds references to is too imprecise for a field-sensitive taint analysis: once the `libraryPointsTo` field is tainted, it could be propagated everywhere and potentially result in many false positives, since the `Library.doItAll()` method is called in every placeholder method.

Problem 2. Framework-based applications often do not have a main method, as the application’s flow of control is usually dictated by the framework. One could use the `Library.doItAll()` method generated by AVERROES as the main entry point. However, `Library.doItAll()` contains no control flow at all, instead callbacks are invoked in arbitrary order. An inter-procedural flow-sensitive taint analysis would miss the data leak in the motivating example (see Section 3.1), if the relevant callbacks are invoked in a wrong order in `Library.doItAll()`.

Problem 3. Repeatedly created objects could lead to false negatives. Figure 3.4 shows an example explaining this problem. On the left, it shows the placeholder method generated for the library method `Handler.sendMessage`. As in the motivating example, the parameter of `Handler.sendMessage` is tainted, then the `libraryPointsTo` field ought to be tainted when a field-sensitive taint analysis analyzes the placeholder method. This tainted field will be then propagated into the `Library.doItAll()` method as it is called in every placeholder method. On the right side, the `libraryPointsTo` field needs to stay tainted when it is assigned to

3.3 EXISTING PROBLEMS WITH AVERROES'S MODEL

```

1 class Handler{
2
3   void sendMessage(Message m){
4     AbstractionLibrary lib = AbstractLibrary.instance;
5     lib.libraryPointTo = this;
6     // taint the libraryPointTo field
7     lib.libraryPointTo = m;
8     lib.doItAll();
9   }
10 }

```

(a) Placeholder method code.

```

1 public class Library extends AbstractLibrary{
2   public void doItAll(){
3     // class instantiation ...
4     Message m1 = new Message();
5     Library.libraryPointsTo = m ; // strong update!
6     // library callbacks
7     Message m2 = (Message) Library.libraryPointsTo;
8     handler.handleMessage(m2);
9   }
10 }

```

(b) Overwritten tainted field.

Figure 3.4: Problem caused by repeatedly created objects (simplified Java code).

a local variable `m2` in `Library.doItAll()` such that `m2` is tainted when it is passed to the call `handler.handleMessage` (the sink is in this method). However, in `Library.doItAll()`, the `libraryPointsTo` field is overwritten by a newly created `Message` object. This means the points-to relationship needs to be removed and the `libraryPointsTo` field should not be tainted any more. This is the so called strong update [DD12], a precise taint analysis usually considers this. As a result, the data leak would be undetected.

Problem 4. AVERROES does not consider Java annotations and results in missing edges in the call graphs. Both Android and Spring frameworks support annotating methods in the application code that will be called reflectively by the framework. In Android, one can use annotation to declare a Java method that can be invoked in JavaScript code. This is the so-called bridge communication [LDR16]. Listing 3.5 and Listing 3.6 show a simplified example from the benchmark app `overlaylocker2_android_samp`. In Listing 3.5, a class `MeSettings` is declared with a method `getPhoneNumber()` annotated with `android.webkit.JavascriptInterface` (line 13-18)¹. At line 8, an instance of `MeSettings` is injected to the JavaScript environment by calling the method `WebView.addJavascriptInterface`. Whenever a web page is loaded on the `WebView` object, a new JavaScript object from the injected object will be created by the Android framework and is accessible with `window.MeSettings`. Invocation on `window.MeSettings` in the JavaScript code will invoke the corresponding method in Java class `MeSettings`.

```

1 class MainActivity extends Activity{
2   @Override
3   protected void onCreate(Bundle savedInstanceState) {
4     super.onCreate(savedInstanceState);
5     setContentView(R.layout.activity_main);
6     WebView webView = findViewById(R.id.webView);
7     webView.getSettings().setJavaScriptEnabled(true);
8     webView.addJavascriptInterface(new MeSettings(), "MeSettings");
9     webView.loadUrl("file:///android_asset/webview.html");
10  }
11 }
12
13 class MeSettings {
14   @android.webkit.JavascriptInterface
15   public String getPhoneNumber(){
16     return ((TelephonyManager) this.mContext.getSystemService("phone")).getLine1Number(); // source
17   }
18 }

```

Listing 3.5: Call Java function in JavaScript code (1).

¹For apps targeting SDK version higher than 16, one has to use the method annotation `android.webkit.JavascriptInterface` to declare methods that can be invoked by JavaScript code. Before that, even no annotation was required.

```

1 <!DOCTYPE html>
2 <html>
3 <head></head>
4 <body>
5   <button onclick="doMalicious()">Try it</button>
6   <script>
7     function doMalicious() {
8       var phoneNumber = window.MeSettings.getPhoneNumber();
9       var xhr = new XMLHttpRequest();
10      xhr.open("POST", "/postman", true);
11      xhr.setRequestHeader('Content-Type', 'application/json');
12      xhr.send(JSON.stringify({phone: phoneNumber})); // sink
13    }
14  </script>
15 </body>
16
17 </html>

```

Listing 3.6: Call Java function in JavaScript code (2).

In Spring, annotations are much more common. One can use it to declare entry point methods (e.g., `PostMapping`) and even class fields that need to be initiated (e.g., `Autowired`) by the framework. We will introduce more details about the Spring framework in Section 3.6.

3.4 The GENCG Approach

In previous section, we discussed why one could not directly use the placeholder library generated by AVERROES for constructing call graphs that can be used by a precise client taint analysis. In this section, we introduce the improvements we made to address the four problems we pointed out. We built our GENCG approach on top of AVERROES and aimed to support both Android apps and Web apps using frameworks like Spring. We refer to our modified version of AVERROES with AVERROES-GENCG. In this section, we focus on the support for Android apps.

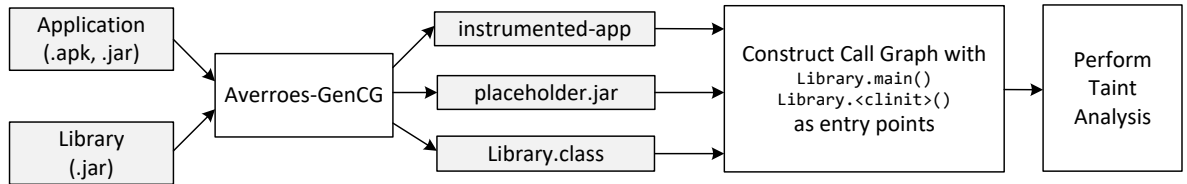


Figure 3.5: Overview of our GENCG approach.

Figure 3.5 shows the overview of our GENCG approach for constructing call graphs for taint analysis. Given an Android app, AVERROES-GENCG first takes both the `apk` file and the `android.jar` file that is shipped with the Android SDK (with stub methods) and generates a placeholder library for `android.jar` and `Library` class. It also generates an `instrumented-app` when certain annotations (see **Problem 4** in Section 3.3) are used in the original app. Otherwise, the `instrumented-app` is simply the original app. We will introduce this as **Improvement 4** in the following subsection.

3.4.1 Main Improvements

```

1      public class averroes.AbstractLibrary extends
      java.lang.Object{
2      public java.lang.Object libraryPointsTo;
3      public static averroes.AbstractLibrary instance;
4      public abstract void doItAll();
5      //default constructor
6      //...
7      }

```

Listing 3.7: The `AbstractLibrary` class generated by AVERROES.

```

1      public class averroes.AbstractLibrary extends
      java.lang.Object{
2      public android.os.Handler LPT_1;
3      public android.os.Message LPT_2;
4      public android.app.Service LPT_3;
5      public android.content.Intent LPT_4;
6      public android.app.Activity LPT_5;
7      public android.os.Bundle LPT_6;
8      public java.lang.Thread LPT_7;
9      public java.lang.Runnable LPT_8;
10     //...
11     public static averroes.AbstractLibrary instance;
12     public abstract void doItAll();
13     //default constructor
14     //...
15     }

```

Listing 3.8: The `AbstractLibrary` class generated by AVERROES-GENCG.

Improvement 1. To address **Problem 1** in Section 3.3, we introduce typed `libraryPointsTo` fields: instead of only using one field (`libraryPointsTo` in Listing 3.7) for all objects, for each type of object that library could point to, we create a field of this type in the `AbstractLibrary` class. We named such typed fields with names starting with the prefix `LPT`. Listing 3.8 shows our version of `AbstractLibrary` class for the motivating example. This can be unsound if objects are casted to other types in the application, i. e., a typed `LPT` field might hold references of other types of objects. But it is more precise than in the original AVERROES, as it does not pollute objects of other types, once a typed `LPT` gets tainted. A more fine-grained separation such as allocation sites is not possible without a deep analysis of both the application and library code, because allocation sites of objects that library can point to can be either in the application code or library code. The Android SDK with stub methods would not be sufficient. Technically, it is also very hard to generate the placeholder library methods, as a method could be called on different objects and with different arguments. Although our type-based separation might still introduce false positives, it turns out to work well for detecting real-world taint flows later in our evaluation.

Improvement 2. To address **Problem 2**, we introduce control flows into the `doItAll()` method as the CFG shown in Figure 3.6 for the motivating example. We do not precisely model which methods should be called in which order, as it requires to study the framework documentation carefully. Our goal is to add control-flow edges such that every possible call sequence is covered. This means also unrealizable call sequences, which could potentially introduce false positives. This is the trade-off for not studying each framework documentation manually. To achieve our goal, we introduce three kinds of edges with if-statements using a nondeterministic predicate p (i.e. `if(Math.random()<0.5)`): skip-method edges (in blue), skip-class edges (in red) and loop edges (in green). In reality, not every callback method is called every time when a library method is called as the placeholder method does, the skip-method edges allow cases that a callback method is not called. Similarly, all calls to methods of a class can be skipped by the skip-class edges. To simulate the effect that a method can be called multiple times in the library, we introduced the loop edges. The combination of skip-method edges and loop edges ensures that all possible orders of method calls are captured, including the lifecycle of Android component classes. For instance, in the part where service is simulated in Figure 3.6, although the `onStartCommand()` appears first in the control-flow graph, the path A-B-F-G-H-J-A-D-E-F-C represents the execution path in which `onCreate()` is called before `onStartCommand()` as defined in the service’s lifecycle.

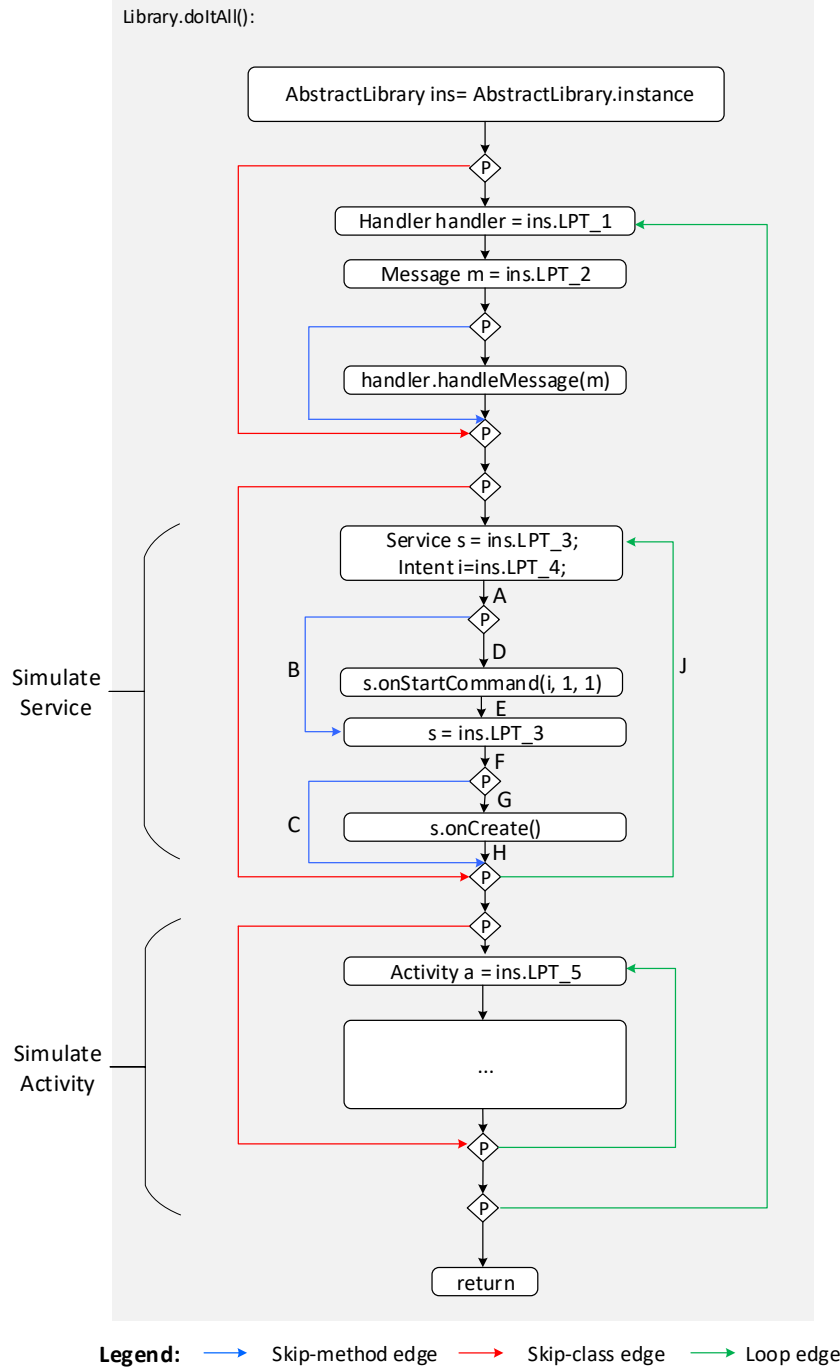


Figure 3.6: The CFG of `Library.doItAll()` by our AVERROES-GENCG. The corresponding code is in Listing 3.9.

Improvement 3. To address **Problem 3**, we move the class instantiation part in `doItAll()` to a separate `main()` method in the `Library` class (see Listing 3.9) to avoid unnecessary strong updates. The `main()` method will be taken by popular analysis frameworks by default as entry point and is only called once. Based on our experience with Android, most objects are only created once by the framework.

```

1 public class Library{
2     public static void main(String[] args){
3         AbstractLibrary r = AbstractLibrary.instance;
4         Activity a1 = new MainActivity();
5         r.LPT_5 = a1;
6         Service s1 = new TaskService();
7         r.LPT_3= s1;
8         Thread t1 = new LooperThread();
9         r.LPT_7 = t1;
10        r.LPT_8 = t1;
11        Handler h1 = new PushMessageHandler();
12        r.LPT_1 = h1;
13    }
14
15    public static Library(){
16        Library lib = new Library();
17        AbstractLibrary.instance = lib;
18    }
19
20    public void doItAll(){
21        AbstractLibrary ins = AbstractLibrary.instance;
22        do{
23            if(p){
24                Handler handler = ins.LPT_1;
25                Message m = ins.LPT_2;
26                if(p){
27                    handler.handleMessage(m);
28                }
29            }
30            if(p){
31                do{
32                    Service s = ins.LPT_3;
33                    Intent i = ins.LPT_4;
34                    if(p){
35                        s.onStartCommand(i, 1, 1);
36                    }
37                    s = ins.LPT_3;
38                    if(p){
39                        s.onCreate();
40                    }
41                } while(p);
42            }
43            if(p){
44                do{
45                    Activity a = ins.LPT_5;
46                    Bundle b = ins.LPT_6;
47                    if(p){
48                        a.onCreate(b);
49                    }
50                    a = ins.LPT_5;
51                    if(p){
52                        a.onStart();
53                    }
54                    a = ins.LPT_5;
55                    if(p){
56                        a.Pause();
57                    }
58                } while(p);
59            }
60        } while(p);
61    } }

```

Listing 3.9: The `Library` class (simplified code) generated by AVERROES-GENCG.

Improvement 4. To address **Problem 4**, our AVERROES-GENCG handles class, method and field annotations that are supported by the framework. These annotations (annotation class signatures) are stored in configuration lists used by AVERROES-GENCG and can be extended easily. For annotated methods and classes, AVERROES-GENCG creates artificial interfaces and instruments annotated classes to be subtypes of these interfaces. These artificial interfaces declare annotated methods and can be seen as a replacement for the annotations. This way it reduces the problem to the resolution of subtyping relationship that can be handled by the original AVERROES. AVERROES-GENCG will generate invocations of those annotated methods in the library callbacks part of the `doItAll()` method. For field annotations, AVERROES-GENCG also handles some common concepts. We will introduce the details about this in Section 3.6 when introducing the application of our approach on the Spring framework, in which annotations are heavily used.

In addition to these four main improvements, we remove the array elements writes and exception handling parts from the `doItAll()`. Our version of `doItAll()` only keeps the library callbacks part. This is unsound, however, we made this choice to avoid false positives caused by the assignments using the `libraryPointsTo` fields in the array elements writes and exception handling.

3.4.2 Sound and Precise Call Graph

Next, we explain how our AVERROES-GENCG allows call graph construction algorithms to produce sound and precise call graphs at least as good as AVERROES does. We focus on providing the intuition how the generated code is handled by the call graph algorithms in SOOT.

The biggest challenge to construct call graphs for object-oriented programs is dynamic dispatch: deciding the runtime type of the receiver of a polymorphic call. A conservative call graph algorithm Class Hierarchy Analysis (CHA) assumes that the receiver could be an object of any subclass of the declared type. Because call graphs constructed with CHA are too imprecise, other call graph algorithms attempt to improve the estimation of the runtime types of receivers. Rapid Type Analysis (RTA) considers object allocation sites and limits the type of a receiver to be only the types of object that have been instantiated in the program. For those two algorithms, there should be no difference in the call graphs using the code generated by our AVERROES-GENCG or the original AVERROES. More precise call graph construction algorithms compute the points-to sets that each variable could point to, which were AVERROES designed for. Variable Type Analysis (VTA) and Declared Type Analysis (DTA) refine call

Table 3.1: The four types of edges in SPARK’s pointer assignment graph (Table 1 in [LH03]).

Statement	Notation	Edge	Inference Rule
Allocation	$x = \text{new } C$	$\text{new } C \rightarrow x$	$\frac{\text{new}_x \rightarrow x}{\text{new}_i \in \text{pt}(x)}$
Assignment	$x = y$	$y \rightarrow x$	$\frac{\text{new}_y \in \text{pt}(y), y \rightarrow x}{\text{new}_y \in \text{pt}(x)}$
Field Store	$x.f = y$	$y \rightarrow x.f$	$\frac{\text{new}_y \in \text{pt}(y), \text{new}_x \in \text{pt}(x)}{\text{new}_y \in \text{pt}(\text{new}_x.f)}$
Field Load	$x = y.f$	$y.f \rightarrow x$	$\frac{\text{new}_y \in \text{pt}(y), \text{new}_x \in \text{pt}(\text{new}_y.f)}{\text{new}_x \in \text{pt}(x)}$

graphs by taking assignments into account. They consider only types that can possibly reach the call site. However, since these two approaches are only field-based, some imprecisions still remain. The most precise algorithm in SOOT is SPARK, which constructs call graphs on-the-fly as the points-to sets of call site receivers are computed. In the following, we demonstrate with the motivating example how the placeholder library generated by AVERROES-GENCG allows constructing sound and precise call graphs using SPARK.

SPARK consists of two steps: (1) build pointer assignment graph (PAG) for the program, (2) propagate the points-to sets along edges in PAG until a fixed point is reached. The pointer assignment graph is used in SPARK to represent the subset-based points-to information of the program. The PAG contains three types of nodes: allocation nodes, variable nodes and field reference node. Allocation nodes represent allocation sites in the program where new objects are created. Variable nodes represent local variables, method parameters, return values and

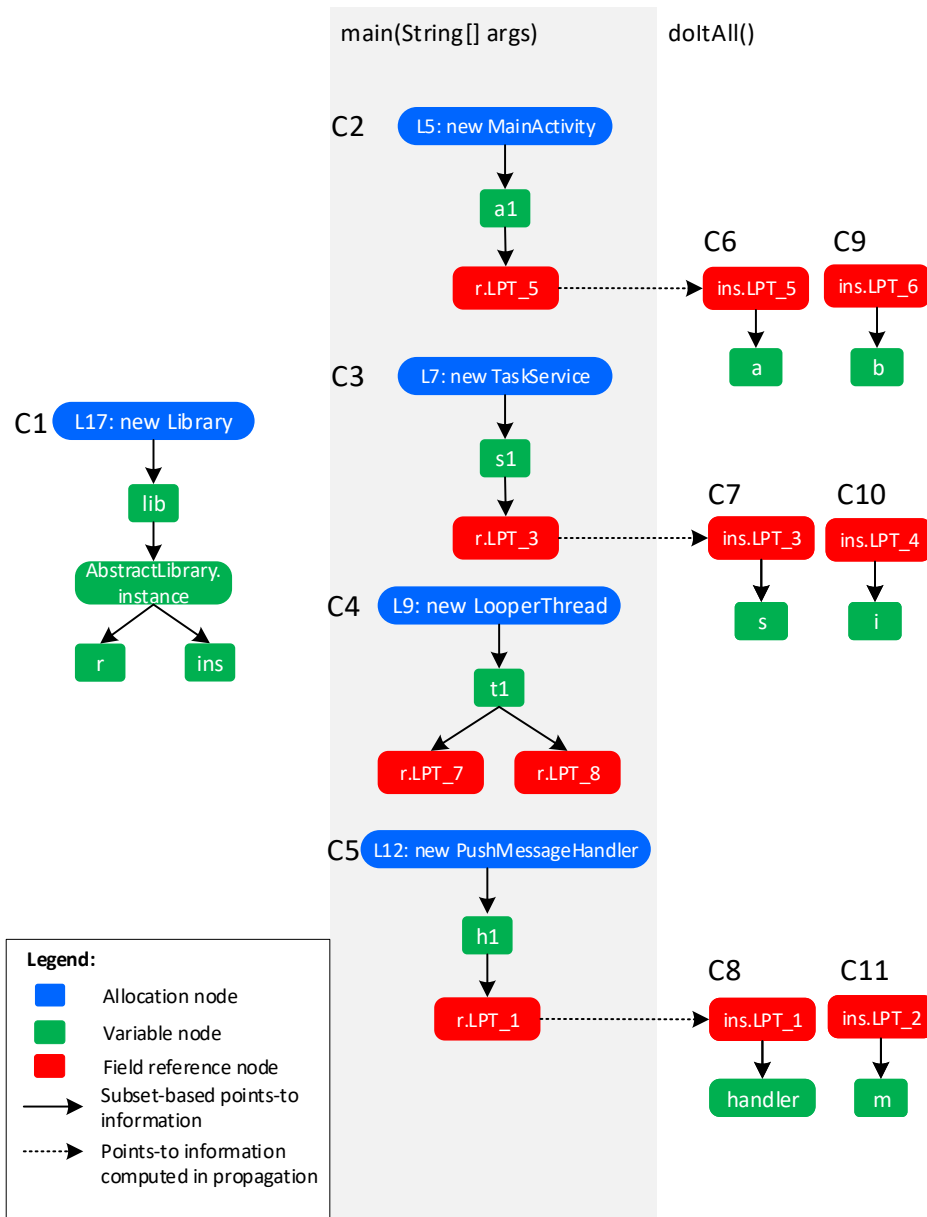


Figure 3.7: Pointer assignment graph for the *Library* class.

static fields. Field reference nodes represent field access expressions. The nodes in PAG are connected by four types of edges based on the rules listed in Table 3.1. The propagation of points-to sets is also based on these rules. An edge $y \rightarrow x$ in PAG models subset-based points-to constraint $pt(y) \subseteq pt(x)$, where $pt(y)$ and $pt(x)$ denote the points-to sets for y and x respectively. The PAG for the `Library` class generated by AVERROES-GENCG in Listing 3.9 is illustrated in Figure 3.7. We use solid edges to represent the edges in the initial PAG. This PAG contains 11 connected components C_1, \dots, C_{11} . The connect component C_1 on the left, starting from the allocation node for a `Library` object, is constructed based on the assignments at line 17-18, line 4 and line 22 in Listing 3.9. As we can see, variable node r from the method `main()` and node ins from the method `doItAll()` are both connected to the variable node `AbstractLibrary.instance`, which is a static field of the `AbstractLibrary` class. The propagation of points-sets in SPARK starts from allocation nodes. It is not hard to see that in the propagation, the allocation site L_{17} will be added to the points-to set of each node in C_1 , especially $pt(r) = pt(ins) = \{L_{17}\}$. Similarly, we will have $pt(a_1) = pt(r.LTP_5) = \{L_5\}$ in C_2 shown in the middle of the figure. Because $pt(r) = pt(ins) = \{L_{17}\}$ and $pt(r.LTP_5) = \{L_5\}$, the propagation will compute $pt(L_{17}.LTP_5) = \{L_5\}$. From $L_{17} \in pt(ins)$, we will get $pt(ins.LTP_5) = pt(L_{17}.LTP_5) = \{L_5\}$. Points-to information computed this way are displayed as dashed edges in the figure. Due to the edge $ins.LTP_5 \rightarrow a$ in C_6 , eventually L_5 will be propagated from $pt(ins.LTP_5)$ to $pt(a)$. The call graph is constructed based on the computed points-to sets. We know now that $pt(a) = \{L_5\}$ in the `doItAll()` method, then the targets of the calls `a.onCreate()`, `a.onStart()` and `a.Pause()` in line 49, 53 and 57 have to be in the `MainActivity` class, since L_5 stands for the allocation site of the `MainActivity` object created in the `main()` method. Because the typed LPT fields in the `AbstractLibrary` class are only assigned with either objects created in the `Library.main()` methods or objects from the application code that are passed as arguments of library calls (see Listing 3.4), there will only be edges from `doItAll()` to methods in the classes of those objects in the call graph.

The call graph for the motivating example using our placeholder library constructed by SPARK is shown in Figure 3.8. For simplicity, we only show the outgoing edges from methods of the `Library` class. The calling behavior of Android is simulated by the methods `Library.main()` and `Library.doItAll()`. In comparison to the actual call graph in Figure 3.1, we can see that for every outgoing edge starting from the Android node in Figure 3.1 to a node X , there is an outgoing edge starting from either `Library.main()` or `Library.doItAll()` to X in the call graph using our approach.

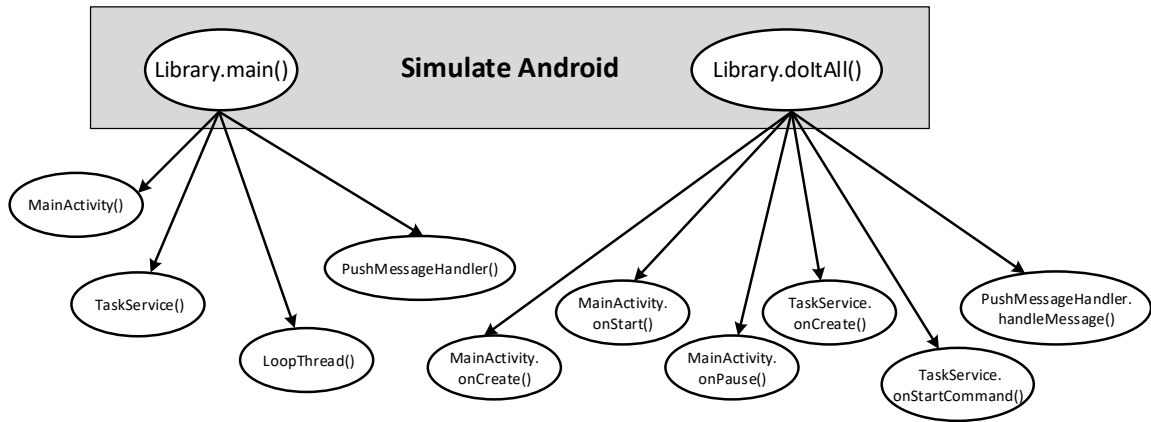


Figure 3.8: Call graph constructed with our GENCG approach.

3.4.3 Supporting Detection of ICC Leaks

Another benefit to use our approach is supporting detection of ICC leaks. In the following, we explain how the generated placeholder library by AVERROES-GENCG allows a field- and flow-sensitive taint analysis like FLOWDROID to detect ICC leaks using `Intent` to exchange sensitive data between different Android components.

To scale, FLOWDROID does not analyze methods from the Android framework nor the JDK. Instead, it uses data-flow summaries of these library methods to model the taint propagation over the library call sites. This has been proven to significantly improve the performance of FLOWDROID [AB16]. However, such approach is only sound whenever the set of summaries is complete, which is not the case in FLOWDROID. For example, there is no summary for library method `ContextWrapper.startService` that passes data to the Android framework which is then passed to the receiver method of the target component as a parameter by the Android framework. Because FLOWDROID’s `DummyMainMethod` simulates the lifecycle of each component to be independent from each other, it does not support passing data between different Android components.

In contrast, our `Library.doItAll()` method combined with the placeholder library methods enables intent data to flow from one component to any other component. Consider the motivating example, Figure 3.9 shows how an intent created in one component (`MainActivity`) is passed to another intent used in a second component (`TaskService`). The key is the field store to `ins.LTP_4` in the placeholder method and field load from `ins.LTP_4` in the `Library.doItAll()` method (highlighted in yellow). `ins.LTP_4` is the typed LPT field for `Intent` generated by our AVERROES-GENCG. It is obvious to see in the figure, that the intents used in both components are aliased to `ins.LTP_4`. If the intent in `MainActivity` is tainted, a field- and flow-sensitive taint analysis like FLOWDROID should also taint `ins.LTP_4` and the intent in `TaskService` when performing the analysis.

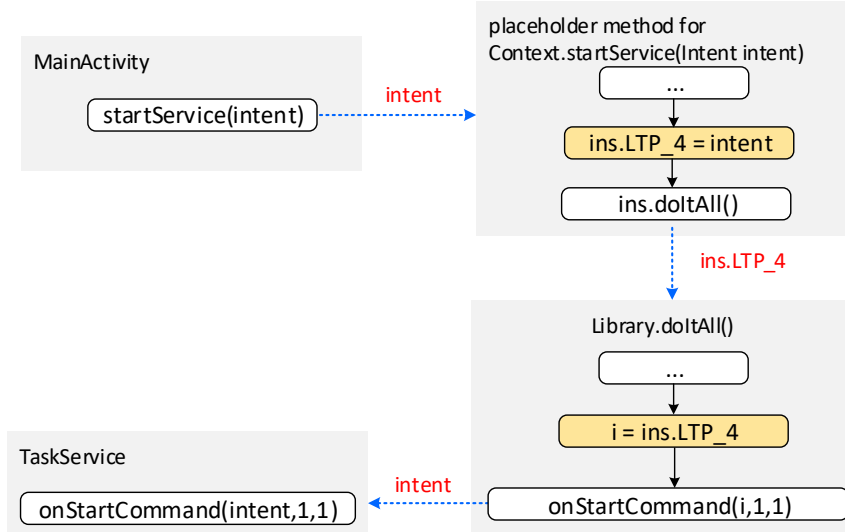


Figure 3.9: How an intent is passed between two components of the motivating example.

We adapted FLOWDROID’s taint analysis to use the call graph generated by our approach and refer to this client FLOWDROID^{Gen}. FLOWDROID^{Gen} takes existing summaries as FLOWDROID does, but also analyzes placeholder library methods from the Android SDK, of which

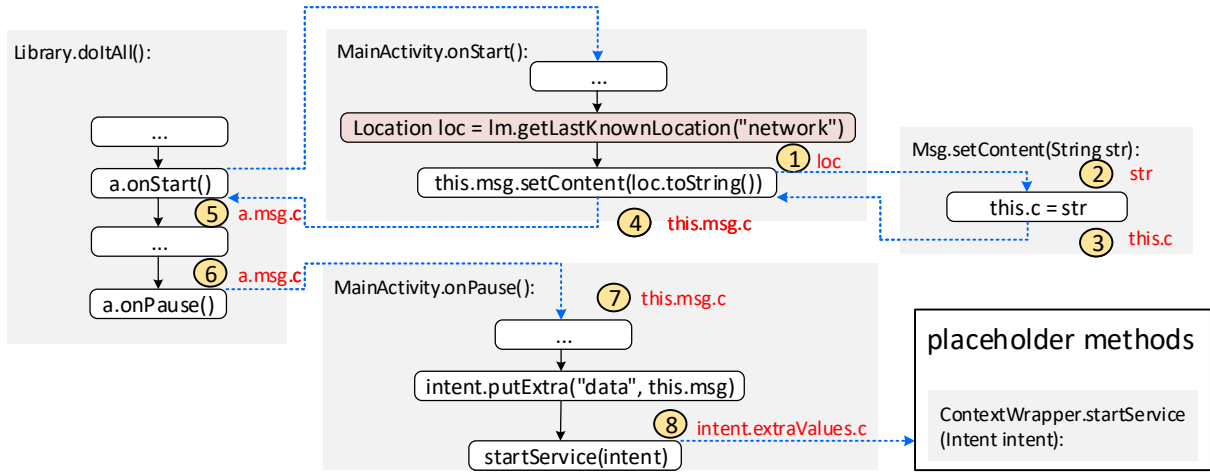


Figure 3.10: Taint propagation in MainActivity.

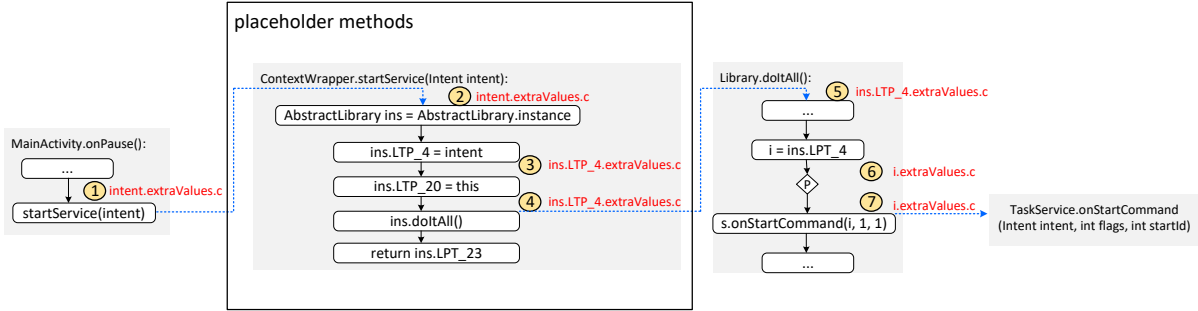


Figure 3.11: Taint propagation from MainActivity to TaskService.

no summaries are available. In the following, we explain how $\text{FLOWDROID}^{\text{Gen}}$ can detect the ICC leak in the motivating example. Consider the step ⑧ in Figure 3.10 at the statement `intent.putExtra("data", this.msg)`: because there exists a summary for this method and a taint `this.msg.c`, $\text{FLOWDROID}^{\text{Gen}}$ generates a new taint `intent.extraValues.c`. Since there is no summary for library method `ContextWrapper.startService`, $\text{FLOWDROID}^{\text{Gen}}$ analyzes the placeholder method for `ContextWrapper.startService`. This enables the taint `intent.extraValues.c` to be propagated into the placeholder method as Figure 3.11 shows.

In the placeholder method of `ContextWrapper.startService`, the parameter is assigned to its corresponding typed LPT₄ field. When evaluating this assignment, $\text{FLOWDROID}^{\text{Gen}}$ will generate a new taint `ins.LTP4.extraValues.c`. Because this taint is on the same object as the receiver of the call site `ins.doItAll()`, this taint will be propagated into the `Library.doItAll()` method (see from step ④ to ⑤ in Figure 3.11). In `Library.doItAll()`, every argument passed to the invocation of a callback method is loaded from its corresponding typed LPT*, e.g., `i=ins.LTP4` for the argument `i` of `s.onStartCommand(i, 1, 1)`. A new taint `i.extraValues.c` is generated and will be propagated to the callee `TaskService.onStartCommand`. As we have seen in this example, analyzing the placeholder method and `Library.doItAll()` allows data passing between different Android component objects. Similarly, as Figure 3.12 shows, the tainted data will be propagated from the method `TaskService.onStartCommand` to `PushMessageHandler.handleMessage` and eventually reaches the sink. Note that there exists no summary in FLOWDROID for `Handler.obtainMessage`. The original `Handler.obtainMessage`

method takes an object as parameter and assigns it to the returned `Message.obj` field. Our AVERROES-GENCG considers such cases and generates placeholder methods that model such setter behavior as Listing 3.10 shows.

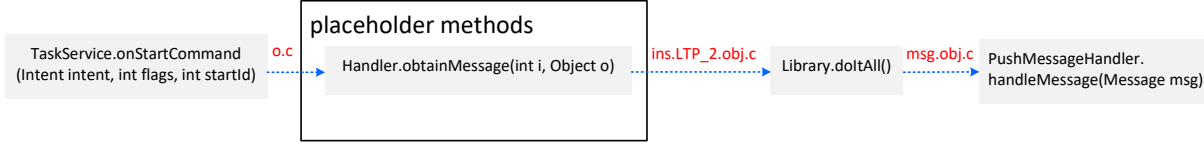


Figure 3.12: Taint propagation from `TaskService` to `PushMessageHandler`.

```

1 public final android.os.Message obtainMessage(int, java.lang.Object){
2   java.lang.Object r0;
3   android.os.Handler r1;
4   android.os.Message $r2, $r3;
5   int i0;
6   averroes.AbstractLibrary r4;
7   r1 := @this: android.os.Handler;
8   i0 := @parameter0: int;
9   r0 := @parameter1: java.lang.Object;
10  r4 = <averroes.AbstractLibrary: averroes.AbstractLibrary instance>;
11  r4.<averroes.AbstractLibrary: java.lang.Object LPT_19> = r0;
12  r4.<averroes.AbstractLibrary: android.os.Handler LPT_2> = r1;
13  $r2 = r4.<averroes.AbstractLibrary: android.os.Message LPT_3>;
14  $r2.<android.os.Message: java.lang.Object obj> = r0; // assign the second parameter to Message.obj.
15  r4.<averroes.AbstractLibrary: android.os.Message LPT_3> = $r2;
16  virtualinvoke r4.<averroes.AbstractLibrary: void doItAll()>();
17  $r3 = r4.<averroes.AbstractLibrary: android.os.Message LPT_3>;
18  return $r3;
19 }

```

Listing 3.10: Placeholder method for `Handler.obtainMessage`

Although using our placeholder library for Android SDK supports detection of ICC leaks, it is not perfect. Because all intents are pointed by one typed LPT field and the placeholder library does not differentiate the source and the target components, it may produce false positives sometimes as we later show in our evaluation.

3.5 Evaluation of GENCG

We evaluate GENCG with FLOWDROID^{Gen} , which is FLOWDROID using our call graphs. Our experiments are designed to address the following research questions:

- RQ1. How completely do the call graphs of FLOWDROID^{Gen} capture the code of known taint flows in TAINTBENCH, compared to FLOWDROID?
- RQ2. How effective is FLOWDROID^{Gen} in detecting taint flows in both DROIDBENCH and TAINTBENCH compared to FLOWDROID?

RQ1. How completely do the call graphs of FLOWDROID^{Gen} capture the code of known taint flows in TAINTBENCH, compared to FLOWDROID?

To answer this question, we executed both FLOWDROID and FLOWDROID^{Gen} to analyze TAINTBENCH apps and dumped the static call graphs constructed with Spark used by the taint analysis as we did in Section 2.5.3.2. We compared all relevant methods specified for the expected taint flows in TAINTBENCH against callees in the serialized call graphs. If a specified method of an expected taint flow is missing in the call graph used by a tool, this flow won't be detected by the tool. In this chapter we focus on these false negatives that are caused by an incomplete call graph. Note that even with a complete call graph, if the analysis is unsound, it can still cause the flow to be undetected. Such cases need to be studied individually, which are not our target in this chapter. Table 3.2 shows our comparison of two different false negatives caused by incomplete call graphs. Type-1 and Type-2 false negatives were introduced in Section 2.5.3.2 in last chapter. While FLOWDROID did not construct call graphs for 6 apps, resulting 28 Type-1 false negatives, FLOWDROID^{Gen} constructed call graphs that capture all of these 28 flows. Also, using our approach the number of Type-2 false negatives is significantly reduced from 42 to 19. In total, 51 more expected taint flows are captured in the call graphs of our approach in comparison to FLOWDROID, which is 25% (51/203) of all expected flows. We further investigated what kinds of edges remain missing in call graphs using our approach. There are two main kinds of missing edges:

- Calls from third-party library (not Android) to callback methods, e.g., `fakeplay`, `hummingbad`. This kind of edges could be constructed if we were to apply AVERROES-GENCG to the corresponding third-party libraries.
- UI callbacks that are defined in layout files, e.g., `repane`. These kind of edges require parsing the layout XML files and finding UI callbacks defined in them. Since such XML configuration is different in every framework and our approach is designed to be general, we did not model this specifically for Android. This could be done, of course, though, with appropriate engineering effort. For instance, one could integrate dynamic tools such as TAMIFLEX [BSS⁺11] that records the actual uses of reflection that occur at runtime. AVERROES can generate the corresponding behavior in the `Library.doItAll()` method. This ability is inherited by AVERROES-GENCG.

Another reason for false negatives is the **Problem 4** discussed in Section 3.3: taint flows cross both Java and JavaScript code via `WebView` supported by the Android framework. As we introduced in **Improvement 4** in Section 3.4.1, AVERROES-GENCG supports detecting of methods with the `android.webkit.JavascriptInterface` annotation and generates invocations of those annotated methods in the `doItAll()` method. Nevertheless, to detect these taint flows, the analysis still needs to track the data flow in the JavaScript code, which is not supported by FLOWDROID.

RQ2. How effective is FLOWDROID^{Gen} in detecting taint flows in both DROIDBENCH and TAINTBENCH compared to FLOWDROID?

Experiment Setup We evaluated FLOWDROID^{Gen} on both DROIDBENCH and TAINTBENCH to see how effective it is in comparison to FLOWDROID. For each benchmark app in both suites, the sources and sinks were configured at app-level (i.e., sources and sinks defined in the benchmark cases of the app) as in Experiment 3 introduced in Section 2.5.1.2. The experiment was executed on a windows laptop with an Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz processor, 16GB RAM and Java 8 (Oracle 1.8.0_202) installed.



Evaluation on DroidBench: The evaluation results on DROIDBENCH are shown in Table 3.3. In this table, for each category, numbers of FLOWDROID^{Gen} that are better than FLOWDROID are marked with a , while worse ones with a . In four categories, i.e., *Android Specific*, *Inter Component Communication*, *Lifecycle*, *Reflection ICC*, FLOWDROID^{Gen} detected more true-positive data leaks than FLOWDROID. The maximal increase of true positives is in the category *Inter Component Communication* with 9 (row 10: 13 - 4) more true positives. However, because

Table 3.2: Comparison of false negatives caused by incomplete call graphs: *Type-1* false negatives are those flows where no call graph could be constructed for the respective apk at all. *Type-2* false negatives are those flows of which relevant methods are missing in the call graph.

(a) Type-1 False Negatives		
Benchmark App	FLOWDROID	FLOWDROID ^{Gen}
chulia	4	0
fakeappstore	3	0
fakemart	2	0
godwon_samp	6	0
samsapo	4	0
slocker_android_samp	5	0
sms_google	4	0
Σ	28	0
(b) Type-2 False Negatives		
Benchmark App	FLOWDROID	FLOWDROID ^{Gen}
chat_hook	1	0
fakedaum	1	0
fakeplay	2	2
hummingbad_android_samp	2	2
overlaylocker2_android_samp	6	6
remote_control_smack	17	1
repane	1	1
save_me	1	1
scipix	2	0
smsstealer_kysn_assassincreed_android_samp	2	0
the_interview_movieshow	1	0
vibleaker_android_samp	4	4
xbot_android_samp	2	2
Σ	42	19

Table 3.3: Comparison of the evaluation results on DROIDBENCH. See detailed result of each benchmark app in Table B.1.

No.	Category	FLOWDROID			FLOWDROID ^{Gen}		
		TP	FP	FN	TP	FP	FN
1	Aliasing	0	2	0	0	3	0
2	Android Specific	6	0	5	7	1	4
3	Arrays and Lists	4	5	0	4	4	0
4	Callbacks	12	2	2	4	1	10
5	Dynamic Loading	3	0	0	3	0	0
6	Emulator Detection	14	0	1	12	0	3
7	Field and Object Sensitivity	2	0	0	2	0	0
8	General Java	11	4	10	11	3	10
9	Implicit Flows	1	0	0	1	0	0
10	Inter Component Communication	4	0	15	13	7	6
11	Lifecycle	15	0	9	20	0	4
12	Native	4	0	1	4	0	1
13	Reflection	1	0	8	1	0	8
14	Reflection ICC	1	0	10	3	1	8
15	Self Modification	0	0	3	0	0	3
16	Threading	5	0	1	5	0	1
17	Unreachable Code	1	0	2	1	0	2
Σ		84	13	67	91	20	60
Precision		0.87			0.82		
Recall		0.56			0.60		
F-measure		0.68			0.69		

TP: True Positive, FP: False Positive, FN: False Negative

○: better result, ~~○~~: worse result

we did not model the XML-based UI callbacks in AVERROES-GENCG to stay general for Java frameworks as we mentioned previously in RQ1, the result of FLOWDROID^{Gen} in the *Callbacks* category is worse than FLOWDROID with 8 less true positives (8 more false negatives), as many of the callbacks in these apps are specified in the app’s layout XML files. In the category *Emulator Detection*, one of the two new false negative of FLOWDROID^{Gen} (app: PI1) was also due to this reason. The other false negative was in the app *PlayStore2*, since AVERROES-GENCG could not generate the *Library* class. The ASM backend threw a *MethodTooLargeException* when generating bytecode for the *Library* class. It turns out that this app *PlayStore2* contains library classes from the Google Play Service (`com.google.android.gms.*`) [Goo12], which are considered as application code by AVERROES-GENCG and it creates objects for these classes in *Library.main()*. This raises another interesting research question: how can one separate library code and application code inside an Apk file? Checking the prefix of common library packages as FLOWDROID is not an optimal option, as malware apps often leverage this as we have discussed in Section 2.5.3.2. Nevertheless, FLOWDROID^{Gen} still detected 7 (row Σ : 91 - 84) more true positives in total, resulting a better recall (0.60 vs. 0.56). Because we did not model the lifecycle of Android components as precisely as FLOWDROID did and only used typed LPT fields to abstract objects of the same type pointed by the Android framework, more false posi-

tives were expected as we explained in **Improvement 1** and **Improvement 2** in Section 3.4.1. There are 6 (row *sum*: 19 - 13) more false positives produced by FLOWDROID^{Gen} in comparison to FLOWDROID. Listing 3.11 shows such a false-positive case detected by FLOWDROID^{Gen}. In this example, the field `this.b.b` is not yet tainted as it is used in the sink at line 10. It is only aliased to `a.b` at line 11. Assume `AbstractLibrary.LPT_1` is the typed LPT field generated by AVERROES-GENCG for the class `Activity`. This field points to an object of the `MainActivity` class initialized in the `Library.main()` method. The field access `this.b.b` at line 10 is equal to the taint with access path `AbstractLibrary.instance.LPT_1.b.b` generated in the taint analysis when evaluating line 11. This taint will be propagated again to the `onCreate()` method through the loop edge for the `Activity` class in the `Library.doItAll()` method (see example of loop edges in Figure 3.6). Note that the loop edges simulate the effect that a method can be called multiple times by the framework. Therefore, the second time the method `aliasFlowTest()` is analyzed, the field `this.b.b` is tainted and a false-positive will be reported. Other new false positives arise in benchmark apps using inter component communications. Although, our approach allows detecting of ICC leaks as we introduced in Section 3.4.3, it does not differentiate the destinations of intents as we mentioned in Section 3.4.3. Those false positives involve unrelated Android components that never receive a tainted intent. Despite the slightly decreased precision, our approach still allows FLOWDROID^{Gen} to achieve better F-measure on DROIDBENCH (0.68 vs 0.66). In conclusion, FLOWDROID^{Gen} is at least as effective as FLOWDROID on DROIDBENCH.

```

1 class MainActivity extends Activity{
2     public void onCreate(Bundle savedInstanceState) {
3         //...
4         aliasFlowTest();
5     }
6     private void aliasFlowTest() {
7         String deviceId = ((TelephonyManager) getSystemService("phone")).getDeviceId(); // source
8         A a = new A();
9         a.b = deviceId;
10        SmsManager.getDefault().sendTextMessage("+49 1234", null, this.b.b, null, null); // sink, no leak
11        this.b = a;
12    }
13 }

```

Listing 3.11: False positive reported by FLOWDROID^{Gen} for the app *FlowSensitivity1* from the aliasing category.

Evaluation on TaintBench: Table 3.4 shows the comparison between FLOWDROID and FLOWDROID^{Gen} evaluated on TAINTBENCH. As we can see in this table, FLOWDROID^{Gen} detected 24 (row *sum*: 64-40) more true positives with just one more false positives (row *sum*: 9-8) than FLOWDROID. The call graphs constructed with our GENCG approach enabled FLOWDROID^{Gen} to analyze more code than FLOWDROID, thus, improved the recall from 0.20 to 0.32. Although a generic approximation like our approach can be noisy, the evaluation results show that it did not affect in detecting real-world malicious taint flows in TAINTBENCH. Even the precision is slightly increased from 0.83 to 0.88.

As we mentioned in Section 3.5, the call graphs used by FLOWDROID^{Gen} contain relevant edges for 50 more expected taint flows. However, still about half (26) of these flows could be not detected in the taint analysis. One major reason is still unmodeled library behaviors. Listing 3.12 shows a false-negative case from the benchmark app *chulia*. The method `Cursor.getString()` invoked at line 11 and line 12 returns sensitive data stored in a database for SMS messages. This behavior is not modeled as a data-flow summary nor captured by the placeholder method body generated by AVERROES-GENCG, as AVERROES-GENCG does not model databases or any intermediate file storage. Such cases are also hard to model statically, dynamic analysis would be more effective [RAMB16]. As a result, no new taints will be generated from the taint query that is tainted.

```

1 String getSms() {
2     StringBuilder stringBuilder = new StringBuilder();
3     Cursor query = getContentResolver().query(Uri.parse("content://sms/"),
4         new String[]{"_id", "address", "person", "body", "date", "type"},
5         null, null, "date desc"); // source, query is tainted.
6     if (query.moveToFirst()) {
7         int index1 = query.getColumnIndex("person");
8         int index2 = query.getColumnIndex("address");
9         do {
10             String person = query.getString(index1); // not tainted
11             String address = query.getString(index2); // not tainted
12             stringBuilder.append("person:");
13             stringBuilder.append(string + ",address:");
14             stringBuilder.append(string2);
15             stringBuilder.append(";");
16         } while (query.moveToNext());
17     } else { stringBuilder.append("no result!");}
18     return stringBuilder.toString();
19 }
20 void onReceive(){
21     leak(getSms()); // sink
22 }

```

Listing 3.12: A simplified false-negative case from the app *chulia* in TAINTBENCH.

Thus, the return value of the method `getSms()` that is passed to the sink is not tainted in the analysis and no leak is reported. Also some malware apps in TAINTBENCH obfuscate with string encryption in combination with reflective method calls, this makes it very difficult for static analyses approaches to understand the purpose of the malicious code. Such cases (e.g. in app *fakemart*) are not handled by the taint analysis of FLOWDROID.

Table 3.4: Comparison of the evaluation results on TaintBench. See detailed result of each benchmark app in Table B.2.

No.	Benchmark App	FLOWDROID			FLOWDROID ^{Gen}		
		TP	FP	FN	TP	FP	FN
1	backflash	11	4	2	11	4	2
2	beita*	0	0	3	①	0	2
3	cajino_baidu	8	1	4	8	1	4
4	chat_hook	8	0	4	⑦	0	5
5	chulia	0	0	4	0	0	4
6	death_ring*	1	0	0	1	0	0
7	dsencrypt*	0	0	1	0	0	1
8	exprespam	0	0	2	①	0	1
9	fakeappstore	0	0	3	①	0	2
10	fakebank*	0	0	5	②	0	3
11	fakedaum	0	0	2	①	0	1
12	fakemart	0	0	2	0	0	2
13	fakeplay	0	0	2	0	0	2
14	faketaobao	0	0	4	③	0	1
15	godwon_samp	0	0	6	④	0	2
16	hummingbad*	0	0	2	0	0	2
17	jollyserv	0	0	1	0	0	1
18	overlay*	0	2	4	①	①	3
19	overlaylocker2*	0	0	7	0	①	7
20	phospy	1	1	1	②	②	0
21	proxy_samp	2	0	15	2	0	15
22	remote*	0	0	17	⑧	0	9
23	repane	0	0	1	0	0	1
24	roidsec	0	0	6	0	0	6
25	samsapo	0	0	4	0	①	4
26	save_me	2	0	23	2	0	23
27	scypiex	0	0	3	①	0	2
28	slocker*	0	0	5	①	0	4
29	sms_google	0	0	4	①	0	3
30	sms_send_locker*	0	0	6	0	0	6
31	smssend*	4	0	1	②	0	3
32	smssilience*	0	0	2	0	0	2
33	smsstealer*	1	0	4	1	0	4
34	stels_flashplayer*	2	0	1	2	0	1
35	tetus	0	0	2	①	0	1
36	the_interview*	0	0	1	0	0	1
37	threatjapan*	0	0	2	0	0	2
38	vibleaker*	0	0	4	0	0	4
39	xbot*	0	0	3	0	0	3
Σ		40	8	163	64	9	139
Precision		0.83			0.88		
Recall		0.20			0.32		
F-measure		0.32			0.47		

TP: True Positive, FP: False Positive, FN: False Negative

①: better result, ⑦: worse result, *: the prefix of the app name.

3.6 Application of GENCG on the Spring Framework

In this section, we demonstrate how one can apply GENCG to the Spring Framework. We focus on explaining how common Spring features are handled by AVERROES-GENCG. Spring is the most popular Java web framework in recent years [Gee20]. It implements the Inversion of Control (IoC) principle, which is also known as dependency injection [Spr02a]. Dependency injection is a technique such that objects define their dependencies through constructor arguments, factory methods arguments or setter arguments. The IoC container in Spring is there to inject dependency objects. In Spring, the objects that are created and managed by its IoC container are called beans. Beans can be defined both via XML configuration, or in Java through annotations and code. The creation of beans or and invocation of their member methods in Spring rely heavily on reflection. This is one of the biggest reasons why popular static analysis frameworks such as SOOT or WALA fail to produce sound call graphs for Spring applications.

3.6.1 Handling Annotated Entry Points

The core part of Spring is designed as a model-view-controller framework that utilizes a class `DispatcherServlet` to dispatch HTTP requests to handlers [Spr02b]. A handler is defined by both a class annotation (e.g., `@RestController`) and a method annotation (e.g., `@PostMapping`) in Spring. Listing 3.13 shows a piece of code from WebGoat [Web14] defining a handler that is vulnerable to SQL injections. In this example, the class `SqlInjectionLesson2` is annotated with `@RestController` and its member method `completed()` is annotated with `@PostMapping`. Just like in Android apps, objects of this class (or beans) are not created anywhere in the application code, but at runtime by the framework, so does the handler method `completed()` for handling POST requests. Without modeling the Spring framework, the method `completed()` will not appear in call graphs constructed by Soot or WALA.

```

1 @RestController
2 public class SqlInjectionLesson2 extends AssignmentEndpoint {
3
4     private final LessonDataSource dataSource;
5
6     public SqlInjectionLesson2(LessonDataSource dataSource) {
7         this.dataSource = dataSource;
8     }
9
10    @PostMapping("/SqlInjection/attack2")
11    @ResponseBody
12    public AttackResult completed(@RequestParam String query) { // source: parameter query
13        return injectableQuery(query);
14    }
15
16    protected AttackResult injectableQuery(String query) {
17        try (var connection = dataSource.getConnection()) {
18            Statement statement = connection.createStatement(TYPE_SCROLL_INSENSITIVE,
19                CONCUR_READ_ONLY);
20            ResultSet results = statement.executeQuery(query); // sink: execute the untrusted SQL query
21            //...
22        } catch (SQLException sqle) { //... }
23    }

```

Listing 3.13: SQL injection case from WebGoat.

We found this concept is quite similar to the entry point classes and lifecycle methods in Android apps. It can be reduced to a subtyping implementation as shown in Listing 3.14. Instead of

Table 3.5: Entry points annotations in Spring.

Entry Point Classes	Entry Point Methods
org.springframework.stereotype.Controller	org.springframework.web.bind.annotation.GetMapping
org.springframework.web.bind.annotation.RestController	org.springframework.web.bind.annotation.PostMapping
org.springframework.stereotype.Component	org.springframework.web.bind.annotation.DeleteMapping
org.springframework.stereotype.Repository	org.springframework.web.bind.annotation.PatchMapping
org.springframework.stereotype.Service	org.springframework.web.bind.annotation.PutMapping
org.springframework.web.servlet.handler.HandlerInterceptorAdapter	org.springframework.web.bind.annotation.RequestMapping
org.springframework.web.servlet.HandlerInterceptor	org.springframework.web.bind.annotation.ExceptionHandler
	org.springframework.web.bind.annotation.InitBinder
	org.springframework.messaging.handler.annotation.MessageMapping

using annotations, the framework could define an interface `SqlInjectionLesson2_GenCG` that declares a method `completed()`, which is called by the framework. The `SqlInjectionLesson2` class that implements this interface would have the same effect as the original class. If we have such an implementation of `SqlInjectionLesson2` in the application code, and the interface in the `placeholder.jar` generated by AVERROES-GENCG, the application-only call graph constructed with our GENCG approach will have a call edge from the `Library.doItAll()` method to the handler method `SqlInjectionLesson2.completed()`.

```

1 // in instrumemnted-app.jar
2 public class SqlInjectionLesson2 implements SqlInjectionLesson2_GenCG {
3     private final LessonDataSource dataSource;
4
5     public SqlInjectionLesson2(LessonDataSource dataSource) {
6         this.dataSource = dataSource;
7     }
8
9     @ResponseBody
10    public AttackResult completed(@RequestParam String query) { // source: parameter query
11        return injectableQuery(query);
12    }
13
14    protected AttackResult injectableQuery(String query) {
15        try (var connection = dataSource.getConnection()) {
16            Statement statement = connection.createStatement(TYPE_SCROLL_INSENSITIVE,
17                CONCUR_READ_ONLY);
18            ResultSet results = statement.executeQuery(query); // sink: execute the untrusted SQL query
19            // ...
20        } catch (SQLException sqle) { // ... }
21    }
22 }
23
24 // in placeholder.jar
25 public interface SqlInjectionLesson2_GenCG {
26     public AttackResult completed(@RequestParam String query);
27 }

```

Listing 3.14: Reduction from annotations to subtyping.

For each entry point class annotation listed in Table 3.5, AVERROES-GENCG checks if there are methods that are annotated with the entry point method annotations in this table. If it is the case, AVERROES-GENCG generates an artificial interface just like the `SqlInjectionLesson2_GenCG` interface in Listing 3.14 for the `placeholder.jar`. It also instruments the original entry point class to implement this interface. This artificial interface will be handled as a library interface by AVERROES-GENCG, thus it invokes the methods of this interface in the `Library.doItAll()` and creates objects for the entry point classes in `Library.main()`. The instrumented classes together with other classes in the application will be written into an

`instrumented-app` jar file (see Figure 3.5). This `instrumented-app` is then used as an input for call graph construction in our approach.

3.6.2 Handling Bean Autowiring

As introduced, the IoC container in Spring creates beans and resolves dependency relationships between beans automatically. This process is called bean autowiring [Bae21]. Three annotations are used in Spring to resolve dependencies: `@Autowired` (defined in Spring), `@Inject` (defined in JSR-330) and `@Resource` (defined in JSR-250). These three annotations work similarly. They can be used to annotate class properties (fields), constructors and setters. We explain here with the `@Autowired` annotation.

```

1 @Controller
2 public class MyController{
3
4     @Autowired
5     private MyService myService; // instantiated by Spring.
6
7     @PostMapping("/register")
8     public void doRegistration(HttpServletRequest request, HttpServletResponse response) throws
        IOException {
9         String username = request.getParameter("username");
10        response = myService.getResponse(username); // missing call edge
11    }
12
13    @Autowired // autowiring on constructor
14    public MyController(MyResource r){
15        this.myResource = r;
16    }
17
18    @Autowired // autowiring on setter
19    public void setResource(MyResource r){
20        this.myResource = r;
21    }
22 }
```

Listing 3.15: Autowiring in Spring.

In Listing 3.15, the field `myService` is annotated with `@Autowired` at line 4. In the application code, there is no code which instantiates this field. Instead, the field will be instantiated by the Spring framework. Without modeling this behavior, the application-only call graph constructed with VTA, RTA or Spark will miss the edge from caller `MyController.doRegistration` to the callee `MyService.getResponse`. This is because these algorithms take object allocation sites into account and limit the type of receiver to types of objects that have been instantiated in the program. In this example, no allocation site (i.e. `new MyService()`) for `myService` can be found in the application code. As a result, the callee `MyService.getResponse` will not be resolved in the call graph construction and potentially result in false negatives for analysis using the constructed call graph.

`@Autowired` can be also used to annotate constructors (see line 13) and setters (see line 18). Both the constructor and the setter in Listing 3.15 are called by Spring with an instance of `MyResource` as an argument. How Spring should instantiate this argument is defined either in a XML file as shown in Listing 3.16 or directly in Java code using the `@Bean` annotation as shown in Listing 3.17. In this example, the argument will be instantiated with its `id` equal to 123 and its `name` equal to `abc` by Spring.

AVERROES-GENCG scans fields and methods in application classes that are declared with these autowiring annotations and handles them in a general way. For each field that is autowired,

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
3 "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
4
5 <beans>
6     <bean id="resourceOne" class="MyResource">
7         <property name="id" value="123"></property>
8         <property name="name" value="abc"></property>
9     </bean>
10 </beans>

```

Listing 3.16: Bean defined in XML configuration.

```

1 @Configuration
2 public class Config{
3
4     @Bean
5     public MyResource resourceOne(){
6         return new MyResource("123", "abc");
7     }
8 }

```

Listing 3.17: Bean defined in Java code.

AVERROES-GENCG instruments the default constructor to instantiate the field. If the type of the field is a concrete class, an object of this type will be created with default property values and assigned to the annotated field. If the type of the field is an abstract class or interface, for each concrete subclass, an object will be created and assigned. Such subtyping relationship can be easily obtained from the class hierarchy AVERROES-GENCG computes. Assume `MyService` in Listing 3.15 is an interface with only two implementations: class `ServiceOne` and class `ServiceTwo`. The generated code for the annotated field is presented in the `MyController` class in Listing 3.18 at line 5- 11. We again use a nondeterministic predicate p (introduced in Section 3.4.1) to model that the field `myService` could be both types of objects. The annotated constructor and setter are regarded as annotated callback methods by AVERROES-GENCG. Invocations of them are generated in the `Library.doItAll()` method as shown at line 18 - 25 in Listing 3.18. AVERROES-GENCG does not search the concrete property values of each bean defined with the annotation `@Bean` or in the XML configuration to stay general. It uses some default values for primitive types and `null` for reference types. We lose precision by doing this, however, call edges to methods of autowired objects will be captured in the call graph construction using the instrumented app.

```

1 public class MyController implements MyController_GenCG {
2     @Autowired
3     private MyService myService;
4
5     public MyController() {
6         // instrumented code for instantiating autowired field
7         if (p)
8             this.myService = new ServiceOne(0);
9         if (p)
10            this.myService = new ServiceTwo(0);
11    }
12    // ...
13 }
14
15 public class Library {
16     public void doItAll(){
17         //...
18         MyController_GenCG c = this.LPT_1;
19         MyResource r = this.LPT_2;
20         if(p)
21             c.<init>(r); // call the annotated constructor
22         MyController_GenCG c = this.LPT_1;
23         MyResource r = this.LPT_2;
24         if(p)
25             c.setResource(r); // call the annotated setter
26         //...
27     }
28
29     public static void main(String[] args) {
30         this.LPT_1 = new MyController();
31         this.LPT_2 = new MyResource();
32         // ...
33     }
34 }

```

Listing 3.18: Instrumented code for handling autowiring.

3.6.3 Implementation Details

Spring applications usually use Spring Boot to create stand-alone executable files [Spr21]. Both `jar` and `war` files are supported by Spring Boot. Since our implementation of AVERROES-GENCG is built on top of Soot and Soot doesn't support analyzing `war` files, we only consider `jar` files created by Spring Boot. Unlike shaded `jar` files (a `jar` file that contains all application classes and library classes) that are hard to differentiate between library classes and application classes, Spring Boot generates nested `jar` files with a clear file structure as shown in Figure 3.13. The application classes are placed under the `BOOT-INF/classes` directory, while the dependencies can be found in `BOOT-INF/lib`. AVERROES-GENCG scans such executable jars and generates an `organized-app.jar` which contains only application classes and an `organized-lib.jar` containing all library classes from the dependency jars in `BOOT-INF/lib`. It then processes these two `jar` files with Soot, builds class hierarchy, instruments classes, generates code etc.

3.6.4 Evaluation with CGBENCH

To evaluate how effective AVERROES-GENCG is on Spring applications, we developed a benchmark suite called CGBENCH with ground-truth documentation. CGBENCH consists of 42 Spring apps classified into 6 categories. 39 of them are micro benchmark apps to demonstrate specific Spring features with one or two built-in taint-style vulnerabilities such as SQL Injection, XSS,

```

exampleApp.jar
|
+-META-INF
|   +-MANIFEST.MF
+-org
|   +-springframework
|       +-boot
|           +-loader
|               +-<spring boot loader classes>
+-BOOT-INF
|   +-classes
|       +-example.app
|           +-example.app.package
|               +-ApplicationClass1.class
|               +-ApplicationClass2.class
|               +-...
|   +-lib
|       +-dependency1.jar
|       +-dependency2.jar
|       +-dependency3.jar
|       +-...

```

Figure 3.13: File structure of the executable jar format supported by Spring Boot.

Log Injection etc. The vulnerabilities in these apps are all expected taint flows. 3 of the 42 apps are bigger apps that are built to demonstrate vulnerabilities. These 3 apps contain multiple Spring features and taint-style vulnerabilities. The ground truth documentation CGBENCH consists of 60 expected and 10 unexpected taint flows. We configured FLOWDROID^{Gen} with the sources and sinks in each benchmark app and analyzed the instrumented app generated by AVERROES-GENCG. Table 3.6 shows the evaluation results of FLOWDROID^{Gen} on CGBENCH. In total, FLOWDROID^{Gen} detected 44 expected taint flows out of 60 and 5 unexpected taint flows out of 10, which makes the precision 0.90, the recall 0.73 and the F-measure 0.81. In the following, we introduce in detail how FLOWDROID^{Gen} performs on each category of CGBENCH.

Table 3.6: Evaluation results of FLOWDROID^{Gen} on CGBENCH.

No.	Category	Expected Taint Flows	Unexpected Taint Flows	True Positives	False Positives
1	HTTP Request Handlers	9	0	8	0
2	Component Classes	5	0	4	0
3	Handler Interceptors	6	0	6	0
4	Parameter Sources	19	0	16	0
5	Configuration	6	0	0	0
6	Demo Apps with Mixed Features	15	10	10	5
Σ		60	10	44	5
Precision					0.90
Recall					0.73
F-measure					0.81

Table 3.7: Benchmark Category: HTTP Request Handlers.

No.	Benchmark App	Features	Vulnerability Type	Expected Taint Flows	True Positives
1	deletemapping	@DeleteMapping, @RestController	Command Injection	2	2
2	getmapping	@GetMapping, @RestController	SQL Injection	1	1
3	patchmapping	@PatchMapping, @RestController	SQL Injection	2	2
4	postmapping	@PostMapping, @Controller	Command Injection	1	1
5	putmapping	@PutMapping, @RestController	SQL Injection	1	1
6	requestmapping	@RequestMapping, @RestController	Open Redirect Attack	1	1
7	exceptionhandler	@ExceptionHandler, @RestController	Reflected XSS	1	0
Σ				9	8

Table 3.7 lists the benchmark apps in CGBENCH that uses Spring annotations to define HTTP request handlers. FLOWDROID^{Gen} detected all vulnerabilities in these apps, except the one in the `exceptionhandler` app. Listing 3.19 shows the code of the false-negative case. Although the call graph used by FLOWDROID^{Gen} captures the call from Spring (i.e. `Library.doItAll()`) to the annotated exception handler `invalidNumberExceptionHandler`, its parameter `ex` is not mapped to the `NumberFormatException` thrown at line 18 containing the untrusted `uid`. This aliasing relationship needs to be modeled in FLOWDROID^{Gen}'s taint analysis.

```

1 @RestController
2 public class MyController {
3     @ExceptionHandler(value = {NumberFormatException.class})
4     public void invalidNumberExceptionHandler(NumberFormatException ex, HttpServletResponse
        response) throws IOException {
5         String uid = ex.getMessage();
6         response.setContentType("text/html;charset=UTF-8");
7         response.setCharacterEncoding("UTF-8");
8         response.getWriter().append(uid); // sink
9     }
10
11     @GetMapping(value = "/", produces = MediaType.TEXT_PLAIN_VALUE)
12     public void retrieveUserInformation(HttpServletRequest request) {
13         String uid = request.getParameter("uid"); // source
14         try {
15             int userID = Integer.parseInt(uid); // throws NumberFormatException if user gives non-numbers
16             // ... retrieves the user information from the database using userID
17         } catch (NumberFormatException ex) {
18             throw new NumberFormatException("invalid user id = " + uid);
19         }
20     }
21 }

```

Listing 3.19: False negative case from the `exceptionhandler` app in CGBENCH.

Table 3.8 lists benchmark apps that use annotations to declare component classes. A component class in Spring is responsible for some operations and will be automatically detected by Spring for dependency injection. These component classes are considered as entry point classes by AVERROES-GENCG (see entry point classes in Table 3.5). FLOWDROID^{Gen} was able to analyze all code that capture the expected taint flows. For the taint flow in the benchmark app `simplerestcontroller`, the user can send a message over a request parameter which is directly returned in the response body without sanitization. It is an injection vulnerability, if the message is some piece of malicious code. Although the execution of the malicious code is not in the Spring application code, to capture this vulnerability one can specify the return value of an annotated request handler to be the sink. However, FLOWDROID^{Gen} doesn't support configuring such sinks, thus, resulted in false negative.

3.6 APPLICATION OF GENCG ON THE SPRING FRAMEWORK

Table 3.8: Benchmark Category: Component Classes

No.	Benchmark App	Features	Vulnerability Type	Expected Taint Flows	True Positives
8	component	@Component, @RestController, @GetMapping	Reflected XSS	1	1
9	repository	@Repository, @RestController, @GetMapping	SQL Injection	1	1
10	service	@Service, @RestController, @GetMapping	Reflected XSS	1	1
11	simplecontroller	@Controller, @RequestMapping	Command Injection	1	1
12	simplerestcontroller	@RestController, @RequestMapping	Command Injection	1	0
Σ				5	4

Table 3.9 lists benchmark apps that demonstrate the usage of Spring handler interceptors. The `HandlerInterceptor` interface declares three methods on where one wants to intercept the HTTP request: `preHandle()` is called before the actual HTTP request handler is executed; `postHandler` is called after the handler; `afterCompletion()` is called after the request is completed and view is rendered. `HandlerInterceptorAdapter` is an abstract class that implements the `HandlerInterceptor` interface and provides some base implementation. All application classes that are subtypes of these two classes are considered as entry point classes by AVERROES-GENCG (see entry point classes in Table 3.5) and callback methods are invoked in the `Library.doItAll()`. As introduced in Section 3.4.1, the skip-method/classes edges and loop edges in `Library.doItAll()` ensure that all possible orders of method calls are captured, including the order where the respective interceptor method is called. All expected taint flows in this benchmark category are detected by FLOWDROID^{Gen}.

Table 3.9: Benchmark Category: Handler Interceptors

No.	Benchmark App	Features	Vulnerability Type	Expected Taint Flows	True Positives
13	handlerinterceptoradapteraftercompletion	@Configuration, HandlerInterceptorAdapter.afterCompletion()	Log Injection	1	1
14	handlerinterceptoradapterposthandle	@Configuration, HandlerInterceptorAdapter.postHandle()	Log Injection	1	1
15	handlerinterceptoradapterprehandle	@Configuration, HandlerInterceptorAdapter.preHandle()	Log Injection	1	1
16	handlerinterceptoraftercompletion	@Configuration, HandlerInterceptor.afterCompletion()	Log Injection	1	1
17	handlerinterceptorposthandle	@Configuration, HandlerInterceptor.postHandle()	Log Injection	1	1
18	handlerinterceptorprehandle	@Configuration, HandlerInterceptor.preHandle()	Log Injection	1	1
Σ				6	6

Table 3.10 lists benchmark apps that contain expected taint flows of which the sources are annotated parameter methods. The original FLOWDROID doesn't support configuring annotated parameter as sources, we extended FLOWDROID^{Gen} with this feature.

Table 3.10: Benchmark Category: Parameter Sources

No.	Benchmark App	Features	Vulnerability Type	Expected Taint Flows	True Positives
19	cookievalue	@CookieValue, @RestController, @GetMapping	SQL Injection	1	1
20	initbinderwithoutvalue	@InitBinder, @RequestParam, @RestController, @GetMapping	Reflected XSS	1	1
21	initbinderwithvalue	@InitBinder, @RequestParam, @RestController, @GetMapping	Reflected XSS	1	1
22	modelattributeonargumentlevel	@ModelAttribute, @RestController, @RequestMapping	Information Leak	1	1
23	modelattributeonargumentlevelwithfalsebinding	@ModelAttribute, @RestController, @RequestMapping	Information Leak	1	1
24	modelattributewithaddattribute	@ModelAttribute, @RestController, @RequestMapping	Reflected XSS, Trust Boundary Violation	2	1
25	modelattributewithreturnvalue	@ModelAttribute, @RestController, @RequestMapping	Trust Boundary Violation	1	0
26	pathvariable	@PathVariable, @RestController, @GetMapping	Trust Boundary Violation	1	1
27	requestattribute	@RequestAttribute, @RestController, @GetMapping	Information Leak	2	2
28	requestheader	@RequestHeader, @RestController, @GetMapping	Open Redirect Attack	1	1
29	requestparam	@RequestParam, @RestController, @GetMapping	SQL Injection	1	1
30	requestpart	@RequestPart, @RestController, @GetMapping	SQL Injection	1	1
31	matrixvariable	@MatrixVariable, @Configuration, @RestController, @GetMapping	SQL Injection	1	1
32	sessionattribute	@SessionAttribute, @RestController, @GetMapping	Stored XSS, Reflected XSS	2	1
33	sessionattributes	@SessionAttributes, @Controller, @GetMapping	Stored XSS, Reflected XSS	2	2
Σ				19	16

Table 3.11 lists benchmark apps that involve configuration, which are not modeled by AVERROES-GENCG at all. Consequently, FLOWDROID^{Gen} detects none of the expected flows in these apps.

Table 3.11: Benchmark Category: Configuration

No.	Benchmark App	Features	Vulnerability Type	Expected Taint Flows	True Positives
34	bean	@Bean	Information Leak	1	0
35	beanwithclassxmlconfiguration	XML Configuration	Information Leak	1	0
36	controlleradvice	@ControllerAdvice	Reflected XSS	1	0
37	springmvcwithfreemarker	Freemarker	SQL Injection	1	0
38	springmvcwithjsp	JSP	SQL Injection	1	0
39	springmvcwiththymeleaf	Thymeleaf	SQL Injection	1	0
Σ				6	0

Table 3.12 shows the bigger demo apps that include multiple vulnerabilities and framework features. There are a few expected taint flows FLOWDROID^{Gen} could not detect, because third-party library methods that can generate new taints are not considered by the analysis, i. e., no summaries are available. Similarly, lack of analyzing third-party methods that are sanitizers caused false positives.

Table 3.12: Demo Apps with Mixed Features

No.	Benchmark App	Expected Taint Flows	Unexpected Taint Flows	True Positives	False Positives
40	onlineshop	6	5	5	3
41	onlinechat	3	0	1	0
42	teleforum	6	5	4	2
Σ		15	10	10	5

3.7 Related Work

Many previous approaches have addressed the challenge of modeling Java frameworks. In terms of Android, FLOWDROID [ARF⁺14] precisely models the Android lifecycle and UI callback handling by creating a dummy main method. AMANDROID [WROR14] and ICCTA [LBB⁺15] extend this model by introducing control and data dependencies between Android components such that inter-component communications are also captured. While these approaches don't analyze the Android framework, DROIDSAFE [GKP⁺15] took another approach by manually crafting stub implementations of the framework. Similar to the placeholder library used in our approach, these stub implementations are analyzed as replacements of the original framework implementations. However, as the authors themselves pointed out, implementing these stubs is labor-intensive and requires expertise in Android. These tools model Android's behavior in its implementation, thus their models can not be reused easily for other tools; however, both Droidel [BGC15] and our approach automatically creating app-specific stubs of the Android framework with a single entry point. While the authors of Droidel acknowledged that their approach is not suitable for flow-sensitive analyses, our evaluation on TAINTBENCH with the flow-sensitive taint analysis in FLOWDROID shows that our approach works well. While our approach only needs the stub version of the Android framework that is available in the Android SDK, Droidel still requires a one-time manual modification of the original Android framework source code to replace usage of reflection with the Droidel's own interfaces. Moreover, Droidel is limited only to Android, but our AVERROES-based approach is designed to be general for Java frameworks.

In terms of modeling Java web frameworks, IBM's TAJ [TPF⁺09] and its follow-up work F4F [SAP⁺11] are among the best-known approaches targeting Java enterprise applications. TAJ is a taint analysis tool that has partly modeled the Apache Struts framework and Enterprise Java Beans in its analysis engine. Adding a new framework support requires engineering effort in the analysis engine. In that way, it is quite similar to most of the taint analysis tools for Android discussed before. The follow-up work F4F is an approach that augments the taint analysis engine to handle new frameworks without modifying the analysis engine. It generates a specification of app-specific framework-related behavior in the specification language WAFL, which can be used to enhance an existing taint analysis engine. Although this approach is an improvement over TAJ regarding reusability, an analysis engine still needs to support the WAFL specification, and new WAFL generators need to be written to support new frameworks. In comparison to this approach, our approach doesn't require any extension of the analysis tool, the generated placeholder library can be processed by any existing Java analysis frameworks (e.g., Soot, WALA). To support a new framework with our approach, one only needs to extend the configurable lists of APIs (i.e., entry point classes/methods/annotations, annotations for dependency injection) that should be considered by AVERROES-GENCG. Similar to F4F, recent work JackEE [AFK⁺20] also introduces a rule-based specification that covers general concepts for modeling Java enterprise framework behaviors. JackEE leverages Doop and its model of a new framework is a collection of logic rules, which can be understood by Doop.

Our work was inspired by and based on AVERROES [AL13] which generates a placeholder library that over-approximates the original library. While AVERROES was focused on the soundness of the call graphs and the authors did not consider application of it by client analyses, the goal of our work is to construct call graphs that are not only sound, but also allow a precise client analysis effectively finding more issues. We discuss the shortcomings of AVERROES when applying it to a precise interprocedural taint analysis and address these in our AVERROES-GENCG. Our evaluation shows that our approach allows an Android taint analysis more effectively finding real-world taint flows.

3.8 Limitations and Threats to Validity

Since our approach is designed to be general, we only consider language-level concepts (e.g., subtyping, annotations) and explicitly did not model framework behaviors that require to parse configuration files (e.g., XML, HTML files). Client analyses which need such information are required to add this support by themselves.

A threat to the external validity in our study is that our evaluation results have limited generalizability to other client analyses. Because our focus is constructing call graphs that allow taint analyses to effectively find more real-world issues, we only evaluated our approach with the taint analysis in FLOWDROID. FLOWDROID uses STUBDROID to generate summaries for handling the taint propagation through common library methods. These summaries cover many methods from the Android framework. Our version FLOWDROID^{Gen} uses existing summaries in FLOWDROID and only analyzes placeholder methods if no summaries are available. Other taint analysis approaches that do not model taint propagation through library methods, with our approach they might still produce imprecise results.

3.9 Conclusion

In this chapter, we proposed GENCG—a general approach to modeling Java frameworks. Our AVERROES-GENCG produces a placeholder library that can be used as a sound replacement of the original framework by precise call graph construction algorithms and further client analyses. We demonstrate its generalization with both the Android and the Spring framework. Experiments on Android show our approach is especially effective in enabling a precise flow-, field- and context-sensitive taint analysis in detection of more real-world issues without introducing much noise. We constructed a micro benchmark suite—CGBENCH—consisting of common taint-style vulnerabilities in Spring-based web applications. We evaluate our approach using this suite and show the effectiveness.

Towards Path-Sensitive Analysis with COVA

In the previous chapter, we introduced our work on constructing complete call graphs for framework-based applications. However, even given a complete enough call graph, analysis precision is still the key factor that impacts the adoption of static analysis tools by developers and security analysts. Existing studies show that static analysis tools are most likely to be adopted if they yield high precision, i.e., a low rate of false positives [CB16, JSMB13]. To achieve high precision, static taint analysis tools incorporate various sensitivities (e.g., field-sensitivity, context-sensitivity, object-sensitivity etc.). However, as Li et al. pointed out in their literature review [LBP⁺17], *path*-sensitivity has been rarely considered by static analysis tools for Android apps. As a result, these tools can only permit one to conclude how many potential taint flows they detect in a given application, but *not of which nature* those taint flows exactly are, i.e., under which path conditions they can occur at runtime, and what could be done further to discard potentially false positives within those taint flows. Analysis tools ignoring code conditioned for software and hardware settings can produce taint flows that developers might not even care about, e.g., the application is only targeting devices with Android version 5.0+ (API level 21+). In this case, developers might want to only see taint flows occurring in code guarded by a version check for the targeted devices. A path-sensitive analysis would be able to exclude uninteresting taint flows.

To provide information that can be used for such cases, we introduce a standalone tool COVA in this chapter. COVA combines both SMT solving and interprocedural data-flow analysis that is flow-, context-, and path-sensitive. It computes partial path constraints that can be used to eliminate false positives along unsatisfiable paths (i.e., path constraint contains contradictory conditions), enhance results produced by a client analysis or even generate user inputs to dynamically validate static findings. The outline of the chapter is as follows: we first motivate the work with an example explaining how complex path conditions in Android apps can be in Section 4.1. We further explain in Section 4.2 that the computation of path constraints is a non-distributive problem and introduce the VASCO framework that can be used for solving such non-distributive problems. We introduce details of COVA’s constraint analysis in Section 4.3 and its architecture in Section 4.4. Evaluation of COVA on CONSTRAINTBENCH—a micro benchmark suite comprising 100 test programs is presented in Section 4.5. In Section 4.6, we introduce the study of Android taint-analyses results using COVA that was published at the ASE conference [LBS19]. To demonstrate the feasibility of using COVA for dynamic validation of static findings, a master thesis [Hau21] supervised by me that extends COVA for this purpose is introduced in Section 4.7. We discuss the limitations in Section 4.8, compare COVA to existing work in Section 4.9, and conclude the chapter in Section 4.10.

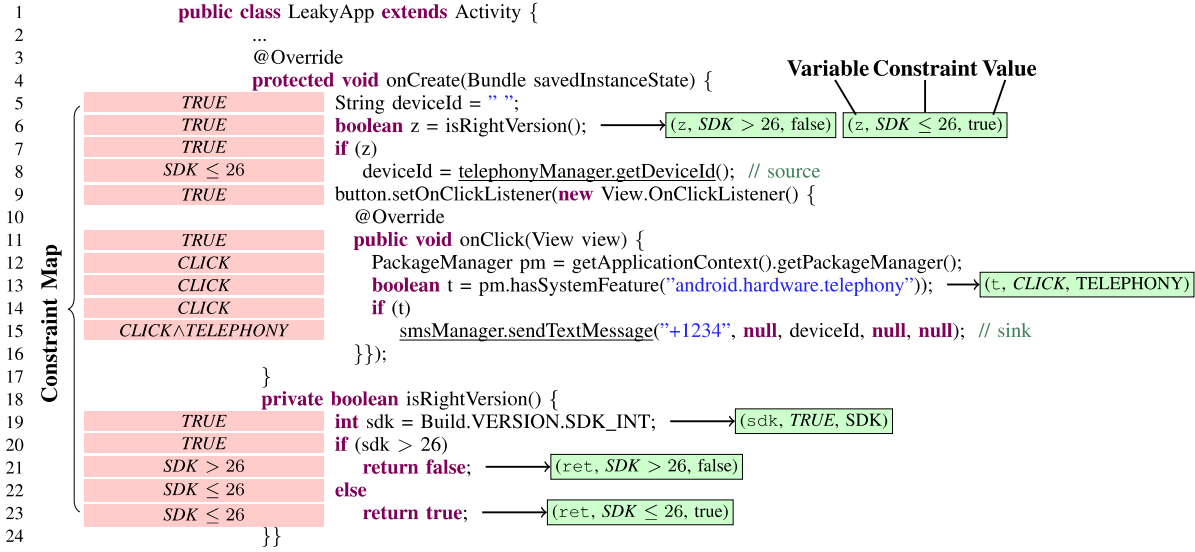


Figure 4.1: A motivating example.

4.1 A Motivating Example

Figure 4.1 shows an `Activity` that contains a data leak—a simplified example. The activity first reads the unique device identifier (line 8), stores it into variable `deviceId` before method `onClick` uses the variable and sends an SMS containing the identifier to the phone number “+1234” (line 15). State-of-the-art static taint-analysis tools for Android, e.g., FlowDroid [ARF⁺14], AmanDroid [WROR14] or DroidSafe [GKP⁺15], are capable of detecting such leaks with a high precision. However, as we observed during our study, these precision dimensions are insufficient when trying to understand *how* and *when* apps leak data.

While any of the mentioned taint-analysis tools report the leak in Figure 4.1, no tool reports that the leak can only occur under a specific execution path, as they are not path-sensitive [QWR18]. The app leaks the device identifier only when it executes the source and sink statements and every statement along the data-flow path. Their execution depends on three path conditions. First, the app must run the correct Android SDK version (line 7), second, the user must trigger the app to execute the `onClick` callback by pressing a button (line 11), and third, a special system feature has to be enabled on the execution device (line 14).

For an automatic analysis of the conditions under which a statement may be executed, we implemented the static analysis tool COVA. COVA computes a *constraint map* which associates each statement of a program with the path conditions required to execute that statement. Client analyses can use the constraint map computed by COVA to enrich or refine their analysis results. For a taint analysis that reports the data leak in the motivating example, the logical formula $C_{source} \wedge C_{sink}$ approximates the conditions under which the data leak happens:

- $C_{source} \equiv SDK \leq 26$: to execute the source, the Android SDK version must be smaller or equal to 26,
- $C_{sink} \equiv CLICK \wedge TELEPHONY$: to execute the sink, the user presses a button (`CLICK`) and the execution device support the telephony feature (`TELEPHONY`).

Both constraints C_{source} and C_{sink} can be simply queried from the constraint map COVA

computes. For taint analysis tools that also report a witness data-flow path, the constraints along the path can be logically joined if more precision is desired. Because path-sensitively analyzing all feasible paths in large programs is known to be expensive, COVA only assumes symbolic values for certain APIs, which we call *constraint-APIs*. For the motivating example, the constraint-APIs COVA considers are:

- `Build.VERSION.SDK_INT`
- `PackageManager.hasSystemFeature`
- `OnClickListener.onClick`

The list of constraint-APIs is configurable in COVA. By default, COVA tracks values from three kinds of constraint APIs for Android apps:

- UI interactions: UI callbacks that are responsible for handling user interactions.
- Environment settings: APIs for getting hardware and software configuration such as platform version, country etc.
- I/O operations: APIs for data input via I/O streams or file system, which are mainly from the `java.io` package.

COVA performs a context-, flow-, and field-sensitive data-flow analysis starting from the entry point of the program. In Figure 4.1, the entry point of the activity is `onCreate` and it is always executable, thus the statement at line 5 has the initial constraint `TRUE`. At each reachable invocation of a constraint-API, COVA generates a data-flow fact, simply referred to by *symbolic taint*. At line 19, COVA creates a symbolic taint (`sdk`, `TRUE`, `SDK`): `sdk` is the variable containing the value returned from the constraint-API; the second entry `TRUE` is the constraint under which the taint reaches the current statement; `SDK` stands for the symbolic value reading from `Build.VERSION.SDK_INT`. COVA then propagates such symbolic taints along the inter-procedural control-flow graph (ICFG) of the program and creates constraints over the symbolic values of taints whenever taints are used in conditional statements.

For instance, the return value of the method `isRightVersion()` is `true` when the SDK version is at most 26. The Android app further branches (indirectly) based on the version at the if-statement at line 20. COVA generates the symbolic taint (`ret`, `SDK ≤ 26`, `true`) for the statement at line 23; the taint encodes that the return value `ret` equals `true` when the version is `SDK ≤ 26`. This taint propagates back to the call site in line 6 as taint (`z`, `SDK ≤ 26`, `true`). Because Android apps intensively interact with user actions, COVA also symbolically represents them. COVA creates a constraint `CLICK` at the entry of the callback method `OnClickListener.onClick` and propagates this constraint to all statements reachable from this method. COVA propagates all symbolic taints from the constraint-APIs and simultaneously computes a constraint for each reachable statement based on available taints at the current statement. Once the data-flow propagation is completed, the constraint map is also computed.

Basically, COVA can be understood as a taint analysis without sinks and the constraint-APIs being the sources. Symbolic taints are propagated as usual along ICFG. At each program point where a path condition is associated with the variable of a symbolic taint, both the constraint carried by each symbolic taint and the constraint of the current statement are updated. Details about how COVA updates the constraints are introduced in Section 4.3.

4.2 Non-Distributivity

Computing the constraint map with a data-flow analysis as shown in Figure 4.1 is a *non-distributive* problem [Kil73], since the execution of a branch may simultaneously depend on two or more values of some constraint-APIs. Listing 4.1 shows a non-distributive case in which line 5 and 6 are executed depending on the values of both `manufacturer` and `model`. Assume the analysis uses *MODEL* to symbolically represent the value of `model` and *MANU* the value of `manufacturer`. A symbolic expression that precisely describes the path constraint for line 5 should be *MODEL.startsWith(MANU)*. However, this expression can only be computed if we know both the value of `model` and `manufacturer` at the same time when evaluating the if-statement at line 4. When using distributive frameworks like IFD-S/IDE [RHS95], each data-flow fact is propagated separately. This means when evaluating the if-statement at line 4 in the flow function, only one value is present. Assume we want to compute the path constraint after the if-statement with only the data-flow fact tracking the variable `model`. Since we do not know whether `manufacturer` is tracked or not and which value it holds, one could just generate a path constraint assuming `manufacturer` holds a symbolic value *X*, e.g., *MODEL.startsWith(X)*. Similarly, when evaluating the if-statement with only the data-flow fact tracking the variable `MANU`, the constraint could be *Y.startsWith(MANU)* if we assume `model` holds a symbolic value *Y*. Now the question is, how can one merge *MODEL.startsWith(X)* and *Y.startsWith(MANU)* to represent a final path constraint for line 5? It is difficult to get a path constraint as precise as *MODEL.startsWith(MANU)*.

Thus, the flow function that precisely evaluates the if statement at line 4 is not distributive/separable (i.e., $f(model \sqcap manufacturer) \neq f(model) \sqcap f(manufacturer)$). An analysis must *jointly* propagate all values to compute the final path constraint for a branch. Furthermore, each value of a constraint-API must be propagated throughout the whole program, since values can flow to fields of objects before the fields are re-accessed elsewhere and aliasing relations must be computed. Computation of aliases is also a well-known non-distributive problem [SDAB16], which is essential for a precise data-flow analysis.

```

1 public void checkDeviceName() {
2     String manufacturer = Build.MANUFACTURER; // constraint-API
3     String model = Build.MODEL; // constraint-API
4     if (model.startsWith(manufacturer)) {
5         doThis(); // MODEL.startsWith(MANUFACTURER)
6     } else
7         doThat();
8     }
9 }
```

Listing 4.1: A non-distributive case.

4.3 The Inter-procedural Constraint Analysis in COVA

We implemented the constraint analysis in COVA within the data-flow framework VASCO [PK13], which solves non-distributive inter-procedural data-flow problems in a context- and flow-sensitive manner.

4.3.1 The VASCO Framework

VASCO uses data-flow values (facts) as contexts for method calls and iterates a worklist of contexts. A value context *X* is defined as a pair $\langle method, entryValue \rangle$ with *entryValue* being the value at the entry of *method* with regard to a call node. Each context *X* has a worklist of

edges, denoted as $X.worklist$. Whenever a context X is initialized (see Algorithm 1), all edges in the CFG of $X.method$ are added to $X.worklist$ (line 13). This is different than in the original algorithm in which only nodes in CFG are iterated. Iterating the edges allows the flow functions to be branch-sensitive, such that our analysis can generate constraints based on which branch of a conditional statement is currently considered. In the procedure `INITCONTEXT`, the `IN` and `OUT` values are initialized with \perp except the entry node $head(X.method)$.

Algorithm 1 Initialization of a value context $X := \langle method, entryValue \rangle$

```

1: procedure INITCONTEXT( $X$ )
2:   ADD( $contextWorklist, X$ )
3:   ADD( $contexts, X$ )
4:    $X.worklist \leftarrow \emptyset$ 
5:    $EXIT[X] \leftarrow \perp$ 
6:   for all node  $n$  in CFG of  $X.method$  do
7:     if  $n = head(X.method)$  then
8:        $IN[X, n] \leftarrow X.entryValue$ 
9:     else
10:       $IN[X, n] \leftarrow \perp$ 
11:     if  $succs(n) \neq \emptyset$  then
12:       for all node  $s \in succs(n)$  do
13:         ADD( $X.worklist, \langle n, s \rangle$ )
14:          $OUT[X, n, s] \leftarrow \perp$ 
15:     else
16:       ADD( $X.worklist, \langle n, n \rangle$ )
17:        $OUT[X, n, n] \leftarrow \perp$ 

```

Algorithm 2 is a modified version of the interprocedural analysis in VASCO tailored for our constraint analysis that computes path constraints. Line 1 defines three global variables: a worklist of context-parametrized CFG edges that has to be processed ($contextWorklist$), a set of value contexts that have been created ($contexts$) and a transition table mapping a value context and a call site to the value context at the callee ($transitions$). The procedure `DOANALYSIS` starts with initializing a context $\langle entryPoint, BI \rangle$ for each entry point method of the program (e.g., for a classic Java program it is the main method). BI is the a data-flow fact holds at the entry point method. The procedure propagates data-flow facts from statement to statement along the ICFG. The flow functions define how a data-flow value changes when it flows from one statement to a successor. The procedure terminates when data-flow facts for all statements reach a fixed point, i.e., no new value context can be computed. The flow functions have generic arguments and they define which information to maintain, generate or kill for each control-flow edge. The flow functions accept a data-flow value $d \in D$ and a control-flow edge $\langle n, m \rangle$ of the ICFG as input, and output a new value $d' \in D$. Similarly to IFDS [RHS95], VASCO differentiates between the following four kinds of functions regarding an edge $\langle n, s \rangle$:

- `NORMALFLOWFUNC`: handles intra-procedural flows where n is not a call site.
- `CALLLOCALFLOWFUNC`: handles intra-procedural flows where n is a call site. It propagates the values of local variables not used at the call site.
- `CALLENTRYFLOWFUNC`: handles an inter-procedural flow from call site n to the first statement m of a callee. It typically maps actual method arguments to formal parameters.

- **CALLEXITFLOWFUNC**: handles an inter-procedural flow from a return statement n to the successor m of a call site. It is the inverse of **CALLENTRYFLOWFUNC** and maps parameters and return value back to the call site.

Algorithm 2 Modified interprocedural analysis in VASCO

```

1: global: contextWorklist, contexts, transitions
2:
3: procedure DOANALYSIS
4:   for all entryPoint  $\in$  entryPoints do
5:     INITCONTEXT( $\langle$ entryPoint, BI $\rangle$ )
6:   while contextWorklist  $\neq \emptyset$  do
7:      $X \leftarrow \text{GETLAST}(\text{contextWorklist})$ 
8:     if  $X.\text{worklist} \neq \emptyset$  then
9:        $\langle n, s \rangle \leftarrow \text{POLLFIRST}(X.\text{worklist})$ 
10:      if  $n \notin \text{heads}(X.\text{method})$  then
11:         $IN[X, n] \leftarrow \sqcup \{OUT[X, p, n] \mid p \in \text{preds}(n)\}$ 
12:         $in \leftarrow IN[X, n]$ 
13:         $out \leftarrow \perp$ 
14:        if  $n$  does not contain a method call then
15:           $out \leftarrow \text{NORMALFLOWFUNC}(X, n, s, in)$ 
16:        else
17:           $m \leftarrow \text{CALLEEMETHOD}(n)$ 
18:           $e \leftarrow \text{CALLENTRYFLOWFUNC}(X, m, n, s, in)$ 
19:           $X' := \langle m, e \rangle$ 
20:          if  $X' \notin \text{contexts}$  then
21:            INITCONTEXT( $X'$ )
22:          else
23:             $x \leftarrow \text{EXIT}[X']$ 
24:             $r \leftarrow \text{CALLEXITFLOWFUNC}(X, m, n, s, x)$ 
25:             $l \leftarrow \text{CALLLOCALFLOWFUNC}(X, n, s, in)$ 
26:             $out \leftarrow l \sqcup r$ 
27:            add edge  $(\langle X, n \rangle, X')$  to transitions
28:          if  $out \neq OUT[X, n, s]$  then
29:            for all  $s' \in \text{succs}(s)$  do
30:              ADD( $X.\text{worklist}, \langle s, s' \rangle$ )
31:             $OUT[X, n, s] \leftarrow out$ 
32:        else
33:          REMOVE(contextWorklist,  $X$ )
34:           $exit \leftarrow \sqcup \{OUT[X, n, n] \mid n \in \text{tails}(X.\text{method})\}$ 
35:          if  $exit \neq \text{EXIT}[X]$  then
36:             $\text{EXIT}[X] \leftarrow exit$ 
37:          for all edges  $(\langle X', c \rangle, X)$  in transitions do
38:            ADD(contextWorklist,  $X'$ )
39:          for all  $s \in \text{succs}(c)$  do
40:            ADD( $X'.\text{worklist}, \langle c, s \rangle$ )

```

4.3.2 Analysis Domain

The domain D for our constraint analysis in VASCO is two-dimensional: $\mathbb{C} \times 2^{\mathbb{T}}$, where \mathbb{C} is a constraint domain and \mathbb{T} is a taint domain. We use $\perp \in D$ to denote an unknown fact. If data-flow fact $(C, T) \in \mathbb{C} \times 2^{\mathbb{T}}$ holds at statement n , then C is the constraint under which statement n is reachable. We seed the data-flow propagation with the fact $BI = (TRUE, \emptyset)$ at each entry point method of the application. The constraint is $TRUE$, as the entry point method is always reachable. At the entry point, the set T is the empty set as no constraint-API call has been encountered.

In general, T is the set of symbolic taints generated at constraint-APIs reaching statement n . Each symbolic taint is a triple $(a, c, v) \in \mathbb{T}$ and consists of an access path (a local variable followed by a finite sequence of fields [Deu94]). The access path a encodes how the value of the constraint-API is heap-referenceable at statement n . The constraint c describes under which conditions the symbolic taint holds. Value v holds the actual value of a . Dependent of the type of value v there are three kinds of symbolic taints:

- **source taints:** v is the value reading from a constraint-API, it is represented symbolically, e. g., $(t, CLICK, TELEPHONY)$ in Figure 4.1
- **imprecise taints:** they are derived from existing taints. v is not exactly the value reading from a constraint-API, but somehow depended on it. For example, Consider a statement $b = a + c$ and a source taint (a, C, A) propagated to this statement. COVA creates an imprecise taint $(b, C, im(A))$, meaning the value of b is somehow affected by A .
- **concrete taints:** Constraint taints are generated by COVA to track concrete values of primitive types if they are depended on values from constraint-API. In such cases, v simply holds the concrete value. For example, $(ret, SDK \leq 26, true)$ in Figure 4.1 is a concrete taint.

We will introduce more about the creation of these taints in Section 4.3.3. Note, at a statement n the constraint c of a taint within the set T and the constraint C are not necessarily equal. For example, in Figure 4.1 the constraint C to reach line 6 is $TRUE$, but the constraint c of the taint $(z, SDK \leq 26, true)$ is $SDK \leq 26$.

The meet operator \sqcup is the logical disjunction \vee for the constraint domain and set union \cup for the taint domain, i.e., $(C_1, T_1) \sqcup (C_2, T_2) = (C_1 \vee C_2, T_1 \cup T_2)$ for two data-flow facts (C_1, T_1) and (C_2, T_2) . For any (C, T) , we define $(C, T) \sqcup \perp = (C, T)$. In the following we separate the flow functions into two parts: the flow functions of the taint domain and of the constraint domain. Let $\langle n, s \rangle$ be a control-flow edge and let (C_{in}, T_{in}) refer to the data-flow fact before n and (C_{out}, T_{out}) denote the fact before m , then we describe the flow function F in form of the result set $(C_{out}, T_{out}) = F(C_{in}, T_{in})$. The analysis operates Jimple, which is a three-addressed code reconstructed from Java bytecode in Soot [LBLH11b]. We define the analysis based on the statements affecting either C or T of a data-flow fact (C, T) . In the following, we will introduce the flow functions of the taint domain and constraint domain separately.

4.3.3 Flow Functions of the Taint Domain

The flow functions of the taint domain mostly follow standard k-limiting access-path based taint tracking data-flow propagation [ARF⁺14, SDAB16]. An access path $x.*$ is a base variable followed by a finite sequence of field accesses, e. g., $x, x.f, x.f.g$. COVA uses Boomerang [SDAB16] to compute the set of aliases for any access path to support strong updates. We use $Aliases(x.*)$ to denote this on-demand alias analysis in Boomerang that returns all aliases of an access path $x.*$ (including $x.*$).

4.3 THE INTER-PROCEDURAL CONSTRAINT ANALYSIS IN COVA

NormalFlowFunc(X, n, s, in): Consider a CFG edge $\langle n, s \rangle$ and an incoming data-flow fact $in = (C_{in}, T_{in})$, this flow function consider the following cases to compute the taint set T_{out} of the outgoing data-flow fact $out = (C_{out}, T_{out})$:

- $n : x = \bullet$ (Any Assignment): The symbol \bullet is a placeholder representing an irrelevant argument. The flow function kills any tainted access path starting with local variable x or its aliases, i. e.,

$$T_{out} = T_{in} \setminus \{(a.\ast, \bullet, \bullet) \in T_{in} | a \in Aliases(x)\}$$

In the following, let $T_{in}^- = T_{in} \setminus \{(a.\ast, \bullet, \bullet) \in T_{in} | a \in Aliases(x)\}$.

- $n : x = y$ (Local Assignment): if any incoming access path matches variable y at the right side, x and its aliases are added to the outgoing taint set. If there is a taint $(y, c, v) \in T_{in}$, then

$$T_{out} = T_{in}^- \cup \{(a, c, v) | a \in Aliases(x)\}$$

- $n : x = z$ (Constant Assignment): if z is a constant value and the constraint C_{in} is not equal to $TRUE$, then the flow function creates a concrete taint (x, C_{in}, z) and

$$T_{out} = T_{in}^- \cup \{(x, C_{in}, z)\}.$$

- $n : x.f = y$ (Non-static Field Store): if any incoming access path matches the variable y at the right side, then the left side and its aliases are added to the outgoing taint set. If there is a taint $(y, c, v) \in T_{in}$, then

$$T_{out} = T_{in} \setminus \{(a.f, \bullet, \bullet) | a \in Aliases(x)\} \cup \{(a.f, c, v) | a \in Aliases(x)\}$$

- $n : x = S.f$ (Static Field Load): if the right side is a static field and this field is labeled as a constraint-API, the flow function generates a source taint $(x, C_{in}, sym(S.f))$ and

$$T_{out} = T_{in}^- \cup \{(x, C_{in}, sym(S.f))\}$$

Hereby C_{in} is the constraint that reaches statement n and sym is a function that generates a unique symbolic value for a given constraint-API. For instance, $(sdk, TRUE, SDK)$ is a source taint created at line 19 in Figure 4.1.

- $n : S.f = y$ (Static Field Store): if any incoming access path matches the variable y at the right side, then the left side and its aliases are added to the outgoing taint set. If there is a taint $(y, c, v) \in T_{in}$, then

$$T_{out} = T_{in} \setminus \{(a, \bullet, \bullet) | a \in Aliases(S.f)\} \cup \{(a, c, v) | a \in Aliases(S.f)\}$$

- $n : x = y \oplus z$ (Binary Operation): \oplus is a binary operator. If at least one variable at the right side is tainted, then the flow function creates an imprecise taint. If $(y, \bullet, v_1) \in T_{in}$ and $(z, \bullet, v_2) \in T_{in}$, then

$$T_{out} = T_{in}^- \cup \{(x, C_{in}, im(v_1, v_2))\}.$$

The symbolic value $im(v_1, v_2)$ means the value of x is affected by v_1 and v_2 . In case the taint of one variable is missing, for example, $(y, \bullet, v_1) \in T_{in}$ and $(z, \bullet, \bullet) \notin T_{in}$, the flow function creates an imprecise taint $(x, C_{in}, im(v_1))$.

- $n : x = \ominus y$ (Unary Operation): \ominus the flow function is analog to the binary operation case and creates an imprecise taint.
- $n : \mathbf{return} \ z$ (Return Statement): if z is constant and constraint C_{in} is not equal to $TRUE$, then the flow function creates a concrete taint (ret, C_{in}, z) and

$$T_{out} = T_{in}^- \cup \{(ret, C_{in}, z)\}$$

Here we use a synthetic access path ret to symbolically represent the returned variable (e.g., $(ret, SDK \leq 26, true)$ at line 23 in Figure 4.1).

CallLocalFlowFunc(X, n, s, in): Consider a CFG edge $\langle n, s \rangle$ in which $n : r = o.f(a_1, \dots, a_k)$ is a call statement. $in = (C_{in}, T_{in})$ is the incoming data-flow fact. Apart from killing taints that start with the overwritten variable r and its aliases, the following cases are considered:

- If f is a constraint-API, then the flow function adds a source taint $(r, C_{in}, sym(f))$ to the outgoing taint set:

$$T_{out} = T_{in} \setminus \{(a.*, \bullet, \bullet) | a \in Aliases(r)\} \cup \{(r, C_{in}, sym(f))\}$$

- If there is a taint $(o, \bullet, v) \in T_{in}$, then the flow function creates an imprecise taint $(r, C_{in}, im(v))$:

$$T_{out} = T_{in} \setminus \{(a.*, \bullet, \bullet) | a \in Aliases(r)\} \cup \{(r, C_{in}, im(v))\}$$

Note that if f is a method from the `java.lang.String` class, the current COVA can handle some string operations (e.g., `equals`, `length`, `contains`, `startsWith`, `endsWith`). Hauptmeier extended COVA to support handle string operations more precisely in his master thesis [Hau21]. Instead of an imprecise taint, a string taint is created in this extension.

CallEntryFlowFunc(X, m, n, s, in): Consider a call statement $n : o.f(a_1, \dots, a_k)$ or $n : r = o.f(a_1, \dots, a_k)$, the callee $m : O.f(p_1, \dots, p_k)$ of n , and the incoming data-flow fact $in = (C_{in}, T_{in})$, the following cases are considered:

- Any taint in T_{in} , whose access path's local variable is an argument of the call (a_1, \dots, a_k) , is mapped to the respective parameter access path within the callee m .
Let $T_1 = \{(p_i.*, c, v) | (a_i.*, c, v) \in T_{in}\}$.
- If there exists a_i being constant and C_{in} is not $TRUE$, the flow function creates a concrete taint (p_i, C_{in}, a_i) for the respective parameter p_i within the callee.
Let $T_2 = \{(p_i, C_{in}, a_i) | a_i \text{ is constant}\}$.
- If $O.f$ is a non-static method, then the receiver variable o is mapped to the this-reference inside the callee.
Let $T_3 = \{(this.*, c, v) | (o.*, c, v) \in T_{in}\}$.
- Access paths $S.*$ representing static data-flow facts, i.e., public static fields, are mapped to the callee, as the callee may change their values.
Let $T_4 = \{(S.*, c, v) | (S.*, c, v) \in T_{in}\}$.

In summary, the outgoing taint set holds at the callee is

$$T_{out} = T_1 \cup T_2 \cup T_3 \cup T_4$$

CallExitFlowFunc(X, m, n, s, in): Consider a call statement $n : o.f(a_1, \dots, a_k)$ or $n : r = o.f(a_1, \dots, a_k)$, the callee $m : O.f(p_1, \dots, p_k)$ of n , and the incoming data-flow fact $in = (C_{in}, T_{in})$, the following cases are considered:

- The flow functions maps any taint from callee back to caller inversely as in **CALLENTRYFLOWFUNC**. In addition to that, aliases at the call site are also considered.
Let $T_1 = \{(a.*, c, v) | (p_i.*, c, v) \in T_{in} \wedge a \in Aliases(a_i)\}$.
- All taints with access path starting with the this-reference is mapped to o and its aliases at the call site.
Let $T_2 = \{(a.*, c, v) | (this.*, c, v) \in T_{in} \wedge a \in Aliases(o)\}$
- Taints $(S.*, c, v)$ representing public static fields are mapped back to the call site.
Let $T_3 = \{(S.*, c, v) | (S.*, c, v) \in T_{in}\}$.
- If the call site is an assignment $n : r = o.f(a_1, \dots, a_k)$, all taints with ret being its access path will be propagated back to the call site.
Let $T_4 = \{(r, c, v) | (ret, c, v) \in T_{in}\}$,

In summary, the outgoing taint set is

$$T_{out} = T_1 \cup T_2 \cup T_3 \cup T_4$$

4.3.4 Flow Functions of the Constraint Domain

To compute path constraint C_{out} based on a given statement n and a path constraint C_{in} and a taint set T_{in} which hold before n , COVA conjoins C_{in} with an extending constraint C_{new} which is created at conditional statements or invocation of UI callbacks, i.e., $C_{out} = C_{in} \wedge C_{new}$. In the following we focus on introducing how C_{new} is computed.

NormalFlowFunc(X, n, s, in): Consider a statement $n : \text{if } (a \oplus b)$ with a comparison operator \oplus , COVA creates C_{new} based on the available taints in T_{in} . The following cases are considered:

- T_{in} contains only taints for variable a (analog for b). Assume there are k taints $(a, c_i, v_i) \in T_{in}$ with $i \in \{1, \dots, k\}$.
 1. If b is a constant value, COVA creates a constraint e_i by substituting the variable a in the formula $a \oplus b$ with its value v_i and conjoining the result with c_i for each taint (a, c_i, v_i) , i.e., $e_i \equiv (v_i \oplus b) \wedge c_i$ if the successor statement m is in the *TRUE* branch of n . For the case m is in the *FALSE* branch $e_i \equiv \neg(v_i \oplus b) \wedge c_i$.
 2. If b is an untracked variable, the formula $v_i \oplus b$ is replaced by the imprecise constraint $im(v_i)$ in e_i , namely: $e_i \equiv (im(v_i) \oplus b) \wedge c_i$ for the *TRUE* branch and $e_i \equiv (\neg im(v_i) \oplus b) \wedge c_i$ for the *FALSE* branch.

C_{new} is computed as follows :

$$C_{new} \equiv (\bigvee_{i=1}^k e_i) \vee c_{miss}, \text{ where } \bigvee_{i=1}^k c_i \vee c_{miss} \equiv C_{in}.$$

Intuitively, if COVA would track all variables in the program, we should know the value of a in all constraints under which this statement **if** $(a \oplus b)$ is reachable. So the taints $(a, c_i, v_i) \in T_{in}$ must share the following invariant $\bigvee_{i=1}^k c_i \equiv C_{in}$. We call such taint set

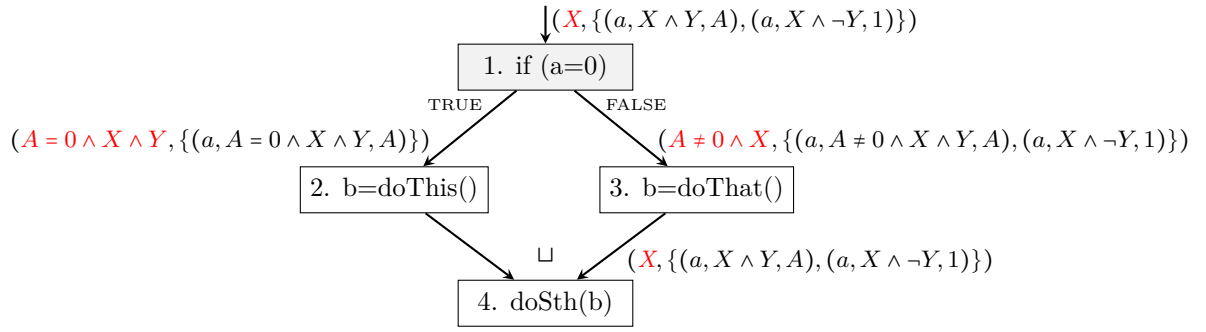


Figure 4.2: An example shows complete taint sets.

complete regarding the variable a . Figure 4.2 illustrates such a case. The path constraints are highlighted in red. Before the *if*-statement at line 1, the path constraint C_{in} is X and the taint set contains two taints $t_1 := (a, X \wedge Y, A)$ and $t_2 := (a, X \wedge \neg Y, 1)$. The taint set is complete for variable a , since the invariant holds: $\bigvee_{i=1}^2 c_i \equiv (X \wedge Y) \vee (X \wedge \neg Y) \equiv X \equiv C_{in}$. In such case, a can not be any other value, rather than A or 1. So c_{miss} is simply *FALSE*. The path constraint C_{out} for taking the *TRUE* branch is computed as follows:

$$\begin{aligned}
 C_{out} &\equiv C_{in} \wedge C_{new} \\
 &\equiv C_{in} \wedge (e_1 \vee e_2) \vee c_{miss} \\
 &\equiv C_{in} \wedge (e_1 \vee e_2) \vee FALSE \\
 &\equiv C_{in} \wedge (A = 0 \wedge X \wedge Y) \vee (1 = 0 \wedge X \wedge \neg Y) \vee FALSE \\
 &\equiv C_{in} \wedge (A = 0 \wedge X \wedge Y) \vee FALSE \vee FALSE \\
 &\equiv X \wedge (A = 0 \wedge X \wedge Y) \\
 &\equiv A = 0 \wedge X \wedge Y
 \end{aligned}$$

The constraint of each taint will also be updated with the respective e_i . In the following, we denote the updated taints with t'_1 and t'_2 :

$$\begin{aligned}
 t'_1 &= (a, e_1, A) = (a, A = 0 \wedge X \wedge Y, A) \\
 t'_2 &= (a, e_2, 1) = (a, FALSE, 1)
 \end{aligned}$$

COVA only propagates taints whose constraints are satisfiable, i. e., they are not equal to *FALSE*. Thus, t'_2 won't be propagated to the *TRUE* branch. The computation regarding the *FALSE* branch is analog. When merging taint sets from both the *TRUE* branch and *FALSE* branch at line 4, taints with the same value are merged to one, i. e., $(a, A = 0 \wedge X \wedge Y, A)$ and $(a, A \neq 0 \wedge X \wedge Y, A)$ becomes $(a, X \wedge Y, A)$.

In practice, we usually have an *incomplete* taint set T_{in} , which the invariant is violated. It means we only know the value of a under some constraint, but not for other constraints that can also drive the execution to the *if*-statement. Figure 4.3 illustrates such a case. The taint set before the statement *if(a<0)* indicates a to hold the value 1 under the constraint X . Before evaluating the statement *if(a<0)*, the constraint $C_{in} \equiv \text{TRUE}$, which means the statement is always reachable. The taint set for a is incomplete, because COVA cannot propagate a taint for a under the constraint $\neg X$ as a holds an *unknown*

4.3 THE INTER-PROCEDURAL CONSTRAINT ANALYSIS IN COVA

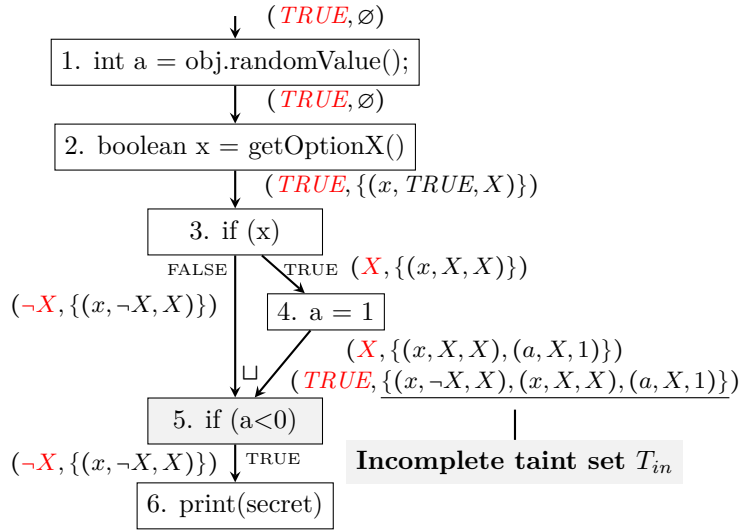


Figure 4.3: An example shows an incomplete taint set. Assume `getOptionX()` is a constrain-API whose return value is represented by the symbolic value X .

return value of the method call `obj.randomValue()`. For variable a , c_{miss} is $\neg X$ and the C_{out} for the $TRUE$ branch is:

$$\begin{aligned}
 C_{out} &\equiv C_{in} \wedge C_{new} \\
 &\equiv C_{in} \wedge (e_1 \vee c_{miss}) \\
 &\equiv C_{in} \wedge (e_1 \vee \neg X) \\
 &\equiv C_{in} \wedge ((1 < 0 \wedge X) \vee \neg X) \\
 &\equiv C_{in} \wedge \neg X \\
 &\equiv \text{TRUE} \wedge \neg X \\
 &\equiv \neg X
 \end{aligned}$$

- T_{in} contains taints for both a and b .

Assume there are k taints $(a, c_i, v_i) \in T_{in}$ and q taints $(b, d_j, w_j) \in T_{in}$. COVA computes e_{ij} by substituting a and b analogously as in the previous case. $e_{ij} = (v_i \oplus w_j) \wedge c_i \wedge d_j$ for the $TRUE$ branch and $e_{ij} = \neg(v_i \oplus w_j) \wedge c_i \wedge d_j$ for the $FALSE$ branch. Let c_{miss} and d_{miss} be the missing-constraints for variable a and b respectively, $C_{new} = (\bigvee_{ij} e_{ij}) \vee c_{miss} \vee d_{miss}$.

For a switch-statement, the flow function is analog.

CallEntryFlowFunc(X, m, n, s, in): For a call $n : o.f(a_1, \dots, a_k)$, $C_{out} = C_{in} \wedge \text{sym}(f)$ if f is a callback from the constrain-APIs. sym is a function of COVA returns a symbolic value for f indicating f is invoked.

CallLocalFlowFunc(X, n, s, in): This function is the same as **CALLENTRYFLOWFUNC**.

CallExitFlowFunc(X, m, n, s, in): The constraint stays unchanged.

4.3.5 Termination

The termination of VASCO is guaranteed by the monotonicity of the flow functions and the finiteness of the lattice of data-flow facts. The flow functions are monotonic, since a variable can either be killed or stay tainted. The on-demand alias analysis queried in the flows functions will always terminate though a timeout we set up for Boomerang. The path constraints are computed based on the taint set and the evaluated statement. The lattice of the taint domain is finite, because we can only extract a finite number of k -limited access paths from the program. Once the taint set achieves the fixed-point, the path constraint will also achieve the fixed-point, since its computation is based on the taint set. VASCO terminates once a fixed point is reached. The result computed by VASCO is a map from $(ctx, n) \in Context \times Statement$ to data-flow facts in $\mathbb{C} \times 2^T$. The C_{in} values before each statement are used to extract the constraint map. Since a statement n can be in multiple contexts, COVA merges the C_{in} values of n from different contexts by logical disjunction.

4.4 Implementation

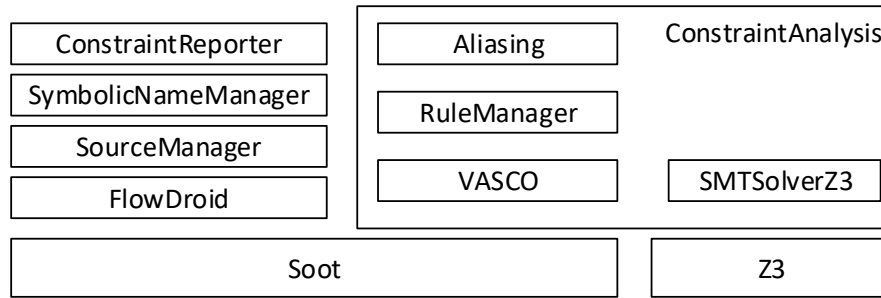


Figure 4.4: COVA’s main components.

We implemented COVA that computes partial path constraints for Java and Android applications based on predefined constraint-APIs. The constraint-APIs are given in configuration files. Figure 4.4 shows the main components in COVA. The core part of COVA computing path constraints is the **ConstraintAnalysis** that is built on top of both Soot [LBLH11b] and Z3 [dMB08]. Z3 is a widely used theorem prover from Microsoft Research. COVA relies on it to evaluate whether a constraint is satisfiable and only propagates taints with satisfiable constraints.

The class **ConstraintAnalysis** extends the **ForwardInterProceduralAnalysis** in VASCO to perform a forward interprocedural constraint analysis as we introduced in Section 4.3. The class **SMTSolverZ3** transforms constraints in COVA’s internal representation to constraints in Z3. The current implementation of COVA fully support constraints in boolean propositional logic, equality logic and linear arithmetic logic. It partially supports string operations, introduced in [Hau21]. The class **RuleManager** is responsible for managing a list of rules applied in the flow functions introduced in Section 4.3. Each rule implements an interface **IRule** as shown in Figure 4.5. The class **UIConstraintCreationRule** is responsible to generate constraints at invocations of UI callbacks. While the class **TaintPropagationRule** is the implementation of all flow functions for the taint domain introduced in Section 4.3.3, the class **TaintConstraintCreationRule** implements flow functions for the constraint domain introduced

in Section 4.3.4.

To reason about aliasing relationship, the `Aliasing` class in Figure 4.4 uses Boomerang [SDAB16], which is a demand-driven flow- and context-sensitive pointer analysis. For Java applications, the call graphs are constructed by Soot. For Android applications, we use the call graphs constructed by FLOWDROID [ARF⁺14]. The `SourceManager` is responsible for determining whether a statement contains a constraint-API. The `SymbolicNameManager` assigns variable to unique symbolic expression and manages the mapping from a symbolic expression to a constraint-API. The `ConstraintReporter` manages the output of COVA. It provides interfaces to query path constraint for a single statement, output path constraint for every reachable statement in the analysis as annotation in Jimple files, etc.

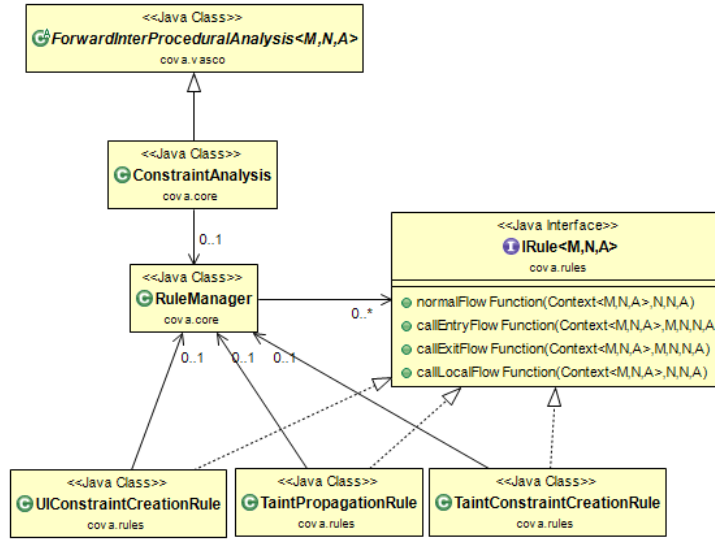


Figure 4.5: The class diagram around constraint analysis.

4.5 Evaluation of COVA

To be able to judge the confidence in the results COVA reports, we developed a micro benchmark suite called CONSTRAINTBENCH. CONSTRAINTBENCH consists of 100 specially crafted test programs with labeled ground truth. It covers test cases in which primitives or heap objects used in conditional statements, nested conditional statements, intra- and inter-procedural conditional dependencies, callback invocations, indirect conditional dependencies, string operations, etc. Originally, as we published our work in [LBS19], COVA did not evaluate string operations. As we introduced in Section 4.3.2, COVA propagates imprecise taints that are derived from source taints, so it could produce imprecise path constraints that are less expressive than one would expect. Listing 4.2 shows such a test case. Let *FA* symbolically represents the value read from

```

1 public void test() {
2     String a = Configuration.fieldA; // constraint-API
3     if (a.equals("abc")) {
4         System.out.println();// FA = "abc"
5     }
6 }

```

Listing 4.2: Test case *String1* from CONSTRAINTBENCH.

```

1 public void test() {
2     int d = Configuration.featureD(); // constraint-API
3     int f = Configuration.featureF(); // constraint-API
4     int[] arr = new int[2];
5     arr[0] = d;
6     arr[1] = f;
7     if (arr[0] > 0) {
8         System.out.println();// (D>0)
9     }
10 }

```

Listing 4.3: Test case *Array1* from CONSTRAINTBENCH.

Configuration.fieldA. A path constraint that precisely expresses how line 4 is reachable at runtime could be $FA = \text{"abc"}$. The path constraint COVA computed was $im(FA)$, which only tells that the execution of statement at line 5 is depended on FA . Hauptmeier extended COVA to support some simple string operations in his master thesis [Hau21]. Thus, the path constraint computed by the current COVA is as precise as the expected $FA = \text{"abc"}$.

Because COVA computes path constraints, the labeled ground truth for a test program is a mapping from a statement to a symbolic formula that expresses the path constraint on the symbolic values over constraint-APIs configured for COVA. If the computed path constraint for a test program is equivalent to the specified path constraint, it is considered as a true positive (TP) case, otherwise, a false positive (FP) case. If no path constraint is computed (i.e., \perp), it is considered as false negative.

Table 4.1 shows the results of the current COVA evaluated on CONSTRAINTBENCH. On

Table 4.1: Evaluation results of COVA on CONSTRAINTBENCH.

No.	Test Category	No. of Tests	TP	FP
1	PrimTypes	19	19	0
2	NonPrimTypes	3	3	0
3	InstanceField	10	10	0
4	StaticField	7	6	1
5	Callbacks	2	2	0
6	Indirect	4	4	0
7	Infeasible	2	2	0
8	InterProcedural	10	9	1
9	Loops	5	5	0
10	Recursion	1	1	0
11	SwitchStmts	2	2	0
12	TryBlock	1	1	0
13	Mixed	2	2	0
14	Imprecise	19	11	8
15	Array	1	0	1
16	Reflection	1	0	1
17	Special Classes	3	0	3
18	StringOperation	8	7	1
	Σ	100	85	15

this suite, COVA achieves a precision of 85% (85/100) and a recall of 85% (85/100). Because the test programs in CONSTRAINTBENCH are small, COVA computes a constraint map for each of them. There is no statement where COVA does not compute a path constraint. However, false negatives can happen in real-world applications, if the call graph COVA operates on is incomplete. 15 cases are false positives. Most false positives are imprecise path constraints, where imprecision is introduced because of library methods that can not be evaluated by COVA. Other false positives cases are due to unhandled language features such as array, reflection, etc. As the example in Listing 4.3 shows, the expected path constraint for the statement at line 8 is $D > 0$. The path constraint COVA computes for line 8 is *TRUE*, because array access is not handled.

4.6 COVA-assisted Qualitative Analysis of Android Taint-Analysis Results

Using COVA, we conducted an experiment to understand the nature of taint flows detected by a static Android taint-analysis tool in real-world applications, and potential avenues to eliminating false positives among those taint flows. Figure 4.6 describes the workflow of COVA when used with a taint-analysis tool. COVA accepts as input an Android application in bytecode format and a set of pre-defined constraint-APIs. COVA then computes the path constraints, i.e., the constraint map, which depend on values from the constraint-APIs. The constraint map computed by COVA is used to refine the data leaks reported by an existing taint-analysis tool, i.e., leaks can be reported with path constraints. We chose FLOWDROID [ARF⁺14] as our evaluation tool, since it is well maintained and, according to previous studies [QWR18, PBW18], beats other tools both in accuracy and efficiency. Although we applied COVA to a taint analysis, COVA is applicable to any other client analysis that can benefit from path information. The experiment intends to answer the following research questions:

- RQ1. What types of taint flows does FLOWDROID report? How common is each type?
- RQ2. How large is the fraction of easily actionable unconditional intra-procedural taint flows, and what characteristics do these flows have?

We next address both questions one after the other.

RQ1. What types of taint flows does FlowDroid report? How common is each type?

Methodology

We randomly sampled 2,000 Android apps from the AndroZoo dataset [ABKT16]. All sampled apps were available in popular app stores (Google Play and Anzhi Market) between year 2016 and 2018. These criteria ensure that we report on the real-world apps from recent years. The apps can be downloaded from this link¹. We used FLOWDROID v2.5.1 in its default configuration. In this configuration, FLOWDROID lists 47 methods as sources² and 122 as sinks. We applied FlowDroid to these 2,000 apps and it reported 1,022 apps to contain data leaks. FLOWDROID reported 28,176 taint flows for these 1,022 apps, which makes it intractable to study every single taint flow in every app. Thus, our methodology follows these two steps:

¹<https://www.kaggle.com/covaanalyst1/cova-dataset>

²46 sources are listed in the configuration file `SourcesAndSinks.txt` and 1 source `android.app.Activity.findViewById(int)` is treated specially by only considering password input fields.

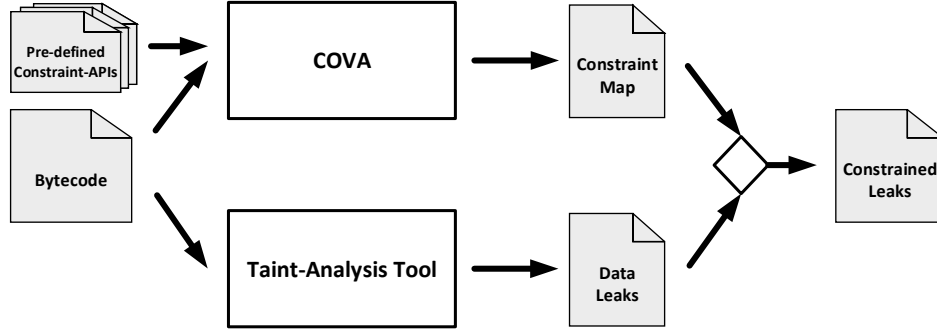


Figure 4.6: The workflow of applying COVA to taint-analysis results.

Step 1: We measured which *source-sink-pairs* appeared in the taint flows and chose the top 3 source-sink-pairs among intra- and inter-procedural taint flows (see Table 4.2) for our case study, since these source-sink-pairs dominate a large amount of taint flows, and among most (88%) of the remaining pairs each pair only appeared in fewer than 50 taint flows (out of 28,176 in total). To determine apps for our case study, we applied *stratified random sampling*: the apps with taint flows using these 6 source-sink-pairs are divided into 6 groups, one for each pair, which we here label with A to F. Due to large amount of reverse engineering and manual work involved in the inspection, we only sampled 10% of the apps of each group. The manual inspection was done in pair by two of the authors. This step allowed us to get an impression on the most common taint flows FLOWDROID reports and potentially identify false-positive patterns that can be used to filter false positives before we classified the taint flows with the path constraints COVA computes in the next step.

Table 4.2: Stratified sampling the top source-sink-pairs among the taint flows.

Gr.	Source	Sink	#Taint Flows	#Apps	#Sampled Apps
Intra-procedural					
A	java.net.URL.openConnection	java.net.HttpURLConnection.setRequestProperty	2,193	535	54
B	android.os.Handler.obtainMessage	android.os.Handler.sendMessage	1,410	199	20
C	java.net.HttpURLConnection.getOutputStream	java.io.OutputStream.write	194	166	17
Inter-procedural					
D	android.database.Cursor.getString	android.app.Activity.startActivityForResult	1,440	156	16
E	java.net.URL.openConnection	java.net.HttpURLConnection.setRequestProperty	862	291	30
F	android.database.Cursor.getString	android.os.Bundle.putString	847	85	9

Step 2: We conducted an experiment in which we applied both FLOWDROID and COVA to the apps in our dataset. An app is passed to both FLOWDROID and COVA (see Figure 4.6). The experiment was designed to classify the taint flows with the following *types*:

- *UI-constrained taint flows* are dependent on UI actions.
- *Configuration-constrained taint flows* are dependent on hardware/software configuration.
- *I/O-constrained taint flows* are dependent on data inputs through streams or file system.

Whenever a taint flow is reported by FLOWDROID, we conjoin the constraints of the source and the sink computed by COVA to obtain the leak-constraint and use it to classify this taint flow. For instance, the leak in our motivating example (see Figure 4.1) will be classified to both UI-constrained and Configuration-constrained, since the leak-constraint $SDK \leq 26 \wedge CLICK \wedge TELEPHONY$ contains symbolic values which stand for configuration (SDK

and *TELEPHONY*) and UI action (*CLICK*) at the same time. This classification also works for imprecise path constraints (e.g., $im(SDK)$ means some unmodeled operation over *SDK*.) COVA computes, since they still contain the symbolic values stand for the relevant constraint-APIs. We collected a list of constraint-APIs from the Android Platform (API level 27) that COVA ought to track:

- 335 APIs for UI actions, which are UI callbacks. We first scanned the whole Android platform with gestural keywords such as click, scroll, etc., to extract a list of possible UI callbacks. Based on this list, callbacks were manually selected.
- 448 APIs for hardware and software configuration. We collected the APIs based on the official Android guide of device compatibility [Goo18a].
- 120 APIs for data input via I/O streams or file system, which are mainly from the *java.io* package.

The selection of the APIs was done by pair-reviewing by two researchers. The list is publicly available with COVA.

Experiment Setup: We set a timeout of 30 minutes per app for COVA. COVA terminated its analysis and computed a complete constraint map for 315 apps. (In cases in which analysis times out, this was most often due to slow constraint solving in Z3, see Section 4.8.) For the remaining 707 apps, COVA only computed partial constraint maps. The experiment was conducted on a virtual machine with an Intel Xeon CPU running on Debian GNU/Linux 9 with Oracle’s Java Runtime version 1.8 (64 bit). The maximal heap size of the JVM was set to 24 GB.

Results

Figure 4.7 shows the different types of taint flows and their fractions in our study. While the false positives were all identified in step 1, the fractions of other types (UI-constrained, Configuration-constrained, I/O-constrained, Infeasible, Unconstrained and intersections) were computed in step 2. The infeasible taint flows are those with unsatisfiable leak-constraints reported by COVA. The fraction of the unconstrained taint flows is only an upper bound. For apps on which COVA timed out, if there is no constraint computed for the source and sink statements of a taint flow in the partial constraint map, we assigned this taint flow with the type “Unconstrained”. In step 1, we studied the most common taint flows while keeping the following questions in mind: Is this taint flow feasible, i.e., could it be a leak? Do some code patterns with the same source-sink-pair exist in the taint flows? To assess the feasibility, we used the data-flow path (in Jimple) between source and sink of each taint flow reported by FLOWDROID and the decompiled code of the apps.

Intra-procedural taint flows, Groups A-C: As shown in Table 4.2, the source-sink-pair (`URL.openConnection`, `URLConnection.setRequestProperty`) appeared most frequently in both intra- and inter-procedural taint flows (group A and E). While the given source method creates a connection object with a given URL, the sink sets the general properties of a HTTP request. This source-sink-pair combination apparently does not constitute a leak, since the connection is not even opened when only calling `URL.openConnection`. One instead still has to call `URLConnection.connect` or equivalent methods (e.g., `URLConnection.getInputStream`) to initiate the communication [FYD⁺06]. During the inspection for the above-mentioned source-sink-pair in group A, we discovered that the reported taint flows share some common patterns.

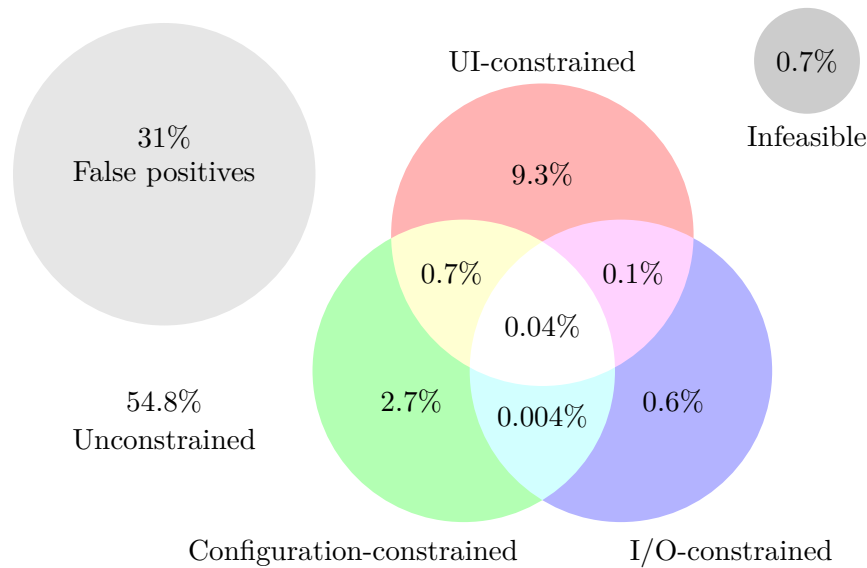


Figure 4.7: Different types of the taint flows.

```

/** code pattern 1 */
URLConnection c = (URLConnection) new URL("http...").openConnection(); //source
c.setDoInput(true);
c.setRequestProperty("User-Agent", "Mozilla/5.0"); //sink, no leak

/** code pattern 2 */
Message m = handler.obtainMessage(); //source
handler.sendMessage(m); //sink, no leak

/** code pattern 3 */
URLConnection c = (URLConnection) new URL("http...").openConnection();
c.setDoOutput(true);
OutputStream s = c.getOutputStream(); //source
s.write(data); //sink, no leak

```

Listing 4.4: False-positive code patterns from group A, B, C.

Code pattern 1 in Listing 4.4 shows an example usage of this source-sink-pair, which is a common way to set up the header of a HTTP request. This is no leak.

Code pattern 2 in Listing 4.4 is another common false-positive pattern we identified in group B. The factory method `Handler.obtainMessage` is regarded as a source by FLOWDROID. This method *creates* a new empty message instance. It does *not* poll a message from the message queue of the Android handler. This method should thus be excluded from the list of sources. Code pattern 3 from group C is a similar case.

In summary, taint flows which fall into these code patterns are false positives. To determine how many taint flows match these code patterns, we extended FLOWDROID to detect these patterns, and re-analyzed the apps in groups A, B and C. In the end, 2,630 (46%) reported intra-procedural taint flows matched these three code patterns. As shown in these code patterns, the root cause of these false positives is that their sources, which FLOWDROID uses in its default configuration, are actually inappropriate, i.e., they do not return sensitive data.

Such a big fraction of false positives caused by this reason cannot be ignored. Thus, we examined *all* 47 sources by reading the Javadoc carefully together with a software developer with more than 5 years experience in Java. Altogether, we identified 11 APIs that were mistakenly

made source/sink (see Table 4.3). These inappropriate sources and sinks resulted in 7,767 reported taint flows, which is 28% of all reported taint flows (intra- and inter-procedural).

About a quarter (11/47) of default sources provided by FLOWDROID are inappropriate and cause more than a quarter (28%) of all reported taint flows being false positives.

Table 4.3: Inappropriate sources and sinks used by FLOWDROID.

Signature
<code>android.os.Handler.obtainMessage()</code>
<code>android.os.Handler.obtainMessage(int,int,int)</code>
<code>android.os.Handler.obtainMessage(int,int,int,Object)</code>
<code>android.os.Handler.obtainMessage(int)</code>
<code>android.os.Handler.obtainMessage(int,Object)</code>
<code>android.app.PendingIntent.getActivity(Context,int,Intent,int)</code>
<code>android.app.PendingIntent.getActivity(Context,int,Intent,int,Bundle)</code>
<code>android.app.PendingIntent.getBroadcast(Context,int,Intent,int)</code>
<code>android.app.PendingIntent.getService(Context,int,Intent,int)</code>
<code>java.net.URLConnection.getOutputStream()</code>
<code>java.net.URL.openConnection()</code> <i>*[regarded as both source and sink]</i>

After a discussion with FLOWDROID’s maintainers, they confirmed the mistake and removed the inappropriate sources and sinks from the default list in FLOWDROID’s GitHub repository.³ This affects the pairs A, B, C and E in Table 4.2.

Inter-procedural taint flows, Groups D-F: Because the source of group E is inappropriate, the manual inspection of this group was unnecessary. We next describe the results of the manual inspection of the remaining groups D and F. The taint flows from group D use the source-sink-pair (`Cursor.getString`, `Activity.startActivityForResult`). Taint flows with this source-sink-pair could be part of a leak when the intent passed to `Activity.startActivityForResult` contains data reading from `Cursor.getString` and the second activity passes this received data to an untrusted sink. Since taint flows using such inter-component communication are outside the scope of FLOWDROID, our goal for inspection was only to check if this partial data-flow is feasible. Surprisingly, 88% of the taint flows from Group D proved to be false positives. All these false positives share a similar code pattern, shown in Listing 4.5. In this example, FLOWDROID taints `this.secret` and reports a leak when the sink method is called on the base object of the taint `this.secret`, which is the `this` object. However, there is no tainted data that flows into the intent passed for the sink method. Such over-approximation in FLOWDROID’s analysis logic, while sometimes useful, is too approximative for the sink `Activity.startActivityForResult`.

Generally, taint flows with taints connecting sources and sinks on the same objects should be filtered. Thus, we extended FLOWDROID with a static analysis that detects such cases and re-analyzed the relevant apps. 330 taint flows matched the false-positive pattern in Listing 4.5. In total, we identified 978 taint flows with taints connecting sources and sinks on the same objects. The sinks appearing in these taint flows are mainly APIs used for inter-component communication. The remaining sinks (e.g. `HttpResponse.execute(HttpUriRequest)`) only make sense when the right parameter was tainted. However, FLOWDROID reported these taint flows when the base object was tainted.

³The link to the commit: <https://github.com/secure-software-engineering/FlowDroid/commit/211b73e32a0ade1ded021f2fc30b0aa647be5862>

```

public class MainActivity extends Activity {
    private String secret;
    public void caller(){
        this.secret = cursor.getString(i); //source
        callee();
    }
    public void callee() {
        Intent i = new Intent();
        this.startActivityForResult(i, ...); //sink, no leak
    }
}

```

Listing 4.5: False-positive code pattern from group D.

In our study, all taint flows reported by FLOWDROID with taints connecting sources and sinks on the same objects are false positives.

The sink `Bundle.putString` used in taint flows from group F is also an API for inter-component communication. Similar to group D, we checked if the reported partial flow is feasible. We found out that while 89% of the reported flows *are* feasible, the false positives all happened in one app and the flows were just for putting the name of the app into the sink. However, the fact that the partial flows are feasible does not mean they are a part of true leaks, since one does not know how the sensitive data stored in `Bundle` were used in other activities, which was not reported by FLOWDROID. In total, we identified that at least one third (31%) of taint flows reported by FLOWDROID in the default configuration are false positives.

In step 2, we classified taint flows that are dependent on the constraint-APIs with COVA. For instance, if the leak-constraint contains symbolic values that relate to the constraint-APIs from UI callbacks, then this taint flow belongs to category “UI-constrained”. Certainly, there can be taint flows which belong to multiple categories. Note that we excluded the false positives we identified in step 1 for the classification.

As the Venn diagram in Figure 4.7 shows, among the 14.2% taint flows whose occurrences are dependent on the constraint-APIs in the categories, the majority are in a single category – UI-constrained, which means they only occur when some specific UI actions are performed. 2.7% of the taint flows may happen under certain environment configurations, and 0.6% are dependent on inputs from I/O operations. The numbers in the Venn diagram’s intersections of different categories indicate that interactions between values read from APIs in different categories are rare but do exist.

Taint flows are seldom conditioned by combinations of UI interactions, environment configurations and I/O operations. Most taints could thus be dynamically confirmed by *different* tools that specialize on the respective category.

Because complex UI dependencies may require a test harness to drive the application with the needed sequences of events, we investigated the complexity of the UI actions. Intuitively, taint flows triggered by a sequence of user actions should exist. A previous study [ZZD⁺12] has found malicious applications in which a user needs to click a series of buttons to trigger the display of a widget which leaks the data. To estimate the complexity, we calculated how many different UI actions are involved in a UI-constrained taint flow by counting the number of symbolic values for UI actions used in the leak-constraint. Note that our constraint encoding is able to distinguish different UI actions.

4.6 COVA-ASSISTED QUALITATIVE ANALYSIS OF ANDROID TAINT-ANALYSIS RESULTS

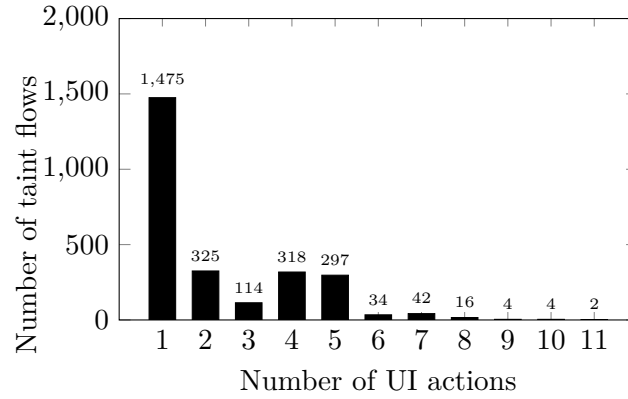


Figure 4.8: The distribution of UI-constrained taint flows.

Figure 4.8 shows the distribution of UI-constrained taint flows. 56.1% (1,475) of taint flows happen after a single UI action. There are only 3.8% of taint flows that may require 6 or more UI actions. Maximally 11 different actions appeared in a leak-constraint. However, executing the taint flow does not require all 11 actions at the same time, since there are disjunctions in the leak-constraint (e.g., $A \vee B$ contains two actions A and B , but one action is sufficient to execute the taint flow).

Despite the existence of sophisticated sequences of UI actions, our results indicate the dynamic exploration of most UI action-related taint flows could be easier than expected.

Among the configuration-constrained taint flows, the distribution is even simpler: the largest number of taint flows (85.6%) require a single configuration option and 13.9% of them are dependent on two options. Only 5 taint flows happen under a complex configuration with more than two options.

The necessary configuration-based conditions for exposing taint flows are easy to be satisfied in the majority of cases.

Table 4.4: Top Constraint-APIs related to the taint flows.

UI Callback	#Flows
android.view.View.OnClickListener.onClick	2,088
android.widget.AdapterView.OnItemClickListener.onClick	623
android.content.DialogInterface.OnClickListener.onClick	595
Configuration	#Flows
android.os.Build.VERSION.SDK_INT	255
android.content.Context.getSystemService("connectivity")	246
android.content.Context.getSystemService("location")	224
I/O Operation	#Flows
java.io.InputStream.read	158
java.io.BufferedReader.readLine	16
java.io.ObjectInputStream.readObject	10

Table 4.4 shows which constraint-APIs from our categories are most frequently used. While

click events are relevant to most taint flows related to UI actions, the Android SDK version plays a considerable role in environment configurations. This is not surprising to us, since the Android operating system remains highly fragmented [Ope15, MSDM16, Goo18b] and developers are challenged to produce applications that are compatible to multiple platform versions. However, the importance of taint flows which only occur in obsolete versions of Android may be limited in practice. Constraints based on I/O operations are mostly checking if the end of a data stream has been reached, e.g., `if(inputStream.read() != -1)`.

Additionally, we observed that for 28% of the 208 infeasible taint flows, their source statements were not executable, since the path constraint is *FALSE*. For 76% of the infeasible taint flows, their sink statements will never be executed at runtime. Such dead code was probably intentionally built in by developers [EHMG15], e.g., during sampling, we inspected code used for logging (sinks of taint flows) that was disabled with a boolean flag for the released APK, but not removed.

RQ2. How large is the fraction of easily actionable unconditional intra-procedural taint flows, and what characteristics do these flows have?

Methodology

Taint flows that are intra-procedural and unconstrained, i.e., the leak-constraint computed by COVA is equal to *TRUE*, are easy to detect and clearly directly actionable for developers. We call such flows “*low-hanging fruits*”. We sought to acquire the characteristics of the “low-hanging fruits”, and thus again applied stratified random sampling with proportion 10% to the taint flows with top source-sink-pairs in Table 4.5.

Table 4.5: Top source-sink-pairs among “low-hanging fruits”.

Gr.	Source	Sink	Sampled/Total Flows
X	android.database.Cursor.getString	android.content.ContentResolver.query	14/137
Y	android.database.Cursor.getString	android.util.Log.e	10/96
Z	android.database.Cursor.getString	android.util.Log.i	7/70

Results

In our study, only 3.5% of the taint flows are “low-hanging fruits” (intra-procedural and unconstrained). However, “low-hanging fruits” are still the majority of the intra-procedural taint flows and they exist in 32% (329) of the apps in our dataset. During the manual inspection for taint flows with top source-sink-pairs in Table 4.5, we found that taint flows with source-sink-pair of group X cannot usually be interpreted as leaks. Listing 4.6 shows a simplified taint flow using this source-sink-pair. The private field `this.secret` is first tainted. FLOWDROID taints the return value of `this.getContentResolver()` since the base object is the prefix of the tainted `this.secret`. Finally, the taint flow is then reported when the sink is called on the tainted `this.getContentResolver()`. This is again an over-approximation FLOWDROID uses similar to the one in Listing 4.5. However, such taint flow *could* be a leak, since the implementation of `Context.getContentResolver` and `ContentResolver.query` could be overridden maliciously.

In comparison to group X, taint flows of group Y and Z are straightforward: they log data from databases. Actually, the log methods from `android.util.Log` are the most frequently used sinks. About half (46%) of the “low-hanging fruits” are leaks in which sensitive information such as data from databases, location information, device ID, the MAC addresses or even passwords are logged. Many of these leaks even have source and sink at the same line of code. In addition, the text that will be logged often specifies what kind of data is being logged.

```

public class MainActivity extends Activity {
    private String secret;
    public void foo(){
        this.secret = cursor.getString(i); //source
        this.getContentResolver().query(...); //sink, no leak
    }
}

```

Listing 4.6: False-positive code pattern from group X.

Besides log methods, sinks for inter-component communication such as `Bundle.putString`, `SharedPreferences.putString` and `Context.sendBroadcast` are also popular among the “low-hanging fruits”. They appeared in 20% of the taint flows. To determine if these taint flows are malicious, additional context must be provided, since benign applications often use these methods for accessing and modifying preference data between activities.

Discussion

First, our results of the qualitative analysis show important ways in which taint-analysis tools can and should be improved. On the one hand, the sources and sinks configured for the tools should be checked more carefully, since an inappropriate source can cause a large amount of false positives, as we determined for FLOWDROID in RQ1. Researchers who used FLOWDROID in the default configuration may need to re-evaluate their conclusions. Even in just a short investigation, we already found 9 papers in which the respective work was built on top of FLOWDROID and inappropriate sources or sinks were used [LBK⁺14, MSTdF17, ZJY⁺17, CKBG18, AKG⁺15, SBF⁺16, TTYR17, RAB14, WWLZ16]. In none of these papers did the authors mention that they have manually checked for false positives that would have been caused by the inappropriate source/sink configurations. Hence, while it is possible that such manual checks were conducted without mentioning them, it is equally possible that the papers report results that are distorted by the presence of those false positives. Given the over 1,000 citations of the FLOWDROID paper, many more such works are likely to exist. On the other hand, some rules used in taint analysis may be not suitable for all sources and sinks, as we have seen in the case shown in Listing 4.5 for FLOWDROID, a taint flow was reported when the base object calling the sink was tainted. However, here the correct way to report a taint flow is when the actual argument (intent) of the sink is tainted. Such cases could be handled easily without increasing analysis complexity. To avoid these false positives we discovered in our study, one could introduce a fine-grained specification (e.g., return value, argument or receiver) of the sources and sinks rather than just a list of APIs that are handled equally as the default one in FLOWDROID. As mentioned in [Arz16], FLOWDROID also allows users to specify the sources and sinks more expressively in a XML file. However, it does not provide such a XML file that is ready to use. A recent commercial analysis engine CodeQL from GitHub also supports fine-grained source/sink/sanitizer specification with its underlying query language QL [Git19a]. Although we only studied the results reported by FLOWDROID, problems we discovered for such a widely used tool may not be a single case among numerous taint-analysis tools produced in academia.

Second, hybrid analysis tools may well be feasible for the case of Android. The results of RQ1 show that to confirm static taint flows dynamically, they should focus on modeling UI actions, but in some cases must be able to set correct environment options as well, and must deal with stream-I/O to some limited extent. Luckily, the overlap between those three classes is small, so that one can probably go a long way even by designing specific, decoupled analysis tools for all three situations. Our tool COVA can further aid the implementation of such hybrid analysis tools: the path constraints it computes can guide dynamic analyses, even in situations

where flows are not conditioned on external stimuli at all.

Third, “low-hanging fruits”, i.e., unconditional intra-procedural flows, are quite common—they exist in 32% of all apps as we show in RQ2. Many of such leaks can be easily fixed, and so even purely static taint-analysis tools can and should prioritize these leaks in the report.

Lastly, we see two actionable aspects of our work: first, our SMT-based path constraints provide a start toward finding concrete executions to demonstrate vulnerabilities. The constraints COVA computes can give an initial suggestion how the program inputs for the execution which exposes the vulnerabilities should be satisfied and can be used to generate inputs. Second, COVA can be used to enhance the comprehension of static-analysis results, since the constraints it computes explain in which circumstance the reported vulnerabilities may occur and how likely they are to occur. Such information is certainly helpful for developers in triaging static-analysis results. In the following section, we introduce a proof-of-concept we implemented addressing test input generation.

4.7 Usage of COVA for Targeted Testing Input Generation

In this section, we introduce the extended COVA for targeted testing of Android apps implemented in the scope a master thesis [Hau21] supervised by me. In the following, COVA refers to the extended COVA. The general idea of this work is to generate concrete inputs for executing any statement that is covered in the constraint map generated by COVA.

4.7.1 Android Testing Frameworks

To run Android apps with concrete inputs at scale, one needs a testing framework that automates the process. There exist a few popular Android testing frameworks to automate scripted

```

1 public void test{
2     Path apkFile = Paths.get("Example.apk");
3     String url = "http://127.0.0.1:4723/wd/hub";
4     // set up Appium Adnroid driver
5     DesiredCapabilities capabilities = new DesiredCapabilities();
6     capabilities.setCapability(MobileCapabilityType.DEVICE_NAME, "Android Emulator");
7     capabilities.setCapability(MobileCapabilityType.APP, apkFile.toAbsolutePath().toString());
8     AndroidDriver<MobileElement> driver = new AndroidDriver<>(new URL(url), capabilities);
9     driver.installApp(apkFile.toAbsolutePath().toString());
10    driver.launchApp();
11    // start recording
12    driver.startRecordingScreen();
13    Thread.sleep(1000);
14    // enter username and password
15    driver.findElement(By.id("username")).sendKeys("Tom");
16    driver.findElement(By.id("password")).sendKeys("abc123");
17    Thread.sleep(1000);
18    // click login button
19    driver.findElement(By.id("login")).click();
20    // save recording
21    String recording= driver.stopRecordingScreen();
22    Files.write(Paths.get("recording.mp4"), Base64.getDecoder().decode(recording));
23    // exit
24    driver.close();
25 }

```

Listing 4.7: Appium sample Java test code.

tests, e. g., Espresso [Goo13b], UI Automator [Goo13c], Robotium [Tec14], and Appium [Fou12]. Espresso is the official white-box testing framework for Android, which requires access to source code for writing test cases. It is usually used for unit testing. UI Automator is a black-box testing framework that provides APIs to test the UI of Android apps. However, it only supports Android devices with API level 18 or higher and does not support **WebView**, which means web-based code can not be tested [Sha21]. Robotium is an open-source gray-box Android testing framework that was mostly widely used in the early days of Android. However, it only supports Java and requires one to recompile or modify the app under test, which is not possible without the source code [T21].

We need a framework that does not require the original source code and works for APKs. Appium used in our work is such a framework. It is cross-platform (Android, iOS and Windows) and supports with native, hybrid, and mobile web apps. It supports testing on Android devices and emulators with all API levels using both UI Automator (API level 18 or higher) and Selendroid (API level lower than 18). One can write test scripts in several programming languages, e. g., Java, JavaScript, Objective-C, Python, PHP, Ruby, C# etc. Listing 4.7 shows a sample Java test code using APIs provided by Appium. This test code installs and launches the app `Example.apk` on an Android emulator, enters inputs for both `username` and `password` text fields and clicks on the `login` button. Our goal of this work is to generate such test code automatically such that selected target code fragments (e. g., potentially insecure or malicious code) will be executed when running the test. As the code shows, a string-based layout element ID (e. g., "username", "password", "login") is required to specify a UI action with Appium.

4.7.2 Extended COVA

Now we introduce the workflow of COVA for targeted testing, which is depicted in Figure 4.9 with 5 steps. The inputs for COVA are an Android APK and a target. A *target* is any statement or basic code block that is desired to be executed at runtime. Currently, COVA only supports

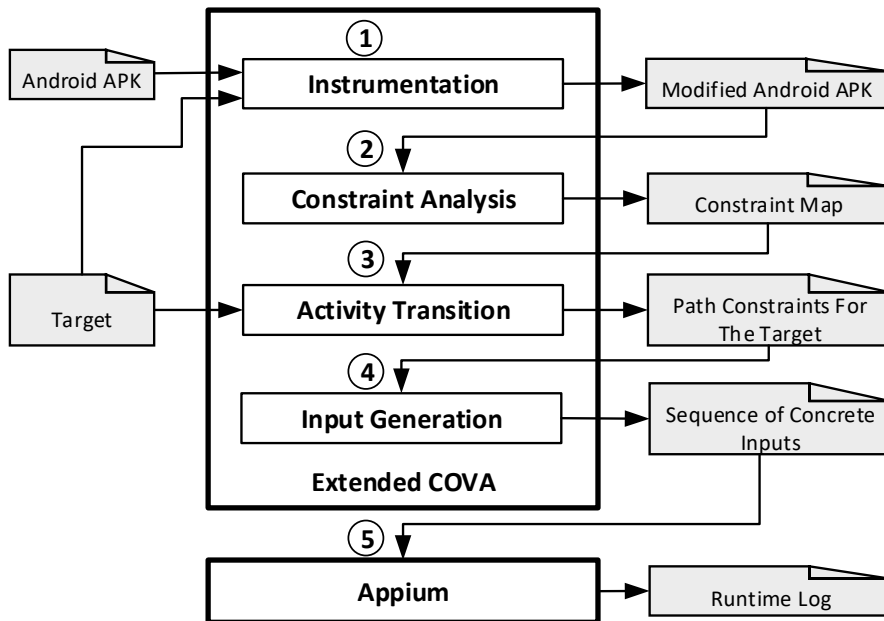


Figure 4.9: The workflow of COVA (extended) for targeted testing.

the definition of an invoke statement or an assign statement as a target.

In step ①, COVA instruments the APK with logging statements at specific code positions, i. e., before each invoke, assign and if statement. A modified APK is generated in this step.

In step ②, the constraint analysis in COVA is performed on the modified APK. Especially, text-based user inputs that are returned from the API `View.findViewById(int)` are considered by COVA. In Android apps, the UI components (e. g., button, text field, etc.) of each activity is declared in a layout XML file (see Listing 4.10a) and can be accessed with `View.findViewById(int)` in code. (see Listing 4.10b). COVA starts taint tracking from the calls to `View.findViewById(int)` and generates a so-called *string taint* once the returned `View` element is stringified (line 6 in Listing 4.10b). Operations on the string taints are modeled in COVA. Note that the compiled Dalvik bytecode does not contain the string-based key in the XML file as Listing 4.10c shows: the key `R.id.weightNum` is replaced by a constant integer ID `2131230857` in the Jimple code generated from the bytecode. COVA also resolves the mapping from integer IDs to string-based keys in the layout XML. This mapping information is needed later for specifying UI actions with the Appium [Fou12] testing framework as we introduced in Section 4.7.1 (see also Listing 4.7).



Figure 4.10: A simplified example from the app *BMI* in F-Droid.

Once COVA computes the constraint map, the next step is to collect a list of constraints required from the main activity to the Android component (i. e., activity, service, broadcast receiver and content providers) in which the target resides. Currently, only activity is considered. In this step ③, starting from the activity that contains the target and an empty list, COVA traverses backwards to the main activity and collects constraints at activity transition statements. Activity transition statements are the statements that call the method `Context.startActivity()`. As the example in Figure 4.11 shows, the target is in `ThirdActivity` and the constraint at the target statement is $C1 = \text{ThirdActivity} : \text{compute} : \text{onClick} \wedge \text{str.length}(\text{ThirdActivity} : \text{editText}) < 10$. This constraint means a UI element (i. e., button) in `ThirdActivity` with the layout element `Id compute` needs to be clicked (`onClick`) and (\wedge) the input text for a UI element (i. e., text field) in `ThirdActivity` with the layout element `id editText` must have the length smaller than 10. In comparison to the original COVA, the extended COVA preserves more precise information (i. e., `layoutid : layoutelementid : onClick(optional)`) about the original UI element. $C1$ is added to the list of constraints at first. Then COVA searches



Figure 4.11: Activity transition: collecting constraints backwards from the target activity `ThirdActivity` to the main activity `MainActivity`.

the activity transition statement that starts `ThirdActivity` and adds the constraint of that statement, i. e., $C2$ which requires a click on a button in `SecondActivity` with the id `process`, to the list. Lastly, $C3$ is also added to the list, since its corresponding statement is the activity transition statement that starts `SecondActivity`. So the list of constraints COVA computes is $\{C1, C2, C3\}$. This list will be reversed and taken as input for step ④.

In step ④, UI actions and text-based inputs are generated according to each constraint in the list. Text-based inputs are generated with Z3 by solving the corresponding constraint over values read from the API `View.findViewById(int)`. Test code with a sequence of UI actions and inputs is generated and sent to the Appium server when the text code is executed in step ⑤. Runtime logs and a recording of the screen are saved after each execution of the app. The logs can be used to check whether the target was executed.

Five apps from F-Droid were chosen to evaluate the effectiveness of the extended COVA. 20 targets were randomly selected for each app and the list of constraints COVA computed in step ③ were checked manually. Table 4.6 shows the evaluation result. Except the app `WorkoutLog`, COVA achieves high precision for the selected targets. The reason why COVA does not work well (low precision and recall) for the `WorkoutLog` app is because the app uses an autocomplete function for text-based input fields. This function also requires prior user data, which can not be modeled statically. Other false positives are due to similar reasons—certain methods or arithmetic operations on text-based inputs are not modeled in COVA, which also lowers both the precision and the recall.

Although this is just a proof of concept, it nonetheless shows promising accuracy in computing constraints over user inputs and the feasibility of using COVA for targeted testing. One could use it for dynamic validation of static findings. Static analysis discovers code that yields the potential for vulnerabilities or for malicious behavior. The vulnerable or malicious code can

Table 4.6: Evaluation results on selected F-Droid apps (table reconstructed from [Hau21]).

	TP	FP	TN	FN	Precision	Recall	F-Measure
BMI	19	1	0	0	95%	100%	97.4%
WorkoutLog	4	5	0	11	44.4%	26.7%	33.3%
NightMode	12	0	3	5	100%	70.1%	82.8%
ItalianSaid	16	1	0	3	94%	84.2%	88.9%

be chosen as targets for COVA. The concrete values recorded during execution of the app can be further used to filter false positives or further analysis.

4.8 Threats to Validity

In this section, we discuss the threats to validity regarding the qualitative study presented in Section 4.6. COVA computes partial path constraints—it only considers control-flow decisions that are dependent on a list of constraint-APIs we collected. However, as we discuss in Section 4.6, we feel that by the way this list was collected, it is comprehensive. We conjoin the constraints of the source and the sink statement, yet, it is possible there are additional constraints caused by intermediate nodes along the data-flow path. However, we did not consider this to keep the constraint solving tractable. Although COVA supports most language features, some corner cases such as reflection or native calls are not covered [LSS⁺15]. In some cases, the taint set computed by COVA may be incomplete due to unknown return values of API method calls such that an over-approximated constraint is computed. For Android applications, we use the call graph constructed by FlowDroid. This call graph, however, is partially incomplete for library methods and some UI callbacks [Arz16, WZR16]—a limitation of FLOWDROID as we also show in Chapter 1.

Since COVA uses Z3 for constraint-solving, the limitations of Z3 are inherited by COVA. In our experiments, an average of 49% percent of the analysis time was occupied by Z3. In fact, this is also one of the main reasons why COVA failed to analyze some apps within the given time budget. In the worst case, 98% of the analysis time for an app was spent for constraint-solving. Increasing the time budget may not help, since Z3 can hit memory pressure and throw exceptions when solving large formulas, which happened in our preliminary experiments. Such exceptions cannot be evaded by increasing the JVM heap size, since they originate from the native code of Z3.

Since COVA failed to analyze some apps in our experiment, our study may have been biased to include only certain kinds of taint flows. Many of the “unconstrained” taint flows might be constrained by multiple factors in reality.

4.9 Related Work

We discuss how our approach relates to previous work in the areas of taint analysis, path conditions, as well as hybrid analysis approaches.

Studies Involving Taint Flows Many researchers have studied Android applications from various perspectives [SGF⁺13, YXA⁺15, AKG⁺15, KWJB13, LKB17, SBF⁺16, CLHS17]. Avdiienko et al. [AKG⁺15] compared the taint flows in benign apps against those in malicious apps, and used machine learning to identify the differences in usage of sensitive data. Unlike COVA, their approach MUDFLOW does not consider path constraints. Keng et al. [KWJB13] monitored 220 Android apps with the dynamic taint-analysis tool TaintDroid [EGH⁺14] to study the

correlation between user actions and leaks. However, their results are limited to the leaks they observed during the runtime. Their results show that many apps leak data due to user actions on certain GUI widgets, which we were able to show statically. Closely related to our approach, Lillack et al. [LKB17] also extended taint analysis to explore the variability of Android apps based on load-time configuration. However, their approach encodes constraint analysis as a distributive problem in the IFDS framework [RHS95]. For our purpose, this model is insufficient, since the execution of a branch may depend simultaneously on two or more configuration options, which IFDS cannot express [Kil73, SRH95].

Path Conditions Many approaches have considered path conditions to increase the accuracy of their analysis. Snelting [Sne96] has shown how exacting and simplifying path conditions can improve slice accuracy. Taghdiri et al. [TSS10] made information flow analysis more precise by incrementally refining path conditions with witnesses that did not yield an information flow in execution. TASMAN [ARHB15] leverages backward symbolic execution as a post-analysis to eliminate false positives in which taint flows along paths are infeasible at runtime. TASMAN is based on the distributive IFDS framework, but constraint computation is not a distributive problem. TASMAN thus needs to approximate in places which COVA can handle precisely. In result, COVA’s computation is more expensive but COVA’s path expressions are also more precise. A general major limitation of symbolic execution is that it cannot explore executions with path conditions which the underlying SMT solver cannot deal with in the given time budget [CS13]. This limitation is shared with COVA. To improve scalability, modern symbolic execution techniques mix concrete and symbolic execution in so-called concolic execution. Anand et al. [ANHY12] propose a concolic execution approach to generate sequences of UI events for Android applications. Schütte et al. [SFT15] also use concolic execution to drive execution to cover target code. They claim that their approach is not limited to any specific kind of conditions, i.e., can handle all kinds of condition (user input, environmental setting and even remote site input). Yet their prototype ConDroid was only designed and evaluated for one specific vulnerability. COVA was evaluated on a wide range of taint flows. The results of our study indicate that tools which seek to expose taint flow dynamically could concentrate on one kind of condition at a time, which is important for scalability.

Hybrid Analysis A number of hybrid approaches, i.e., combinations of static and dynamic analysis, have been proposed for Android malware detection [ZZD⁺12, YQL⁺16, Ken16, RATP17, WL16, XGL⁺15]. SmartDroid by Zheng et al. [ZZD⁺12] statically detects UI interaction sequences that lead to sensitive API calls, and it exposes those behaviors dynamically. Yang et al. [YQL⁺16] propose a hybrid approach in which they first identify the possible attack-critical path with static mining algorithms based on sensitive APIs and existing malware patterns, then execute the program in a focused scope under dynamic taint analysis. Wong et al. [WL16] demonstrate IntelliDroid, a tool which generates a reasonably small set of inputs statically to trigger malicious behavior of applications. Their evaluation shows that one only needs to execute a very small part of the application to expose malicious behaviors. Recent work of Rasthofer et al. [RATP17] combines a set of static and dynamic analyses with fuzzing to generate execution environments to expose hidden malicious behaviors efficiently.

4.10 Conclusion

In this chapter, we addressed the path-sensitivity that is often omitted by most static taint analysis approaches. To enhance a client analysis with path sensitivity, we developed a tool called COVA for tracking user-defined APIs through the program and computing path constraints

based on these APIs. The path constraints COVA computes can be used refined the results of any client analysis. We conducted a COVA-supported study which gathers information about the nature of static taint-analysis results, particularly with FLOWDROID. Our study shows important ways how static taint-analysis tools can be improved and how information about taint-flows conditioned on different factors can be used for future taint-analysis research, particularly with the aim of further eliminating false positives. We extended COVA to not only compute the path constraint of a target statement, but also generate concrete user inputs that are required to execute this statement at runtime. Experiments with a small set of apps from F-Droid show the feasibility of using this approach to generate valid user inputs for testing randomly selected statements, which is a step towards dynamic validation of static findings.

Integrating Static Analyses into IDEs with MagpieBridge

In the previous chapters, we introduced our GENCG and COVA approaches, which provide support for more effective taint analysis in terms of improving recall and precision. However, to make the analysis useful also requires effective ways of presenting the findings to users. Evaluations of static taint analyses have been mostly restricted to automated experiments where the analyses are run in “headless” mode as command-line tools, e. g., FlowDroid [ARF⁺14], Aman-droid [WROR14], IccTA [LBB⁺15], HybriDroid [CLHS17], TAJ [TPF⁺09], Andromeda [TPC⁺13]. The analysis findings are either printed as terminal output or stored in files which often do not follow any standard, paying little to no attention to usability aspects on the side of the user. We observed that static taint analysis is not the only case that needs better presentation of findings. There are also many analyses that address code quality issues, such as FindBugs [HP07], SpotBugs [Tea17], PMD [Tea11] for common programming flaws (e. g., unused variables, dead code, empty catch blocks, unnecessary creation of objects, etc.) and TRACKER [TC10] for resource leaks. Other analyses target code performance, such as J2EE transaction tuning [FDC04]. There are also specialized analyses for specific domains, such as Ariadne [DSAR18] for machine learning. These analyses collectively represent a large amount of work, as they embody a variety of advanced analyses for a range of popular programming languages. To make this effort more tractable, many analyses are built on existing program analysis frameworks that provide state-of-the-art implementations of commonly-needed building blocks such as call graph construction, pointer analysis, data-flow analysis and slicing, which in turn all rest on an underlying abstract internal representation (IR) of the program. Doop [BS09, Doo09], Soot [LBLH11a, Soo00], Safe [Saf14], Soufflé [Sou15] and WALA [WAL06] are well-known.

While development of these taint analyses and other analyses has been a broad success of programming language research, there has been less adoption of such analyses in tools commonly used by developers, i.e., in interactive development environments (IDEs) such as Eclipse [IF01], IntelliJ [Jet01], PyCharm [Jet10], Android Studio [Goo13a], Spyder [Ray09] and editors such as Visual Studio Code [Mic15a], Emacs [DAMvfsd76], Atom [Git14], Sublime Text [HQ08], Monaco [Mic16b] and Vim [Moo91]. There have been some positive examples: the J2EE transaction analysis shipped in IBM WebSphere [IBM98], Andromeda was included in IBM Security AppScan [IBM07], both ultimately based on Eclipse technology. Similarly, CogniCrypt comprises an Eclipse plugin that exposes the results of its crypto-misuse analysis directly to the developer within the IDE. Each of these tools involved a substantial engineering effort to integrate a specific analysis for a specific language into a specific tool. Table 5.1 shows the amount of code in plugins for analyses is a significant fraction of code in the analysis itself. Given that degree of needed effort, the sheer variety of popular tools and potentially-useful analyses makes

Tool	Analysis (LOC)	Plugin (LOC)	Plugin/Analysis
FindBugs	132,343	16,670	0.13
SpotBugs	121,841	16,266	0.13
PMD	117,551	33,435	0.28
CogniCrypt	11,753	18,766	1.60
DroidSafe	41,313	8,839	0.21
Cheetah	4,747	864	0.18
SPLlift	1,317	3,317	2.52

Table 5.1: Comparison between the LOC (lines of Java code) for analysis and the LOC for plugin.

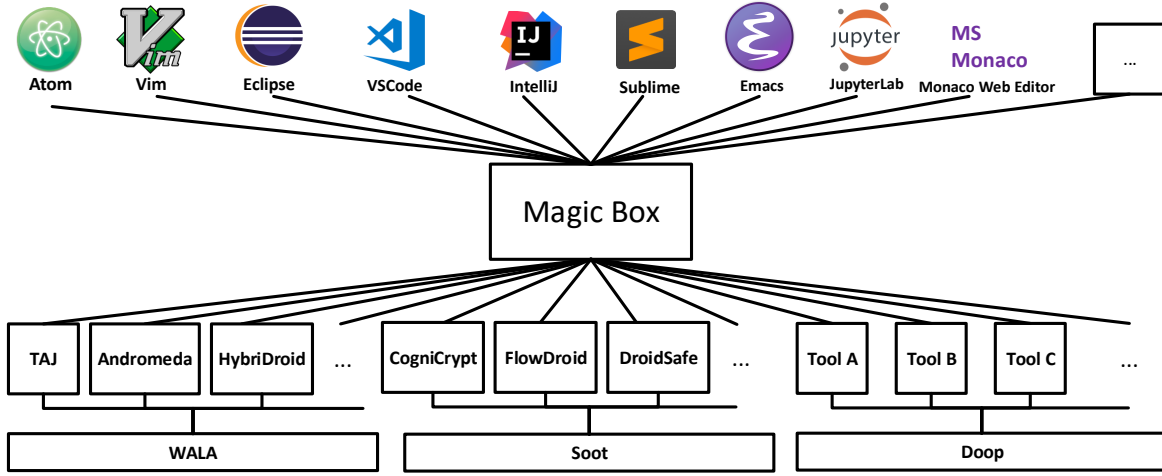


Figure 5.1: The desired solution: a magic box that connects arbitrary static analyses to arbitrary IDEs and editors.

it impractical to build every combination.

While the difficulty of integrating such tools into different development environments has led to poor adoption of these tools in practice, it also makes empirical evaluations of them with real users challenging. As many recent studies show [JSMB13, CB16, DAL⁺17a], if static analysis tools do not yield actionable results, or if they do not report them in a way that developers can understand, then the tools will not be adopted. So to develop and evaluate such tools, researchers need ways to bring tools into IDEs more easily and quickly.

The ideal solution is the magic box shown in Figure 5.1, which adapts any analysis to any editor,¹ and presents the results computed by the analysis, e.g., security vulnerabilities, using common idioms of the specific tool, e.g., problem lists or hovers, allowing user interactions, e.g., button clicks or configuration.

In this chapter, we present MAGPIEBRIDGE,²—our magic box which uses three mechanisms to realize a large fraction of this ultimate goal:

1. Since many analyses are written using program analysis frameworks, MAGPIEBRIDGE can focus on supporting the core data structures of these frameworks. For instance, analyses

¹Note: In the following, when we write *editor*, we mean any code editor, which comprises IDEs.

²In a Chinese legend, a human and a fairy fall in love, but this love angers the gods, who separate them on opposite sides of the Milky Way. However, on the seventh day of the seventh lunar month each year, thousands of magpies form a bridge, called 鹊桥 in Chinese and Queqiao in pinyin, allowing the lovers to meet.

based on data-flow frameworks can be supported if the magic box can render their data-flow results naturally. Furthermore, while there are multiple frameworks, they share many common abstractions such as data flow and call graphs, which allows one to support multiple frameworks with relative ease.

2. More and more editors support the Language Server Protocol (LSP) [Mic16a], a protocol by which editors can obtain information from arbitrary “servers”. LSP is designed in terms of idioms common to IDEs, such as problem lists, hovers and the like. Thus, MAGPIEBRIDGE can take information from a range of analyses and render it in a few common tooling idioms. LSP support in each editor then displays these in the natural idiom of the editor.
3. While LSP does not support customized UI elements such as icons or buttons being added to the editors, many editors can render web pages in editor tabs and provide APIs for this purpose. Thus, MAGPIEBRIDGE utilizes such support to allow building complex user interfaces beyond the native editor.

MAGPIEBRIDGE was published at the ECOOP conference in 2019 [LDB19]. At the time it was published, we only considered LSP in MAGPIEBRIDGE, which leads to limited customization of the integrated analyses (see Section C.1). However, as I learned more about the editors over the course of this dissertation, the third mechanism above is added to MAGPIEBRIDGE to support complex user interfaces inside an editor. Moreover, the APIs in MAGPIEBRIDGE have been updated based on practical use cases. The chapter is based on the latest version of the MAGPIEBRIDGE open-source project at <https://github.com/MagpieBridge/MagpieBridge>. The outline of this chapter is as follows: we first discuss related work in Section 5.1. We present the MAGPIEBRIDGE workflow, explaining the common APIs we defined for enabling IDE integration of analyses in Section 5.2. In Section 5.3, we demonstrate IDE integration of a few existing analyses using MAGPIEBRIDGE. We discuss limitations and conclude the chapter in Section 5.5.

5.1 Related Work

In this section, we first introduce the IDE integration of existing analysis tools and frameworks. Since our work is based on language server protocol, we give some background about it and compare it to other communication protocols between editors and analyses.

Existing tools and frameworks Given the importance of programming tools for IDEs, there have been a variety of efforts to provide them, both commercial and open source. We here survey some significant ones, focusing on those that use WALA [FD12] or Soot [LBLH11a, VRCG⁺10] and hence are most directly comparable to our work.

There have been a few commercial tools, notably IBM AppScan [IBM07] and RIGS IT Xanitizer [Gmb13]. Both products make use of WALA and target JavaScript among other languages. They comprise views to display analysis results as annotations to the source code, and allow for some triaging of the often longish lists of potential vulnerabilities within the IDEs. Among other issues, AppScan finds tainted flows and allows the user to focus on a specific flow through the program, although the user needs to decide what flow is of interest.

There has been a wider variety of open-source tools. WALA has been used in e. g., JOANA [HS09, GS15]. Soot is used in the widely adopted open-source crypto-misuse analyzer Eclipse CognitoCrypt [KNR⁺17], and is also part of the research tools Cheetah [DAL⁺17a], SPLift [BTR⁺13] and DroidSafe [GKP⁺15]. All tools named so far integrate with the Eclipse IDE. JOANA focuses on Java, including Android, and provides a range of advanced analyses based on information

flow control. CogniCrypt is a tool to detect misuses of cryptographic APIs in Java and Android applications. Its current UI integration is relatively basic, offering simple error annotations in the program code and the problems view. CogniCrypt further comprises an XText-based [EB10] Eclipse plugin that allows developers to edit API-specification files using syntax highlighting and code completion. Those specification files directly determine the definition of the static analysis. SPLift is a research tool to analyze Java-based software product lines. Its UI is an extension to FeatureIDE [TKB⁺14], which allows it to show variations in the product line’s code base through color coding. Detected programming errors are shown as code annotations and in the problems view. FeatureIDE itself is also an extension to Eclipse. Cheetah is a research prototype for the just-in-time static taint analysis within IDEs. In Cheetah, the analysis is triggered upon saving a source-code file, but in its case the analysis is automatically prioritized to provide rapid updates to the error messages in those code regions that are in the developer’s current scope. From there the analysis works its way outwards, potentially reporting errors in farther parts of the program only after several seconds or even minutes. Due to this mechanism, Cheetah requires the IDE to provide information about which file edit caused the analysis to be triggered, and what the project layout looks like. Cheetah also provides a somewhat richer UI integration than the previously named tools. For instance, when users select an individual taint-flow message in the problems view, it highlights in the code all statements involved in that particular taint, and also shows a list of those statements in a separate view—useful in case those are scattered across multiple source code files.

Analysis based on Doop [BS09, Doo09] has been experimentally integrated into the ProGuard optimizer for Android applications [Vra17]. This is a once-off integration rather than a framework for Doop analyses, and it is focused on the build process rather than the IDE itself. Still, it reflects the special-purpose integrations that show how analysis tends to be used.

Until now, program-analysis frameworks have focused on making it easier to develop analyses, with supportive infrastructure for basics such as scalable call graph, pointer analysis, and data-flow analysis. There have been presentations³ and tutorials⁴ at conferences which have provided both introductions and detailed tutorials for analysis construction; however, until now, there has been little focus on assisting with integrating such analyses into usable tools.

Language Server Protocol (LSP) The Language Server Protocol (LSP) [Mic16a] is a JSON-based RPC protocol originally developed by Microsoft for its Visual Studio Code to support different programming languages. LSP follows a client/server architecture, in which “clients” are typically meant to be code editors, i.e., IDEs such as IntelliJ, Eclipse, etc., or traditional editors such as Visual Studio Code, Vim, Emacs or Sublime Text. Those clients can trigger certain actions in “servers”, e.g., by opening a source-code file. Those servers can be of different flavours, but LSP allows them to contribute certain contents to the editor’s user interface, such as code annotations, list items or hovers. We will give concrete examples, including screenshots, in Section 5.3. As we show in this work, the LSP’s design lends itself to implement static code analysis tools as servers. In such a design, clients trigger analysis servers through LSP, and those servers communicate back their results through LSP as well, causing analysis results to automatically be shown in the client through the respective editor’s native interfaces.

SASP and SARIF The Static Analysis Server Protocol (SASP) [Gra18], although similar in name to LSP, is a distinctly different protocol. Started in 2017 by the static code analysis vendor GrammaTech, it describes a standardized communication protocol to facilitate communication between static analysis tools and consumers of their results. Compared to LSP, it supports

³e.g., <https://souffle-lang.github.io/pdf/SoufflePLDITutorial.pdf>

⁴e.g., <http://wala.sourceforge.net/wiki/index.php/Tutorial>

a richer data-exchange format that is explicitly fine-tuned to static analysis. This is realized through the Static Analysis Results Interchange Format (SARIF) [OAS18, Gra18] that SASP uses to communicate static-analysis results from servers to clients. Generally, SASP therefore promises a more tight coupled integration compared to LSP static analyses into editors, potentially needing more work on the server. Also, as of now, SASP and SARIF have seen little adoption by tool vendors. Currently, the standard is mostly put forward by GrammaTech, which through SASP offers third-party static analysis tools to allow a triaging of those tools’ results in GrammaTech’s CodeSonar [Gra05]. SARIF exporters currently exist for some few static analysis tools, including CogniCrypt [KNR⁺17], the Clang Static Analyzer [Gro07], Cppcheck [Mar07], and Facebook Infer [Fac15], which makes them amenable for an integration through SASP. However, right now, CodeSonar appears to be the only client ready to consume SARIF results, and it is unclear whether this will change in the near future. It is for this reason that MAGPIEBRIDGE builds, for now, on top of LSP instead of SASP and SARIF. Furthermore, SASP is still in the early stage of its development and there exists no formal specification of the protocol [Gra18], which makes it hard to compare it to LSP in detail and hard to use for our work. Once it matures, one could potentially use SASP as an intermediate format between the analyzers and MAGPIEBRIDGE, so that MAGPIEBRIDGE translates SASP to LSP. As this dissection is written, there exists already implementation in MAGPIEBRIDGE that translates SARIF results to LSP diagnostics.

5.2 Approach

In this section, we introduce how MAGPIEBRIDGE leverages both the Language Server Protocol (LSP) and Hypertext Transfer Protocol (HTTP) to integrate program analyses into editors. MAGPIEBRIDGE provides a default implementation of a LSP (language) server, which we call the **MagpieServer**. Figure 5.2 shows how **MagpieServer** runs multiple analyses and handles the communications in both LSP and HTTP. This server communicates with the LSP clients (editors) to report analysis results. When customized user interfaces are desired, the **MagpieServer** is at the same time a HTTP server that generates a web page and handles requests triggered by user interactions in the web page. The web page can be displayed in the LSP client, if the client supports rendering web pages, e.g., Visual Studio Code’s WebView APIs [Mic15b]. For LSP clients that do not have this support, the web page is displayed in the default web

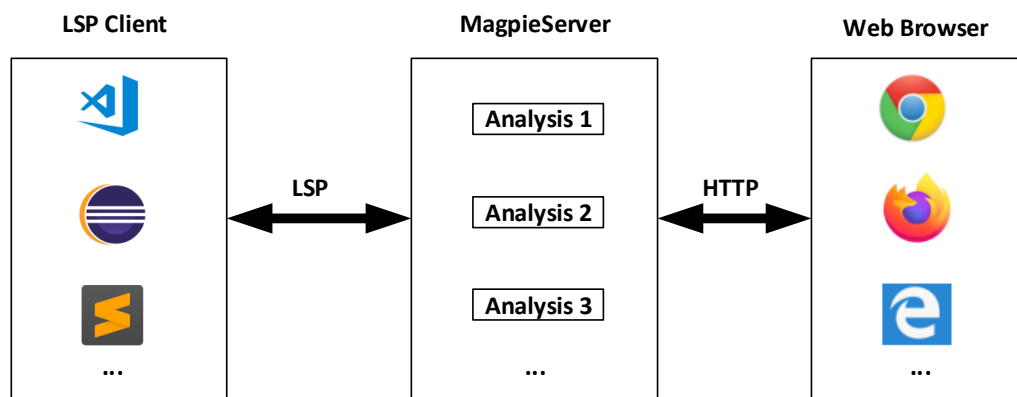


Figure 5.2: Overview of the communications supported by MAGPIEBRIDGE.

browser configured on the host machine. In the following, we first introduce the workflow of MAGPIEBRIDGE and relevant APIs in terms of these two communications in Section 5.2.1. Afterwards, we introduce how MAGPIEBRIDGE was built to support various analysis tools that are built on top of popular program analysis frameworks in Section 5.2.2.

5.2.1 The MAGPIEBRIDGE Workflow

MAGPIEBRIDGE uses the Language Server Protocol to integrate program analyses into editor and IDE clients. MAGPIEBRIDGE is implemented using the Eclipse LSP4J [CDvfsd16] LSP implementation based on JSON-RPC [Gro05], but MAGPIEBRIDGE hides LSP4J details and presents an interface in terms of high-level analysis abstractions. The overall workflow in Figure 5.3 shows how analysis can be registered, triggered by common coding activities inside the editors, and analysis results can be automatically displayed.

There are multiple mechanisms by which LSP-based tools can be used, but the most common mechanism is that an IDE or editor is configured to launch any desired tools. Each tool is built as a jar file based on the `MagpieServer`, with a main method that creates a `MagpieServer` (Listing 5.1), then adds the desired program analyses (`Analysis` in Listing 5.2) with `addAnalysis`, and then launches `MagpieServer` with `launch` so that it receives messages. There are two kinds of `Analysis` defined in MAGPIEBRIDGE: the `ServerAnalysis` and the `ToolAnalysis`. Both interfaces extend the `Analysis` interface. While the `ServerAnalysis` interface is used for analyses which are written in Java, the `ToolAnalysis` interface is designed for analyses that are written in other programming languages and have command line interfaces. The `ToolAnalysis` defines

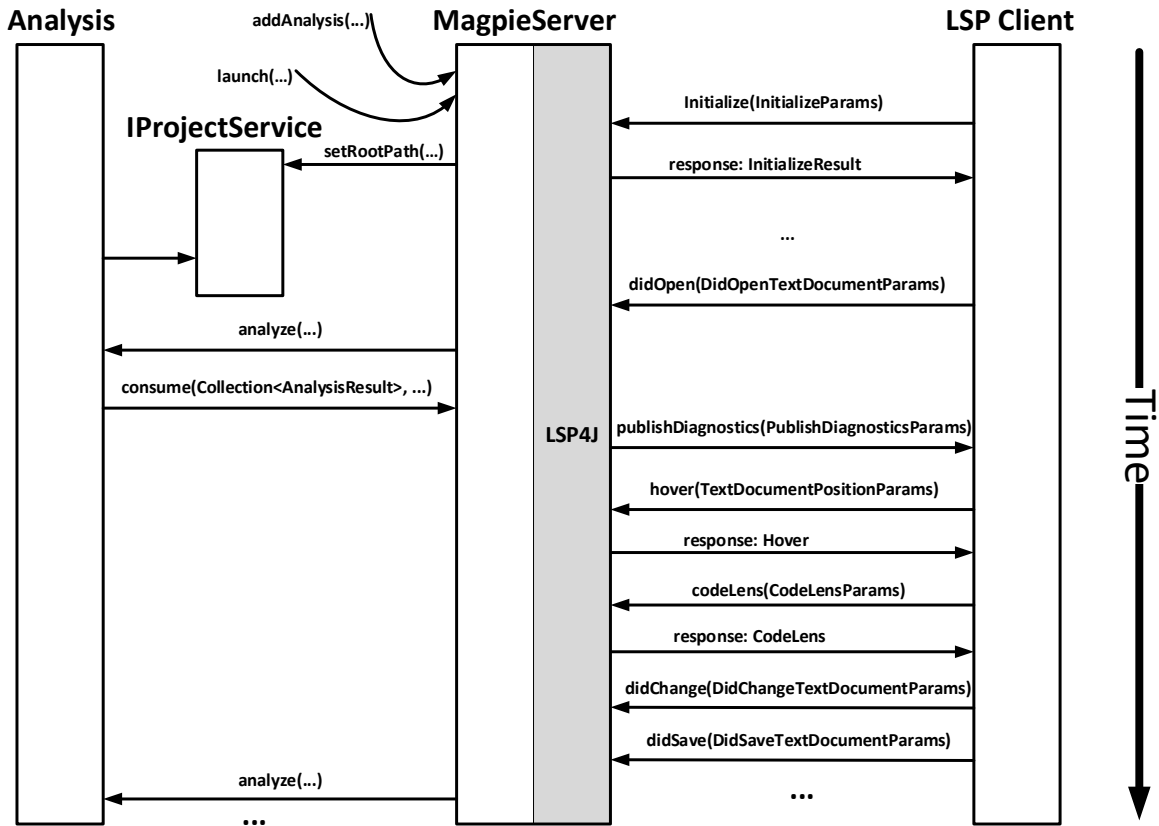


Figure 5.3: Overall MAGPIEBRIDGE workflow without customized web interfaces.

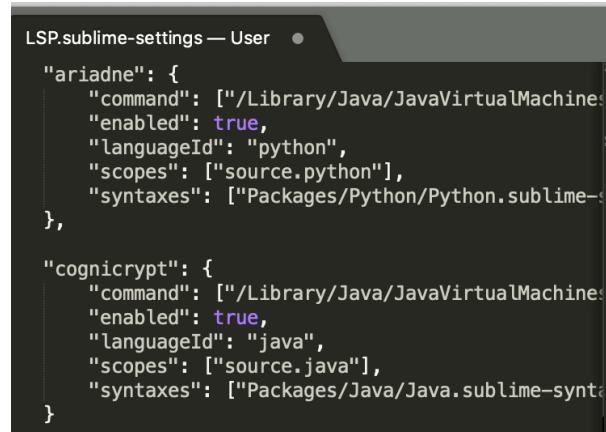


Figure 5.4: Configuration for Sublime Text to launch MagpieServer.

additional APIs to `Analysis` that allow definition of commands that need to be supported in the IDE integration. An `Analysis` (`ServerAnalysis` or `ToolAnalysis`) can be added to run on the `MagpieServer` with the `addAnalysis` and `launch` edges in Figure 5.3. With a jar that creates `MagpieServer` with analyses being added, `MAGPIEBRIDGE` can be used simply by configuring an editor to launch it. Figure 5.4 shows our Sublime Text setup to launch both Ariadne and CogniCrypt analyses. The user merely obtains jar files of the analyses and sets up Sublime Text to launch each of them for the appropriate languages. That is all the setup that is needed.

Based on LSP4J, there are several mechanisms for sending and receiving messages. Most clients/editors simply launch the server and then expect it to handle messages using standard I/O (e.g., Eclipse, IntelliJ, Emacs and Vim); however some clients expect to talk using a well-known socket (e.g., Spyder), Web-based tools communicate using WebSockets (e.g., Jupyter and Monaco) and only few tools support both standard I/O and socket (e.g., Visual Studio Code). Our `MagpieServer` supports all these channels out of the box and can be configured to communicate with a client using any of the channels. Once `MagpieServer` is launched, it interacts with the client tool using standard LSP mechanisms:

- The first step is initialization. The client sends an `initialize` message to the server, which includes information about the project being analyzed, such as its project root path. `MagpieServer` calls `setRootPath` on each `IProjectService` (service that resolves the project scope such as source code path and library code path) instance to initialize project path information. `MAGPIEBRIDGE` currently understands Eclipse, Maven, Gradle and Npm projects. `MagpieServer` also sends the response `InitializeResult` which declares its capabilities back to the client. This is shown in the upper portion of Figure 5.3.
- Subsequently, the client informs `MagpieServer` whenever it works with a file: the `didOpen`, `didChange` and `didSave` messages are sent to the server whenever files are opened, edited and saved respectively. These messages allow `MAGPIEBRIDGE` to call the `analyze` method whenever anything changes. This is shown with the `didOpen` and `analyze` edges in Figure 5.3. The analysis provider/writer can decide the granularity of when `MagpieServer` actually runs analysis and how much analysis it does via the `ServerConfiguration` argument passed to the constructor of `MagpieServer` as shown in Listing 5.1.
- As shown in the rest of Figure 5.3, analysis uses the `consume` method to report analysis results of type `AnalysisResult` (Listing 5.4) to `MagpieServer`, which handles them via

the appropriate LSP mechanism, specified by the `kind` method (Listing 5.4), which returns a `Kind` (Listing 5.5):

Diagnostic denotes issues found in the code, corresponding to lists of errors and warnings that might be reported by a compiler. Tools typically report them either in a list of results or highlight the results directly in the code. When the program analysis provides such results via `consume`, `MagpieServer` reports them to the client tool with the LSP `publishDiagnostics` API.

Hover denotes annotations to be displayed for a specific program variable or location. It could be used to report e.g., the type of a variable or the targets of a function call. Tools often show them when the cursor highlights a specific location. When the program analysis provides such results via `consume`, `MagpieServer` keeps them and reports them to the client tool as responses to LSP `hover` API calls by the client tool.

CodeLens denotes information to be added inline in the source code, analogous to generated comments. Tools typically report them as distinguished lines of text inserted between lines of source code. When the program analysis provides such results via `consume`, `MagpieServer` keeps them and reports them to the client tool as responses to LSP `codeLens` API calls by the client tool. A `CodeLens` can also be actionable. When the user clicks on it, encoded commands defined in the `command` method of `AnalysisResult` (Listing 5.4) will be executed.

These analysis results have a `position` method that returns a `Position` (Listing 5.6) denoting the source location to which the result pertains. The result requires a precise location based on starting and ending line and column numbers, which is required by the LSP protocol. Note that the `Position` of `MAGPIEBRIDGE` implements the Java `Comparable` interface; `MAGPIEBRIDGE` exploits this to store analysis results in `NavigableMap` structures so that it can find the nearest result if a user hovers in a location near a result, e.g., some whitespace immediately after a variable or expression.

To enable complex interactions between the user and the analysis, `MAGPIEBRIDGE` can be configured to start a HTTP server and communicate with a web front end inside the LSP client or a web browser (if the client does not support rendering web pages):

- When the `MagpieServer` is configured to show a web interface, it creates and starts a local HTTP server via `CreateAndStartLocalHttpServer` when it responds to the client with `InitializeResult` as shown in Figure 5.5. `MAGPIEBRIDGE` has a default HTTP handler called `MagpieHttpHandler`.
- The URL hosting the web interface is then opened by the web browser engine and a GET request is sent to the `MagpieServer`. HTTP requests are handled by the `MagpieHttpHandler`. The `MagpieHttpHandler` generates a web page for the analysis, which is based on the `getConfigurationOptions()` and `getConfiguredActions()` methods implemented for the analysis in Listing 5.2. For each analysis, one can define a list of `ConfigurationOptions` (see Listing 5.7) that should be shown in the web interface. A configuration option can have different `OptionTypes` as shown in Listing 5.8: `checkbox` stands for an option with boolean value (e.g., an analysis rule set is enabled or disabled), `text` stands for text-based inputs (e.g., the path to configuration files) and `container` stands for an option group that contains multiple options. Similarly, actions (e.g., run the analysis) upon buttons can be defined for an analysis. A `Configuration Action` is defined as a `Runnable` object as shown in Listing 5.9. Figure 5.6 shows the default web interface generated by `MAGPIEBRIDGE`, in which the configuration options are listed on the left and the actions are

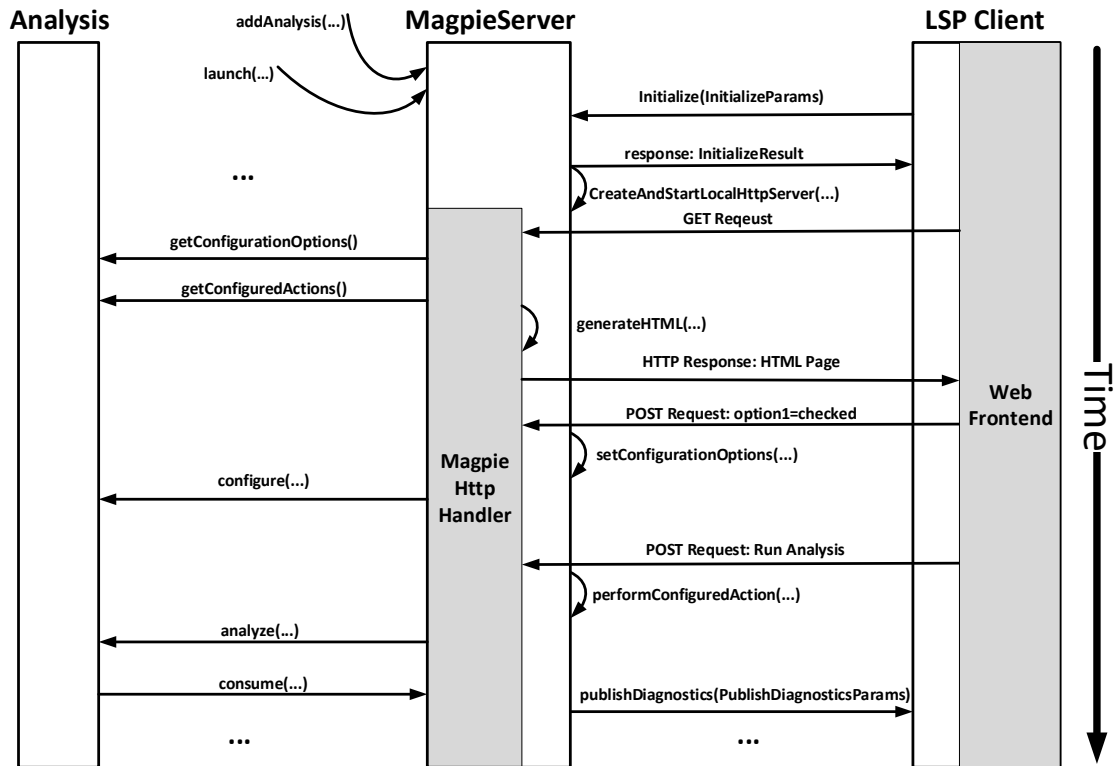


Figure 5.5: Overall MAGPIEBRIDGE workflow with customized web interfaces.

listed on the right. One can also customize a web page with more styles as the one shown in Figure 5.7.

- Users can select configuration options in the web page and submit them back to the **MagpieServer**. POST requests encoding user's selection and actions inside the web page are generated and handled by the **MagpieServer**. In Figure 5.5, the user first selected a configuration option `option1=checked`, this analysis is the configured by with option via the `configure` method (see Listing 5.2). Then the user clicks a `Run Analysis` button to trigger the analysis to run. The analysis results are then shown in the editors via LSP mechanisms.

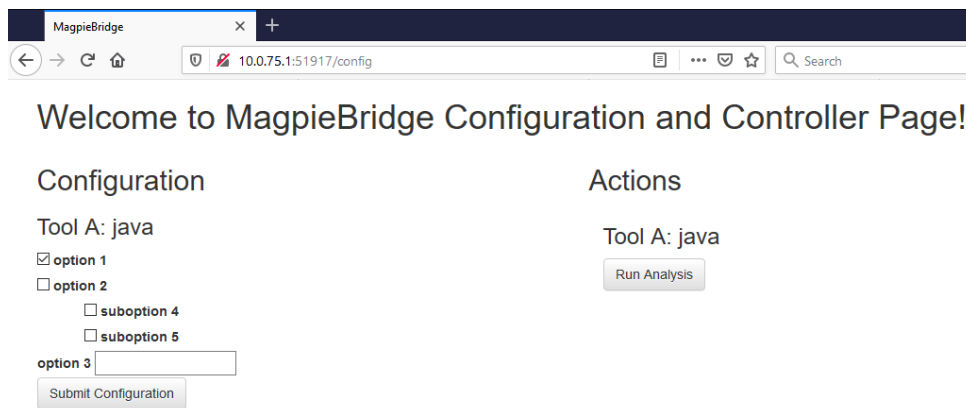


Figure 5.6: The default web interface generated by MAGPIEBRIDGE.

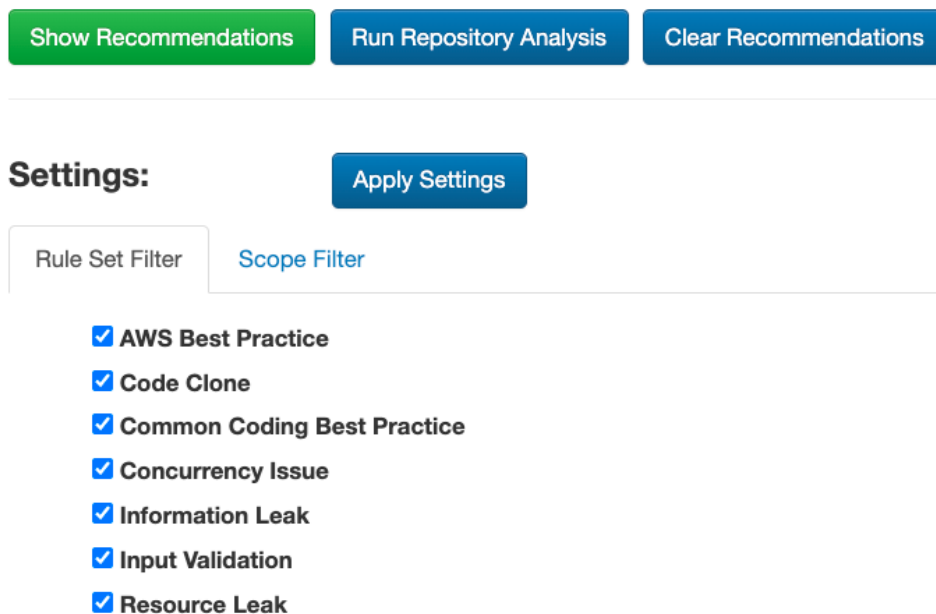


Figure 5.7: A customized web interface we designed for Amazon CodeGuru Reviewer (later introduced in Chapter 6).


```

public class MagpieServer implements AnalysisConsumer, LanguageServer, LanguageClientAware{
    protected LanguageClient lspClient;
    protected Map<String, IProjectService> languageProjectServices;
    protected Map<String, Collection<Either<ServerAnalysis, ToolAnalysis>>> languageAnalyses;

    public MagpieServer(ServerConfiguration config) {...}
    public void addProjectService(String language, IProjectService projectService){...}
    public void addAnalysis(Either<ServerAnalysis, ToolAnalysis> analysis, String... languages){...}
    public void doAnalyses(String language){...}
    public void consume(Collection<AnalysisResult>, String source){...}

    protected void createAndStartLocalHttpServer() {...}
    public List<ConfigurationOption> setConfigurationOptions(Map<String, String> options) {...}
    public void performConfiguredAction(NameValuePair actionName, NameValuePair sourceName) {...}
    ...
}

```

Listing 5.1: The core of the server.

```

public interface Analysis<T extends AnalysisConsumer> {
    public String source();
    public void analyze(Collection<? extends Module> files, T server, boolean rerun);
    public default List<ConfigurationOption> getConfigurationOptions() {
        return new ArrayList<>();
    }
    public default List<ConfigurationAction> getConfiguredActions() {
        return new ArrayList<>();
    }
    public default void configure(List<ConfigurationOption> configuration) {}
    public default void cleanUp(){};
}

```

Listing 5.2: Interface for defining analysis on the server.

```

public interface IProjectService {
    public void setRootPath(Path rootPath);
    public String getProjectType();
}

```

Listing 5.3: Interface for defining service which resolves project scope.

```

public interface AnalysisResult {
    public Kind kind();
    public String toString(boolean useMarkdown);
    public Position position();
    public Iterable<Pair<Position,String>> related();
    public DiagnosticSeverity severity();
    public Pair<Position, String> repair();
    public String code();
    public default Iterable<Command> command() { return Collections.emptySet(); }
}

```

Listing 5.4: Interface for defining analysis result.

```

public enum Kind { Diagnostic, Hover, CodeLens }

```

Listing 5.5: Enum for defining kinds of analysis results.

```

public interface Position extends Comparable {
    public int getFirstLine();
    public int getLastLine();
    public int getFirstCol();
    public int getLastCol();
    public int getFirstOffset();
    public int getLastOffset();
    public URL getURL();
}

```

Listing 5.6: Interface for defining position

```

public class ConfigurationOption {
    private String id;
    private final String name;
    private final OptionType type;
    private List<ConfigurationOption> children;
    private String value;
    private String source;
    private Object extra;
    ...
}

```

Listing 5.7: Class for defining a configuration option of the analysis.

```

public enum OptionType { checkbox, text, container }

```

Listing 5.8: Enum for defining types of configuration options.

```

public class ConfigurationAction {
    private String id;
    private String name;
    private Runnable action;
    private String source;
    ...
}

```

Listing 5.9: Class for defining an action allowed by the analysis.

5.2.2 The MAGPIEBRIDGE System

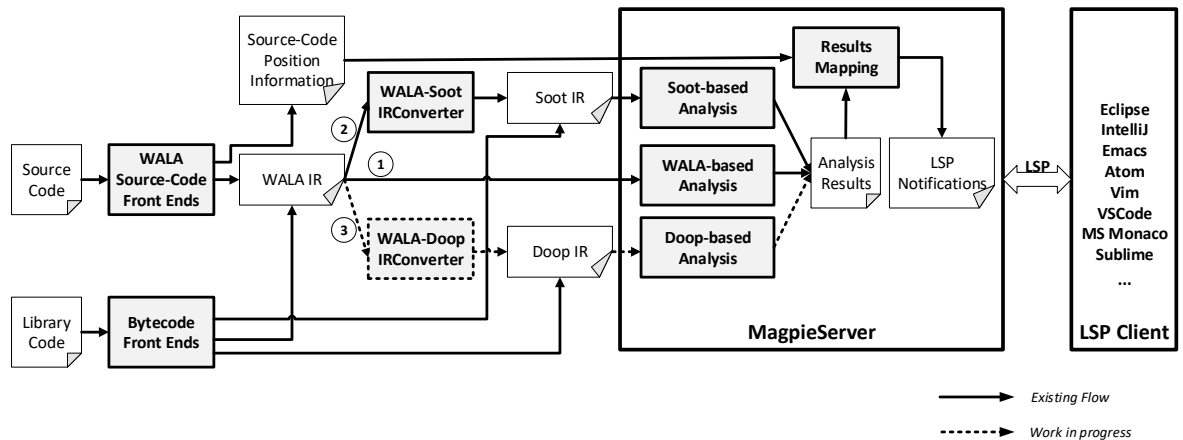
We explain our MAGPIEBRIDGE system with an overview in Figure 5.8. MAGPIEBRIDGE needs to support various analysis tools that were built on top of different frameworks, e.g., TAJ, Andromeda and HybriDroid use WALA, while CogniCrypt, FlowDroid and DroidSafe rely on Soot and many other analyses are based on Doop. These analysis frameworks have different IRs, which MAGPIEBRIDGE needs to use to generate analysis results. One key requirement for all the frameworks supported by MAGPIEBRIDGE is very precise source-code mappings, since in LSP all the messages communicate using starting and ending line and column numbers. In the following we explain how MAGPIEBRIDGE achieves this requirement for WALA-based analyses, Soot-based analyses and Doop-based analyses respectively.

5.2.2.1 WALA-based Analysis

The simplest code path in MagpieBridge (flow ① in Figure 5.8) uses WALA source language front ends for creating IR on which to perform analysis. WALA comprises both bytecode and source-code front ends for different languages (Java, Python and JavaScript), and the source-code front end preserves source-code positions very well. This information can be consumed later in the LSP notifications, since it is kept in WALA’s IR. WALA’s IR is a traditional three-address code in Static Single Assignment (SSA) form, which is translated from WALA’s Common Abstract Syntax Tree (CAst).

The approach to source-code front ends for WALA is using existing infrastructure for each supported language: Eclipse JDT for Java, Mozilla Rhino for JavaScript and Jython for Python. Each of these front ends is maintained with respect to its respective language standards, and all the front ends provide precise mappings of source locations for constructs. To provide a detailed source mapping for the generated IR, each WALA function body has an instance of `DebuggingInformation` (Listing 5.10) which allows MAGPIEBRIDGE to map locations from requests to IR elements at a very fine level.

Listing 5.10 details how much source mapping information is available. `getCodeBodyPosition` is the source range of the entire function, and `getCodeNamePosition` is the position of just the name in the body. `getInstructionPosition` is the source position of a given IR instruction. `getOperandPosition` is the source position of a given operand in an IR instruction. `getParameterPosition` is the position of a given parameter declaration in the source.



```

public interface DebuggingInformation {
    Position getCodeBodyPosition();
    Position getCodeNamePosition();
    Position getInstructionPosition(int instructionOffset);
    String[][] getSourceNamesForValues();
    Position getOperandPosition(int instructionOffset, int operand);
    Position getParameterPosition(int param);
}

```

Listing 5.10: Debugging information interface.

5.2.2.2 Soot-based Analysis

Soot comprises a solid Java bytecode front end. The bytecode only has the line number of each statement. This is not sufficient to support features such as hover, fix and codeLens in an editor. For those features, position information about variables, expressions, calls and parameters are necessary. However, they are lost in the bytecode. Soot further comprises source-code front ends. Such front ends, however, require frequent updates due to the frequently changing specification of the Java source language, which has caused Soot's source-code front ends to become outdated. Besides, Soot IR was not designed to keep precise source-code position information, e.g., there is no API for getting the parameter position in a method. Our approach is to take WALA's

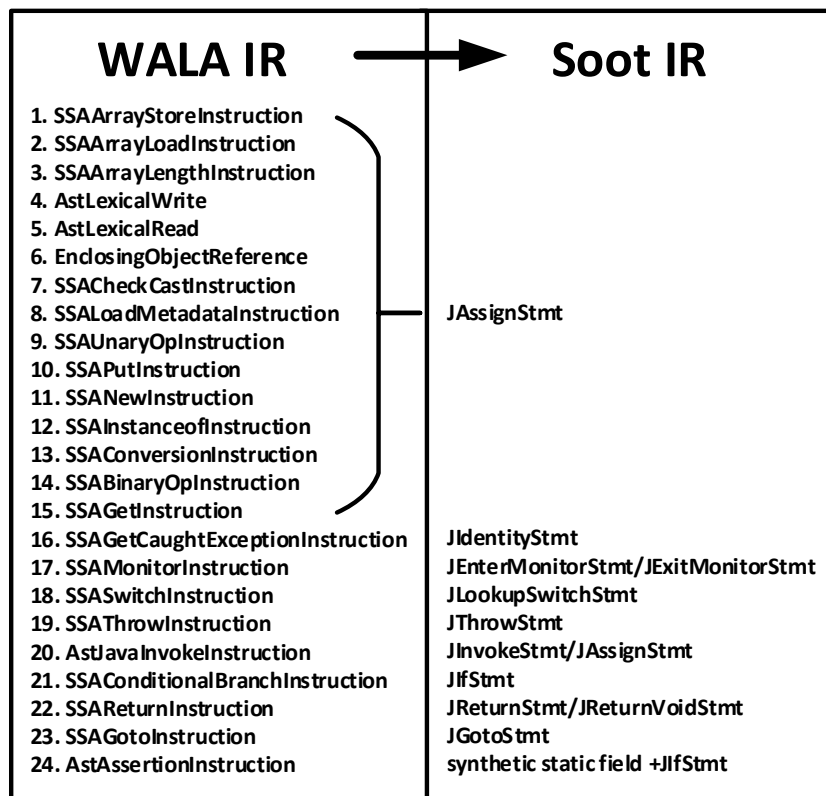


Figure 5.9: Conversion from WALA IR to Soot IR.

```

public class ExampleAnalysis implements ServerAnalysis{

    @Override
    public String source(){
        return "Example Analysis"
    }

    @Override
    public void analyze(Collection<Module> sources, MagpieServer server){
        ExampleTransformer t = getExampleTransformer();
        loadSourceCodeWithWALA(sources);
        JavaProjectService service = (JavaProjectService) server.getProjectService("java");
        loadLibraryCodeWithSoot(service.getLibraryPath());
        runSootPacks(t);
        List<AnalysisResult> results = t.getAnalysisResults();
        server.consume(results);
    }
    ...
}

public class Example{

    public static void main(String... args){
        MagpieServer server = new MagpieServer();
        IProjectService service = new JavaProjectService();
        ExampleAnalysis analysis = new ExampleAnalysis();
        String language = "java";
        server.addProjectService(language, service);
        server.addAnalysis(language, analysis);
        server.launch(...);
    }
}

```

Listing 5.11: The MagpieServer runs a Soot-based analysis.

source-code front end to generate WALA IR and convert it to Soot IR. Soot has multiple IRs, the most commonly used IR is called Jimple [VRCG⁺10]. Jimple is also a three-address code and has Java-like syntax, but is simpler, e.g., no nested statements. Opposed to WALA IR, Jimple is not in SSA-form. Both WALA and Soot are implemented in Java and manipulate the IR through Java objects. This makes the conversion between the IRs feasible. In particular, we have implemented the WALA-Soot IRConverter and defined the common APIs (Listing 5.4) to encode analysis results, as well as the **MagpieServer** (Listing 5.1) that hosts the analysis. Currently the WALA-Soot IRConverter only converts WALA IR generated by WALA’s Java source-code front end. In fact, WALA uses a pre-IR before generating the actual WALA IR in SSA-form, and this non-SSA pre-IR is actually the IR that we convert to Jimple. Since also Jimple is not in SSA, this conversion is more direct. This pre-IR contains 24 different instructions as shown in Figure 5.9. After studying both IRs, we found out that 15 instructions in WALA IR can be converted to JAssignStmt in Jimple. Most of the times the conversion is one-to-one, only a few cases are one-to-many. The precise source-code position information from WALA IR is encapsulated in the tags (annotations) of the converted Soot IR. In the future, we plan to convert WALA IR from front ends of other languages such as Python and JavaScript to a potentially extended version of the Soot IR.

The flow ② in Figure 5.8 for integrating Soot-based analysis starts by dividing the analyzed program code into application source code and library code (which can be in binary form). The source code is parsed by one of WALA’s source-code front ends and it outputs WALA IR, as

well as precise source code position information associated in the IR. For a Soot-based analysis, the WALA IR is translated by a WALA-Soot IRConverter into Soot IR (Jimple). The library code is parsed by Soot's bytecode front end and then complements the program's IR obtained from the source code. The Soot IR in Figure 5.8 thus consists of two parts: Jimple converted by the WALA-Soot IRConverter, which represents the source-code portion/application code of the program, and Jimple generated by Soot's bytecode front end which represents the library code. Based on the composite Soot IR, Soot further conducts a call graph and optionally also pointer analysis, which can then be followed by arbitrary data-flow analyses.

Listing 5.11 shows an example of running a Soot-based analysis `ExampleTransformer` (analyses are called transformers in Soot) on the `MagpieServer`. The `ExampleTransformer` accesses the program through the singleton object `Scene` in order to analyze the program. Once the `MagpieServer` receives the source code, the method `loadSourceCodeWithWALA` parses the source code, converts it to Soot IR with the WALA-Soot IRConverter and stores the IR in the `Scene`. The class `JavaProjectService` resolves the library path for the current project. `loadLibraryCodeWithSoot` loads the necessary library code from the path and adds the IR into `Scene`. The method `runSootPacks` invokes Soot to build call-graph and run the actual analysis. The analysis results will be then consumed by the server. In this example, only the source files sent to the server are analyzed together with the library code. However, it can be configured to perform a whole-program analysis, since the source code path can also be resolved by `JavaProjectService`.

We explain how the class `JavaProjectService` which implements `IProjectService` resolves the full Java project scope, i.e., source code path and library code path. As specified in LSP, the editors send the project root path (`rootURI`) to the server in the first request `initialize`. Library and source code path can be resolved by using the build-tool dependency plugins (e.g., caching results of `mvn dependency:list`) or parsing the configuration (e.g., `pom.xml`, `build.gradle`) and source code files located in the root path. Project structure conventions for different kinds of projects are also considered in `MAGPIEBRIDGE`. For more customized projects, `MAGPIEBRIDGE` also allows the user to specify the library and source code path manually as program arguments.

5.2.2.3 Doop-based Analysis

Doop uses Datalog to allow for declarative analysis specifications, encoding instructions as Datalog relations as well as instruction source positions. There is code to convert from the WALA Python IR to Datalog, and that captures both the semantics of statements as well as source mapping, and these declarations capture the information needed for analysis tool support. For instance, there is a Datalog relation that captures instruction positions and is generated directly from WALA IR:

```
.decl Instruction_SourcePosition(?insn:Instruction,
    ?startLine:number, ?endLine:number, ?startColumn:number, ?endColumn:number)
```

This code has been used experimentally for analysis using Doop of machine code written in Python. This code path could be used to express analyses in editors using `MAGPIEBRIDGE`, and such work is under development.

5.3 Integration of Existing Static Tools

To make MAGPIEBRIDGE more concrete, we use two illustrative analyses, based on different frameworks—Soot and WALA, respectively—for different languages—Java and Python—in different domains—security and bug finding—both in a range of editors:

CogniCrypt analyzes how cryptographic APIs are used in a program, and reports a variety of vulnerabilities such as encryption protocols being misused or when protocols are used in situations where they should not. The tool then also gives suggestions on how to fix the problem. CogniCrypt comprises a highly efficient demand-driven, inter-procedural data-flow analysis [SAB19] based on Soot, and has its own Eclipse-based plugin. As Table 5.1 shows, its plugin actually required substantially more code than the analysis itself. The plugin also is limited to Eclipse. We illustrate what it looks like to use CogniCrypt in multiple tools using MAGPIEBRIDGE. To keep exposition simple, we focus on a case in which a weak encryption mode is used (Electronic Codebook Mode, ECB). In the general case the analysis can also report complex flows through the program. Screenshots in Figure 5.10, Figure 5.11, Figure 5.12 and Figure 5.13 show the same crypto warning reported by CogniCrypt in different editors. As we can see, only the call `Cipher.getInstance` with the insecure parameter is marked in each editor. We also compared our MAGPIEBRIDGE-based CogniCrypt to the existing CogniCrypt Eclipse Plugin. Details about this comparison can be found in Section C.1.1.

Ariadne analyzes how tensor (multi-dimensional array) data structures are used in machine-learning code written in Python, and reports a range of information. It presents basic tensor-shape information for program variables, and finds and fixes certain kinds of program bugs. A key operation is reshaping a tensor: the `reshape` operation takes a tensor and a new shape, and returns a new tensor with the desired shape when that is possible. To simplify complex tensor semantics, a tensor can be reshaped only when its total size is equal to size of the desired new shape. Another operation is performing a convolution, e. g., `conv2d`, which requires the input tensor to have a specific number of dimensions. We illustrate cases of these bugs, and how they are shown in multiple editors (Figure 5.14, Figure 5.15, Figure 5.16, Figure 5.17, and Figure 5.18).

We illustrate how the aspects of LSP used by MAGPIEBRIDGE are rendered in a variety of editors; while there are common notions such as a list of diagnostics, different tools make different choices in how those elements are displayed. We describe in turn several LSP aspects and how analysis information is displayed using them.

5.3.1 Diagnostics

The most straightforward interface is for an analysis to report a set of issues, but even this simple concept is handled differently in different editors.

- Some editors have a problem view, i.e., a list summarizing all outstanding issues. An example of this interface is Sublime Text, illustrated in Figure 5.12 where a warning about weak encryption is shown in a list.
- Some editors do not have such a list, but choose to highlight issues directly in the code. An example of this interface is Monaco, illustrated in Figure 5.11; the same warning about weak encryption is shown inline. To minimize clutter, editors typically make such warnings as hovers, and we show it displayed in Monaco. A somewhat different visualization of the same idea is in Figure 5.17, in which Atom shows an invalid use of `reshape` in Tensorflow.

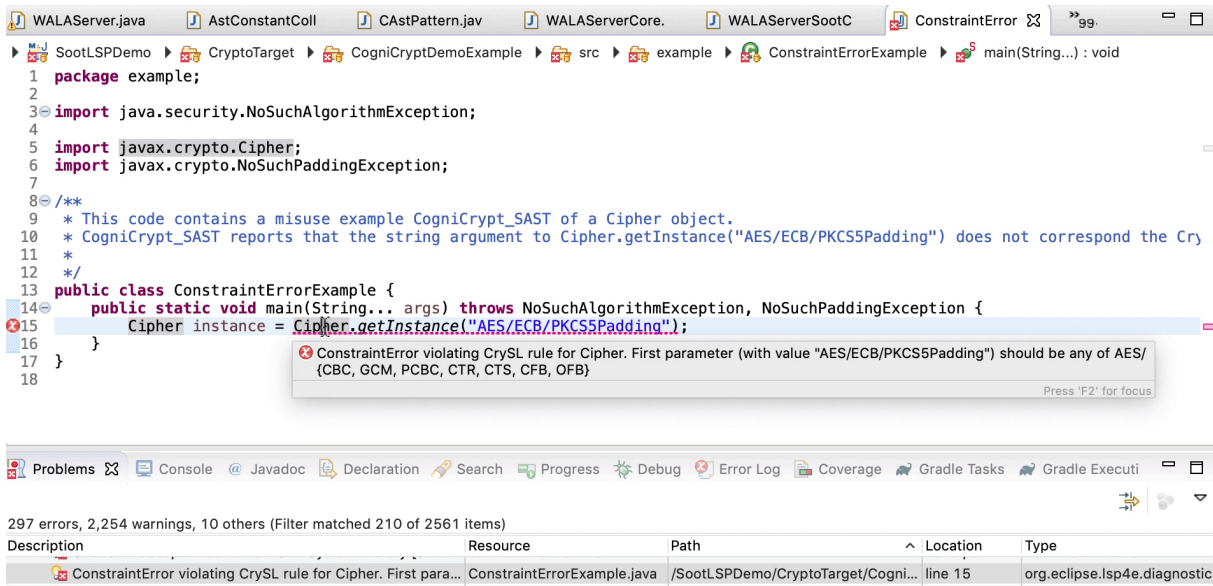


Figure 5.10: Insecure crypto warning in Eclipse.

Monaco Example

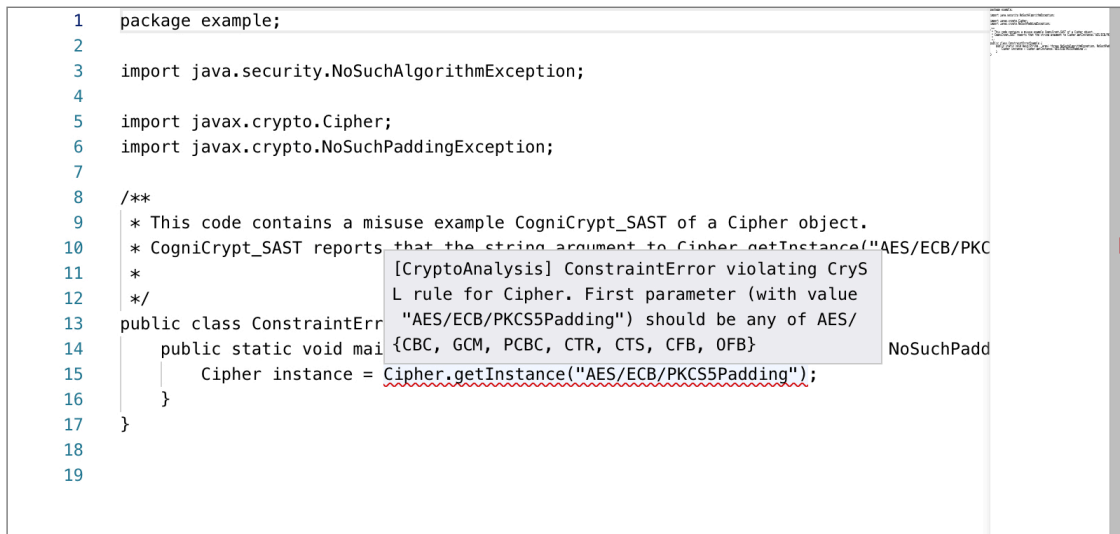


Figure 5.11: Insecure crypto warning in Monaco.

- Some editors do both. An example of this interface is Eclipse, illustrated in Figure 5.10 where a warning about weak encryption is shown both inline and in a list. Again to minimize clutter, the inline message is realized via a hover.

Note that all issues displayed here are computed by the very same analysis in all editors and rendered as the same LSP objects; however, they appear natural in each editor, due to the editor-specific LSP client implementations.

5.3.2 Code Lenses

Code lenses look like comments, but are inserted into the code by analyses and are used to reflect generally-useful information about the program. An example is shown in Figure 5.14,

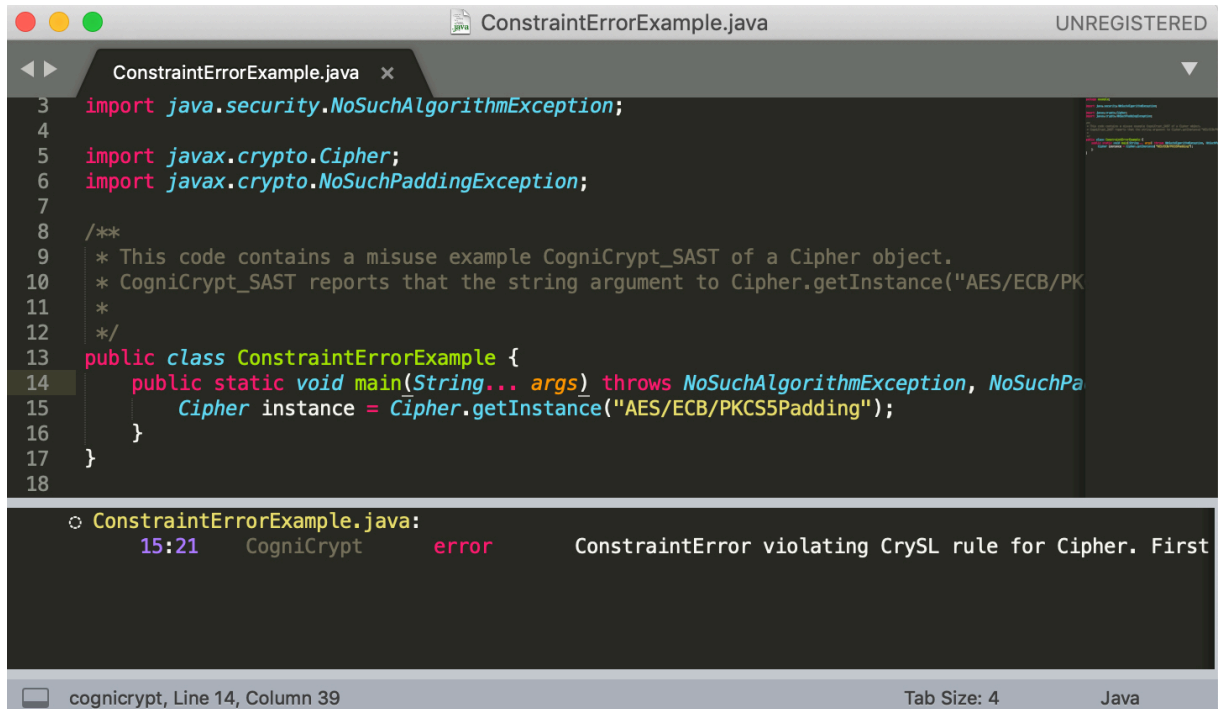


Figure 5.12: Insecure crypto warning in Sublime Text.

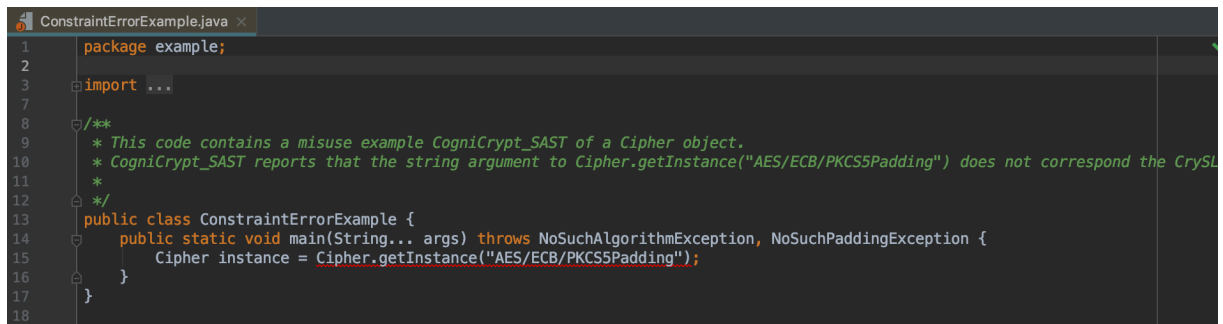


Figure 5.13: Insecure crypto warning in IntelliJ.

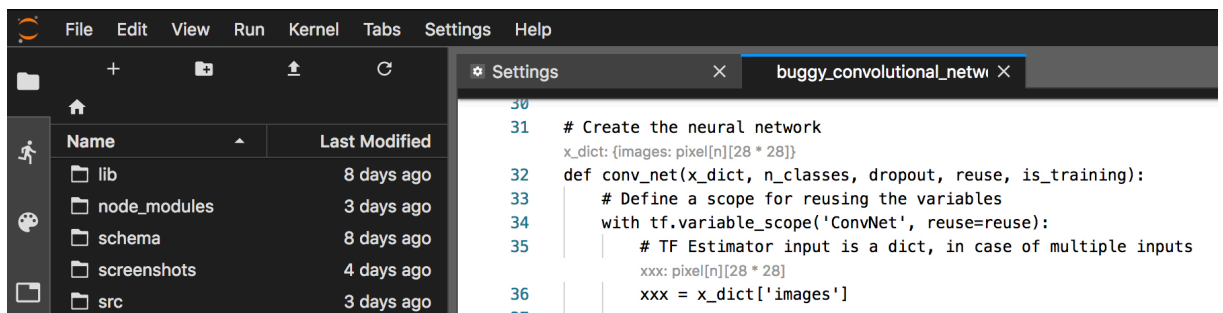


Figure 5.14: Code lenses showing tensor types in JupyterLab.

in which the shapes of tensors are listed explicitly for various program variables and function arguments.

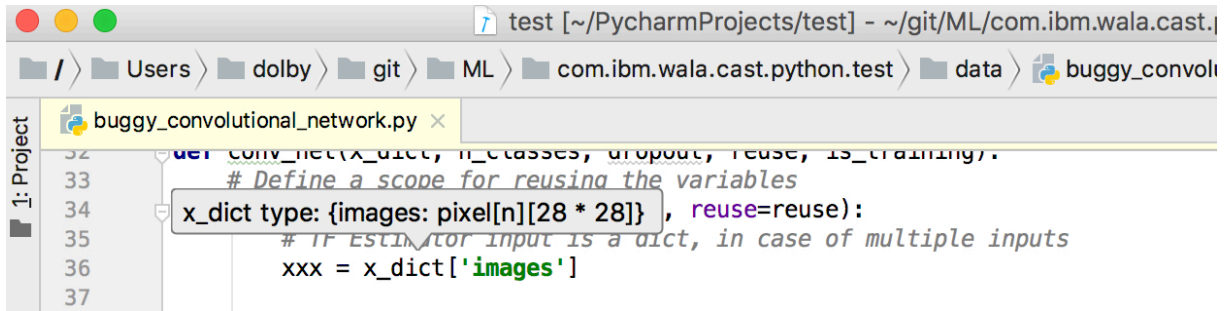


Figure 5.15: Hover tip showing tensor types in PyCharm.

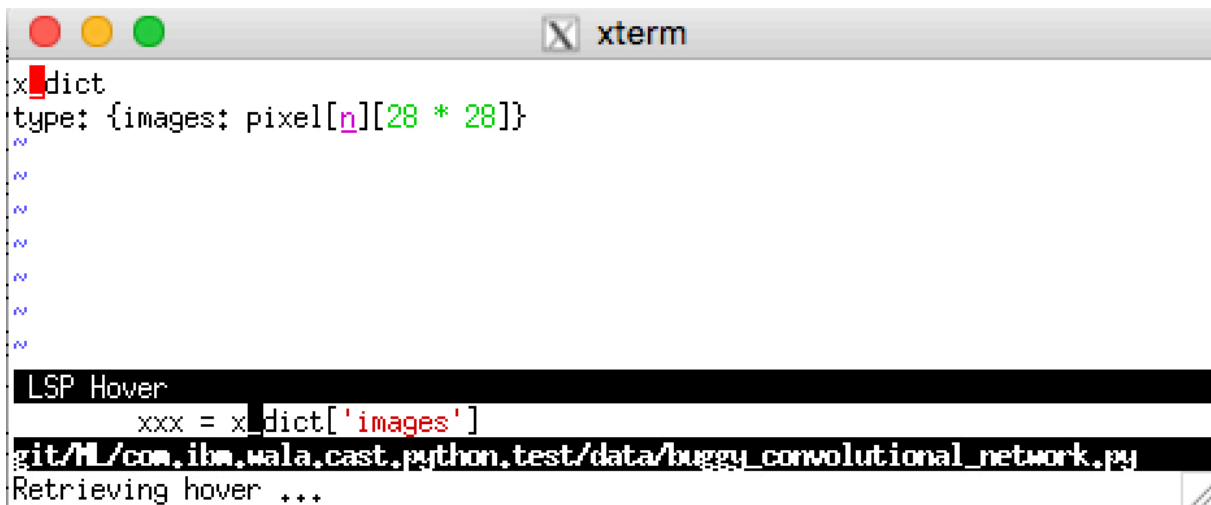


Figure 5.16: Hover tip showing tensor types in Vim.

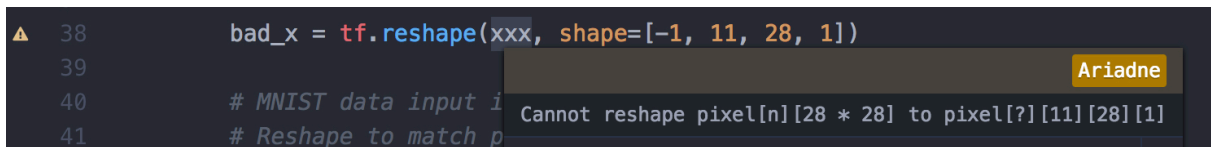


Figure 5.17: Diagnostic warning showing an incompatible reshape in Atom.

5.3.3 Hovers

Hovers are used to reflect generally-useful information about the program, but, unlike code lenses, they are visible only on demand. As such, an analysis can sprinkle them liberally in the program and they will not be distracting since they are only visible when needed. Different tools have different ways of user interaction. In Figure 5.15, the user hovers over the variable `x_dict` in PyCharm to reveal the shape of tensors that it holds. In Figure 5.16, the user enters a Vim command with the cursor over the variable `x_dict`.

5.3.4 Repairs

LSP provides the ability to specify fixes for diagnostics; a diagnostic can specify replacement text for the text to which the given diagnostic applies. The method `repair()` in the interface `AnalysisResult` is designed exactly for this purpose (see Listing 5.4). Figure 5.18 shows an example of this: the top half shows an error report in Visual Studio Code that a call to `conv2d`

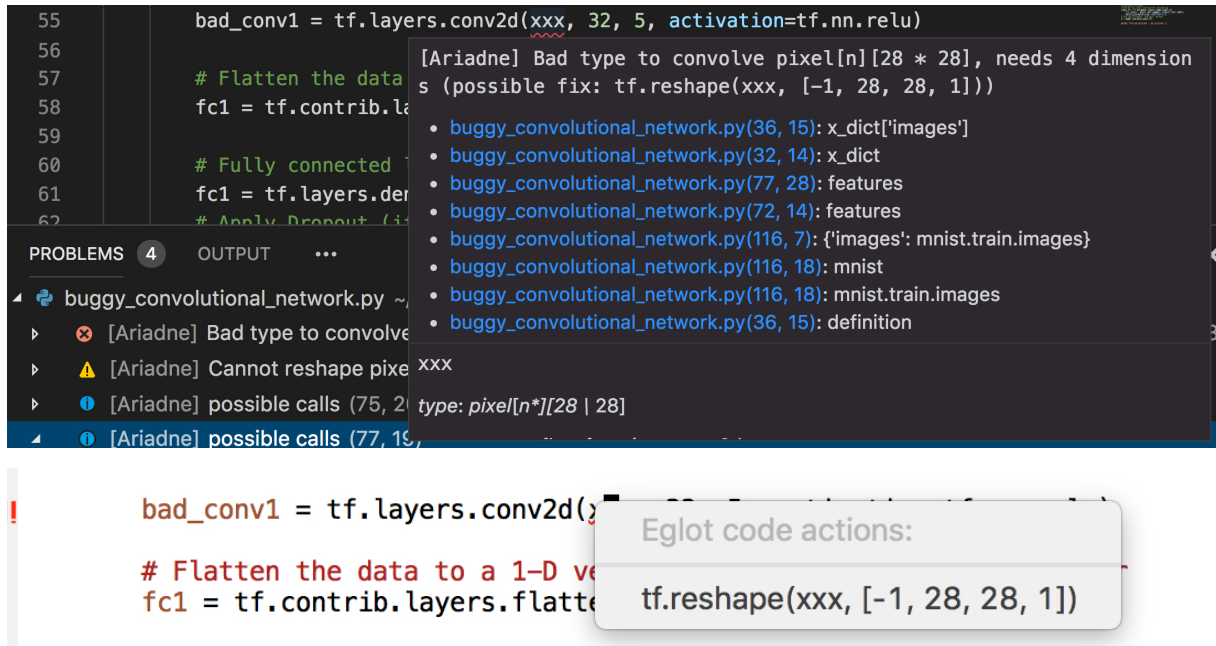


Figure 5.18: Diagnostic error showing fixable incorrect dimensions for `conv2d`. Error shown in Visual Studio Code and quick fix in Emacs.

is invalid, since such calls require a tensor with four dimensions whereas the provided argument has only 2. However, the analysis determines that a plausible fix is to **reshape** the provided argument to have more dimensions, and the lower part of the figure shows a prompt, in Emacs, suggesting a **reshape** call to insert.

5.4 More Tool Integrations

In addition to CogniCrypt and Ariadne, which we demonstrated extensively, we also integrated COVA (introduced in Chapter 4)⁵, the taint analysis tool FlowDroid⁶ and the Facebook Infer analyzer [Fac15]⁷ into IDEs with MAGPIEBRIDGE.

We integrated COVA with MAGPIEBRIDGE into IDEs to show path constraints on demand. The idea behind this integration is that developers can use it to identify how code fragments are affected by which configuration options when maintaining and testing their code. For example, developers want to test a newly implemented feature in a suitable environment. They could choose the “show constraint” code action next to the code fragment they want to test as shown in Figure 5.19. The IDE integration of COVA will show the path constraint of the queried code fragment based on both software and hardware configuration options as in Figure 5.20. On the left side, the constraint-APIs (from where the configuration option values are read) that are tracked by COVA are displayed as a mapping from their symbolic values to the API signatures. Below the constraint-APIs, a witness path shows how these constraint-APIs are used in the code. On the right side, the path constraint and witness path are both shown in a hover message indicating that the SDK version must be smaller than 15 and the model of the device is “HTC”. With such information developers can then set up the appropriate testing environment.

FlowDroid is a Soot-based tool and is written in Java. For its integration we implemented

⁵Integration prototype for COVA available at <https://github.com/secure-software-engineering/COVA>

⁶Integration prototype for FlowDroid available at <https://github.com/MagpieBridge/FlowDroidLSPDemo>

⁷Integration prototype for Facebook Infer available at <https://github.com/MagpieBridge/InferIDE>

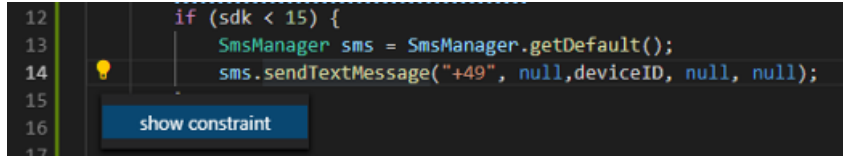


Figure 5.19: Requesting path constraint from COVA by using the pop-up “show constraint” code action.

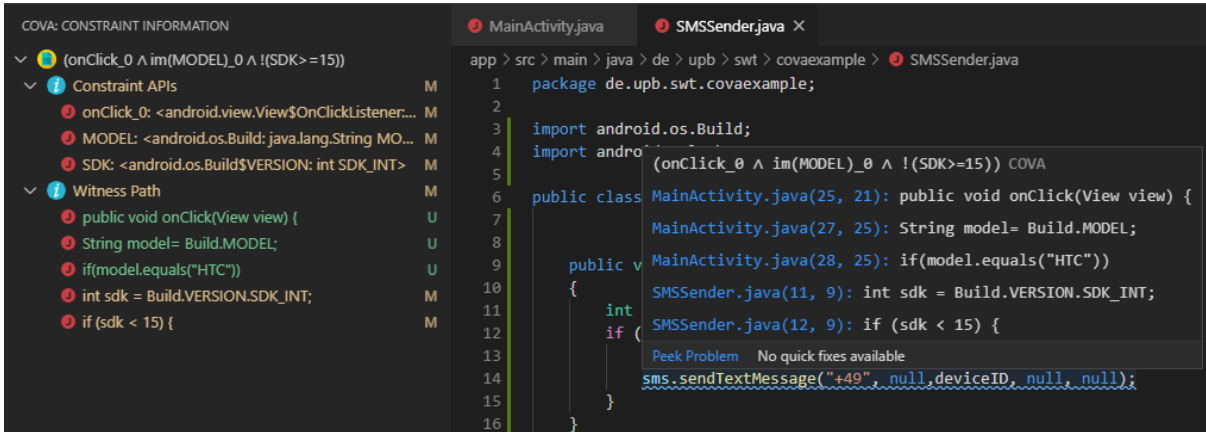


Figure 5.20: Left: Constraint-APIs and witness path; Right: path constraint and witness in hover message.

a **ServerAnalysis** (see Section 5.2.1) that calls the `SetupApplication.runInfoFlow` of FlowDroid with the default configuration in the `analyze` method. Because FlowDroid only analyzes bytecode, we modified it to have separate code loading. The modified FlowDroid uses the IR-Converter (see Section 5.2.2) to load the source code and Soot’s byte code front end to load the library code. Figure 5.21 shows FlowDroid analyzing the data flow starting from a parameter of the HTTP request, finding a cross-site scripting vulnerability in a web app which can be exploited by attackers, and showing a witness trace of it in Visual Studio Code. The expressions in the witness are shown precisely, which is possible since the IRConverter of MAGPIEBRIDGE is able to run FlowDroid unchanged on the converted IR and recover precise source mappings. As far as we know, this has never been done before with FlowDroid. MAGPIEBRIDGE then renders this precise trace (encoded as `related` of `AnalysisResult` in Listing 5.4) from FlowDroid in the IDE, also the first time this has been done. We would like to have both COVA and FlowDroid integrated into Android Studio, however, the only open-source LSP support for Android Studio we found was unfortunately too buggy.

Infer is a static analyzer that finds quality bugs in Java, C/C++ code, e.g., null pointer exceptions, resource leaks, race conditions, etc. It is written in OCaml and provides a command line interface. We implemented a **ToolAnalysis** (see Section 5.2.1) that runs the command “`infer run -reactive`” by default for Java projects that are built by Maven and Gradle. Users can also define any command they would like infer to run via the default web interface generated by MAGPIEBRIDGE as introduced in Section 5.2.1. This was realized by setting providing a text-based `ConfigurationOption`. Figure 5.22 shows how infer warns about a resource leak in the web editor Gitpod [Git18]. We also integrated it for Eclipse, IntelliJ and Visual Studio Code. Our Visual Studio Code Extension InferIDE is publicly freely available at the market place of Visual Studio Code. For IntelliJ, we developed a plugin⁸ to support

⁸IntelliJ LSP support public available at <https://github.com/MagpieBridge/IntelliJLSP>

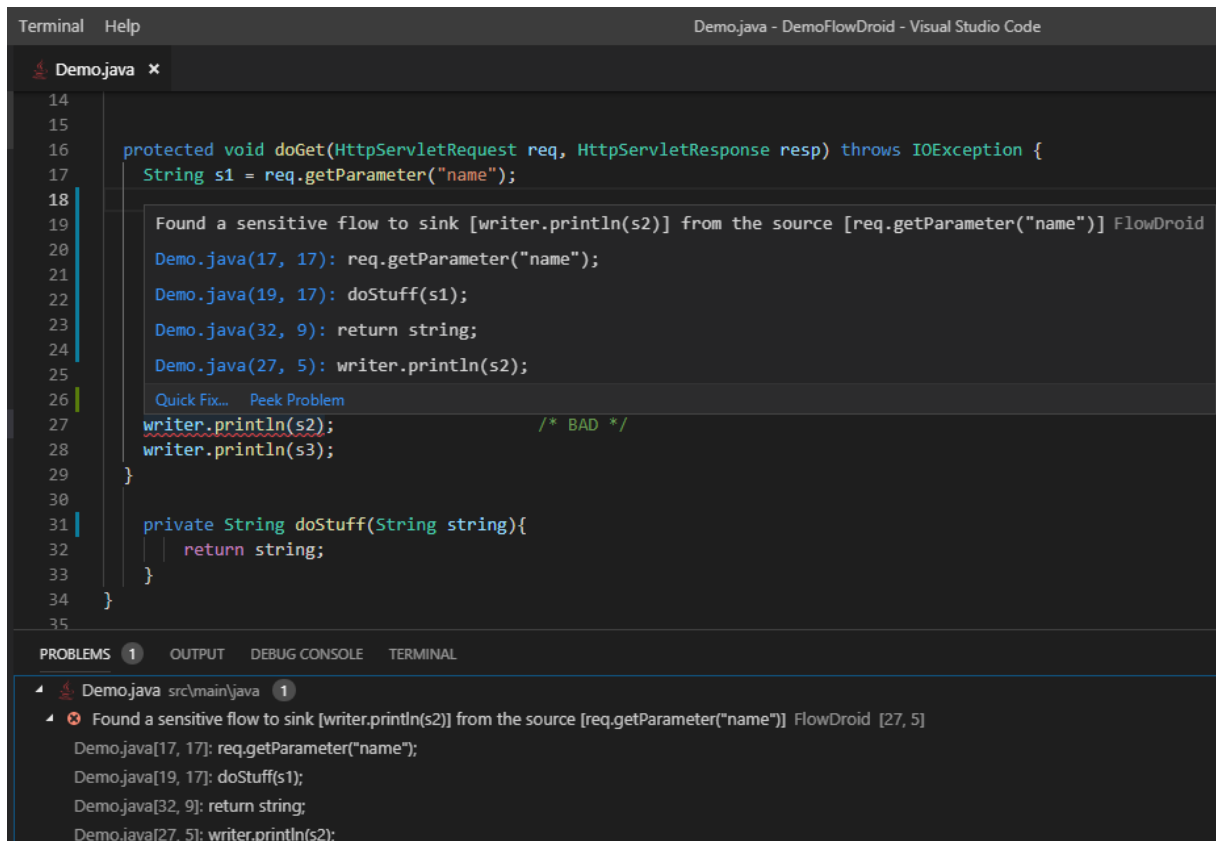


Figure 5.21: A taint flow reported by FlowDroid in Visual Studio Code.

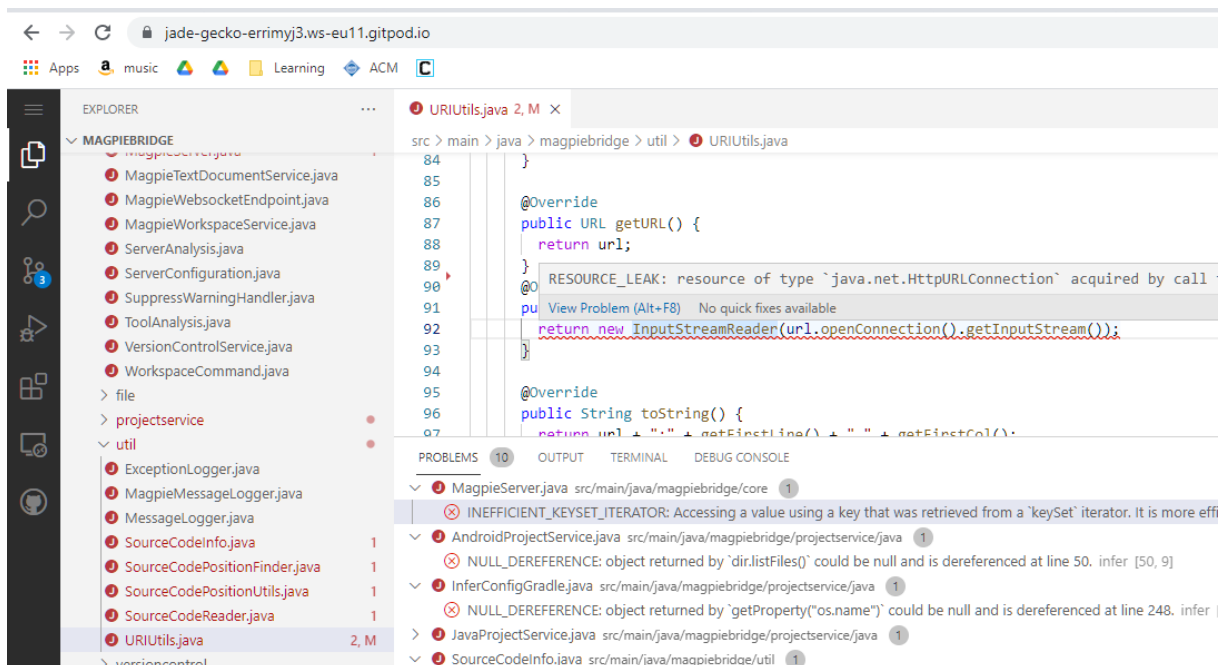


Figure 5.22: Infer warning in the web editor Gitpod.

some basic LSP features (i.e., `publishDiagnostics`, `hover`, `codeActions`, `showMessage` and `showMessageRequest`). This plugin also supports displaying a web page directly in IntelliJ via

an extended LSP notification `showHTML` tailored for MAGPIEBRIDGE. Another application of MAGPIEBRIDGE is to render the taint flows in IDEs with the TB-VIEWER for the TAINTBENCH work introduced Chapter 2. The latest integration with MAGPIEBRIDGE was the cloud-based Security Application Testing Tool (SAST)—CodeGuru Reviewer—from Amazon Web Services. Details about this integration will be introduced in Chapter 6.

5.5 Conclusion

The difficulty of integrating static tools into different IDEs and editors has caused little adoption of the tools by developers and researchers, and MAGPIEBRIDGE addresses this problem by providing a general approach to integrating static analyses into IDEs and editors. MAGPIEBRIDGE uses the increasingly popular Language Server Protocol and supports the rich analysis frameworks WALA and Soot. We demonstrate the generalizability of MAGPIEBRIDGE by using it to integrate multiple analyses into IDEs from both academia (CogniCrypt, Ariadne, COVA and FlowDroid) and industry (Facebook Infer and Amazon CodeGuru Reviewer). When we published MAGPIEBRIDGE in 2019, MAGPIEBRIDGE had some limitations (Section C.1.2) which are inherited from LSP, e. g., lack of customized user interfaces. To overcome these limitations, we proposed and extended MAGPIEBRIDGE to use web interfaces for building more complex user interfaces in this chapter. We believe this will lead to a promising future of MAGPIEBRIDGE. Hopefully, MAGPIEBRIDGE will open more and more doors for developers to access static analyses and foster developers to write more secure softwares.

IDE Support for Cloud-based SAST Tools

In previous chapters, we have been focused on the technical aspects of improving static taint analysis for the real world. In this chapter, we focus on the users of static analysis tools—software developers who write code and are expected to use analysis tools to find vulnerabilities in their code. As MAGPIEBRIDGE, introduced in the last chapter, makes IDE integration of static analyses much easier, the next step is to understand what specific features are expected by developers in terms of IDE integration of static analysis for security testing. In this chapter, we introduce a novel study [LSSB21a] we conducted with software developers at Amazon Web Services which targets the increasingly popular cloud-based Static Application Security Testing (SAST) tools, in which static taint analysis is often implemented.

Many companies are providing static analysis as a service, e.g., Coverity Scan [Syn08], Veracode [Ver06], Checkmarx [Che06] and LGTM [Git19b]. These tools fit well into CI/CD, since CI/CD allows time for deep static analyses (e.g., inter-procedural data-flow analysis) of the complete code base without taking up resources on a user’s machine. There are many benefits for performing static analysis tasks in the cloud. From the user’s perspective, it can provide central storage and tracking of analysis results. Cloud-based SAST tools usually offer hooks to integrate with popular CI/CD systems, such as GitHub Actions, Jenkins, or Travis CI and offer a browser-based dashboard for developers to manage findings. From the supplier’s perspective, parallelism in the cloud can improve the performance of these tools. As reported by Microsoft [KBL16], moving static analysis for Windows drivers to the cloud significantly reduced the analysis time spent for the `svb_bugbash` suite with 22.5x speedup. In addition, the cloud environment provides a central configuration of the analysis. SAST tool suppliers can tune the analysis engine to keep the false-positive rate low and update the analysis rule set without shipping constant updates to customers.

Despite all these benefits of doing static analysis in the cloud, multiple studies have shown that the ideal reporting location for static analysis is the developers’ IDE [CB16, DWA20]. So, there is a disconnect between the typical workflow, where SAST tools perform deep analysis in CI/CD, and developers’ expectation of interacting with these tools much earlier in the development lifecycle, directly from the IDE. Some cloud-based SAST tools provide an IDE integration to trigger an analysis manually from the IDE. E.g., Veracode Static for IDE [Ver17] allows developers to upload binaries to the cloud, start a scan on demand, and triage the findings from the IDE. Does this style of IDE integration meet developers’ expectation?

Integrating such a cloud-based SAST tool into the developers’ workflow comes with a set of challenges. In CI/CD, it is acceptable if an analysis spends several minutes computing in the cloud. How would such waiting time impact the user experience in the IDE? Another challenge

of designing such an IDE integration is dealing with the desynchronization between the code that is analyzed in the cloud and the code in developers’ IDE. While the analysis is running remotely, developers might write more code which makes the analysis results “out-of-date”. How should such results be displayed in the IDE? Especially for long-running analysis, are developers aware of the time to run it?

The main goal of our work in this chapter is to explore how IDE support for a purely cloud-based static analysis, that is typically used in CI/CD, should be designed to meet the expectations of developers. We identify the key design elements for such IDE support, and investigate whether it fits better into developers’ workflow in comparison to a web-based solution. Specifically, does it encourage more usage of the analysis, improve developers’ performance (i.e., less time to fix code) and perceived usability? To investigate whether an IDE solution can improve developers’ workflow, we conducted a user study (due to COVID-19, all interviews and usability tests were done remotely). The four stages of the user study were:

1. **User Interviews** (Section 6.2): We started by interviewing developers to understand their expectations of how cloud-based analyses should be triggered from an IDE, how the findings should be displayed there, and what UX features developers would like.
2. **Prototyping** (Section 6.3): Guided by the user interviews, we developed an IDE prototype for the existing tool CODEGURU REVIEWER [Ser20], using its infrastructure for CI/CD.
3. **Second-round Interviews** (Section 6.4): We presented the IDE prototype to the same developers of stage 1 to evaluate whether the design met their expectations. While developers were satisfied with most features implemented in the prototype, they found existing mechanisms for CI/CD, e.g., code uploading via Git, were cumbersome in the IDE.
4. **Usability Testing** (Section 6.5): Finally, we assessed our IDE prototype with a larger group of developers. In this test, we applied both quantitative and qualitative research methods to determine if the IDE solution was an improvement. We found that, using the IDE prototype, developers performed code scans three times more often than using the web-based solution. Our measurements also show a promising reduction in time for fixing code. However, bringing the findings of the tool into the IDE did not necessarily improve developers’ workflow. Specifically, they expected:
 - more education on capabilities and limitations of cloud-based SAST tools,
 - real-time feedback on analysis progress (e.g., progress bars),
 - quick validation of each fix, which implies incremental analysis on code changes,
 - seamless analysis of code (e.g., an analysis button without going through steps such as uploading it),
 - more interactive ways to suggest rescan, integrated into current workflows.

6.1 Background

Our study was conducted with developers at Amazon Web Services (AWS). We focused on the cloud-based SAST tool—CODEGURU REVIEWER, which is used as part of the CI/CD process inside AWS. At AWS, every commit to its code bases is required to go through a code review process first. Teams can configure different SAST tools, including CODEGURU REVIEWER, in their code review process. CODEGURU REVIEWER has an expected running time of under 10 minutes. Currently, CODEGURU REVIEWER integrated in the code review process only gets triggered to run (along with other quality assurance tools) when developers submit code changes

to a remote repository via an internal pull request tool. This internal tool pushes code to a detached branch and developers have to wait until CODEGURU REVIEWER and other quality assurance tools finish running. The findings of these tools are displayed in a web application. Developers have to address the findings before merging the code changes. Usually, developers address the findings together with comments from their teammates. However, developers told us they would like to get findings from CODEGURU REVIEWER before their code reviews, which is not the case in CI/CD, so our focus of this study is to explore how IDE support could be an improvement over the current flow and whether an IDE solution is better than a web solution. CODEGURU REVIEWER also provides a public API and a web interface to trigger a scan of a specific commit and fetch the findings. We used this API to build an IDE prototype.

6.2 User Interviews

First, we wanted to identify developers' expectations from IDE support for cloud-based SAST tools. With user interviews we aimed to answer the following two research questions:

RQ1: What do developers expect from IDE support for a cloud-based SAST tool?

RQ2: How could such IDE support fit into developers' workflow?

6.2.1 Methodology

Research Methods We wanted to understand how IDE support for a cloud-based SAST tool could fit better into developers' workflow in comparison to a web solution. Thus, we interviewed developers who already used a cloud-based SAST tool in practice. We conducted *semi-structured* interviews with developers using *contextual inquiry* [SN93]. Contextual inquiry allows us to understand how developers work with CODEGURU REVIEWER on a day-to-day basis. Before the interview, we sent each participant a link to one of their code reviews on which CODEGURU REVIEWER detected issues. During the interview, each participant was first asked to talk through their code review regarding CODEGURU REVIEWER's findings. Participants were asked to demonstrate how they fixed those issues in their IDEs together with the vulnerable code. Afterwards, while they had their IDEs opened, they were asked about their expectations on the IDE support and also to describe the features and demonstrate them in their IDEs if possible. The detailed questions list can be found in Section D.1. To differentiate from common static tools that run analysis on the same machine as the IDE, we explicitly told participants that the analysis is running in the cloud and that a scan takes minutes. Each interview lasted 45 minutes to one hour.

Participants We interviewed nine participants who were all *software development engineers* from different teams and countries within AWS. To ensure that participants were already familiar with SAST tools and willing to use them, we started by finding developers who were involved with code reviews on which CODEGURU REVIEWER had found issues (n=328) and then invited developers (n=252) who replied to the CODEGURU REVIEWER findings. All the interviewees had experience using static analysis tools (CODEGURU REVIEWER (9), FindBugs (7), CheckStyle (6), ESLint (1), SonarQube (1), Coverlay (1), IntelliJ built-in static tools (1)). In the following, we denote the nine developers with P1-9.

Data Collection All interviews were recorded and transcribed. They were carried out over video conferencing and all participants shared their screens during the interview so that their IDE activity could be captured.

Data Analysis The responses were analyzed using *thematic analysis*. We used both deductive (codes derived from the questions we prepared for the interviews) and inductive (codes derived from the responses) coding [FMC06]. The codebook contains 21 codes that were discussed and agreed upon by two researchers. The list of codes and their definitions can be found in Table D.1. The coding itself was first done by the researcher who conducted the interviews. To ensure reliability in the coding, a second researcher checked and discussed all coded data together with the first researcher. Adjustments were made where disagreement occurred. We applied an inductive approach to extract emerging themes which could be used to answer our research questions. We hit saturation [GBJ06, GSS68] after the 7th participant, whereby no new information was obtained.

6.2.2 Result of the User Interviews

The analysis produced five themes: Analysis Triggering Mechanism, Result Retrieval Mechanism, Result Display Mechanism, UX Features, and Workflow Integration. In the following, we will talk about how the first four themes of the IDE solution could fit into developers' workflow (the fifth theme).

6.2.2.1 Analysis Triggering Mechanism

In this section, we introduce how developers expect cloud-based SAST tools to be triggered from their IDEs, how code in their IDEs could be uploaded to the cloud and other expectations on this topic.

Ways of Triggering: The participants mentioned four ways the analysis should be triggered from the IDE: manual ($n=8/9$)¹, build ($n=6/9$), fully-automated ($n=4/9$), and semi-automated ($n=1/9$). The most mentioned way was *manual*. 8 participants said the analysis should be manually triggered by clicking a button in the IDE or by pressing a key shortcut. Participants would like control over the timing when their code is analyzed as P7 told us:

“I would want to control it by myself. If I would have a simple button to do the analysis, in preparation I will do the testing, before sending the code review I would upload the code to get the review by the machine.”

Most participants ($n=6/9$) also would like the analysis to be triggered in the project *build* process. Participants expected it to work this way, since they used other lightweight static analysis tools like FindBugs that can be configured as a build target.

Some participants ($n=4/9$) mentioned that the analysis should be triggered in a *fully-automated* way. The developers don't want to do anything else to trigger the analysis except writing the code. Real-time feedback from the analysis was expected. P9 explained us the reason:

“I don't want to introduce new behavior [...] If there is a button, during my normal flow, I'm very likely to forget that button.”

P7 mentioned a *semi-automated* way; he expected that the analysis can be configured to run when he presses Control + S to save a file.

Code Uploading: Since the analysis is running remotely, we interviewed the participants to understand how they expected the code to be uploaded to the cloud. The participants mentioned two ways: uploading with analysis triggering ($n=6/9$) and continuous uploading ($n=4/9$). The majority of participants ($n=6/9$) expected the code, especially the changes, to be uploaded when

¹We denote the numbers in fractions with the denominator being the sample size.

the user triggers the analysis. Their responses indicate that they expected the IDE support will do it for them. Some participants expected the code changes or diffs to be continuously uploaded in the background.

Developers have two mental models for how cloud-based static analyses should be triggered from the IDE—via active triggering (manual and build) or passive triggering (fully-automated and semi-automated). Developers with the first mental model would like to control the timing when they want feedback from the analysis. They actively search and fix issues once they are done with their coding task. The others prefer not thinking of the timing when they want feedback, they expect the IDE solution to provide feedback right after they make mistakes. Developers want to interact with the analysis as seamlessly as possible in a way that matches their individual workflows.

6.2.2.2 Result Retrieval Mechanism

All participants expected the IDE support to retrieve analysis results automatically from the cloud. They did not want to download an analysis report from the cloud and import it into their IDEs.

Timing: All participants expected the result to be retrieved to their IDEs directly after the analysis is completed. This can be in the build phase, if the build triggers the analysis; after the user manually triggered the analysis; or while coding if real-time analysis is possible. Three participants mentioned that it would be sufficient if the result could be retrieved before they published code reviews. Although we told participants that the analysis is as time-consuming as CODEGURU REVIEWER, their responses indicated that they were not aware of CODEGURU REVIEWER’s capabilities in terms of performance. They used phrases like “*after several seconds*” and “*at most 10 seconds*”. Some developers told us that they usually go on working on other tasks after submitting a code review and get notification emails when the analysis result is ready. They only check the result (of multiple tools) after their teammates review their code. This probably explains why some developers don’t have a sense of the analysis time of a specific tool.

Despite usage of cloud-based SAST tools in the CI/CD process, some developers were not aware of the capabilities and limitations of these tools, e.g., they were unaware how long CODEGURU REVIEWER takes to run.

Project Scope: The participants mentioned four project scopes: entire project (n=7/9), diffs (n=6/9), selection (n=4/9) and real-time changes (n=2/9). Scope has a twofold meaning: either they only want the code in respective project scope to be analyzed or they only want to see the result in the scope. Seven of them expected to see the result of the entire project they were working on. Five of the seven also wanted partial code to be analyzed or to only see the result in partial code. Partial code can be diffs or selection (e.g., selected packages, files or methods). We also noticed that the scope often comes with developer’s primary goal as P9 explained:

“If I just added some code, I am really interested in modifications I made. [...] If I am working on making the code better, I would want to see all the issues.”

Six participants mentioned that they would like to see the analysis result in their code changes (diffs) if they knew previously they passed all the analysis checks. Only two participants ex-

pressed that they would like to see analysis result in real-time changes, e.g., P6 said: *“If I write something, the plugin would tell me immediately: are you sure if you want to do this?”*

Which part of the analysis result to be displayed in the IDE depends on what developers’ primary goals are. If they are interested in improving overall code quality, showing findings in the entire project is preferred. If their primary goal is to implement a feature, they would like to see only findings that are context-close to the code they are working on (e.g., diffs).

6.2.2.3 Result Display Mechanism

When talking about how the analysis result should look in the IDE, many participants demonstrated their expectations in their IDEs with compiler errors. All participants suggested to *visually highlight* or *underline* the problematic code and display a *warning message* which explains the issue when the user *hovers* over the line of code. In addition, all participants believed the *severity* of the issue should be included in the warning, because it helps them to prioritize tasks. Some developers expected only critical issues to be shown and they must fail or block the build as P8 told us:

“If there is a failure [...] you have to fix it. However, if there is a warning [...], it is basically ignored. It is useless.”

Many participants (n=4/9) suggested to have a *list view of all issues* which allows *direct navigation to the line of code* when clicking on it. One of them expected to see issues grouped in packages. Three participants would have liked to have quick fixes attached to the warnings. P8 would like to *“have code snippet (vulnerable code)”* attached to the warnings such that he *“can easily see what the problem was”*.

Display of Invalid Result: Since the analysis is running in the cloud, by the time the analysis result is back to the user’s IDE, the user might have made more changes to the code. Thus, we interviewed developers to understand how they expected these invalid or old results to be handled.

Five of the nine participants expected to see only issues where the code is still in place, otherwise, *“it is misleading”* as P1 told us. Also they did not want to spend time on investigating issues which might not be there anymore due to code changes. P9 suggested:

“The plugin can see this suggestion was for this particular line or file, if this line or file changed, the suggestion will not be displayed.”

Also, two participants wanted to be informed about the code changes and a rescan (rerun the analysis) to be suggested by the IDE support, as P3 told us:

“Developers should be informed if they make changes to the code after they trigger the analysis, they would have to redo the analysis for the changes. They should be informed that the changes after triggering the analysis wouldn’t be considered. If we show out-of-date recommendations in the IDE, the developers should be informed that these recommendations are for the past and they might be not valid now.”

Developers expect to be warned in their code just like the way their IDEs usually show compiler errors. They do not want to spend time on issues which are out-of-date and expect the IDE solution to remind them to rescan.

6.2.2.4 UX Features

The most mentioned feature by participants was quick fix ($n=5/9$), as P7 describes how he expected it to work: “*you type Alt+Enter, it will offer you some fixes*”. Four participants would like to suppress warnings, either false positives or issues which are less severe. P1 expected it to be “*a list of previously suppressed warnings to re-enable them or something like checkboxes*”. P3 suggested to import a configuration file containing suppressed warnings as CheckStyle does. P9 would like to “*add a line of comment such as ‘disable CODEGURU REVIEWER’ to the code to suppress.*”

Participants also expected to customize the rule set of the analysis ($n=3/9$) and even the warning severities ($n=3/9$) to decide which warnings should be displayed. Both warning suppression and customization of rule set were mentioned as developers talked about features that would be beneficial for their teams.

Developers expect the IDE solution to not only pinpoint issues in their code, but also to fix them. They do not fully trust static analyses based on previous experience. They expect to suppress or prioritize warnings based on their own judgment.

6.3 Prototyping

Based on what we learned from the user interviews and the public API of CODEGURU REVIEWER [Ser21b], we developed an IDE prototype as a Visual Studio Code (VS Code) extension for CODEGURU REVIEWER. In the following, we introduce this prototype with respect to the themes derived from the interviews.

Analysis triggering, result retrieval and display The prototype provides a control panel for users to interact with CODEGURU REVIEWER in VS Code as shown in Figure 6.1. The “Show Recommendations” button allows users to view the recommendations (findings) provided

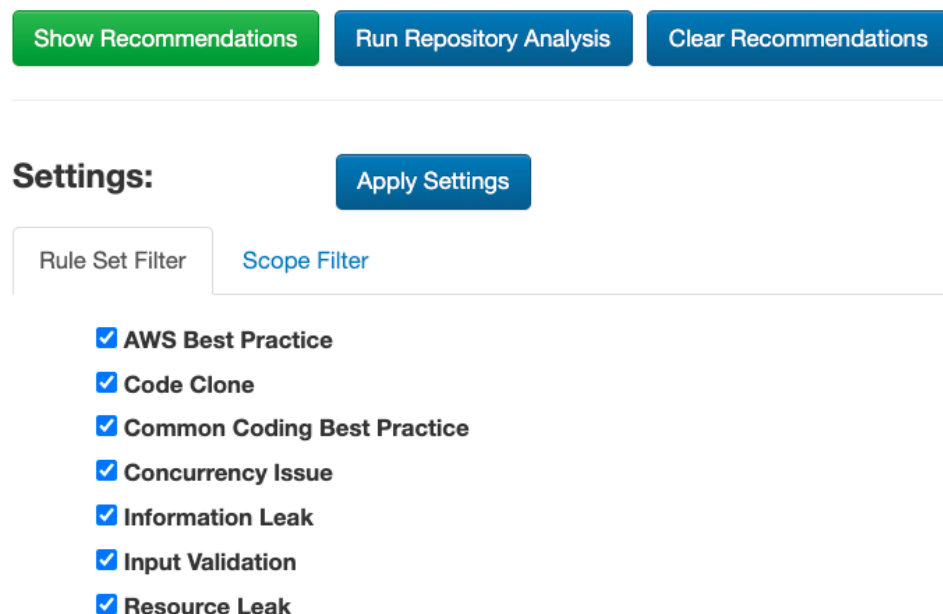


Figure 6.1: Control panel of our IDE prototype.

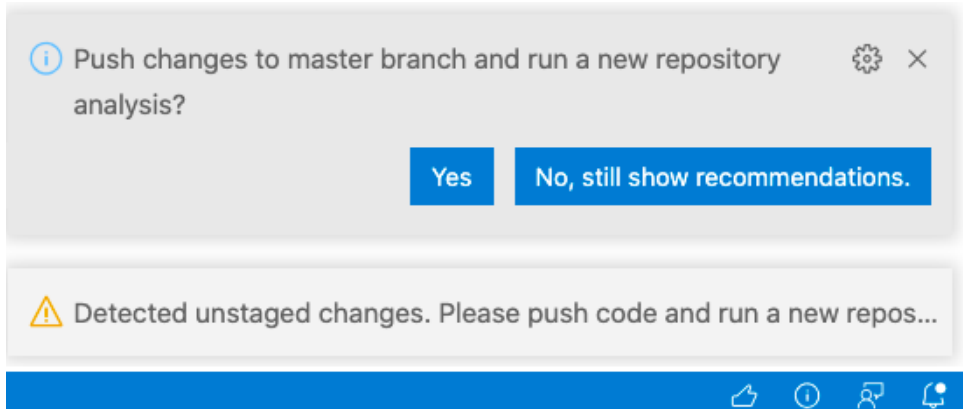


Figure 6.2: Reminder notifications asking users to rescan.

by CODEGURU REVIEWER directly in the IDE. The prototype automatically compares the local code version to the remote code version and fetches the result to the IDE if a matched analysis is found. If there are local code changes which haven't been uploaded to the remote repository, the prototype displays pop-up notifications to remind the user for a rescan as shown in Figure 6.2. The user can choose to display the result of the most recent analysis on the current branch with the “No, still show recommendations” button. Only recommendations in unchanged files will be displayed in the IDE, since developers told us they would not want to spend time on issues which might be invalid (see Section 6.2.2.3). If the user chooses to push code and rescan, a pop-up window will ask for a commit message and code changes will be pushed to the remote repository. After that, the prototype triggers a new analysis in the cloud. A notification will then be shown to tell the user about the estimated analysis time (5 to 10 minutes according to the official documentation) and the result will be automatically retrieved once the analysis is completed. The “Run Repository Analysis” button allows the user to run a new analysis on the remote repository. Similarly, it also reminds the user to push code if there are uncommitted code changes.

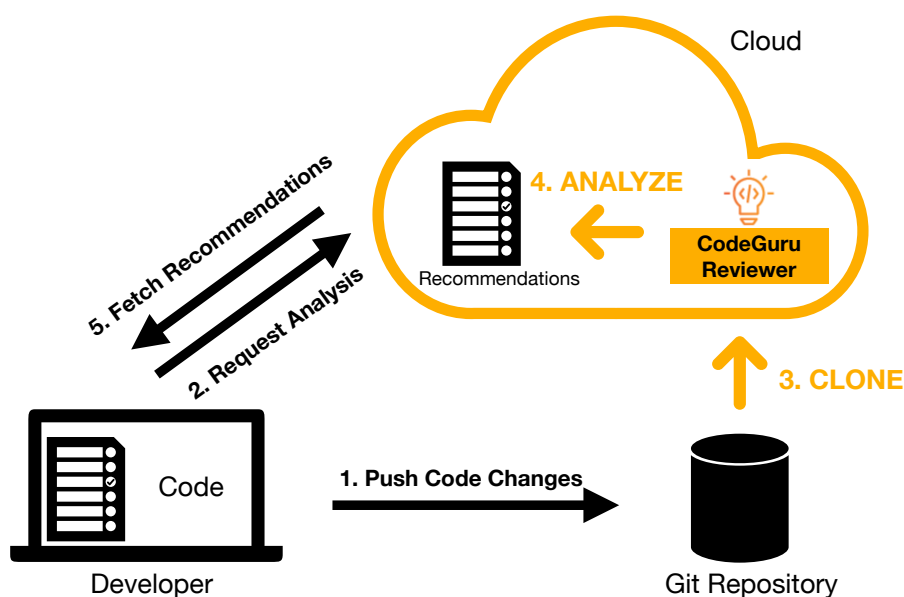



Figure 6.3: A typical workflow using our IDE prototype.

Figure 6.3 shows a typical workflow with five steps using our IDE prototype:

1. Developer modifies code and pushes changes to a remote Git repository.
2. Developer clicks the “Run Repository Analysis” button to request CODEGURU REVIEWER to run a new analysis on the Git repository.
3. CODEGURU REVIEWER receives the request and clones the Git repository.
4. CODEGURU REVIEWER analyzes the cloned repository and generates recommendations.
5. The IDE prototype automatically fetches the recommendations once CODEGURU REVIEWER finishes the analysis or the developer clicks the “Show Recommendations” button to fetch the recommendations to the IDE.

At step 4, while CODEGURU REVIEWER is running, the developer can continue working on the code or switch to other tasks.

Recommendations are displayed in a list view at the bottom of the IDE as shown in Figure 6.4. They are organized in groups according to the source files. From the interviews, we learned that some developers expect issues with fix suggestions to be prioritized. For recommendations with fix suggestions, we used the red marker  as an attentional cue that the warning was actionable to help developers quickly get to the code. It also indicates these issues are more severe and must be fixed. Although CODEGURU REVIEWER itself does not report the severity of an issue, our consultation with the engineers of CODEGURU REVIEWER revealed that when the tool provides fix suggestions then it’s typically for more severe issues. Yellow markers were used for all other findings. These two markers are the default markers provided by VS Code. We also included weakness types and code snippets in the recommendations, which were not provided by CODEGURU REVIEWER before. Clicking on a recommendation in the list navigates to the line of code. The code is highlighted and underlined as shown in Figure 6.5. Recommendations are also displayed in hover messages when the user hovers over the code. The hover message supports markdown, thus, the URLs to best practices in the recommendations are also clickable. Except quick fix, we addressed all expectations on result display from developers as introduced in Section 6.2.2.3. Although we would have liked to provide quick fixes, this feature was not supported by CODEGURU REVIEWER at the time and most issues cannot be easily fixed by adding/removing/replacing a code string.

Other UX features The prototype was built to support warning suppression and rule set customization, because these were the most wanted features by developers. Warning suppression is provided as a code action (automatic refactoring source code) attached to the recommendation

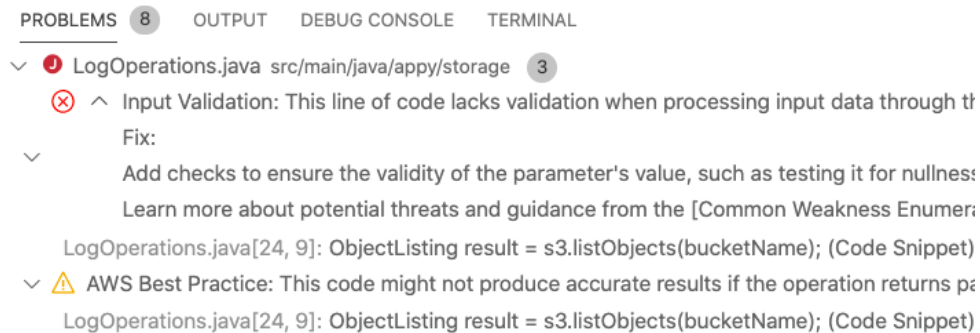


Figure 6.4: Recommendations are displayed in a list view.

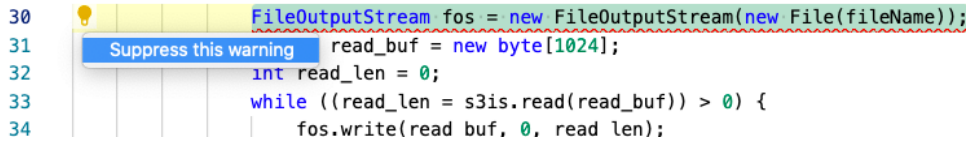


Figure 6.5: Warning suppression is shown as a code action.

as shown in Figure 6.5. When the user chooses to suppress a warning, the line of code will not be marked as an issue anymore and a special line code comment “SUPPRESS CODEGURU REVIEWER” is automatically added. Users can also manually add the suppression comment to code. No warning will be shown at lines with that comment.

Because we could not change CODEGURU REVIEWER to allow its rule set to be customized, we implemented a rule set filter to allow users to select/unselect the weakness types as shown under the settings section in Figure 6.1. Only recommendations with the selected weakness types would be displayed in the IDE. Developers also mentioned they would like to limit the display of findings to specific packages or classes, so the prototype provided a scope filter to select files or packages they were interested in. VS Code also has a built-in filter for the warning markers such that users can filter the recommendations based on the severities in the issue list. The configuration in the rule set filter and scope filter were locally stored by the prototype.

6.4 Second-round Interviews

After implementing the prototype, we re-invited the 9 developers we interviewed for a second-round interview. Five (P1, P4, P7, P8 and P9) of them accepted our invitations. These second-round interviews allowed us to fix minor issues prior to the usability test with a larger group of developers to ensure the usability feedback was focused on core issues rather than surface-level design concerns. The interviews were structured by demonstrating the features of the prototype addressing the topics from previous interviews. Each interview was about 30 minutes. After demonstrating a feature, we reminded the participant what she/he told us in the previous interview and asked how the feature differs from what she/he expected. We transcribed and coded the interviews to assess user sentiment (negative and positive) across the themes extracted from the first round of interviews.

Participants were all very positive about how analysis results were automatically retrieved and displayed in the IDE. They also liked the warning suppression and filters feature. Four participants didn’t expect the code needs to be pushed to the remote repository to trigger the analysis. P7 explained:

“Because for me it was like making dirty commits and I don’t like it.”

P8 gave us his reason:

“I am not using test branch at all, I am only using the mainline.”

He felt it was not helpful if he needs to setup a remote branch for his changes to run the analysis before sending a code review.

Although participants were critical about pushing code to the remote Git repository, we could not change the public API of CODEGURU REVIEWER to support other channels. Regarding old findings in changed files, P8 expected *“to see something even I change the file, unless I change exactly that line.”* After we explained that there might be case that an issue is fixed when new lines are added to the file, he responded with *“I know the system doesn’t know if it is fixed, but I would like to keep track of what was the issue.”*

However, the prototype reminds the user to rescan if there are local changes and the findings displayed in the IDE will not be removed unless the user clears them intentionally or requests for a new scan. Before we started the usability test with a larger group of developers, we tested the prototype with six developers and fixed bugs discovered in the interviews and during the test.

While code uploading mechanism via Git push is widely accepted in CI/CD integration, some developers found it cumbersome in the IDE due to different working habits, e.g., they only commit once per feature or do not use the Gitflow [Atl20] workflow.

6.5 Usability Testing

6.5.1 Methodology

Study Design To test if the IDE solution was an improvement over the web-based solution, we conducted a within-subjects usability test with developers. In comparison to between-subjects studies, it eliminates problems concerning individual differences [CGK12]. We wanted to compare the condition with the IDE prototype to the web-based solution of CODEGURU REVIEWER in AWS CONSOLE, where users can request analyses for their Git repositories and view recommendations in a web browser. For simplicity, we use *IDE* to represent our IDE prototype and *Web* to represent AWS CONSOLE in the following. We prepared two tasks, X and Y. In each task, participants were asked to fix issues in a prepared Java application either with help of the IDE prototype or AWS CONSOLE. All issues can be detected by the analysis engine of CODEGURU REVIEWER. The prepared applications use AWS services with the AWS SDK for Java [Ser21a] and each of them contains 8 issues with different weakness types. The test applications and issue list can be found in [LSSB21b]. Although the official documentation of CODEGURU REVIEWER gives 5-10 minutes as the average analysis time, for the two applications used in our study, the analysis time was just 3 minutes for each.

We applied 4 different treatments to participants as listed in Table 6.1. T1, T2 are the treatments in which participants first test *IDE*, while in C1, C2 participants started with *Web*. From the study in [DWA20], we learned that the typical length of a working session with a SAST tool of developers is 10-30 minutes (see Table 2 in [DWA20]. The authors refer to SAST tools with “dedicated tools”). Thus, we chose 30 minutes as the session length in our study. In each session, the participants were given maximally 30 minutes to solve the task. After each session, participants were asked to fill out an exit-survey (see the survey in Section D.3) and take an interview with us to examine how participants used the tools and how the tools affected their behaviors.

Table 6.1: Four treatments.

Treatment	Session 1		Session 2	
	System	Task	System	Task
T1 (n=8/32)	IDE	X	Web	Y
T2 (n=8/32)	IDE	Y	Web	X
C1 (n=8/32)	Web	X	IDE	Y
C2 (n=8/32)	Web	Y	IDE	X

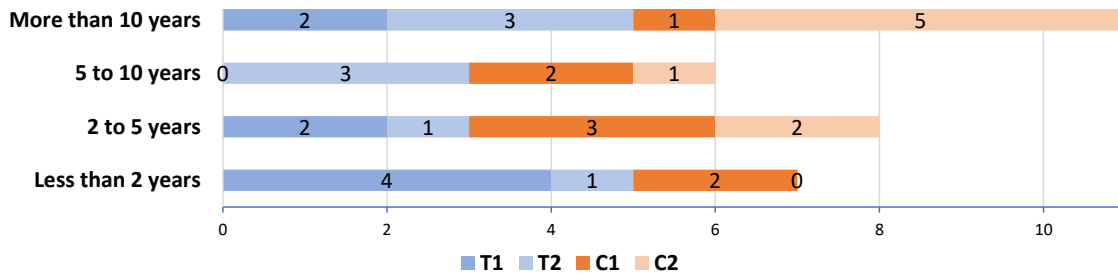


Figure 6.6: Years of professional coding experience in Java.

Participants We sent 1323 invitation emails to different mailing lists at AWS. In the email, we asked people to fill out a demographic survey if they accepted our invitations. We received 49 survey responses and, based on the responses, we removed participants who do not write code in Java. We chose 32 of them for our study. The participants were located in 9 different countries. 75% ($n=24/32$) of them never used CODEGURU REVIEWER before. Half ($n=16/32$) of them write code in VS Code and 78% ($n=25/32$) had written applications before with the AWS SDK. Figure 6.6 shows their professional coding experience in Java. More than half of them ($n=17/32$) have at least 5 years experience. We refer to these 32 participants with G1-32.

Study Setup The participants were assigned round-robin to one of the four treatments. In all treatments, participants were asked to perform the tasks in VS Code. After a brief introduction to the study, the participants were given the tasks in written form. We explicitly told the participants that CODEGURU REVIEWER is a cloud-based SAST tool and the expected analysis time to be a few minutes. We ran CODEGURU REVIEWER on the test application before each session and made sure that participants saw the CODEGURU REVIEWER’s findings displayed in either VS Code or in AWS CONSOLE before they started doing the task. We also provided participants user guides of the tested tool, i.e., IDE prototype or AWS CONSOLE. We told participants they could read them if they had questions. Participants were asked to solve the tasks independently without any help from us. They were also asked to give us clear signals as they started and finished the tasks to record the time.

After each session, participants were asked to fill out an exit-survey containing the 10 System Usability Scale (SUS) questions [Bro96]. In the survey, we also asked participants to evaluate the difficulty of the task using a Likert scale, select features of the tool they thought were helpful, estimate the number of issues they fixed, and answer some open-ended questions.

In each treatment group, we randomly chose 3 participants for monitoring. We asked these 12 participants to share their screen with us and think aloud as they were performing the tasks. The others were not monitored. Because not all participants could take interviews with us after their sessions due to scheduling constraints, we only interviewed 24 of them after they completed the exit-surveys. We asked them about the experience using the tool for the given task and whether the tool worked as they expected.

6.5.2 Quantitative Analysis

Developers tend to have different working habits when it comes to fixing issues in code as we learned from previous interviews. While some developers tend to validate the fix every time they address an issue, others fix all issues at a time and check them at once. We wanted to investigate how different solutions for a cloud-based SAST tool impact developers’ interaction with the tool. Our within-subjects design allows us to test the effect on individual participants. We also wanted to investigate whether our IDE prototype was sufficient to improve developers’

performance in code fixing and perceived usability of the tool. With the quantitative data we collected during the test, we answer the following questions:

RQ3: Does the IDE prototype encourage developers to interact more with the cloud-based SAST tool in comparison to the AWS CONSOLE?

RQ4: Do developers fix issues more efficiently with the IDE prototype in comparison to the AWS CONSOLE?

RQ5: Do developers perceive the IDE prototype to be more usable than the AWS CONSOLE?

Behavior (RQ3): Our alternative hypothesis for RQ3 is:

H1: Using the IDE prototype developers will rescan more frequently than using the AWS CONSOLE.

To test H1, we logged how many times each participant ran repository analysis successfully. Participants ran the analysis three times more often from the IDE (117 in total) than from the AWS CONSOLE (36 in total) as Figure 6.7 shows. We used the Shapiro-Wilk test [SW65] to test whether our data is normally distributed. Since the data doesn't distribute normally, we applied a two-tailed Wilcoxon signed-rank test [Woo08] with $\alpha = 0.05$, which is non-parametric and used for repeated measures. Although our hypothesis is one-sided, we used two-tailed testing to ensure the statistical power in both directions [RN10]. We present the results in Table 6.2 with participants grouped by their treatments.

The statistics in Table 6.2 (a) suggests that there is a significant difference ($W < W_{crit}$ and p -values are much smaller than 0.05) in number of scans. Using the IDE prototype developers performed analysis significantly more often than using the web-based solution, despite the fact that the analysis engine was the same and the tasks were similar. This indicates that the IDE solution fits better into developers' workflow. Developers wanted validation of their fixes more often when addressing static findings and the IDE prototype allowed them to run analysis easier.

Based on survey responses to the question “How did you know that you fixed the issues?”, participants were more confident about the number of fixed issues they estimated in the *IDE* condition. While 8 participants gave the answer saying that they didn't know in the *Web* condition, only 3 participants were not sure as they used the IDE prototype. Later in Section 6.5.3 we will introduce the opinions of developers and how they felt their workflows were impacted in the two conditions.

The IDE prototype also logged the total number of usage by all participants for each feature

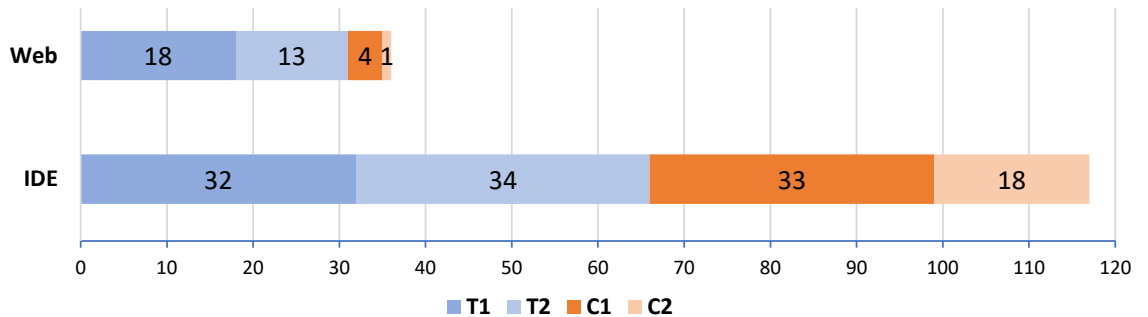


Figure 6.7: How often did participants rescan (run analysis)? The numbers shown in each bar are the total number of rescans performed by all participants in the associated treatment group in the condition.

Table 6.2: Results of two-tailed Wilcoxon signed-rank tests with $\alpha = 0.05$. N is sample size without ties. W_{crit} is the critical value for N and α . p-values < 0.05 are marked with *.

(a) Number of Scans				
Group	N	W-value	W_{crit} at N (p<0.05)	p-value
T1, T2	15	21	25	* 0.0264
C1, C2	12	7	13	* 0.0121
All	27	46	107	* 0.0006
(b) Average Time to Fix an Issue				
Group	N	W-value	W_{crit} at N (p<0.05)	p-value
T1, T2	14	46	21	0.682
C1, C2	14	50	21	0.873
All	28	194	116	0.841
(c) SUS-Score				
Group	N	W-value	W_{crit} at N (p<0.05)	p-value
T1, T2	14	27	21	0.110
C1, C2	14	46	21	0.682
All	28	160	116	0.327

Table 6.3: IDE feature usage.

Feature	#Usage	Feature	#Usage
Show Recommendations	318	Rule Set Filter	9
Run Repository Analysis	84	Warning Suppression	2
Clear Recommendations	20	Scope Filter	0

as shown in Table 6.3. Participants clicked the “Show Recommendations” button 318 times, which is 3.8 times than they clicked the “Run Repository Analysis” button. The huge difference between the usage of these two buttons suggested that participants didn’t choose to rescan but opted for displaying old findings as they were doing the tasks. While most participants actively clicked the “Run Repository Analysis” button to rescan (84 times), some participants took suggestions from the IDE prototype (33 times) and selected “Yes” in the pop-up notification to rescan as shown in Figure 6.2. Other features were rarely used. This is likely due to the short time planned for each session. We asked developers to select features they thought were useful in the survey, 13 of the 32 participants selected warning suppression, 8 for the rule set filter and 5 for the scope filter.

Performance (RQ4): Our alternative hypothesis is:

H2: Given an application containing issues that can be detected by CODEGURU REVIEWER, developers using the IDE prototype will be faster than using the AWS CONSOLE to fix an issue.

While 20 participants did not finish the task (timed out) in the session using the IDE prototype, the number with the AWS CONSOLE is 21. In three of the four groups (T2, C1 and C2), participants fixed more issues in the *IDE* than in the *Web* condition as shown in Figure 6.8 (An issue was considered fixed if CODEGURU REVIEWER didn’t report it again.). Surprisingly, participants fixed the same number of issues (157) in total when using the IDE prototype and the AWS CONSOLE. This is close to the number of issues participants estimated in the exit-survey, i.e., 165 in the *Web* and 166 in the *IDE* condition.

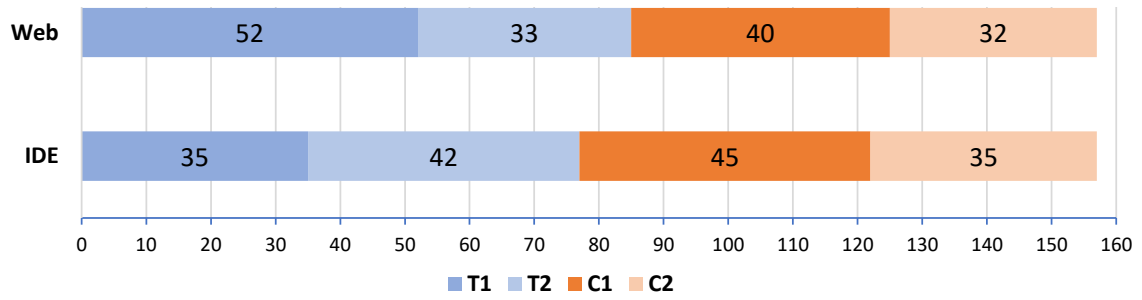
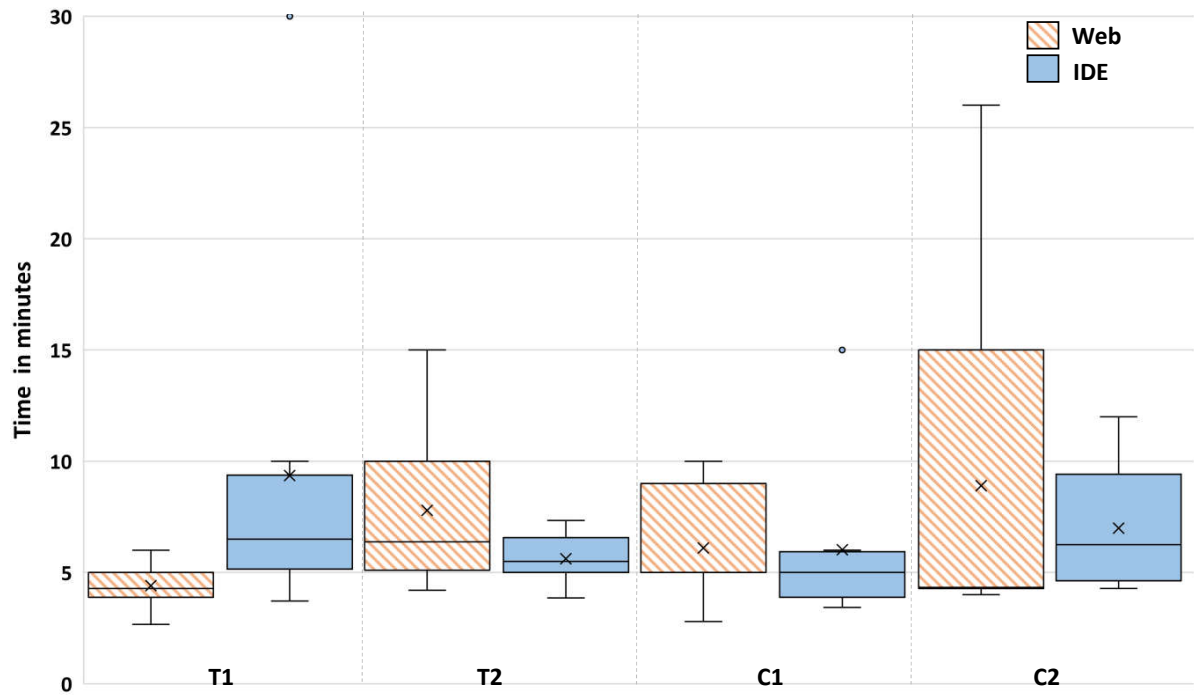


Figure 6.8: How many issues did participants fix?

Figure 6.9: Boxplots for average time participants used to fix an issue. Average values are marked with \times .

For each participant, we computed the average time used to fix an issue. Since one participant didn't fix any issue with AWS CONSOLE, we excluded this data from the test. Our test failed to reject the null-hypothesis as the statistics shown in Table 6.2 (b). However, as the boxplots in Figure 6.9 show, there is a promising time reduction (average and maximum values are lower) in the IDE condition among most groups of participants (T2, C1 and C2). In these groups, developers' performance was also more consistent in the *IDE* condition, since the boxes are smaller.

Usability (RQ5): Although we were comparing a research prototype to a commercial tool designed by professional UX designers, we formulated the alternative hypothesis in an optimistic way.

H3: Developers will rate the IDE prototype with higher SUS-scores.

We evaluated the survey responses to the 10 SUS questions and computed the SUS-scores. The higher the score is, the better the perceived usability. Again, we applied Wilcoxon signed-

rank test to the SUS-scores and the result is shown in Table 6.2 (c). There is no statistically significant difference between the two conditions regarding the SUS-scores.

However, we found out that participants tended to rate the tool they tested later with higher SUS-scores as the boxplots in Figure 6.10 show. We observed that many participants who started testing the IDE prototype at first (i.e., T1 and T2) were actually not expecting the analysis to be time-consuming. These participants were more confused as the IDE prototype did not display results immediately as they tried to run the analysis. Note that 75% ($n=24/32$) of the participants never used CODEGURU REVIEWER before. In contrast, participants who tested the *Web* condition first had a better sense of the asynchronous nature and the analysis time as they tested the IDE prototype later. We will discuss this further with the qualitative data in Section 6.5.3.

Although we pseudo-randomly sampled participants into groups to control for key factors, like experience with pre-existing tools, we had a few issues specifically affecting group T1. The data were affected by four participants in T1 who fixed more issues in the *Web* condition (less time per fix as shown in Figure 6.9) and rated extremely low SUS scores in the IDE condition. Participant G34 had more than 10 years of development experience and was very familiar with SAST tools used in CI (including CODEGURU REVIEWER), so he was extremely fast in the *Web* condition (only used half of the time as in the IDE) and fixed more issues. In the interviews, participants G20 and G21 mentioned that they did not understand there was no local analysis and got confused in the *IDE* condition, while in the *Web* condition it was straightforward for them and they could better focus on the task. G2 was observed spending time exploring the features of the IDE prototype rather than fixing issues. He gave a much lower SUS score for *IDE* than for *Web*.

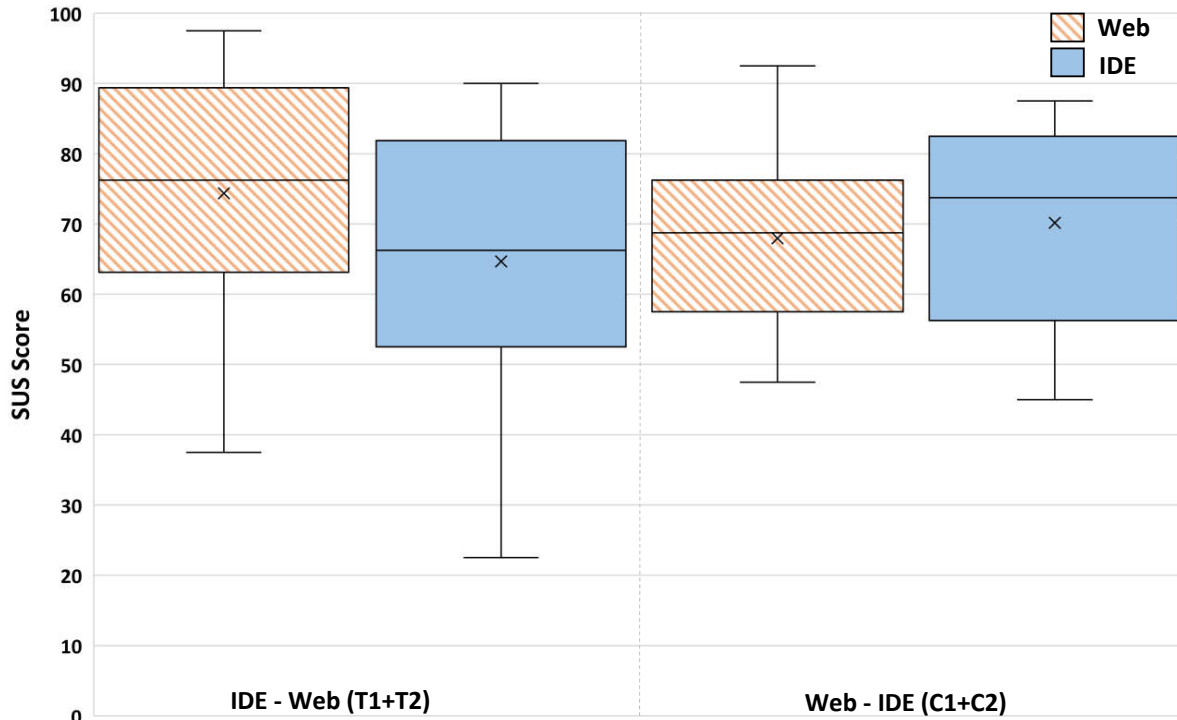


Figure 6.10: Boxplots for SUS-Scores.

Developers’ perceived usability of the IDE prototype was impacted by both their pre-existing expectations (on IDE integration of lightweight static analyses) and familiarity with the cloud-based SAST tool.

6.5.3 Qualitative Analysis

We coded the after-session interviews and survey responses to “*Tell us what needs to be improved*”. We reused the codes from previous interviews introduced in Section 6.2 and Section 6.4 and also added 12 new codes (see codes in Table D.1). We answer the following questions:

RQ6: How does the IDE prototype differ from the expectations of developers?

RQ7: How did developers think the IDE prototype impacted their workflows?

6.5.3.1 RQ6: How does the IDE prototype differ from the expectations of developers?

The positive things developers mentioned about the IDE prototype were similar to those we heard from previous interviews introduced in Section 6.4. Moreover, developers were very satisfied with the quality of the analysis result. They felt the recommendations were precise and informative. This is probably the reason why warning suppression was rarely used in the test. In the following, we focus on the major issues of the IDE prototype pinpointed by developers. We also identified some issues in the AWS CONSOLE and reported them to the engineering team.

Analysis Triggering Mechanism: The biggest issue mentioned by participants (n=10/32) was uploading code via Git. Some participants felt uncomfortable to push code without a code review. Others felt less confident to push code without testing it locally. Although the test applications provide unit tests, these participants didn’t run them at first, instead, they expected to get feedback before testing.

Many participants (n=8/32) were expecting fast local validation of fixes when they clicked on “Run Repository Analysis” button, as G8 said:

“when I modified the code, it was strange to see the squiggly lines here and there saying there was an error.”

Although there were pop-up notifications shown in the IDE mentioning the analysis time and suggesting rescan, some participants seemed to pay little attention to those pop-ups. These participants are mostly from the groups who tested the IDE prototype at first.

Developers expected IDE support to allow usage of cloud-based SAST tools before their code goes into the next phases (test, CI/CD) in the development lifecycle. They wanted the analysis of code without going through steps such as uploading it.

Result Retrieval and Display Mechanisms: Most problems came from the analysis time and poor indication of the analysis progress in the IDE. CODEGURU REVIEWER needs about 3 minutes for reanalyzing each test application, which is shorter than the official average analysis time (5-10 minutes) given by CODEGURU REVIEWER. Still, it was “*painfully slow to the point I was worried the plugin was unresponsive*” as G15 told us. As mentioned above, pop-ups were not sufficient for informing participants about the analysis time. As G8 told us, although he read the pop-ups he thought it was a “*generic message*” and “*didn’t consider it would be the exact time*”. More than one third of the participants (n=13/32) we interviewed expected to see a progress bar or a dynamic display of analysis status in the IDE. They wanted to “*see what it*

(the service) is currently doing, so not just all of a sudden, a pop-up comes up saying the result is ready.” This is also the pain point in the AWS CONSOLE, since there is no progress indication either.

Using the IDE prototype, many participants told us they kept working after started a new scan. Most confusion came from displaying old findings. Some participants didn’t rescan, but chose to see them in the IDE and clicked the “Show Recommendations” button multiple times as they were fixing the issues. They thought this button performed local validation of a fix, although this button actually checks code version and only displays the findings in unchanged files. This led to “*the problem (finding) disappeared and I had trouble to get them again.*” as G12 described. This can probably be resolved by reading the user manual or a mechanism that checks local changes better. Other participants felt “*the outdated messages are annoying, I’d rather have them disappear when a line gets updated, to invalidate them.*”, since they were expecting the prototype to be as reactive as lightweight static analysis tools.

Pop-ups were less effective in educating developers on mechanisms built in the IDE solution. If the analysis is not returning results instantaneously or developers’ activities in the IDE are not blocked, the display of old findings and unpredictable waiting time for new findings are “deal breakers”. This suggests developers need clear visual cues (e.g., progress bars) to understand when the analysis is running and if findings are outdated.

UX Features: Five participants mentioned that they were not aware that the analysis was running remotely, thus, they were expecting real-time feedback due to previous experience. As G20 told us that he didn’t think about the analysis was running in the cloud. He said:

“My initial perception is that this is going to be some sort of static analysis tool. I was expecting it to be a similar experience to other static analysis tools I used before.”

These five participants suggested more obvious visual indication for the asynchronous nature of the IDE solution.

Another feature that was missed by participants for both the IDE prototype and AWS CONSOLE is a way to keep track of addressed issues. G9 told us he was using the rating buttons (thumb up and down symbol) in the AWS CONSOLE “*as almost a checklist. I know which I have done because I marked them helpful.*” Although quick fix was the most mentioned feature in our first user interviews introduced in Section 6.2, only 3 participants mentioned it this time. Developers understood that it was harder to provide quick fixes for more complex issues detected by a SAST tool than a linter. Two participants suggested that the IDE prototype should forbid or auto-cancel multiple scans on the same commit, since “*it’s a wasted action*”.

Even though most developers were aware that the cloud-based SAST tool performs a deeper analysis and acknowledged that the analysis takes longer, they still complained about the waiting time and that findings and code ran out of sync. This suggests that using the same visual components for the cloud-based analysis that are also used by lightweight local static analyzers (e.g., problem-list windows, error markers on code) may create unrealistic expectations about the behavior of the tool.

6.5.3.2 RQ7: How did developers think the IDE prototype impacted their workflows?

As we show for RQ3 the IDE prototype impacts developers’ behavior in fixing code, the qualitative data also indicates the same. Although developers interacted with the same analysis engine,

they approached the tasks differently in the two conditions. A participant used a metaphor to express the different feelings:

“The thing in the AWS CONSOLE felt like integration test. Having it in the IDE was like unit test.”

In the AWS CONSOLE, because it is in a browser, participants felt a disconnect “*between running the analysis and editing*”. A few participants perceived the list of findings as a task list as G9 told us:

“I saw the task list and I went to work on that code. It just didn’t click for me that I can go back to the AWS CONSOLE and rerun the analysis.”

They felt that they were supposed to pick a workflow in which they would only rescan once they addressed all issues. Not seeing the result immediately was less frustrating, because it was clear that there was no synchronization. While some participants chose to address all issues at once, others felt their workflows were paused in the *Web* condition as G29 told us:

“I was somehow encouraged to wait to see what happens. I felt that if go back working, I would not be aware when the execution finishes.”

Using the IDE prototype, without switching between the browser and the IDE, some developers rescanned more often. A participant who addressed all issues at once in the *Web* condition said:

“(In the IDE) It was like I saw the thing turned red, I fixed it, kept iterating on it until error-free, then I move on to the next one. [...] So I was expecting some feedback. I changed something, hit on ‘Run Repository Analysis’[...]”

Another participant told us he felt encouraged to change code and rescan even before the previous scan is completed such that he could work more efficiently.

Using the IDE prototype, developers wanted to validate their fixes more often and felt encouraged not to wait for the analysis execution, but continue working on fixing other issues.

Despite of all the problems identified in the IDE prototype, many participants expressed that they would prefer the IDE support to interact with cloud-based SAST tools. G21 told us:

“I would prefer IDE, because less time wasted having to go through other screens. I can push code, and let analysis run on branch, while continuing workflow in the IDE. Less context switching.”

G1 also preferred the prototype but wished it was more interactive:

“It runs analysis on the file you are working on and tells you if you fixed it correctly or not.”

Also G22 said:

“The IDE integration is the better path, because it’s closer to the activity being performed: writing code.”

Despite the delay of the analysis, he would “*much rather see a list of suggested problems/fixes in my editor than changing screens back-and-forth.*”

6.6 Threats To Validity

External Validity: We conducted our study with developers of a single company. Among them, only a few participants were female. This may lead to limited generalizability of our findings to the whole developer community. However, the participants were located in 9 different countries, have years of professional experience, and work on different kinds of products. Furthermore, we only studied the effect of IDE integration for one cloud-based SAST tool and one IDE. As we demonstrated, the response time of the SAST tool is a major factor, so our findings cannot be generalized to tools that are significantly faster or slower. A follow-up study can determine this effect by artificially introducing delays when retrieving findings. Our prototype uses MAGPIEBRIDGE [LDB19] (see Chapter 5) which is based on language server protocol [Mic16a] that integrates with most modern IDEs, so we expect the impact of the IDE choice to be minimal, but developers’ familiarity with an IDE could affect the study.

We only compared the IDE solution to the web-based solution of CODEGURU REVIEWER regarding repository analysis. We did not consider the impact on development lifecycle management with the issue board in the web-based solution. It is likely that project managers and team leaders would have a higher preference for the web-based solution. Moreover, we did not consider cost, security, and trust. Some participants were critical about pushing code for a rescan and proposed to hide the action, however, real customers of CODEGURU REVIEWER might not want their code to be uploaded silently due to security concerns.

Internal Validity: The first threat is the session time. Some participants told us they felt stressed and they did not have enough time to fix all issues. While the available time limited the performance of some participants, it also simulated the pressure of software development on tight deadlines, e.g. before releases. We also observed that some participants were less motivated to fix code, but more interested in playing with the features of the tools.

Another threat was attrition. We had four developers (who did the first interviews with us) not attend the second-round interviews and two (their data are not included in this work) did not participate in the usability test. We are aware that our findings are likely to be based on a biased sample of developers who have higher motivation to use static analysis tools or cloud services. Moreover, the tasks in the usability test were artificial. Due to unfamiliarity with the projects or the used Java libraries, some participants may have performed worse than in real development situations. However, we only selected developers who have professional experience in Java and the majority ($n=25/32$) of them used AWS SDK before.

Regarding the impact of developers’ familiarity with the IDE, we applied Wilcoxon-singed rank test to the sample grouped by tasks and grouped by the experience with VS Code, the result indicated there was no significant difference between the groups. Regarding the issues detected by CODEGURU REVIEWER, they were all true positives.

Lastly, the IDE prototype was not designed by professional UX designers, but researchers. It is likely that developers would perceive a significant improvement of the usability using a professionally designed IDE solution in comparison to the web-based solution. Although we were comparing a prototype to a web application with real customers, our result indicates that the prototype is not worse.

6.7 Related Work

The usability of static analysis tools has been studied by many researchers. Johnson et al. interviewed experienced developers to understand why developers were not widely using static analysis tools [JSMB13]. They found out that false positives and bad warnings were the major

reasons for developers’ dissatisfaction. Christakis and Bird surveyed developers at Microsoft to understand what developers want and need from static analysis tools [CB16]. Their study shows that developers would like static analysis tools to detect more critical issues for them such as security or concurrency issues and display the findings directly in their IDEs. Beller et al. studied the usage of static analysis tools on open source projects [BBMZ16]. They found out that most open source developers only use static tools sporadically and they need to be made aware of the benefits of using these tools. Vassallo et al. studied developers’ behavior using static analysis tools over different development stages [VPP⁺18]. They found out that severity is the most important factor for developers in prioritizing issues to fix, which was confirmed in our study. Steidl et al. suggested to prioritize issues that are easy to refactor [SE14]. Their study indicates prioritizing by low refactoring costs matches greatly the developers’ opinions. In our study, we also heard expectation of such prioritization mechanism from some developers. A more recent study from Nguyen Quang Do et al. took a user-centric approach to understand why developers use static analysis tools and which decision they make when using these tools [DWA20]. According to their study with developers at Software AG, IDEs are still the ideal reporting locations wanted by developers. However, we observed that there exists a disconnect between the typical usage of cloud-based SAST tools in CI/CD and developers’ wish to interact with them earlier in the lifecycle, in their IDEs. Our work focuses on exploring how IDE support for cloud-based SAST tools that are typically used in CI/CD should be designed. We approached the exploration from developers’ perspective with a user study. We found out that developers expected more than just seeing the findings of these tools in their IDEs.

In recent years, we see an increased interest in studies that apply static analysis tools at scale [Bol16, KBL16, ZSO⁺17, VPBG18, IMW19]. Facebook’s static analyzer *infer* has detected over 100,000 issues that have been resolved by Facebook’s developers since 2014 [DFLO19]. As reported by Google [SAE⁺18], their static analysis platform *Tricorder* could analyze 50,000 code review changes per day. More than 5,000 issues per day were tagged to be fixed by developers. These studies discuss tools that are integrated in the code review process. Our work builds on the results of these papers and asks the question how we can give developers access to cloud-based SAST tools directly through their IDEs, and if this improves developers’ workflows. We share challenges and lessons learned in the exploration that can be beneficial for suppliers that wish to build such IDE support.

Many researchers have studied the impact of cloud services on the user experience [WTK⁺08, UKJS10]. Kaisa Väänänen-Vainio-Mattila et al. studied the user perceptions of *Wow*—a positive user experience when using cloud services [VPP⁺11]. They proposed a few design implications for achieving *Wow* such as pushing dynamic features to keep the user stimulated. Tang et al. interviewed users of file synchronizing and sharing services to understand the cloud-based user experience [TBM13]. They found out that users’ understanding and usage of cloud functionalities are limited by their existing practices. Similarly, we also learned that developers’ expectations of IDE support for cloud-based SAST tools were affected by their awareness of the limitations of these tools, and their previous experience with lightweight analysis tools. Developers’ overexpectations hugely impacted the perceived usability when interacting with our IDE prototype. Through a usability test, we identified important design elements and mechanisms required for a better tool support.

6.8 Conclusion

To investigate how IDE support for cloud-based SAST tools should be designed, we conducted a multiple-staged user study. We first interviewed developers at AWS to understand their expectations. Developers’ feedback indicates that they expected the IDE support for cloud-

based analyses to behave similar to the lightweight static analysis tools they already use in their daily work. Their responses also indicate that they have limited understanding of the capabilities and limitations of SAST tools. Guided by the user interviews, we developed an IDE prototype that was positively confirmed by the same group of developers. We tested this IDE prototype on 32 developers. This usability test showed that allowing developers to interact with a cloud-based SAST tool through their IDE significantly increased their interaction with the tool, i.e., they ran the analysis much more frequently than using the web-based solution. This might impact the code quality in a long time span. Moreover, we found promising reduction in fix time even in our small-size study. A larger longitudinal study on this impact should be conducted in the future. However, we also found out that reusing the same visual components for the cloud-based analysis that are also used by lightweight static analyzers (e.g., problem-list windows, error markers on code) created confusion and that developers need clear visual cues to understand the asynchronous nature of cloud-based analyses.

Conclusion and Future Work

Designing static analysis tools for the real-world is challenging. Analysis ideally should be sound, precise and scale to large applications. Soundness requires the analysis model all possible executions of the program under analysis and not miss issues. To achieve this, analyses are often over-approximated, which means analyses might model program behaviors that could never happen. In contrast, precision requires analyses to avoid modeling of such unrealizable behaviors and produce false positives as less as possible. However, to make analysis scale, people often opt to lower the precision since it is widely perceived that a more precise analysis often needs more time to run.

Finding the best trade-off between precision, soundness, and scalability of a static taint analysis requires good benchmarks to be tested on. However, common benchmarks for static taint analysis are micro benchmarks, which are not representative for real-world applications in terms of size and complexity as we see in Chapter 2. Analyses that work well on micro benchmarks are often less effective on real-world apps. To fill this gap, we contributed TAINTBENCH—the first real-world malware benchmark suite with a well-documented ground truth for Android taint analysis in Chapter 2. TAINTBENCH allows us to evaluate state-of-the-art Android taint analyses tools and reveals insights of these tools that have not been gained with micro benchmarks. We found evaluated tools have very low recall on TAINTBENCH and new releases of these tools often performed worse than old ones. Although FLOWDROID produces the best results on TAINTBENCH, our in-depth investigation reveals that incomplete call graph modeling is one of the main reasons it failed to detect 35% of the malicious taint flows in TAINTBENCH. This finding leads to our second contribution of this thesis in Chapter 3—GENCG, which is an approach to build more complete call graphs. We show our GENCG approach is general in the sense that it is not limited to any framework or any specific analysis tool. Experiments on both Android and Spring frameworks with a client taint analysis show our approach enables detection of more real-world taint flows without introducing much noise (false positives).

In terms of improving precision of static taint analyses, the third contribution of this thesis addresses path-sensitivity, which is one of the least considered sensitivities by existing taint analysis approaches. Chapter 4 presents an approach COVA that computes partial path constraints that can be used to enhance results produced by a client analysis which is not path-sensitive. Client analyses can use COVA to eliminate false positives due to unrealizable path constraints, as well as explain under which conditions an issue might happen. Using COVA we conducted a qualitative study of taint flows from a large set of real-world android apps from popular Android app markets. The qualitative data we collected in this study explains circumstances under which taint flows may actually occur. This is a start toward finding concrete executions to validate

static findings. Our proof-of-concept extension of COVA demonstrates the feasibility of using COVA to generate concrete user inputs to drive execution to desired program points.

Making sound, precise and scalable static analysis is unfortunately not enough for the real world. Often, software developers are confronted with the task of learning and applying a large number of technologies in far too short a time. Using these technologies securely represents a big hurdle. Often, security is not explicitly tested. Despite the fact that security analyses like static taint analysis can help here, current analysis tools are unfortunately not well received by developers. There are only few analyses integrated into tools that are commonly used by developers, i. e., in IDEs. Many analyses are produced in academia and stay in academia, which is a huge waste of efforts. Even with the existence of IDE integration, most analyses target only one specific IDE due to the sheer amount of engineering effort involved. To make static analysis more accessible for developers, researchers need ways to bring them into IDEs more easily. Our fourth contribution in Chapter 5 is MAGPIEBRIDGE, which provides a solution to integrate static analyses into multiple IDEs with less effort in comparison to traditional approaches. Although the focus of this thesis is static taint analysis, we designed MAGPIEBRIDGE to be generally applicable and demonstrate its generalizability with multiple analyses from both academia and industry.

As MAGPIEBRIDGE builds a bridge between static analyses and IDEs, the next step is to understand what specific wishes developers have for IDE integration of static analyses. In collaboration with scientists from Amazon Web Services, the last contribution of this thesis in Chapter 6 explores how IDE support for a purely cloud-based static analysis, as is typically used in CI/CD, should be designed to meet the expectations of developers. We wanted to enable usage of static analyses earlier in the software development lifecycle, i. e., from testing/deployment to development. We conducted a multiple-staged user study with software developers at Amazon Web Services. In this study, we built a prototype of the IDE support for a cloud-based SAST tool—Amazon CodeGuru Reviewer and evaluated this prototype with software developers with a usability test. The usability test showed that allowing developers to interact with a SAST tool through their IDE significantly increased their interactions with the tool, which might positively impact the code quality in a long time span. The challenges and lessons learned in this study opens avenues to future research directions. While developers expect quick validation of code fixes in their IDEs, existing SAST tools are usually not able to give real-time feedback. This implies the need of incremental analysis on code changes.

In conclusion, this thesis tackles three existing problems (see Chapter 1) to improve real-world applicability of static taint analysis. We hope that the presented benchmarks, approaches, their implementations and shared insights can help static taint analysis designers to create better analyses, and foster the adaption of static taint analyses by software developers in building more secure software. We end this thesis by discussing a few research directions for the future:

- In benchmarking Android taint analysis tools with TAINTBENCH, we saw taint flows in real-world applications are often consists of multiple sub-flows in different programming languages (Java, JavaScript, C/C++). Combining and composing taint analysis approaches for different languages to detect such cross-language taint flows is an emerging research direction.
- With COVA we discovered the usefulness of path constraints in refining Android taint analysis results. However, we still did not fully solve the scalability problem in obtaining path-sensitivity. A possible improvement is to make the analysis in COVA on demand. As we demonstrate with the extended COVA, it is promising to use it for target testing or guided fuzzing for Android applications. Currently, there are only few approaches for Android applications [RATP17, AYS⁺21] and it is even less explored for Java web

applications [MAHF18].

- Our GENCG approach enables static analysis to analyze more reachable code, while it is still limited by not modeling reflections through configuration files that are used by framework-based applications. We believe incorporating actual execution traces into our approach can help to produce more sound and precise call graphs.
- When interacting with security analysis tools from an IDE, developers expect them to be as responsive as possible. They want to get analysis results instantly for code they are editing. However, making whole-program analysis to terminate in a few seconds is still very challenging for modern softwares. In recent years, there have been a few approaches doing incremental analysis [AB14, SEB21, SEV20, SCS21], however, analysis tools in industry rarely do so [EN08]. It is an interesting research question whether the proposed approaches are suitable for industry code and how efficient they would be.

Bibliography

- [AB14] Steven Arzt and Eric Bodden. Reviser: efficiently updating ide-/ifds-based data-flow analyses in response to incremental program changes. In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 288–298. ACM, 2014.
- [AB16] Steven Arzt and Eric Bodden. Stubdroid: automatic inference of precise data-flow summaries for the android framework. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 725–735. ACM, 2016.
- [ABKT16] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, pages 468–471, 2016.
- [AFK⁺20] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. Static analysis of java enterprise applications: frameworks and caches, the elephants in the room. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 794–807. ACM, 2020.
- [AKG⁺15] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 426–436, 2015.
- [AL13] Karim Ali and Ondrej Lhoták. Averroes: Whole-program analysis without the whole program. In Giuseppe Castagna, editor, *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, volume 7920 of *Lecture Notes in Computer Science*, pages 378–400. Springer, 2013.
- [Ama17] Amandroid. https://mvnrepository.com/artifact/com.github.arguslab/amandroid_2.12/3.1.2, November 2017. Accessed: 2021-08-03.
- [Ama18] Amandroid*. https://mvnrepository.com/artifact/com.github.arguslab/amandroid_2.12/3.2.0, December 2018. Accessed: 2021-08-03.

- [And11] Androguard. <https://github.com/androguard/androguard>, 2011. Accessed: 2021-08-03.
- [ANHY12] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 59, 2012.
- [ARF⁺14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Ochteau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of PLDI*. ACM, 2014.
- [ARHB15] Steven Arzt, Siegfried Rasthofer, Robert Hahn, and Eric Bodden. Using targeted symbolic execution for reducing false-positives in dataflow analysis. In Anders Møller and Mayur Naik, editors, *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2015, Portland, OR, USA, June 15 - 17, 2015*, pages 1–6. ACM, 2015.
- [Arz16] Steven Arzt. *Static Data Flow Analysis for Android Applications*. PhD thesis, Technische Universität Darmstadt, Dec 2016.
- [ASH⁺14] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. DREBIN: effective and explainable detection of android malware in your pocket. In *Proceedings of the 21st NDSS*. The Internet Society, 2014.
- [Atl20] Atlassian. Gitflow workflow, 2020. Accessed: 2021-08-03.
- [AYS⁺21] Yousra Aafer, Wei You, Yi Sun, Yu Shi, Xiangyu Zhang, and Heng Yin. Android smarttvs vulnerability discovery via log-guided fuzzing. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.
- [Bae21] Baeldung. Guide to spring autowired. <https://www.baeldung.com/spring-autowire>, 2021. Accessed: 2021-08-03.
- [BBMZ16] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 470–481. IEEE Computer Society, 2016.
- [BGC15] Sam Blackshear, Alexandra Gendreau, and Bor-Yuh Evan Chang. Droidel: a general approach to android framework modeling. In Anders Møller and Mayur Naik, editors, *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2015, Portland, OR, USA, June 15 - 17, 2015*, pages 19–25. ACM, 2015.
- [BGH⁺06] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21th OOPSLA*. ACM, 2006.

- [BKKL⁺20] Manuel Benz, Erik Krogh Kristensen, Linghui Luo, Nataniel P. Borges Jr., Eric Bodden, and Andreas Zeller. Heaps’n leaks: How heap snapshots improve android taint analysis. In *Proceedings of the 42nd International Conference on Software Engineering*, 2020.
- [BLYW17] Amiangshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of AsiaCCS*. ACM, 2017.
- [Bol16] Claude Bolduc. Lessons learned: Using a static analysis tool within a continuous integration system. In *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 37–40. IEEE, 2016.
- [Bro96] John Brooke. Sus: a “quick and dirty” usability. *Usability evaluation in industry*, page 189, 1996.
- [BS09] Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: better together. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, pages 1–12, 2009.
- [BS18] Zohreh Bohluli and Hamid Reza Shahriari. Detecting privacy leaks in android apps using inter-component information flow control analysis. In *Proceedings of the 15th ISCISC*. IEEE, 2018.
- [BSS⁺11] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 241–250. ACM, 2011.
- [BTR⁺13] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Spllift: statically analyzing software product lines in minutes instead of years. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 355–364, 2013.
- [CB16] Maria Christakis and Christian 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*, page 332–343, New York, NY, USA, 2016. Association for Computing Machinery.
- [CDvfsd16] Frederic Gagnon Christian Dietrich and various free software developers. Eclipse lsp4j. <https://projects.eclipse.org/proposals/eclipse-lsp4j>, 2016. Accessed: 2021-08-03.
- [CGK12] Gary Charness, Uri Gneezy, and Michael A Kuhn. Experimental methods: Between-subject and within-subject design. *Journal of Economic Behavior & Organization*, 81(1):1–8, 2012.
- [Che06] Checkmarx. Checkmarx, 2006. Accessed: 2021-08-03.
- [CHN16] Nguyen Cam, Pham Hau, and Tuan Nguyen. Android security analysis based on inter-application relationships. In *Information Science and Applications (ICISA) 2016*, pages 689–700. Springer, 01 2016.

- [CK94] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *Transactions on Software Engineering IEEE*, 1994.
- [CKBG18] Paolo Calciati, Konstantin Kuznetsov, Xue Bai, and Alessandra Gorla. What did really change with the new release of the app? In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 142–152, 2018.
- [CLHS17] Hongyi Chen, Ho-fung Leung, Biao Han, and Jinshu Su. Automatic privacy leakage detection for massive android apps via a novel hybrid approach. In *IEEE International Conference on Communications, ICC 2017, Paris, France, May 21-25, 2017*, pages 1–7, 2017.
- [Con12] Contagio Mobile. <http://contagiominidump.blogspot.com>, 2012. Accessed: 2021-08-03.
- [Con18] Contagio Mobile Malware, 2018. Accessed: 2021-08-03.
- [CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.
- [DAL⁺17a] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson 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*, pages 39–42, 2017.
- [DAL⁺17b] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson R. Murphy-Hill. Just-in-time static analysis. In Tefvik Bultan and Koushik Sen, editors, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 307–317. ACM, 2017.
- [DAMvfsd76] Guy L. Steele Jr. David A. Moon and various free software developers. Emacs. <https://www.gnu.org/software/emacs>, 1976. Accessed: 2021-08-03.
- [DD12] Arnab De and Deepak D’Souza. Scalable flow-sensitive pointer analysis for java with strong updates. In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 665–687. Springer, 2012.
- [DEB16] Lisa Nguyen Quang Do, Michael Eichberg, and Eric Bodden. Toward an automated benchmark management system. In *Proceedings of the 5th SOAP@PLDI*. ACM, 2016.
- [Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k -limiting. In *Proceedings of the ACM SIGPLAN’94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, pages 230–241, 1994.
- [DFLO19] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O’Hearn. Scaling static analyses at facebook. *Commun. ACM*, 62(8):62–70, July 2019.

- [dMB08] Leonardo Mendonça de Moura and Nikolaj 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*, pages 337–340, 2008.
- [Doo09] Doop. <http://doop.program-analysis.org>, 2009. Accessed: 2021-08-03.
- [Dro16] DroidBench 3-0. <https://github.com/secure-software-engineering/DroidBench/tree/develop>, September 2016. Accessed: 2021-08-03.
- [DSAR18] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. Ariadne: analysis for machine learning programs. In Justin Gottschlich and Alvin Cheung, editors, *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18–22, 2018*, pages 1–10. ACM, 2018.
- [DtJd20] JADX: Dex to Java decompiler. <https://github.com/skylot/jadx>, 2020. Accessed: 2021-08-03.
- [DWA20] Lisa Nguyen Quang Do, James Wright, and Karim Ali. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering*, 2020.
- [EB10] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.
- [EGC⁺14] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick D. McDaniel, and Anmol Sheth. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM*, 2014.
- [EGH⁺14] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick D. McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29, 2014.
- [EHMG15] Michael Eichberg, Ben Hermann, Mira Mezini, and Leonid Glanz. Hidden truths in dead software paths. In *Proceedings of the 10th ESEC/FSE*. ACM, 2015.
- [EN08] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electron. Notes Theor. Comput. Sci.*, 217:5–21, 2008.
- [F-D20] F-Droid. <https://F-Droid.org>, 2020. Accessed: 2021-08-03.
- [Fac15] Facebook. Facebook infer. <https://fbinfer.com>, 2015. Accessed: 2021-08-03.
- [FD12] Stephen Fink and Julian Dolby. Wala—the tj watson libraries for analysis, 2012.
- [FDC04] Stephen Fink, Julian Dolby, and L. Colby. Semi-automatic j2ee transaction configuration. <https://dominoweb.draco.res.ibm.com/32e774866c89774a85256f0400669309.html>, 2004. Accessed: 2021-08-03.

- [Flo17] FlowDroid. <https://github.com/secure-software-engineering/soot-infoflow-android/wiki>, April 2017. Accessed: 2021-08-03.
- [Flo19] FlowDroid*. <https://github.com/secure-software-engineering/FlowDroid/releases/tag/v2.7.1>, January 2019. Accessed: 2021-08-03.
- [FMC06] Jennifer Fereday and Eimear Muir-Cochrane. Demonstrating rigor using thematic analysis: A hybrid approach of inductive and deductive coding and theme development. *International Journal of Qualitative Methods*, 5(1):80–92, 2006.
- [Fou12] JS Foundation. Appium. <https://appium.io>, 2012. Accessed: 2021-08-03.
- [FYD⁺06] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective tpestate verification in the presence of aliasing. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 133–144, 2006.
- [GBJ06] Greg Guest, Arwen Bunce, and Laura Johnson. How many interviews are enough? an experiment with data saturation and variability. *Field Methods*, 18(1):59–82, 2006.
- [Gee20] GeeksforGeeks. Top 10 most popular java frameworks for web development. <https://www.geeksforgeeks.org/top-10-most-popular-java-frameworks-for-web-development>, 2020. Accessed: 2021-08-03.
- [Git14] GitHub. Atom. <https://atom.io>, 2014. Accessed: 2021-08-03.
- [Git18] Gitpod. Gitpod. <https://www.gitpod.io>, 2018. Accessed: 2021-08-03.
- [Git19a] GitHub. Codeql. <https://codeql.github.com/docs/codeql-overview>, 2019. Accessed: 2021-08-03.
- [Git19b] GitHub. Lgtm, 2019. Accessed: 2021-08-03.
- [Git20] GitHub. Github actions. <https://docs.github.com/en/actions>, 2020. Accessed: 2021-08-03.
- [GKP⁺15] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. Information flow analysis of android applications in droidsafe. In *Proceedings of the 22nd NDSS*. The Internet Society, 2015.
- [GM14] Xi Ge and Emerson R. Murphy-Hill. Manual refactoring changes with automated refactoring validation. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 1095–1105, 2014.
- [Gmb13] RIGS IT GmbH. Xanitizer. <https://www.rigs-it.com/xanitizer>, 2013. Accessed: 2021-08-03.
- [Goo12] Google. Google play services. <https://developers.google.com/android/reference/packages>, 2012. Accessed: 2021-08-03.
- [Goo13a] Google. Android studio. <https://developer.android.com/studio>, 2013. Accessed: 2021-08-03.

- [Goo13b] Google. Espresso test framework. <https://developer.android.com/training/testing/espresso>, 2013. Accessed: 2021-09-09.
- [Goo13c] Google. Ui automator. <https://developer.android.com/training/testing/ui-automator>, 2013. Accessed: 2021-09-09.
- [Goo18a] Google. Device compatibility, 2018. Accessed: 2021-08-03.
- [Goo18b] Google. Platform Versions, 2018. Accessed: 2021-08-03.
- [Goo21] Google. Package name in android manifest. <https://developer.android.com/guide/topics/manifest/manifest-element.html#package>, 2021. Accessed: 2021-08-03.
- [Gra05] GrammaTech. Codesonar. <https://www.grammatech.com/products/codesonar>, 2005. Accessed: 2021-08-03.
- [Gra18] GrammaTech. Static analysis results: A format and a protocol: Sarif and sasp. <http://blogs.grammatech.com/static-analysis-results-a-format-and-a-protocol-sarif-sasp>, 2018. Accessed: 2021-08-03.
- [Gro05] JSON-RPC Working Group. Json-rpc. <https://www.jsonrpc.org>, 2005. Accessed: 2021-08-03.
- [Gro07] LLVM Developer Group. Clang static analyzer. <https://clang-analyzer.llvm.org>, 2007. Accessed: 2021-08-03.
- [GS15] Dennis Giffhorn and Gregor Snelting. A new algorithm for low-deterministic security. *International Journal of Information Security*, 14(3):263–287, Jun 2015.
- [GS17] Neville Grech and Yannis Smaragdakis. P/taint: Unified points-to and taint analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.
- [GSS68] Barney G Glaser, Anselm L Strauss, and Elizabeth Strutzel. The discovery of grounded theory; strategies for qualitative research. *Nursing research*, 17(4):364, 1968.
- [Hau21] Fynn Hauptmeier. Targeted testing input generation for android applications. Master’s thesis, Paderborn University, 2021.
- [HDMD15] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for android. In *Proceedings of ISSTA*. ACM, 2015.
- [HP07] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE ’07, pages 9–14, New York, NY, USA, 2007. ACM.
- [HQ08] Sublime HQ. Sublime text. <https://www.sublimetext.com>, 2008. Accessed: 2021-08-03.
- [HS09] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, December 2009.

- [HZZ⁺14] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th ICSE*. ACM, 2014.
- [IBM98] IBM. Ibm websphere. <https://www.ibm.com/cloud/websphere-application-platform>, 1998. Accessed: 2021-08-03.
- [IBM07] IBM. Appscan. <https://www.hcltechsw.com/appscan>, 2007. Accessed: 2021-08-03.
- [IF01] IBM and Eclipse Foundation. Eclipse. <https://www.eclipse.org>, 2001. Accessed: 2021-08-03.
- [IMW19] Nasif Imtiaz, Brendan Murphy, and Laurie Williams. How do developers act on static analysis alerts? an empirical study of coverity usage. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 323–333. IEEE, 2019.
- [Jet01] JetBrains. IntelliJ. <https://www.jetbrains.com/idea>, 2001. Accessed: 2021-08-03.
- [Jet10] JetBrains. Pycharm. <https://www.jetbrains.com/pycharm>, 2010. Accessed: 2021-08-03.
- [JSMB13] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. Why don’t software developers use static analysis tools to find bugs? In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE ’13, San Francisco, CA, USA, May 18-26, 2013*, pages 672–681. IEEE Computer Society, 2013.
- [KBL16] Rahul Kumar, Chetan Bansal, and Jakob Lichtenberg. Static analysis using the cloud. *arXiv preprint arXiv:1610.08198*, 2016.
- [Ken16] Joseph Chan Joo Keng. Automated testing and notification of mobile app privacy leak-cause behaviours. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 880–883, 2016.
- [Ker21] Jörn 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.
- [Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, pages 194–206, 1973.
- [KNR⁺17] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, et al. Cognicrypt: supporting developers in using cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 931–936. IEEE Press, 2017.
- [KWJB13] Joseph Chan Joo Keng, Tan Kiat Wee, Lingxiao Jiang, and Rajesh Krishna Balan. The case for mobile forensics of private data leaks: towards large-scale

- user-oriented privacy protection. In *Asia-Pacific Workshop on Systems, APSys '13, Singapore, Singapore, July 29-30, 2013*, pages 6:1–6:7, 2013.
- [LBB⁺15] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Ochteau, and Patrick D. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th ICSE*. IEEE Computer Society, 2015.
- [LBK⁺14] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Ochteau, and Patrick D. McDaniel. I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis. *CoRR*, abs/1404.7431, 2014.
- [LBLH11a] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.
- [LBLH11b] Patrick Lam, Eric Bodden, Ondrej Lhotak, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. *Cetus '11*, 2011.
- [LBLH11c] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Proceedings of CETUS*, 2011.
- [LBP⁺17] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Ochteau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. *Inf. Softw. Technol.*, 88:67–95, 2017.
- [LBS19] Linghui Luo, Eric Bodden, and Johannes Späth. A qualitative analysis of android taint-analysis results. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 102–114. IEEE, 2019.
- [LDB19] Linghui Luo, Julian Dolby, and Eric Bodden. Magpiebridge: A general approach to integrating static analyses into ides and editors (tool insights paper). In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*, volume 134 of *LIPICs*, pages 21:1–21:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
- [LDR16] Sungho Lee, Julian Dolby, and Sukyoung Ryu. Hybridroid: static analysis framework for android hybrid applications. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 250–261. ACM, 2016.
- [LH03] Ondrej Lhoták and Laurie J. Hendren. Scaling java points-to analysis using SPARK. In Görel Hedin, editor, *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*, volume 2622 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2003.

- [LKB17] Max Lillack, Christian Kastner, and Eric Bodden. Tracking Load-time Configuration Options. *IEEE Transactions on Software Engineering*, 5589(c):1–1, 2017.
- [LL05] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th USENIX Security Symposium*. USENIX Association, 2005.
- [LPP⁺21] Linghui Luo, Felix Pauck, Goran Piskachev, Manuel Benz, Ivan Pashchenko, Martin Mory, Eric Bodden, Ben Hermann, and Fabio Massacci. Taintbench: Automatic real-world malware benchmarking of android taint analyses. *Empirical Software Engineering*, 2021.
- [LSS⁺15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundness: a manifesto. *Commun. ACM*, 58(2):44–46, 2015.
- [LSSB21a] Linghui Luo, Martin Schäf, Daniel Sanchez, and Eric Bodden. Ide support for cloud-based static analyses. In *Proceedings of the the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2021*, 2021.
- [LSSB21b] Linghui Luo, Martin Schäf, Daniel Sanchez, and Eric Bodden. Test applications and issue list, 2021. Accessed: 2021-08-03.
- [Luo21] Linghui Luo. A general approach to modelling java framework behaviors. In *The ACM Student Research Competition at the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2021*, 2021.
- [MAHF18] Hongpeng Man, Jing An, Wei Huang, and Wenqing Fan. Jsefuzz: Vulnerability detection method for java web application. In *2018 3rd International Conference on System Reliability and Safety (ICSRS)*, pages 92–96, 2018.
- [Mar07] Daniel Marjamäki. Cppcheck. <http://cppcheck.sourceforge.net>, 2007. Accessed: 2021-08-03.
- [Mau15] Maurício Aniche. Java code metrics calculator (CK). <https://github.com/mauricioaniche/ck>, 2015. Accessed: 2021-08-03.
- [MDSK21] Nicole Perlroth Michael D. Shear and Clifford 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.
- [Mic15a] Microsoft. Visual studio code. <https://code.visualstudio.com>, 2015. Accessed: 2021-08-03.
- [Mic15b] Microsoft. Vs code webview apis. <https://code.visualstudio.com/api/extension-guides/webview>, 2015. Accessed: 2021-08-03.
- [Mic16a] Microsoft. Language server protocol, 2016. Accessed: 2021-08-03.
- [Mic16b] Microsoft. Monaco. <https://microsoft.github.io/monaco-editor/index.html>, 2016. Accessed: 2021-08-03.

- [Mic20a] Trend Micro. New tekya ad fraud found on google play. <https://blog.trendmicro.com/trendlabs-security-intelligence/new-tekya-ad-fraud-found-on-google-play>, 2020. Accessed: 2021-08-03.
- [Mic20b] Microsoft. VSC - Visual Studio Code. <https://code.visualstudio.com>, 2020. Accessed: 2021-08-03.
- [Moo91] Bram Moolenaar. Vim. <https://www.vim.org>, 1991. Accessed: 2021-08-03.
- [MR17] Joydeep Mitra and Venkatesh-Prasad Ranganath. Ghera: A repository of android app vulnerability benchmarks. In *Proceedings of the 13th PROMISE*. ACM, 2017.
- [MSDM16] Patrick Mutchler, Yeganeh Safaei, Adam Doupé, and John C. Mitchell. Target fragmentation in android apps. In *2016 IEEE Security and Privacy Workshops, SP Workshops 2016, San Jose, CA, USA, May 22-26, 2016*, pages 204–213, 2016.
- [MSHN17] Alfonso Murolo, Fabian Stutz, Maria Husmann, and Moira C. Norrie. Improved developer support for the detection of cross-browser incompatibilities. In *Web Engineering - 17th International Conference, ICWE 2017, Rome, Italy, June 5-8, 2017, Proceedings*, pages 264–281, 2017.
- [MSTdF17] Omid Mirzaei, Guillermo Suarez-Tangil, Juan E. Tapiador, and José María de Fuentes. Triflow: Triaging android applications using speculative information flows. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, pages 640–651, 2017.
- [NWA⁺17] Duc-Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. A stitch in time: Supporting android developers in writing secure code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1065–1077, 2017.
- [OAS18] OASIS. Sarif specification. <https://github.com/oasis-tcs/sarif-spec>, 2018. Accessed: 2021-08-03.
- [OLD⁺15] Damien Ochteau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick D. McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 77–88. IEEE Computer Society, 2015.
- [OMJ⁺13] Damien Ochteau, Patrick D. McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 543–558, 2013.
- [Ope15] OpenSignal. Android Fragmentation, 2015. Accessed: 2021-08-03.
- [PBW18] Felix Pauck, Eric Bodden, and Heike Wehrheim. Do android taint analysis tools keep their promises? In *Proceedings of ESEC/FSE*. ACM, 2018.

- [PDB19] Goran Piskachev, Lisa Nguyen Quang Do, and Eric Bodden. Codebase-adaptive detection of security-relevant methods. In *Proceedings of the 28th ISSTA*. ACM, 2019.
- [PK13] Rohan Padhye and Uday P. Khedker. Interprocedural data flow analysis in soot using value contexts. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program analysis, SOAP 2013, Seattle, WA, USA, June 20, 2013*, pages 31–36, 2013.
- [PRL⁺19] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th PLDI*. ACM, 2019.
- [PW19] Felix Pauck and Heike Wehrheim. Together strong: cooperative android app analysis. In *Proceedings of ESEC/FSE*. ACM, 2019.
- [PZ19] Felix Pauck and Shikun Zhang. Android app merging for benchmark speed-up and analysis lift-up. In *Proceedings of the 2nd A-Mobile@ASE*. IEEE, 2019.
- [QWR18] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ISSTA*. ACM, 2018.
- [RAB14] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *Proceedings of the 21st NDSS*. The Internet Society, 2014.
- [RAMB16] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.
- [Ras16] Siegfried Rasthofer. *Improving Mobile-Malware Investigations with Static and Dynamic Code Analysis Techniques*. PhD thesis, Technische Universität Darmstadt, December 2016.
- [RATP17] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. Making malory behave maliciously: targeted fuzzing of android execution environments. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 300–311. IEEE / ACM, 2017.
- [Ray09] Pierre Raybaut. Spyder. <https://www.spyder-ide.org>, 2009. Accessed: 2021-08-03.
- [RCJ13] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*. ACM, 2013.
- [REHM17] Michael Reif, Michael Eichberg, Ben Hermann, and Mira Mezini. Hermes: assessment and creation of effective test corpora. In *Proceedings of the 6th SOAP@PLDI*. ACM, 2017.

- [Reu21] Markus Reuter. Schon wieder desaströse sicherheit-slücke in luca app. <https://netzpolitik.org/2021/it-sicherheit-schon-wieder-desastroese-sicherheitsluecke-in-luca-app>, 2021. Accessed: 2021-09-21.
- [RHS95] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL’95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61, 1995.
- [RKG04] Atanas Rountev, Scott Kagan, and Michael Gibas. Static and dynamic analysis of call chains in java. In *Proceedings of ISSTA*. ACM, 2004.
- [RM20] Venkatesh-Prasad Ranganath and Joydeep Mitra. Are free android app security analysis tools effective in detecting known vulnerabilities? *Empirical Software Engineering*, 25(1):178–219, 2020.
- [RN10] Graeme D Ruxton and Markus Neuhäuser. When should we use one-tailed hypothesis testing? *Methods in Ecology and Evolution*, 1(2):114–117, 2010.
- [SAB19] Johannes Späth, Karim Ali, and Eric Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proc. ACM Program. Lang.*, 3(POPL):48:1–48:29, January 2019.
- [SAE⁺18] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspán. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4):58–66, March 2018.
- [Saf14] Safe. <https://github.com/sukyoung/safe>, 2014. Accessed: 2021-08-03.
- [SAP⁺11] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4F: taint analysis of framework-based web applications. In Cristina Videira Lopes and Kathleen Fisher, editors, *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*, pages 1053–1068. ACM, 2011.
- [SBF⁺16] Lovely Sinha, Shweta Bhandari, Parvez Faruki, Manoj Singh Gaur, Vijay Laxmi, and Mauro Conti. Flowmine: Android app analysis via data flow. In *13th IEEE Annual Consumer Communications & Networking Conference, CCNC 2016, Las Vegas, NV, USA, January 9-12, 2016*, pages 435–441, 2016.
- [SCS21] Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. Demanded abstract interpretation. In Stephen N. Freund and Eran Yahav, editors, *PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 282–295. ACM, 2021.
- [SDAB16] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:26, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

- [SDOF07] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The emperor’s new security indicators. In *2007 IEEE Symposium on Security and Privacy (SP ’07)*, pages 51–65, May 2007.
- [SE14] Daniela Steidl and Sebastian Eder. Prioritizing maintainability defects based on refactoring recommendations. In Chanchal K. Roy, Andrew Begel, and Leon Moonen, editors, *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, pages 168–176. ACM, 2014.
- [SEB21] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. Incremental whole-program analysis in datalog with lattices. In Stephen N. Freund and Eran Yahav, editors, *PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 1–15. ACM, 2021.
- [Sec19] IBM Security. Cost of a data breach report 2019. <https://www.ibm.com/downloads/cas/RDEQK07R>, 2019. Accessed: 2021-08-03.
- [Ser20] Amazon Web Services. Amazon codeguru reviewer. <https://aws.amazon.com/codeguru>, 2020. Accessed: 2021-08-03.
- [Ser21a] Amazon Web Services. Aws sdk for java, 2021. Accessed: 2021-08-03.
- [Ser21b] Amazon Web Services. Public api of amazon codeguru reviewer, 2021. Accessed: 2021-08-03.
- [SEV20] Helmut Seidl, Julian Erhard, and Ralf Vogler. Incremental abstract interpretation. In Alessandra Di Pierro, Pasquale Malacaria, and Rajagopal Nagarajan, editors, *From Lambda Calculus to Cybersecurity Through Program Analysis - Essays Dedicated to Chris Hankin on the Occasion of His Retirement*, volume 12065 of *Lecture Notes in Computer Science*, pages 132–148. Springer, 2020.
- [SFT15] Julian Schütte, Rafael Fedler, and Dennis Titze. Condroid: Targeted dynamic analysis of android applications. In *29th IEEE International Conference on Advanced Information Networking and Applications, AINA 2015, Gwangju, South Korea, March 24-27, 2015*, pages 571–578, 2015.
- [SGF⁺13] Ryan Stevens, Jonathan Ganz, Vladimir Filkov, Premkumar T. Devanbu, and Hao Chen. Asking for (and about) permissions used by android apps. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR ’13, San Francisco, CA, USA, May 18-19, 2013*, pages 31–40, 2013.
- [Sha21] Lingkai Shao. Top 5 android automated testing frameworks with code examples. <https://bitbar.com/blog/top-5-android-testing-frameworks-with-examples/>, 2021. Accessed: 2021-09-09.
- [SHR⁺00] Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In Mary Beth Rosson and Doug Lea, editors, *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000*, pages 264–280. ACM, 2000.

- [Sie13] Siegfried Rasthofer. The android logging service – a dangerous feature for user privacy? <https://blogs.uni-paderborn.de/sse/2013/05/17/privacy-threatened-by-logging>, 2013. Accessed: 2021-08-03.
- [SN93] Douglas Schuler and Aki Namioka. *Participatory design: Principles and practices*. CRC Press, 1993.
- [Sne96] Gregor Snelting. Combining slicing and constraint solving for validation of measurement software. *Static Analysis SE - 23*, 1145(Springer):332–348, 1996.
- [Son20] Jitendra Soni. This dangerous malware got around google play store security. <https://www.techradar.com/news/phantomlance-malware-breaches-google-play-store-security>, 2020. Accessed: 2021-08-03.
- [Soo00] Soot. <https://github.com/Sable/soot>, 2000. Accessed: 2021-08-03.
- [Sou15] Souffle. <https://github.com/oracle/souffle/wiki>, 2015. Accessed: 2021-08-03.
- [Spr02a] Spring. Spring ioc container and beans. <https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html>, 2002. Accessed: 2021-08-03.
- [Spr02b] Spring. Spring web mvc framework. <https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html>, 2002. Accessed: 2021-08-03.
- [Spr21] Spring. The executable jar format. <https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-executable-jar-format.html>, 2021. Accessed: 2021-08-03.
- [SRH95] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Lecture Notes in Computer Science*, 915:651–665, 1995.
- [Sta19] Statcounter. Operating system market share worldwide jan - dec 2019. <https://gs.statcounter.com/os-market-share#monthly-201901-201912-bar>, 2019. Accessed: 2021-08-03.
- [Sta20] 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-record>, 2020. Accessed: 2021-08-03.
- [SW65] S. S. SHAPIRO and M. B. WILK. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, dec 1965.
- [Syn08] Synopsys. Coverity scan, 2008. Accessed: 2021-08-03.
- [T21] Amaan T. Appium vs robotium. <https://dzone.com/articles/appium-vs-robotium>, 2021. Accessed: 2021-09-09.

- [TBM13] John C. Tang, Jed R. Brubaker, and Catherine C. Marshall. What do you see in the cloud? understanding the cloud-based user experience through practices. In Paula Kotzé, Gary Marsden, Gitte Lindgaard, Janet Wesson, and Marco Winckler, editors, *Human-Computer Interaction - INTERACT 2013 - 14th IFIP TC 13 International Conference, Cape Town, South Africa, September 2-6, 2013, Proceedings, Part II*, volume 8118 of *Lecture Notes in Computer Science*, pages 678–695. Springer, 2013.
- [TC10] Emina Torlak and Satish Chandra. Effective interprocedural resource leak detection. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10*, pages 535–544, New York, NY, USA, 2010. ACM.
- [Tea11] PMD Team. Pmd. <https://pmd.github.io>, 2011. Accessed: 2021-08-03.
- [Tea17] SpotBugs Team. Spotbugs. <https://spotbugs.github.io>, 2017. Accessed: 2021-08-03.
- [Tec14] Robotium Tech. Robotium. <https://github.com/RobotiumTech/robotium>, 2014. Accessed: 2021-09-09.
- [TKB⁺14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.
- [TPC⁺13] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 210–225, 2013.
- [TPF⁺09] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: Effective taint analysis of web applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09*, pages 87–97, New York, NY, USA, 2009. ACM.
- [TSS10] Mana Taghdiri, Gregor Snelting, and Carsten Sinz. Information flow analysis via path condition refinement. In *Formal Aspects of Security and Trust - 7th International Workshop, FAST 2010, Pisa, Italy, September 16-17, 2010. Revised Selected Papers*, pages 65–79, 2010.
- [TTYR17] Ke Tian, Gang Tan, Danfeng Daphne Yao, and Barbara G. Ryder. Redroid: Prioritizing data flows and sinks for app security transformation. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation, FEAST@CCS 2017, Dallas, TX, USA, November 3, 2017*, pages 35–41, 2017.
- [UKJS10] Ilkka Uusitalo, Kaarina Karppinen, Arto Juhola, and Reijo Savola. Trust and cloud services-an interview study. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 712–720. IEEE, 2010.

- [VCG⁺99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of CASCAN*. IBM, 1999.
- [Ver06] Veracode. Veracode, 2006. Accessed: 2021-08-03.
- [Ver17] Veracode. Veracode static for ide, 2017. Accessed: 2021-08-03.
- [Vir14] VirusShare. <https://virusshare.com>, 2014. Accessed: 2021-08-03.
- [VPBG18] Carmine Vassallo, Fabio Palomba, Alberto Bacchelli, and Harald C Gall. Continuous code quality: are we (really) doing that? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 790–795, 2018.
- [VPP⁺11] Kaisa Väänänen-Vainio-Mattila, Jarmo Palviainen, Santtu Pakarinen, Else Lagerstam, and Eeva Kangas. User perceptions of wow experiences and design implications for cloud services. In Alessandro Deserti, Francesco Zurlo, and Francesca Rizzo, editors, *Designing Pleasurable Products and Interfaces, DPPI '11, Milano, Italy, June 22-25, 2011*, pages 63:1–63:8. ACM, 2011.
- [VPP⁺18] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall. Context is king: The developer perspective on the usage of static analysis tools. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 38–49, 2018.
- [Vra17] Christos V. Vrachas. Integration of static analysis results with proguard optimizer for android applications. *Bachelor Thesis*, 2017.
- [VRCG⁺10] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCAN First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.
- [WAL06] WALA. <https://github.com/wala/WALA>, 2006. Accessed: 2021-08-03.
- [Web14] WebGoat. <https://github.com/WebGoat/WebGoat>, 2014. Accessed: 2021-08-03.
- [WL16] Michelle Y. Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *Proceedings of the 23rd NDSS*. The Internet Society, 2016.
- [WLR⁺17] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In *Proceedings of the 14th DIMVA*, volume 10327 of *Lecture Notes in Computer Science*. Springer, 2017.
- [Woo08] R. F. Woolson. *Wilcoxon Signed-Rank Test*, pages 1–3. American Cancer Society, 2008.
- [WR13] Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for javascript. In Mauro Pezzè and Mark Harman, editors, *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 336–346. ACM, 2013.
- [WROR14] Fengguo Wei, Sankardas Roy, Xinming 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.

- [WTK⁺08] Lizhe Wang, Jie Tao, Marcel Kunze, Alvaro Canales Castellanos, David Kramer, and Wolfgang Karl. Scientific cloud computing: Early definition and experience. In *10th IEEE International Conference on High Performance Computing and Communications, HPCC 2008, 25-27 Sept. 2008, Dalian, China*, pages 825–830. IEEE Computer Society, 2008.
- [WWLZ16] Songyang Wu, Pan Wang, Xun Li, and Yong Zhang. Effective detection of android malware based on the usage of data flow apis and machine learning. *Information & Software Technology*, 75:17–25, 2016.
- [WWZ⁺20] Jie Wang, Yunguang Wu, Gang Zhou, Yiming Yu, Zhenyu Guo, and Yingfei Xiong. Scaling static taint analysis to industrial SOA applications: a case study at alibaba. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE ’20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1477–1486. ACM, 2020.
- [WZR16] Yan Wang, Hailong Zhang, and Atanas Rountev. On the unsoundness of static analysis for android guis. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2016, Santa Barbara, CA, USA, June 14, 2016*, pages 18–23, 2016.
- [XCLM11] Jing Xie, Bill Chu, Heather Richter Lipford, and John T. Melton. ASIDE: IDE support for web application security. In *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*, pages 267–276, 2011.
- [XGL⁺15] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 899–914, 2015.
- [YQL⁺16] Tianda Yang, Kai Qian, Lei Li, Dan Chia-Tien Lo, and Lixin Tao. Static mining and dynamic taint for mobile security threats analysis. In *2016 IEEE International Conference on Smart Cloud, SmartCloud 2016, New York, NY, USA, November 18-20, 2016*, pages 234–240. IEEE Computer Society, 2016.
- [YS17] Ayman Youssef and Ahmed F. Shosha. Quantitative dynamic taint analysis of privacy leakage in android arabic apps. In *Proceedings of the 12th International Conference on Availability, Reliability and Security, ARES ’17, New York, NY, USA, 2017*. Association for Computing Machinery.
- [YXA⁺15] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 303–313, 2015.
- [ZJ12] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 95–109. IEEE Computer Society, 2012.

- [ZJY⁺17] Dali Zhu, Hao Jin, Ying Yang, Di Wu, and Weiyi Chen. Deepflow: Deep learning-based malware detection by mining android application for abnormal usage of sensitive data. In *2017 IEEE Symposium on Computers and Communications, ISCC 2017, Heraklion, Greece, July 3-6, 2017*, pages 438–443, 2017.
- [ZSO⁺17] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 334–344. IEEE, 2017.
- [ZTD19] Jie Zhang, Cong Tian, and Zhenhua Duan. Fastdroid: efficient taint analysis for android applications. In *Proceedings of the 41st ICSE*. IEEE / ACM, 2019.
- [ZZD⁺12] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of SPSM@CCS*. ACM, 2012.

Supplementary Material of Chapter 2

A.1 Usability Test

To show how usable the tools in our TAINTBENCH framework are, we conducted a controlled experiment with experts to test JADX with TB-EXTRACTOR and Visual Studio Code (VSC) with TB-VIEWER. Thereby we wanted to answer the following research questions:

- RQ1** Do users spend less time to inspect and document taint flows using TB-EXTRACTOR and TB-VIEWER than using plain JADX and Visual Studio Code?
- RQ2** Do users perceive TB-EXTRACTOR and TB-VIEWER to be more usable than plain JADX and Visual Studio Code?

A.1.1 Participants

The TAINTBENCH framework is designed for experts and it is hard to find suitable users. We sent emails to researchers who work in area of program analysis and developers who have experience in developing static analysis tools. We were able to recruit five experts to participate in our study. Four of them are researchers (PhD students). One of them is a software engineer who has experience in developing static analyzers. All participants are very familiar with taint analysis. We denote them with User 1–5 in the following.

A.1.2 Study Design

Due to the low number of participants, we designed a within-subjects study for each tool, i.e., each participant tests all the conditions. We compare the condition with tool support to without tool support. Table A.1 shows the tasks we designed for the study. While tasks **VSC** (control condition) and **VSC+TB-Viewer** (experimental condition) are used to test TB-VIEWER, the tasks **Jadx** (control condition) and **Jadx+TB-Extractor** (experimental condition) are for testing TB-EXTRACTOR.

Tasks for testing TB-Viewer: For TB-VIEWER, the two tasks are about the inspection of taint flows. These tasks simulate the manual inspection one has to do when evaluating a tool’s precision. The participants were asked to judge whether taint flows reported by a taint analysis tool are false positives or not. We prepared six taint flows reported by FLOWDROID when applying it to an app from our benchmark suite. To avoid unfair distribution, we intentionally chose

Table A.1: Descriptions of tasks.

Task	Description
<i>VSC</i> (control)	The participant was given plain VS Code, decompiled source code of an app X and 3 taint flows in X. The participant was asked to judge whether these taint flows are true positives or false positives in VS Code.
<i>VSC+TB-Viewer</i> (experimental)	The participant was given VS Code with TB-Viewer installed, decompiled source code of the app X and 3 taint flows in X (different 3 than in task VSC). The participant was asked to judge whether these taint flows are true positives or false positives in VS Code.
<i>Jadx</i> (control)	The participant was given the Jadx decompiler, an apk Y from our suite, and two expected taint flows specified for the apk. The participant was asked to document these two flows in TAF-format.
<i>Jadx+TB-Extractor</i> (experimental)	The participant was given the Jadx decompiler extended with TB-Extractor, the apk Y from our suite, and two expected taint flows (different 2 than in task Jadx) specified for the apk. The participant was asked to document these two flows in TAF-format.

six true-positive taint flows that we think to be similarly complex. However, the participants are not aware of this and they have to triage the taint flows by searching through and looking at relevant code.

In task **VSC**, the participants were given Visual Studio Code and decompiled source code of the app. We asked them to inspect three taint flows that are documented in an XML file (in AQL-Answer format¹). For each flow we provide information only about the source and the sink but not about the data-flow paths, as this is also the case when dealing with popular Android taint analysis tools.² For each participant, these three taint flows are *randomly* chosen from the six taint flows. In task **VSC+TB-Viewer**, the participants are asked to inspect the remaining three taint flows. In addition, they used Visual Studio with the extension TB-VIEWER installed. TB-VIEWER can read the taint flows from this XML file and display them directly in Visual Studio Code as described in Section 2.4.3.

To minimize the ordering/learning effects, we randomize the order of these two tasks for the participants. We make sure that the participants do not always start with task **VSC** nor **VSC+TB-Viewer**.

Tasks for testing TB-Extractor: For TB-EXTRACTOR, the participants are asked to document taint flows that are determined in an Android app. Manual inspection and discovering taint flows is a skillful and time-consuming task. To simplify the study, we give the participants taint flows found by us. In other words, they do not need to search taint flows by themselves, but only documenting them. We chose four taint flows from our baseline of an benchmark app. As described in Section 2.4.3, in Visual Studio Code (including TB-VIEWER) each taint flow is

¹<https://github.com/FoelliX/AQL-System/wiki/Answers>

²Note that FLOWDROID does provide an option to compute and output data-flow paths in its current version, not, however, in the version used for this study. To construct the ground truth, we preferred not to use the current version but instead the version from the REPRODROID paper due to the many false negatives that the current FLOWDROID version creates

displayed with detailed information about source, sink, its attributes and its intermediate flows as well as a general description. Note that we ensure the tasks **VSC** and **VSC+TB-Viewer** to be conducted before the tasks for TB-EXTRACTOR. Thus, the participants already know TB-VIEWER when conducting the tasks for TB-EXTRACTOR.

In task **Jadx**, the participants are given JADX together with the chosen app. They are asked to document two randomly chosen taint flows from the four prepared taint flows with a code editor. They are given a template JSON file using TAF-format in which they only need to fill in the information (code, line number etc.) about taint flows copied from JADX. The TAF-format is explained to the participants before they start the task. In task **Jadx+TB-Extractor**, the participants are given JADX *with* TB-EXTRACTOR. We play a short tutorial video (six minutes) to them. This video explains how to document taint flows with the extended JADX. The participants are required to document the other two taint flows. Similar to the tasks for testing TB-VIEWER, we also randomize the order of these two tasks for the participants.

A.1.3 Data Collection

We conducted the experiment with participants remotely via a video conference tool. Participants were asked to share their screen with us all the time. After a brief introduction to the study and guide for installation of our tools, we gave our tasks to the participants in written form and asked them to solve the tasks independently without our help. In each session, the participants were given maximally 30 minutes to solve each task. We measured the actual time each participant spent for each task. We asked the participants to give us a clear signal when they started and finished each task. After each task, the participants filled out an exit-survey. In this survey, they were asked to evaluate the ten statements from the System Usability Scale (SUS) and tell about their feeling when using the system to do the task. SUS is a questionnaire that is designed to measure the usability of the a system [Bro96]. The survey and the descriptions of all tasks used can be found on our website.³

A.1.4 Results

Figure A.1 shows the results of our experiment. The measured time is used to answer RQ1 and the SUS score for answering RQ2.

RQ1 (Time) All participants solved the tasks more efficiently with the support of TB-VIEWER and TB-EXTRACTOR than the ones without. Averagely, the time used for task **VSC+TB-Viewer** is reduced from 14.8 to 11 minutes in comparison to task **VSC**. With the support of TB-EXTRACTOR, the average time for solving the task is even halved (from 21.6 to 10.8).

RQ2 (Usability) Overall, the participants responded positively to both tools. Except **User 2**, all users gave both TB-VIEWER and TB-EXTRACTOR high SUS scores ranging from 80 to 100, which means the usability of both tools is *excellent* or at least *good* from their point of view. **User 2** explained to us why he rated **VSC+TB-Viewer** with low scores. He felt it was very cumbersome to do the task without the data-flow path of a taint flow displayed in the editor. However, this information is not given in the results of FLOWDROID, thus, TB-VIEWER cannot provide this feature. Actually, if the information about the data-flow path is given in the analysis results, TB-VIEWER can actually display this information as done for the taint flows in our baseline (see Figure 2.3). While **User 2** complained more about the analysis results missing data-flow path, other users felt well supported by the tool while solving the task. For example, **User 1** told us about his positive feeling about TB-VIEWER:

³<https://taintbench.github.io/userstudy>

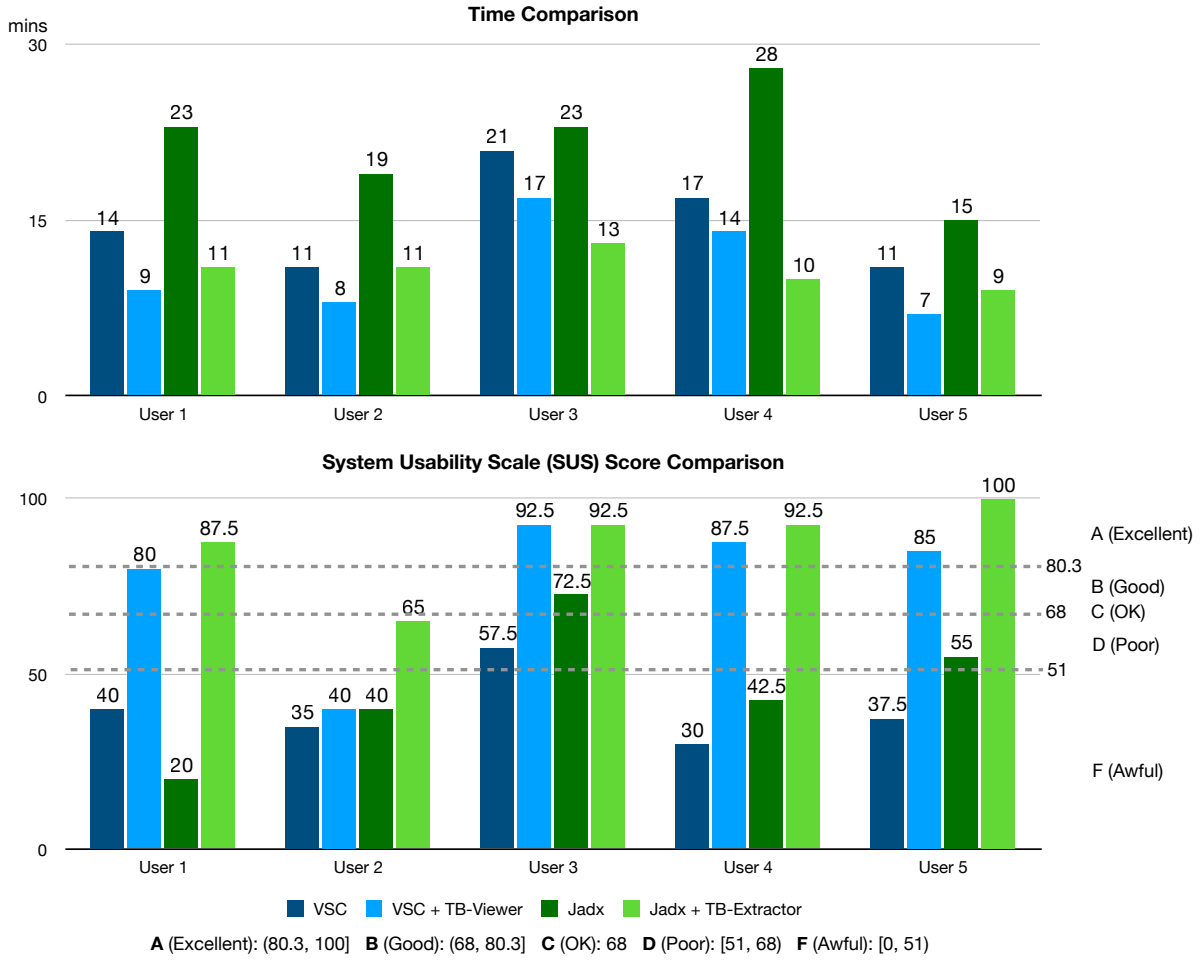


Figure A.1: Experimental results.

“Not having to switch back and forth constantly between VS Code and the XML file took away a lot of possible problem vectors. Like scrolling too far, misreading a line, misunderstanding what a particular line in the XML means. Even though the tool didn’t provide a lot more functionality (the ‘jump to’ feature is much appreciated) than the XML-based solution, I still felt more secure with my results in the end.”

Also **User 4** had similar feeling when solving task **VSC+TB-Viewer** and wrote:

“Finding the sources and sinks is much easier than without the system. Still it is not always easy to find the path between source and sink. Overall, the task is much easier to solve than without the system and gives higher confidence in giving the correct evaluation.”

In the documentation tasks, without TB-EXTRACTOR, all participants felt doing the task was very tedious and error-prone. They all made some mistakes (e.g., wrong line number, wrong method signature) in the documentation. In contrast, the taint flows documented with TB-EXTRACTOR were all correct. The participants felt TB-EXTRACTOR was self explanatory and easy to use. Especially **User 5** who had to document taint flows for other work before, gave a full SUS score (100) to TB-EXTRACTOR and spent the least time (9 minutes) for the task. **User 3**’s comments also show TB-EXTRACTOR eases the task:

Task **Jadx**: *“Absolutely cumbersome to use. A lot of busy work. No support by the tool at all.”*

Task **Jadx+TB-Extractor**: *“Easy to use! However, the description of the taint flow is in a separate window. But the window is well designed.”*

User 3’s comment on task **Jadx** reflects probably one of the main reasons for why in previous evaluations of taint analysis tools the ground truth was rarely documented. In summary, we see that TB-EXTRACTOR allows participants to document taint flows more efficiently and correctly.

A.2 Figures

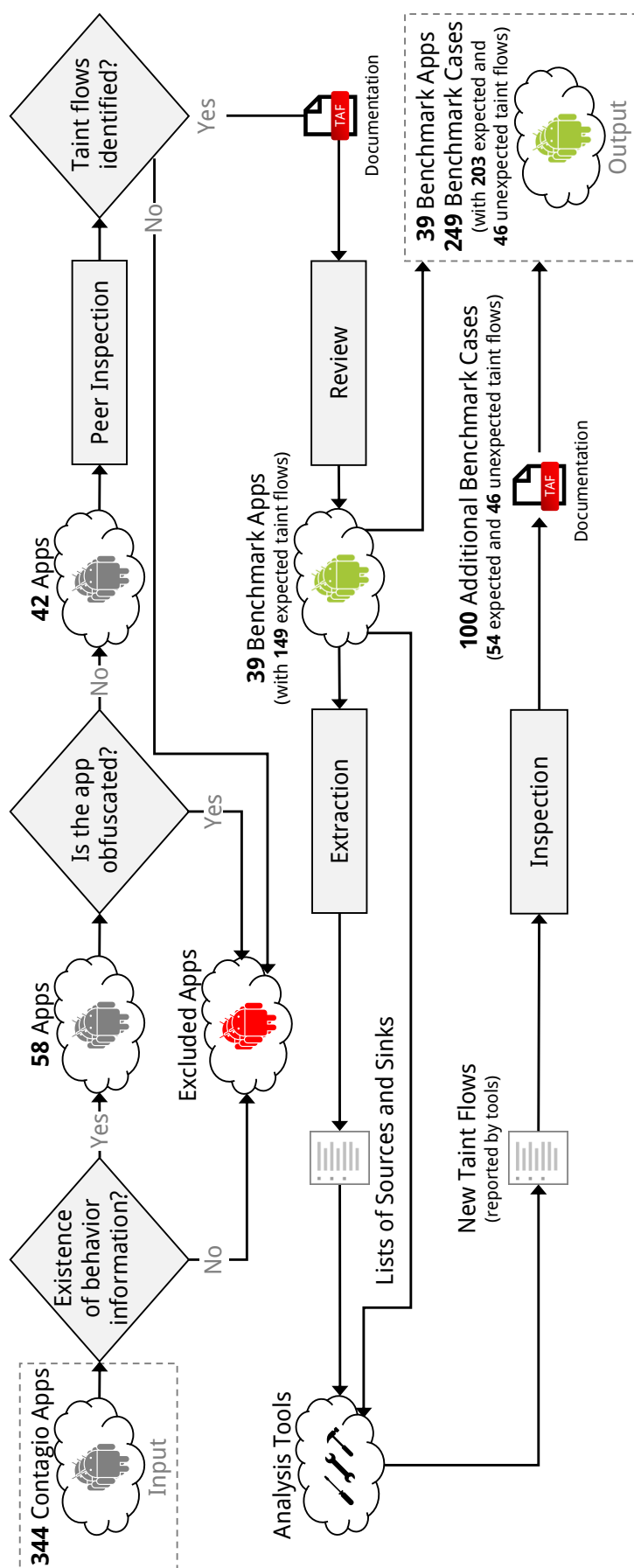


Figure A.2: The construction process of the TaintBENCH suite.

B

Supplementary Material of Chapter 3

B.1 Figures

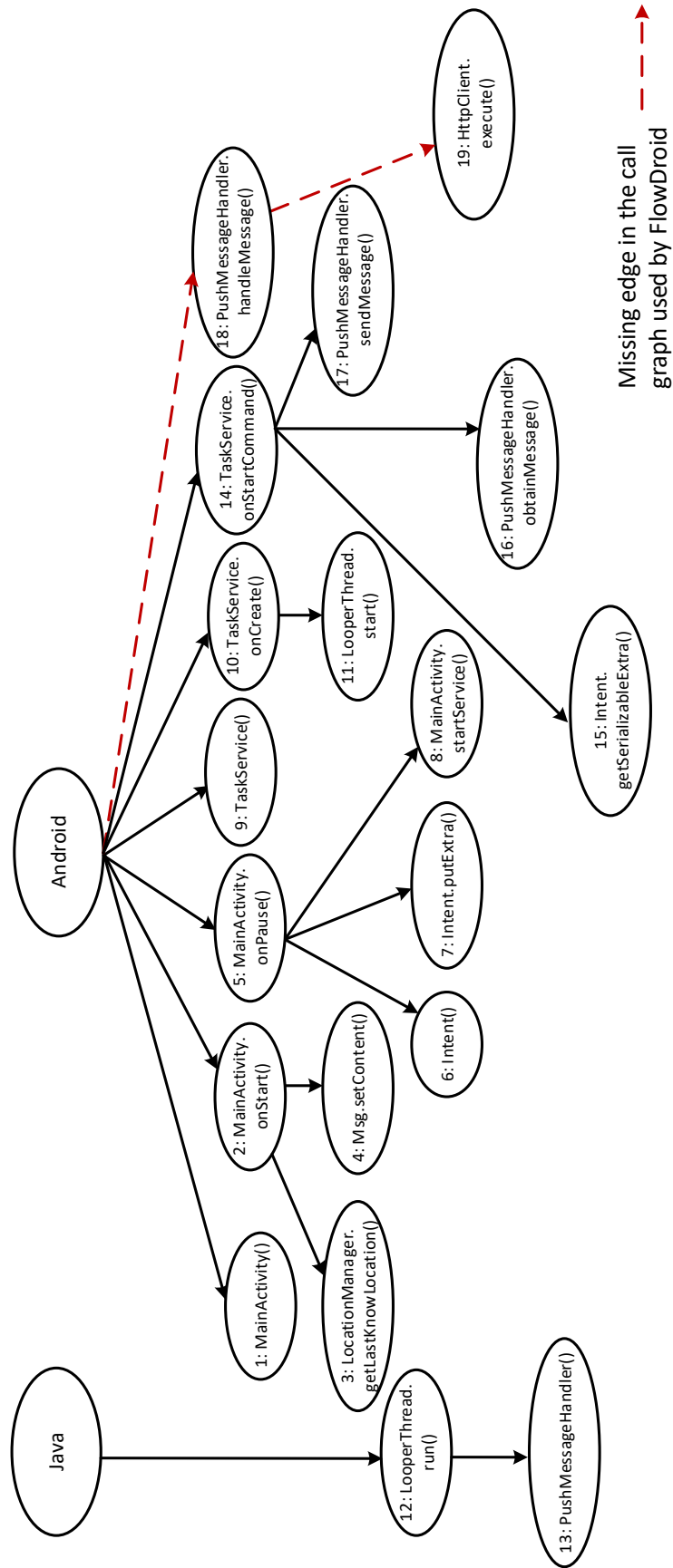


Figure B.1: Essential subgraph of the actual call graph of the motivating example.

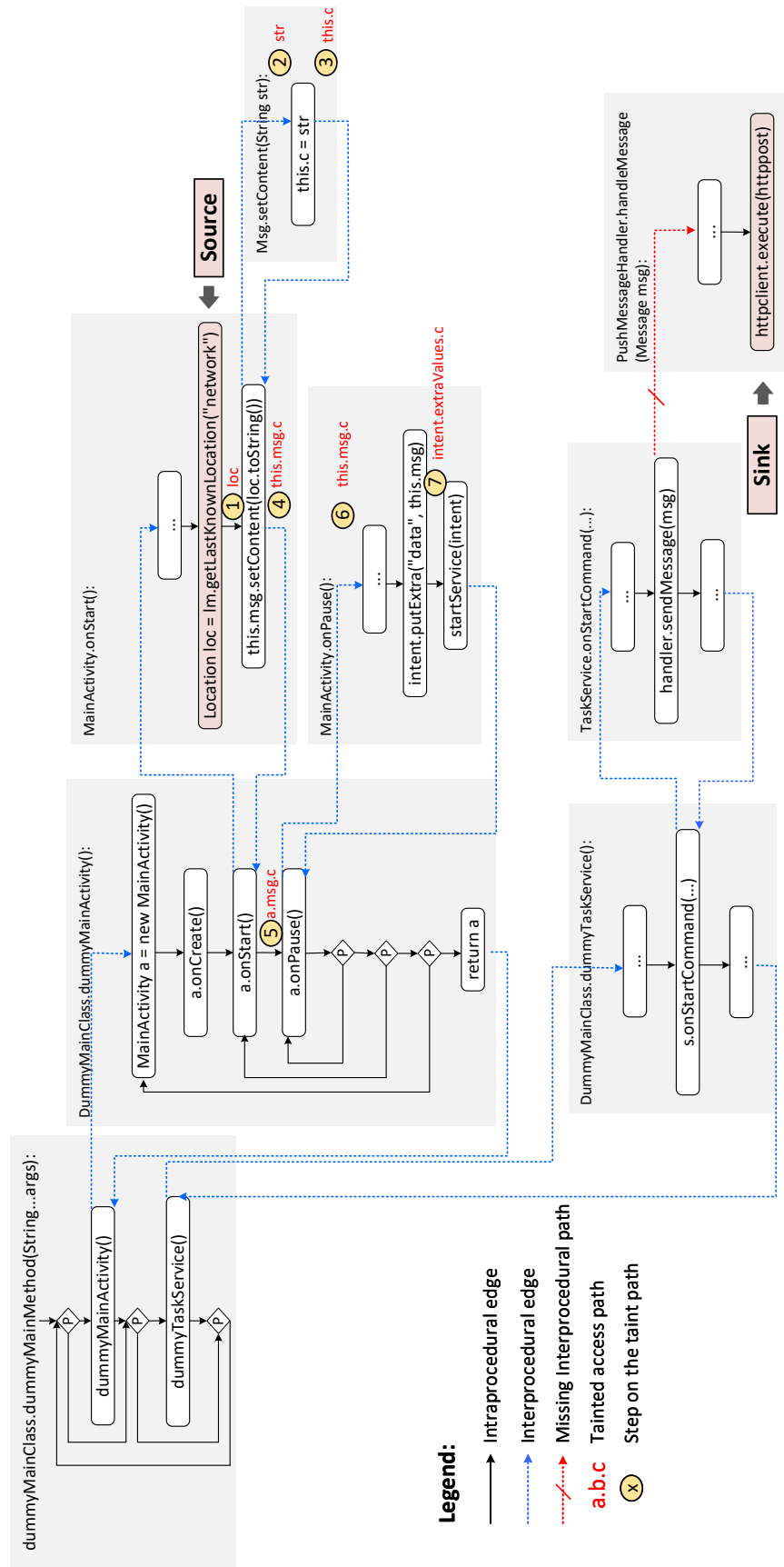


Figure B.2: FlowDROID’s taint analysis on generated ICFG for the motivating example.

B.2 Tables

Table B.1: Detailed evaluation results on DROIDBENCH.

Category: Aliasing						
Benchmark App	FLOWDROID			FLOWDROID ^{Gen}		
	TP	FP	FN	TP	FP	FN
FlowSensitivity1	0	0	0	0	1	0
Merge1	0	1	0	0	1	0
SimpleAliasing1	0	1	0	0	1	0
StrongUpdate1	0	0	0	0	0	0
Category: Android Specific						
Benchmark App	FLOWDROID			FLOWDROID ^{Gen}		
	TP	FP	FN	TP	FP	FN
ApplicationModeling1	1	0	0	1	0	0
DirectLeak1	1	0	0	1	0	0
InactiveActivity	0	0	0	0	1	0
Library2	1	0	0	1	0	0
Obfuscation1	1	0	0	1	0	0
Parcel1	0	0	1	0	0	1
PrivateDataLeak1	0	0	1	0	0	1
PrivateDataLeak2	0	0	1	0	0	1
PrivateDataLeak3	0	0	1	0	0	1
PublicAPIField1	0	0	1	1	0	0
PublicAPIField2	1	0	0	1	0	0
View1	1	0	0	1	0	0
Category: Arrays and Lists						
Benchmark App	FLOWDROID			FLOWDROID ^{Gen}		
	TP	FP	FN	TP	FP	FN
ArrayAccess1	0	1	0	0	1	0
ArrayAccess2	0	1	0	0	1	0
ArrayAccess3	1	0	0	1	0	0
ArrayAccess4	0	0	0	0	0	0
ArrayAccess5	0	1	0	0	0	0
ArrayCopy1	1	0	0	1	0	0
ArrayToString1	1	0	0	1	0	0
HashMapAccess1	0	1	0	0	1	0
ListAccess1	0	1	0	0	1	0
MultidimensionalArray1	1	0	0	1	0	0

Table B.1 Continued.

Category: Callbacks						
Benchmark App	FLOWDROID			FLOWDROID ^{Gen}		
	TP	FP	FN	TP	FP	FN
AnonymousClass1	1	0	0	0	0	1
Button1	1	0	0	0	0	1
Button2	2	1	1	0	0	3
Button3	1	0	0	1	0	0
Button4	1	0	0	0	0	1
Button5	0	0	1	0	0	1
LocationLeak1	1	0	0	0	0	1
LocationLeak2	1	0	0	0	0	1
LocationLeak3	1	0	0	0	0	1
MethodOverride1	1	0	0	1	0	0
MultiHandlers1	0	0	0	0	0	0
Ordering1	0	0	0	0	0	0
RegisterGlobal1	1	0	0	1	0	0
RegisterGlobal2	1	0	0	1	0	0
Unregister1	0	1	0	0	1	0
Category: Emulator Detection						
Benchmark App	FLOWDROID			FLOWDROID ^{Gen}		
	TP	FP	FN	TP	FP	FN
Battery1	1	0	0	1	0	0
Bluetooth1	1	0	0	1	0	0
Build1	1	0	0	1	0	0
Contacts1	1	0	0	1	0	0
ContentProvider1	1	0	0	1	0	0
DeviceId1	1	0	0	1	0	0
File1	1	0	0	1	0	0
IMEI1	0	0	1	0	0	1
IP1	1	0	0	1	0	0
PI1	1	0	0	0	0	1
PlayStore1	1	0	0	1	0	0
PlayStore2	1	0	0	0	0	1
Sensors1	1	0	0	1	0	0
SubscriberId1	1	0	0	1	0	0
VoiceMail1	1	0	0	1	0	0

Table B.1 Continued.

Category: Field and Object Sensitivity						
Benchmark App	FLOWDROID			FLOWDROID ^{Gen}		
	TP	FP	FN	TP	FP	FN
FieldSensitivity1	0	0	0	0	0	0
FieldSensitivity2	0	0	0	0	0	0
FieldSensitivity3	1	0	0	1	0	0
FieldSensitivity4	0	0	0	0	0	0
InheritedObjects1	1	0	0	1	0	0
ObjectSensitivity1	0	0	0	0	0	0
ObjectSensitivity2	0	0	0	0	0	0
Category: General Java						
Benchmark App	FLOWDROID			FLOWDROID ^{Gen}		
	TP	FP	FN	TP	FP	FN
Clone1	0	0	1	0	0	1
Exceptions1	1	0	0	1	0	0
Exceptions2	1	0	0	1	0	0
Exceptions3	0	1	0	0	1	0
Exceptions4	0	0	1	0	0	1
Exceptions5	0	0	1	0	0	1
Exceptions6	1	0	0	1	0	0
Exceptions7	0	0	0	0	0	0
FactoryMethods1	0	0	1	0	0	1
Loop1	1	0	0	1	0	0
Loop2	1	0	0	1	0	0
Serialization1	0	0	1	0	0	1
SourceCodeSpecific1	0	0	1	0	0	1
StartProcessWithSecret1	1	0	0	1	0	0
StaticInitialization1	0	0	1	1	0	0
StaticInitialization2	1	0	0	1	0	0
StaticInitialization3	0	0	1	0	0	1
StringFormatter1	0	0	1	0	0	1
StringPatternMatching1	1	0	0	1	0	0
StringToCharArray1	1	0	0	1	0	0
StringToOutputStream1	1	0	0	1	0	0
UnreachableCode	0	0	0	0	0	0
VirtualDispatch1	1	1	0	0	0	1
VirtualDispatch2	0	1	1	0	1	1
VirtualDispatch3	0	1	0	0	1	0
VirtualDispatch4	0	0	0	0	0	0

Table B.1 Continued.

Category: Inter Component Communication						
Benchmark App	FLOWDROID			FLOWDROID ^{Gen}		
	TP	FP	FN	TP	FP	FN
ActivityCommunication1	1	0	0	1	0	0
ActivityCommunication2	0	0	1	1	1	0
ActivityCommunication3	0	0	1	1	1	0
ActivityCommunication4	0	0	1	1	1	0
ActivityCommunication5	0	0	1	1	1	0
ActivityCommunication6	0	0	1	1	0	0
ActivityCommunication7	0	0	1	1	1	0
ActivityCommunication8	0	0	1	1	1	0
BroadcastTaintAndLeak1	0	0	1	1	0	0
ComponentNotInManifest1	0	0	1	1	1	0
EventOrdering1	0	0	1	0	0	1
IntentSink1	1	0	0	1	0	0
IntentSink2	1	0	0	0	0	1
IntentSource1	0	0	1	0	0	1
ServiceCommunication1	0	0	1	0	0	1
SharedPreferences1	0	0	1	0	0	1
Singletons1	1	0	0	0	0	1
UnresolvableIntent1	0	0	2	2	0	0
Category: Implicit Flow						
Benchmark App	FLOWDROID			FLOWDROID ^{Gen}		
	TP	FP	FN	TP	FP	FN
ImplicitFlow1	1	0	0	1	0	0
ImplicitFlow6	0	0	0	0	0	0
Category: Threading						
Benchmark App	FLOWDROID			FLOWDROID ^{Gen}		
	TP	FP	FN	TP	FP	FN
AsyncTask1	1	0	0	1	0	0
Executor1	1	0	0	1	0	0
JavaThread1	1	0	0	1	0	0
JavaThread2	1	0	0	1	0	0
Looper1	1	0	0	1	0	0
TimerTask1	0	0	1	0	0	1

Table B.1 Continued.

Category: Native						
Benchmark App	FLOWDROID			FLOWDROID ^{Gen}		
	TP	FP	FN	TP	FP	FN
JavaIDFunction	0	0	1	0	0	1
NativeIDFunction	1	0	0	1	0	0
SinkInNativeCode	1	0	0	1	0	0
SinkInNativeLibCode	1	0	0	1	0	0
SourceInNativeCode	1	0	0	1	0	0
Category: Reflection ICC						
Benchmark App	FLOWDROID			FLOWDROID ^{Gen}		
	TP	FP	FN	TP	FP	FN
ActivityCommunication2	0	0	1	1	1	0
AllReflection	0	0	1	0	0	1
OnlyIntent	0	0	1	0	0	1
OnlyIntentReceive	1	0	0	1	0	0
OnlySMS	0	0	1	0	0	1
OnlyTelephony	0	0	1	1	0	0
OnlyTelephonyDynamic	0	0	1	0	0	1
OnlyTelephonyReverse	0	0	1	0	0	1
OnlyTelephonySubstring	0	0	1	0	0	1
SharedPreferences1	0	0	2	0	0	2
Category: Reflection						
Benchmark App	FLOWDROID			FLOWDROID ^{Gen}		
	TP	FP	FN	TP	FP	FN
Reflection1	1	0	0	1	0	0
Reflection2	0	0	1	0	0	1
Reflection3	0	0	1	0	0	1
Reflection4	0	0	1	0	0	1
Reflection5	0	0	1	0	0	1
Reflection6	0	0	1	0	0	1
Reflection7	0	0	1	0	0	1
Reflection8	0	0	1	0	0	1
Reflection9	0	0	1	0	0	1
Category: Self Modification						
Benchmark App	FLOWDROID			FLOWDROID ^{Gen}		
	TP	FP	FN	TP	FP	FN
BytecodeTamper1	0	0	1	0	0	1
BytecodeTamper2	0	0	1	0	0	1
BytecodeTamper3	0	0	1	0	0	1

Table B.1 Continued.


Category: Lifecycle						
Benchmark App	FLOWDROID			FLOWDROID ^{Gen}		
	TP	FP	FN	TP	FP	FN
ActivityEventSequence1	1	0	0	1	0	0
ActivityEventSequence2	1	0	0	1	0	0
ActivityEventSequence3	0	0	1	1	0	0
ActivityLifecycle1	1	0	0	1	0	0
ActivityLifecycle2	1	0	0	1	0	0
ActivityLifecycle3	1	0	0	1	0	0
ActivityLifecycle4	1	0	0	1	0	0
ActivitySavedState1	0	0	1	0	0	1
ApplicationLifecycle1	1	0	0	1	0	0
ApplicationLifecycle2	1	0	0	1	0	0
ApplicationLifecycle3	1	0	0	1	0	0
AsynchronousEventOrdering1	1	0	0	1	0	0
BroadcastReceiverLifecycle1	1	0	0	1	0	0
BroadcastReceiverLifecycle2	0	0	1	0	0	1
BroadcastReceiverLifecycle3	0	0	1	0	0	1
EventOrdering1	1	0	0	1	0	0
FragmentLifecycle1	1	0	0	1	0	0
FragmentLifecycle2	0	0	1	1	0	0
ServiceEventSequence1	0	0	1	1	0	0
ServiceEventSequence2	0	0	1	1	0	0
ServiceEventSequence3	0	0	1	1	0	0
ServiceLifecycle1	1	0	0	1	0	0
ServiceLifecycle2	1	0	0	1	0	0
SharedPreferenceChanged1	0	0	1	0	0	1
Category: Dynamic Loading						
Benchmark App	FLOWDROID			FLOWDROID ^{Gen}		
	TP	FP	FN	TP	FP	FN
DynamicBoth1	1	0	0	1	0	0
DynamicSink1	1	0	0	1	0	0
DynamicSource1	1	0	0	1	0	0
Category: Unreachable Code						
Benchmark App	FLOWDROID			FLOWDROID ^{Gen}		
	TP	FP	FN	TP	FP	FN
SimpleUnreachable1	0	0	0	0	0	0
UnreachableBoth	0	0	1	0	0	1
UnreachableSink1	0	0	1	0	0	1
UnreachableSource1	1	0	0	1	0	0

Table B.2: Detailed evaluation results on TAINTBENCH.

\odot = True positive \odot = False negative \bullet = No flow reported for the app \times = False positive, Empty cell = Flow not reported

Benchmark App	Flow ID															
backflash	1	2	3	4	5	6	7	8	17	19	20	21	22	9	10	11
FLOWDROID	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot			*
FLOWDROID ^{Gen}	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot			*
backflash	12	13	14	15	16	18	23	24								
FLOWDROID	*	*	*													
FLOWDROID ^{Gen}	*	*	*			*										
beita_com*	1	2	3													
FLOWDROID	\bullet	\bullet	\bullet													
FLOWDROID ^{Gen}	\odot	\odot	\odot													
cajino_baidu	1	2	3	4	5	6	7	8	9	10	12	15	11	13	14	
FLOWDROID	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot			*	
FLOWDROID ^{Gen}	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot			*	
chat_hook	1	2	3	4	5	6	7	8	9	10	11	12	13			
FLOWDROID	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot				
FLOWDROID ^{Gen}	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot				
chulia	1	2	3	4												
FLOWDROID	\bullet	\bullet	\bullet	\bullet												
FLOWDROID ^{Gen}	\odot	\odot	\odot	\odot												
death_ring*	1															
FLOWDROID	\odot															
FLOWDROID ^{Gen}	\odot															
dsencrypt*	1															
FLOWDROID	\bullet															
FLOWDROID ^{Gen}	\bullet															
exprespam	1	2														
FLOWDROID	\bullet	\bullet														
FLOWDROID ^{Gen}	\odot	\odot														
fakeappstore	1	2	3													
FLOWDROID	\bullet	\bullet	\bullet													
FLOWDROID ^{Gen}	\odot	\odot	\odot													
fakebank*	1	2	3	4	5											
FLOWDROID	\bullet	\bullet	\bullet	\bullet	\bullet											
FLOWDROID ^{Gen}	\odot	\odot	\odot	\odot	\odot											
fakedaum	1	2														
FLOWDROID	\bullet	\bullet														
FLOWDROID ^{Gen}	\odot	\odot														
fakemart	1	2														
FLOWDROID	\bullet	\bullet														
FLOWDROID ^{Gen}	\bullet	\bullet														
fakeplay	1	2														
FLOWDROID	\bullet	\bullet														
FLOWDROID ^{Gen}	\bullet	\bullet														

Table B.2 Continued.

⊕ : True positive ⊖ : False negative  : No flow reported for the app * : False positive, Empty cell = Flow not reported

x : Expected flow x : Unexpected flow

































Benchmark App	Flow ID																		
faketaobao	1	2	3	4															
FLOWDROID																			
FLOWDROID ^{Gen}	⊕	⊕	⊖	⊕															
godwon_samp	1	2	3	4	5	6													
FLOWDROID																			
FLOWDROID ^{Gen}	⊕	⊕	⊖	⊖	⊕	⊕													
hummingbad*	1	2																	
FLOWDROID																			
FLOWDROID ^{Gen}																			
jollyserv	1																		
FLOWDROID	⊖																		
FLOWDROID ^{Gen}	⊖																		
overlaylocker2*	1	2	3	4	5	6	7												
FLOWDROID	⊖	⊖	⊖	⊖	⊖	⊖	⊖												
FLOWDROID ^{Gen}	⊖	⊖	⊖	⊖	⊖	⊖	⊖												
overlaylocker2*	8	9	10	11	12	13	14	15	16	17	18	19							
FLOWDROID																			
FLOWDROID ^{Gen}									*										
overlay_android*	1	2	3	4	5	6													
FLOWDROID	⊖	⊖	⊖	⊖	*	*													
FLOWDROID ^{Gen}	⊕	⊖	⊖	⊖															
phospy	1	2	3	4	5														
FLOWDROID	⊕	⊖	*																
FLOWDROID ^{Gen}	⊕	⊕		*	*														
proxy_samp	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	17			
FLOWDROID	⊕	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖			
FLOWDROID ^{Gen}	⊕	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖			
proxy_samp	18	16	19	20															
FLOWDROID	⊖																		
FLOWDROID ^{Gen}	⊖																		
remote_control*	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15				
FLOWDROID	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖				
FLOWDROID ^{Gen}	⊖	⊖	⊕	⊕	⊖	⊖	⊕	⊕	⊖	⊖	⊖	⊖	⊕	⊕	⊕				
remote_control*	16	17																	
FLOWDROID	⊖	⊖																	
FLOWDROID ^{Gen}	⊖	⊕																	
repane	1																		
FLOWDROID																			
FLOWDROID ^{Gen}																			
roidsec	1	2	3	4	5	6													
FLOWDROID																			
FLOWDROID ^{Gen}																			
samsapo	1	2	3	4	5														
FLOWDROID																			
FLOWDROID ^{Gen}	⊖	⊖	⊖	⊖	*														

Table B.2 Continued.

\odot = True positive \odot = False negative \odot = No flow reported for the app \odot = False positive, Empty cell = Flow not reported
 \odot : Expected flow \odot : Unexpected flow

Benchmark App	Flow ID															
save_me	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
FLOWDROID	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot
FLOWDROID ^{Gen}	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot
save_me	17	18	21	22	23	24	25	30	31	19	20	26	27	28	29	
FLOWDROID	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot
FLOWDROID ^{Gen}	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot
scipix	1	2	3													
FLOWDROID	\odot	\odot	\odot													
FLOWDROID ^{Gen}	\odot	\odot	\odot													
slocker*	1	2	3	4	5											
FLOWDROID	\odot	\odot	\odot	\odot	\odot											
FLOWDROID ^{Gen}	\odot	\odot	\odot	\odot	\odot											
smssend*	1	2	3	4	5											
FLOWDROID	\odot	\odot	\odot	\odot	\odot											
FLOWDROID ^{Gen}	\odot	\odot	\odot	\odot	\odot											
smssilience*	1	2	3	4												
FLOWDROID	\odot	\odot	\odot	\odot												
FLOWDROID ^{Gen}	\odot	\odot	\odot	\odot												
smsstealer*	1	2	3	4	5											
FLOWDROID	\odot	\odot	\odot	\odot	\odot											
FLOWDROID ^{Gen}	\odot	\odot	\odot	\odot	\odot											
sms_google	1	2	3	4												
FLOWDROID	\odot	\odot	\odot	\odot												
FLOWDROID ^{Gen}	\odot	\odot	\odot	\odot												
sms_send*	1	2	3	6	7	8	4	5								
FLOWDROID	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot								
FLOWDROID ^{Gen}	\odot	\odot	\odot	\odot	\odot	\odot	\odot	\odot								
stels*	1	2	3													
FLOWDROID	\odot	\odot	\odot													
FLOWDROID ^{Gen}	\odot	\odot	\odot													
tetus	1	2														
FLOWDROID	\odot	\odot														
FLOWDROID ^{Gen}	\odot	\odot														
the_interview_movieshow	1															
FLOWDROID	\odot															
FLOWDROID ^{Gen}	\odot															
threatjapan_uracto	1	2														
FLOWDROID	\odot	\odot														
FLOWDROID ^{Gen}	\odot	\odot														
vibleaker*	1	2	3	4												
FLOWDROID	\odot	\odot	\odot	\odot												
FLOWDROID ^{Gen}	\odot	\odot	\odot	\odot												
xbot*	1	2	3													
FLOWDROID	\odot	\odot	\odot													
FLOWDROID ^{Gen}	\odot	\odot	\odot													

Supplementary Material of Chapter 5

C.1 Comparison Between MAGPIEBRIDGE-Based Approach and Plugin-Based Approach

While MAGPIEBRIDGE enables analyses to run in a larger set of IDEs, the question remains of how the support in any specific IDE using MAGPIEBRIDGE compares to a custom-built plugin for that same IDE. Because most analysis tools do not have integration with most IDEs, we are going to focus our comparison on one existing combination: the CogniCrypt plugin for Eclipse. Afterwards, we discuss in more general terms the range of functionality exploited by custom plugins that is supported by LSP. Note that the comparison in this section is based on the version of MAGPIEBRIDGE when it was first published at the ECOOP conference [LDB19].

C.1.1 Comparison Between MAGPIEBRIDGE-Based CogniCrypt and CogniCrypt Eclipse Plugin

The CogniCrypt Eclipse Plugin [KNR⁺17] consists of two components: code generation, which generates secure implementations for user-defined cryptographic programming tasks, and cryptographic misuse detection, which runs static code analysis in the background and reports insecure usage of cryptographic APIs. MAGPIEBRIDGE focuses on analysis, and so we do not consider the code-generation component here. For comparison, we integrated the static crypto analysis of CogniCrypt with MAGPIEBRIDGE into Eclipse IDE.

Figure C.1 and Figure C.2 are screenshots in which the original CogniCrypt Eclipse Plugin reports insecure crypto warnings. In comparison, Figure C.3 shows our CogniCrypt-integration with MAGPIEBRIDGE. Figure C.1 shows two buttons that CogniCrypt adds to the toolbar: “Generate Code For Cryptographic Task” and “Apply CogniCrypt Misuse to Selected Project”. By clicking the latter, one triggers the misuse detection using the plugin in its default configuration. The plugin can also be configured to trigger the analysis whenever a Java file is saved. On the other hand, MAGPIEBRIDGE-based CogniCrypt starts the analysis automatically whenever a Java file is opened or saved. In either case, after the analysis has been run, any detected misuses are indicated in Eclipse in several ways, which the corresponding numbers show in Figure C.1 and Figure C.3:

1. In the Package Explorer view, the error ticks appear on the affected Java element and their parent elements.
2. In the Problems view, the detected misuses are listed as errors.

C.1 COMPARISON BETWEEN MAGPIEBRIDGE-BASED APPROACH AND PLUGIN-BASED APPROACH

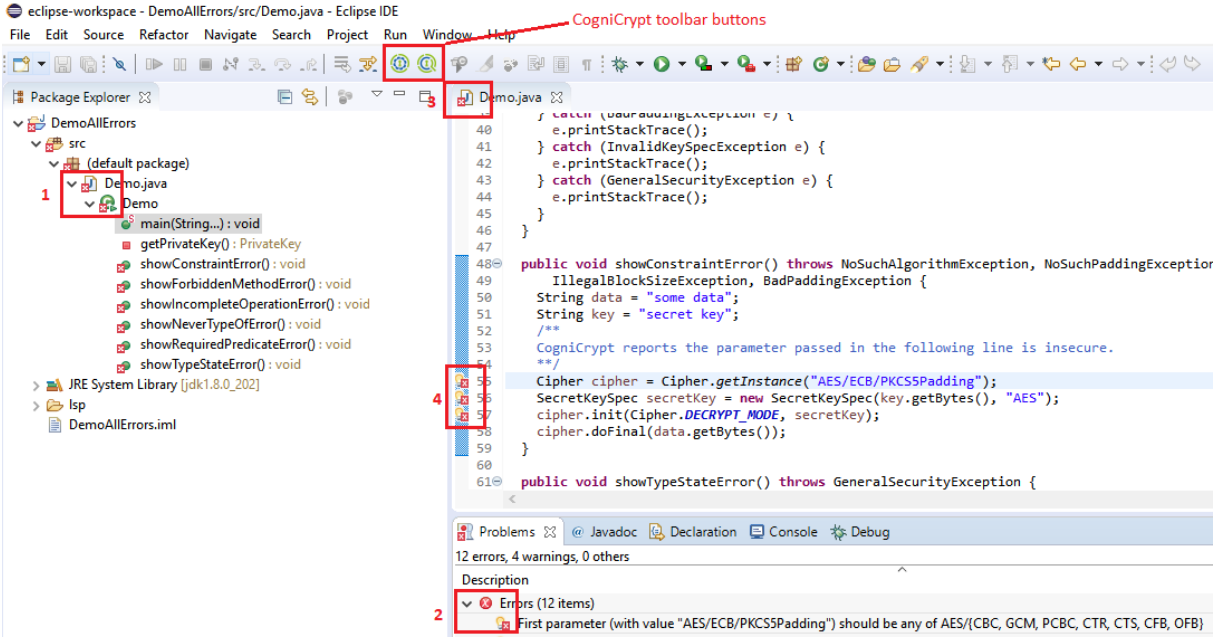


Figure C.1: The appearance of CogniCrypt Eclipse Plugin.

3. The editor tab is annotated with an error marker.

4. In the editor's vertical ruler / gutter, an error marker is displayed near the affected line.

As shown in Figure C.2, one can hover over an error marker next to the affected line to view the description of the misuse. The appearance of the MAGPIEBRIDGE-based and plugin-based CogniCrypt is rather similar, with just a few differences:

- MAGPIEBRIDGE-based CogniCrypt does not change the appearance of the IDE. To work with the MagpieServer which runs the crypto analysis, end-users do not have to do anything different. The analysis runs automatically whenever a Java file is opened or saved by an end-user. In contrast, in the Eclipse Plugin, one can trigger the analysis manually, or (optionally) have it started automatically whenever a file is saved.
- Results are indicated similarly in the CogniCrypt Eclipse Plugin MAGPIEBRIDGE-based CogniCrypt; however, in MAGPIEBRIDGE-based CogniCrypt in addition to the error markers, squiggly lines appear under the affected lines.
- In MAGPIEBRIDGE-based CogniCrypt, the hover message also includes a quick fix that can replace the insecure parameter `AES/ECB/PKCS5Padding` with a secure parameter `AES/CBC/PKCS5Padding` automatically. Since MAGPIEBRIDGE preserves the precise source code position from the WALA source-code front end, e.g., the exact code range (starting/ending line/column numbers) of each parameter of a method call, we were able to build such quick fix easily with the `codeAction` feature supported by LSP. Such quick fix is not available in the CogniCrypt Eclipse Plugin, although the warning message already indicates what a secure parameter should look like.

Another difference is that, since MAGPIEBRIDGE does not add buttons to the IDE, it needs to invoke the analysis automatically. When the end-user changes the opened file, the MagpieServer clears the warnings when it receives the `didChange` notification from the IDE. The analysis is then restarted whenever the end-user saves the file, i.e., the MagpieServer receives a `didSave`

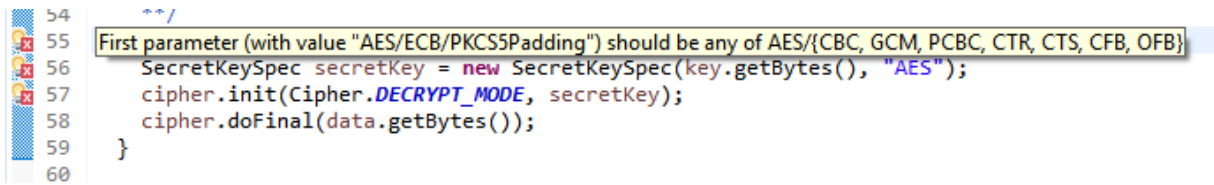


Figure C.2: CogniCrypt Eclipse Plugin: insecure crypto warning message shown by hovering.

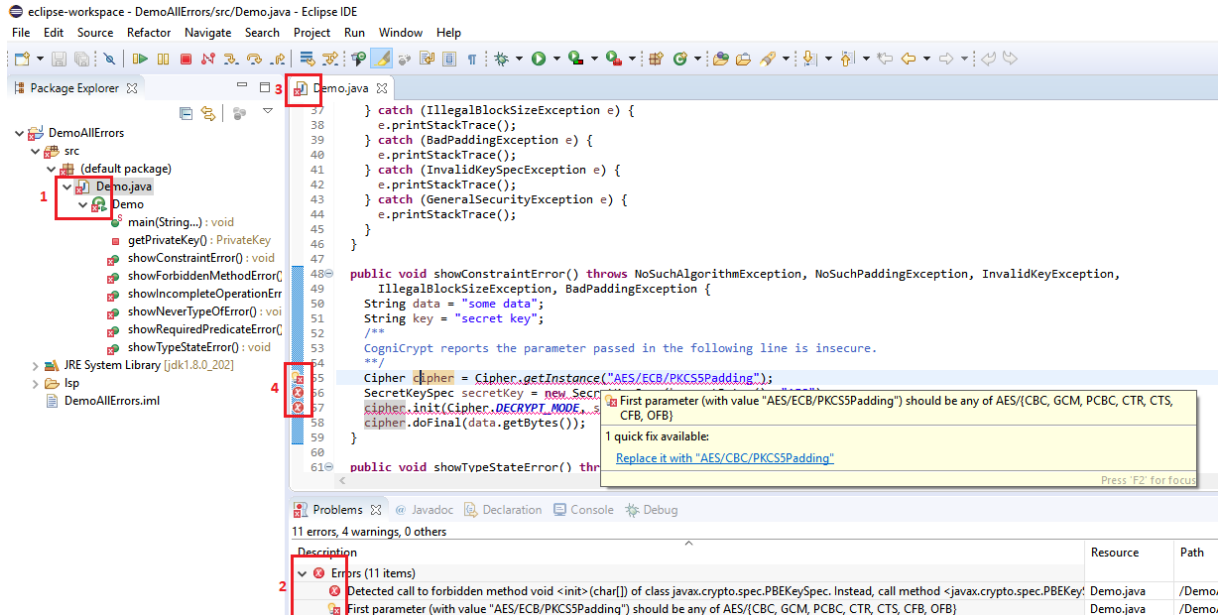


Figure C.3: The appearance of MAGPIEBRIDGE-based CogniCrypt: insecure crypto warning message and quick fix shown by hovering.

notification. Once the MagpieServer receives the notification from the Eclipse IDE, it resolves the source code and library code path required for the inter-procedural crypto analysis. This analysis is all asynchronous, so that the analysis always runs in the background and updated error messages are shown once they are available. If they want to, end-users have the ability to connect and disconnect the MagpieServer at runtime, e.g., via “Preferences” in Eclipse IDE.

C.1.2 Comparison to Other Plugin-Based Approaches

As shown in Figure C.4, LSP offers a set of UI features to present the analysis results to end-users that are sufficient to capture the majority of UI features used in a range of existing plugins for a single analysis tool in a specific IDE. Most of the plugin approaches we identified were implemented as Eclipse plugins (Cheetah[DAL⁺17a], SpotBugs[Tea17] and ASIDE[XCLM11]), but some of them were created for other popular IDEs such as Android Studio (FixDroid[NWA⁺17]), IntelliJ (wIDE[MSHN17]) and Visual Studio (GhostFactor[GM14]). Figure C.4 shows the comparison between features that can be supported with LSP to features supported by these existing plugin approaches.

Some plugins do use IDE features that are not explicitly supported by LSP; however, there are often analogs in LSP that could be used instead. For instance, Cheetah uses a custom view, essentially a separate window panel in the IDE, to show an example data-flow trace for a bug; in LSP, related information capturing a trace can be attached to problems as illustrated in Figure 5.18. Other uses of custom views and wizards are mainly for analysis configuration.

C.1 COMPARISON BETWEEN MAGPIEBRIDGE-BASED APPROACH AND PLUGIN-BASED APPROACH

Simple forms of such analysis configuration could be supported by the message protocol in LSP.

One minor feature unsupported by LSP appeared in the plugins: customized icons (see Figure C.5, Figure C.6 and Figure C.7) are not supported by the LSP-based approach, since that requires changes to the appearance of the IDEs, which LSP intends not to. Although studies have shown customized icons are useful to catch end-users' attention [NWA⁺17, XCLM11, SDOF07], it is not clear if it is more effective than the default error icon supported by each editor.

As we can see in Figure C.4, the major features such as hover tips, warning marker and code highlighting, which are supported by a majority of the plugins, can be supported by an LSP-based approach. However, LSP support varies across IDEs, both in what features are handled and how they are shown. In LSP, hover tips are specified as the `hover` request sent from the client to the server, warning marker can be realized by the `publishDiagnostics` notification and `documentHighlight` is the corresponding request for code highlighting. However, the implementation of `documentHighlight` varies from editor to editor, since the specification for this feature in LSP is unclear. Most plugins listed in Figure C.4 support code highlighting. This features means changing the background color of affected lines of code as shown in Figure C.5, Figure C.6 and Figure C.7. While Visual Studio Code limits this feature to only highlights all references to a symbol scoped in a file, sublime Text choses an underline for highlighting (see Figure C.9). In addition, there is no possibility with LSP to specify the background color used in this feature, all editors have their pre-defined colors.

Some advanced features such as code actions (we have shown quick fix with MAGPIEBRIDGE-based CogniCrypt), pop-ups and code change detections can also be supported by LSP. There are two interfaces (`showMessage` and `showMessageRequest`) defined in LSP which are implemented as pop-up windows in editors. Figure C.10 shows a message sent from a server to the Eclipse IDE that is displayed in a pop-up window. Where more interactions are required, the interface `showMessageRequest` allows to pass actions and wait for an answer from the client. Figure C.11 shows a pop-up windows with a message and available actions in Visual Studio Code.

Features that are not supported by LSP for now can be extended to LSP in the future, since LSP is a moving target with ever-growing functionality and support. One just has to keep in mind that, as the LSP is extended, the IDEs/editors that support it, might require extensions

Feature Comparison								
Feature	LSP-based Approach	FixDroid (Android Studio)	wIDE (IntelliJ)	GhostFactor (Visual Studio)	Cheetah (Eclipse)	SpotBugs (Eclipse)	ASIDE (Eclipse)	# Plugins support the feature
Warning Marker	✓	✓	☐	✓	✓	✓	✓	5
Code Highlighting	✓	✓	✓	☐	✓	☐	✓	4
Code Actions (quick fix, code)	✓	✓	☐	✓	☐	☐	✓	3
Hover Tips	✓	✓	✓	✓	✓	✓	✓	6
Pop-ups	✓	✓	✓	☐	☐	☐	☐	2
Code Change Detection	✓	✓	☐	☐	☐	☐	✓	2
Customized Icons	☐	✓	☐	☐	✓	☐	✓	3
Customized Views	☐	☐	✓	☐	✓	☐	✓	3
Customized Wizards	☐	☐	✓	☐	☐	☐	☐	1

Figure C.4: Feature comparison between LSP-based approach and other plugin-based approaches.

as well.

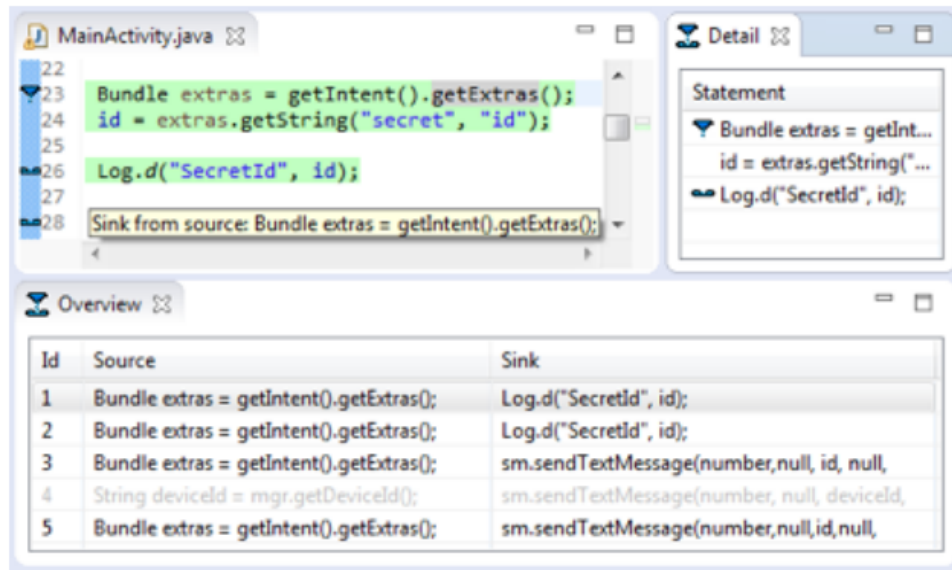


Figure C.5: Cheetah: code highlighting, hover tips, customized icon and views.

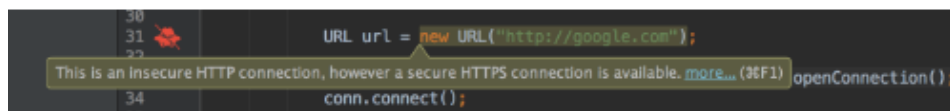


Figure C.6: FixDroid: code highlighting, hover tips and customized icon.

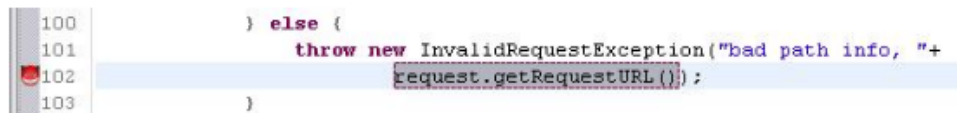


Figure C.7: ASIDE: code highlighting and customized icon.

C.1 COMPARISON BETWEEN MAGPIEBRIDGE-BASED APPROACH AND PLUGIN-BASED APPROACH

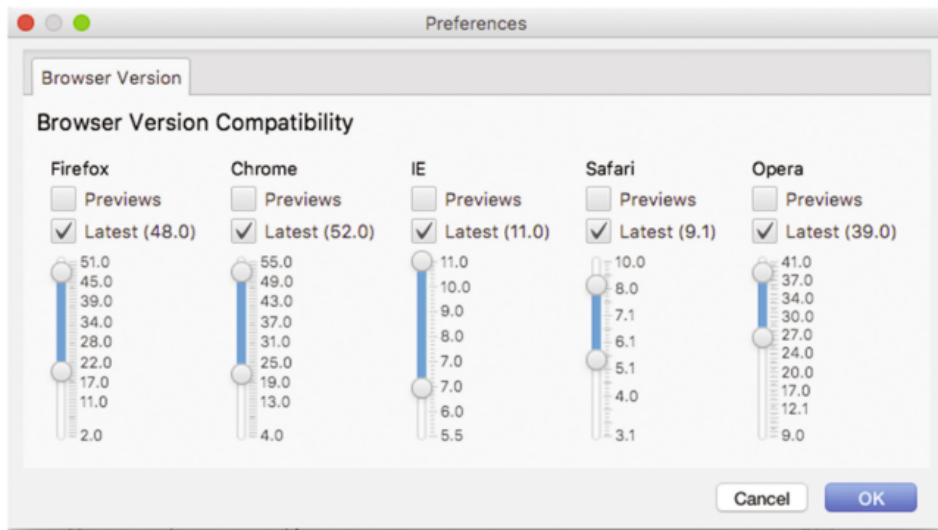


Figure C.8: wIDE: customized wizard.

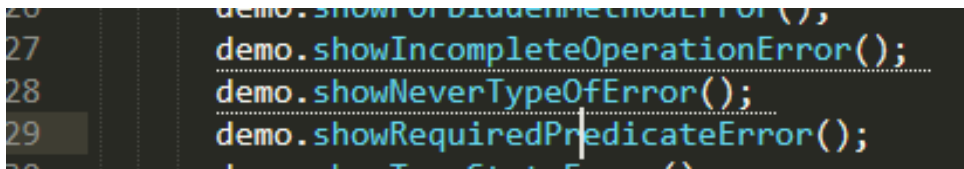


Figure C.9: Highlighting in Sublime Text.

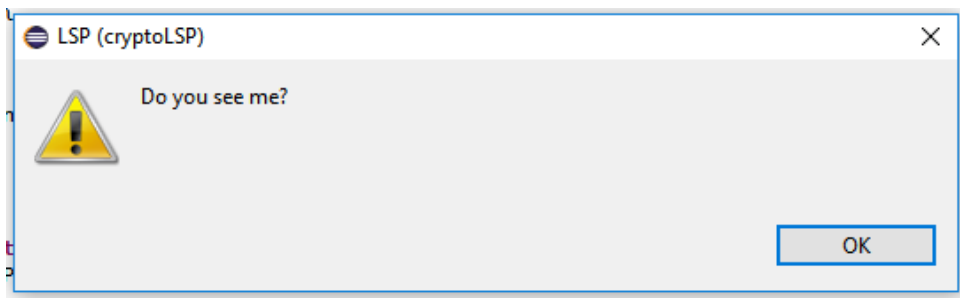


Figure C.10: Pop-up in Eclipse.

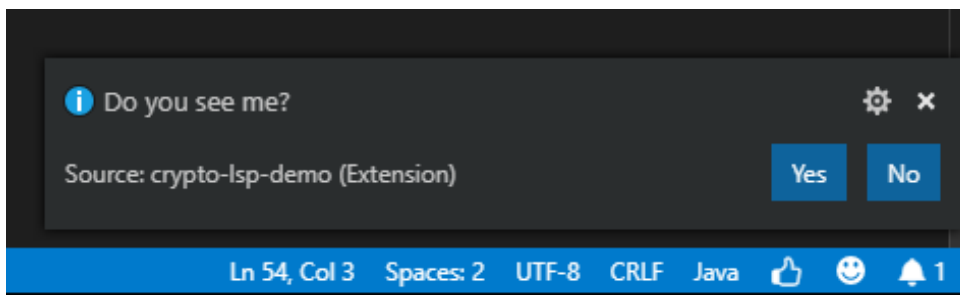


Figure C.11: Pop-up with actions in Visual Studio Code.

Supplementary Material of Chapter 6

D.1 Script For User Interviews

Below is a script we used in user interviews introduced in Section 6.2, Chapter 6:

1. Tell the participant: We want to design an IDE integration of a cloud-based SAST tool that you would like to use, so we would like to you to tell us what your expectations are. The SAST tool performs complex static analyses in the cloud and it is time-consuming.
2. Talk through the prepared code review and ask the participant to demonstrate how issue was fixed in IDE.
 - Did you fix the issues mentioned in the recommendations provided by CODEGURU REVIEWER
 - When and how did you do it?
 - Can you demonstrate it in your IDE?
3. Talk about experience with other static analysis tools
 - Can you tell us about your experience with other static analysis tools?
 - How do you use it in your daily workflow?
 - Can you demonstrate it?
4. Tell the participant: We want to ask you a few questions regarding how the IDE integration is expected to work. Please demonstrate your ideas in your IDE if it is possible.
 - Since the analysis is running in the cloud, how do you expect the code to be uploaded?
 - When would you trigger the analysis?
 - How should it be triggered?
 - Do you think you would want to interrupt the analysis? Why?
 - When and how should the analysis result be retrieved from the cloud?
 - Which parts of the result do you expect to see?
 - Where and how should the analysis result be shown in your IDE?
 - Should the findings be classified? How should they be classified?
 - How should outdated result be shown due to local code changes?
 - What kinds of UX features do you expect?
 - What kinds of features do you think will be helpful for your team?

D.2 Codes

Table D.1: Codes

Codes from User Interviews		
Theme	Code	Definition
Analysis Triggering Mechanism	Code Uploading	How should the code be uploaded
	Analysis Triggering	How should the analysis be triggered
	Partial Code Uploading	Expectation on uploading partial code
	Analysis Termination	Wether analysis should be able to interrupted/terminated
	Time of Uploading	When should the code be uploaded
	Time of Analysis Triggering	When is the time to trigger analysis
Result Retrieval Mechanism	Time of Looking for Issues	When to look for issues
	Acquisition of Warnings	How should warnings be obtained from the cloud
	Partial Code Uploading	Expectation on uploading partial code
	Analysis Scope	Which parts of the code should be analyzed and which warnings should be shown
	Analysis Time	How long the time should take
	Time of Warning Display	When should the warnings be displayed
Result Display Mechanism	Warning Appearance	Where should the warnings be displayed and how should the warnings look like?
	Warning Classification	Wether and how should the warnings be classified or labelled with severities
	Display of Invalid Warnings	Wether and how should the invalid (old) warnings be displayed due to latency
	Configuration	What configuration options should be provided
UX Features	Other IDE Features	Other common IDE features such as fixes, warning suppression
	Team Features	Features that are useful for the team
	Habit	Working habit
	Time of Fixing Issues	When to fix issues
Workflow Integration	Time of Looking for Issues	When to look for issues
	IDE Usage	Which IDE the developer uses
	Other Static Analysis Tools	Experience with other static analysis tools

Table D.1 Continued.

Codes from Usability Tests		
1. Dimension		
Theme	Code	Definition
Analysis Triggering Mechanism	Code Uploading (IDE/Web)	See definition above
	Time of Analysis Triggering (IDE/Web)	See definition above
	Analysis Triggering (IDE/Web)	See definition above
	Acquisition of Warnings (IDE)	See definition above
Result Retrieval Mechanism	Analysis Time (IDE/Web)	See definition above
	Analysis Scope (IDE/Web)	See definition above
	Time of Warning Display (IDE/Web)	See definition above
Result Display Mechanism	Warning Classification (IDE/Web)	See definition above
	Warning Appearance (IDE/Web)	See definition above
	Display of Invalid Warnings (IDE)	See definition above
	Quality of the analysis result	Comments on the quality of analysis result
UX Features	View switching (Web)	Developers mentioned switching between IDE and web browser
	Other IDE Features	See definition above
	Other Web Features	Other features for Web mentioned by developers
	Configuration (IDE/Web)	See definition above
Workflow Integration	Habit	See definition above
	Other Static Analysis Tools	See definition above
	Time of Looking for Issues (IDE/Web)	See definition above
	IDE Usage	See definition above
Issues and Causes	Perception on workflow (IDE/Web)	Developers talked about their perception on the workflow
	Tool preference	Developers talked about their preference in IDE or Web
	Why and when to give feedback	Developers talked about why and when they gave feedback to a finding (thumb up/down)
	Don't read (IDE/Web)	Developers mentioned that they didn't read pop-ups or user guides
	Tutorial required (IDE/Web)	Developers mentioned that they didn't understand how a feature works
	Technical issue (IDE/Web)	Technical issue caused by the tools, e.g. failed to push code due to configuration of git credentials
	Confusion (IDE/Web)	Developers mentioned that they were confused.
	Study setup	Developers mentioned issues due to study setup, e.g. too many issues for too less time
2. Dimension	Other comment (IDE/Web)	Other comments given by developers
	Positive feedback	Positive feedback from developers
	Negative feedback	Negative feedback from developers

D.3 Survey For Usability Tests

Demographic Questions:

1. How many years of professional coding experience in Java do you have? (choose one)
 - Less than 2 years
 - 2 to 5 years
 - 5 to 10 years
 - More than 10 years
2. Have you used CODEGURU REVIEWER before? (choose one)
 - Yes
 - No
3. Do you use Visual Studio Code to write code? (choose one)
 - Yes
 - No
4. Have you written an application with the AWS SDK for Java before? (choose one)
 - Yes
 - No

Exit-survey (questions regarding the test sessions):

1. The task that I just completed was ... (choose one)
1 (Very difficult) 2 3 4 5 (Very easy)
2. I think that I would like to use this system frequently. (choose one)
1 (Strongly disagree) 2 3 4 5 (Strongly agree)
3. I found the system unnecessarily complex. (choose one)
1 (Strongly disagree) 2 3 4 5 (Strongly agree)
4. I thought the system was easy to use. (choose one)
1 (Strongly disagree) 2 3 4 5 (Strongly agree)
5. I think that I would need the support of a technical person to be able to use this system. (choose one)
1 (Strongly disagree) 2 3 4 5 (Strongly agree)
6. I found the various functions in this system were well integrated. (choose one)
1 (Strongly disagree) 2 3 4 5 (Strongly agree)
7. I thought there was too much inconsistency in this system. (choose one)
1 (Strongly disagree) 2 3 4 5 (Strongly agree)
8. I would imagine that most people would learn to use this system very quickly. (choose one)
1 (Strongly disagree) 2 3 4 5 (Strongly agree)

D.3 SURVEY FOR USABILITY TESTS

9. I found the system very cumbersome to use. (choose one)
1 (Strongly disagree) 2 3 4 5 (Strongly agree)
10. I felt very confident using the system. (choose one)
1 (Strongly disagree) 2 3 4 5 (Strongly agree)
11. I needed to learn a lot of things before I could get going with this system. (choose one)
1 (Strongly disagree) 2 3 4 5 (Strongly agree)
12. Which features do you think that were useful (multiple choices)? (select all that apply)
IDE:

- Fetch recommendations of a matching scan to Visual Studio Code
- List recommendations in a problems view in Visual Studio Code
- Direct navigation to code by clicking on a recommendation
- Highlight relevant code and display recommendations in hover messages
- Recommendations are classified with weakness types
- Create repository analysis from Visual Studio Code
- Clear recommendations in Visual Studio Code
- Search recommendations in the problems view
- Customization of rule set filtering
- Customization of scope filtering
- Warning suppression
- Give feedback (thumbs up/down)
- Other

Web:

- List scans in the AWS Console
 - List recommendations in the AWS Console
 - Direct navigation to code in Git repository by clicking on a recommendation
 - Create repository analysis from the AWS Console
 - Search scans in the AWS Console
 - Search recommendations in the AWS Console
 - Give feedback (thumbs up/down)
 - Other
13. If you chose “Other” feature in the last question, tell us which
14. How many issues did you fix? _____
15. How did you know that you fixed the issues? _____
16. How did you decide which issues to fix first? _____
17. How likely are you to use CODEGURU REVIEWER with our Visual Studio Code extension/AWS Console for such tasks in the future? (choose one)
Very unlikely Unlikely Likely Very likely
18. Tell us what needs to be improved. _____