



UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Secure Software Engineering

Adapting Taint Analyses for Detecting Security Vulnerabilities

by
GORAN PISKACHEV

Doctoral Thesis
Submitted in partial fulfillment of the requirements for the degree of
Doktor der Naturwissenschaften (Dr. rer. nat.)

Advisor:
Prof. Dr. Eric Bodden

Paderborn, February 1, 2023

Abstract. The first applications of static code analysis established the foundation for compiler optimization techniques. Recently, applications in detecting security vulnerabilities have gained attention, especially in industry, where security has become more relevant for most digital companies. With that, static analysis tools, first mostly known for code refactoring and code-smell detection, evolved into so-called, *static analysis security testing* (SAST) tools. Many existing SAST tools use as their core engine a *taint analysis* that can be configured to detect different security vulnerabilities.

Previous empirical studies on static analysis tools reported that users face usability issues due to many false warnings, missing essential findings, or long-running times. However, if the users configure the tools properly, these issues can be avoided. Many tool vendors enable the tool configuration via specification of custom taint-flow in domain-specific languages. However, the configuration is a manual step, which requires highly skilled users, who need (1) knowledge of the static analysis, (2) a good understanding of the target program being analyzed, and (3) a solid understanding of security vulnerabilities. In practice, the users have different backgrounds (software developers, software architects, security experts, managers), and lack at least one of these requirements. Therefore, when used, the tools run primarily in their default configurations leading to the usability issues stated earlier, resulting in low acceptance and low popularity among the users.

In this thesis, we propose methods and tools that (semi-)automate the adaptation of taint analysis to the target programs and enable a broader range of users to configure the tools according to their needs. In particular, we focus on software developers as primary users, and security engineers as secondary users.

Initially, we conduct an empirical study to understand better the needs of the users who already have experience with taint analysis or SAST tools. Our study confirms that the users' background is diverse, yet most users are motivated to interact with the SAST tools to configure and adapt them to the target program. Based on this, we propose SWAN, a fully automatic machine-learning approach for inferring security-relevant methods from the target program required for the specification of taint-flows. Our experimental results show that SWAN achieves high precision and recall for a set of popular security vulnerability types. Its limitations can be improved via SWAN_{ASSIST}, a semi-automated active machine-learning approach that incorporates feedback from the user to improve the results of SWAN. Finally, to provide a usable way for configuring the taint analysis with new security vulnerabilities, we propose *fluentTQL*, the first developer-oriented domain-specific language for specifying taint-flows. In a user study, *fluentTQL* shows to have excellent usability in comparison to a commercial state-of-the-art DSL, CODEQL. *fluentTQL* is independent of the underlying taint analysis and is implemented as part of SECUCHECK, our taint analysis tool running in multiple integrated development environments, as preferred by most software developers, confirmed in our empirical study.

The achieved results move forward the design and use of taint analysis tools. We show that the adaptation of the tools can be improved for the end users via machine-learning and user-centric design.

Zusammenfassung. Die ersten Anwendungen der statischen Codeanalyse bildeten die Grundlage für Compiler-Optimierungstechniken. Seit den letzten Jahren haben Anwendungen zum Aufspüren von Sicherheitslücken an Aufmerksamkeit gewonnen, vor allem in der Industrie, wo die Sicherheit für die meisten digitalen Unternehmen an Bedeutung gewonnen hat. Damit haben sich die statischen Analysewerkzeuge, die zunächst vor allem für Code-Refactoring und Code-Smell-Erkennung bekannt waren, zu so genannten Static Analysis Security Testing (SAST)-Werkzeugen entwickelt. Viele bestehende SAST-Tools verwenden als Kern eine Taint-Analyse, die so konfiguriert werden kann, dass sie verschiedene Sicherheitslücken erkennt.

Frühere empirische Studien über statische Analysewerkzeuge berichteten, dass Benutzer aufgrund einer hohen Anzahl falscher Warnungen, fehlender wichtiger Ergebnisse oder langer Laufzeiten Probleme mit der Benutzerfreundlichkeit haben. Wenn die Benutzer die Werkzeuge richtig konfigurieren, können diese Probleme vermieden werden. Viele Tool-Anbieter ermöglichen die Tool-Konfiguration über die Spezifikation eines benutzerdefinierten Taint-Flows in domänenspezifischen Sprachen. Diese Konfiguration ist jedoch ein manueller Schritt, der hochqualifizierte Benutzer erfordert, die (1) Kenntnisse in statischer Analyse, (2) ein gutes Verständnis des zu analysierenden Zielprogramms und (3) ein solides Verständnis von Sicherheitslücken haben müssen. In der Praxis haben die Benutzer unterschiedliche Hintergründe (Software Entwickler, Softwarearchitekten, Sicherheitsexperten, Manager) und ihnen fehlt mindestens eine dieser Voraussetzungen. Daher laufen die Tools bei ihrer Verwendung meist in ihren Standardkonfigurationen, was zu den bereits erwähnten Problemen mit der Benutzerfreundlichkeit führt und eine geringe Akzeptanz und Beliebtheit bei den Benutzern zur Folge hat.

In dieser Arbeit schlagen wir Methoden und Werkzeuge vor, die die Anpassung der Taint-Analyse an die Zielprogramme (halb-)automatisieren und es einem breiteren Benutzerkreis ermöglichen, die Werkzeuge nach ihren Bedürfnissen zu konfigurieren. Insbesondere konzentrieren wir uns auf Softwareentwickler als primäre Nutzer sowie auf Sicherheitsingenieure als sekundäre Nutzer. Zunächst führen wir eine empirische Studie durch, um die Bedürfnisse der Benutzer besser zu verstehen, die bereits Erfahrung mit Taint-Analyse oder SAST-Tools im Allgemeinen haben. Unsere Studie bestätigt, dass der Hintergrund der Benutzer unterschiedlich ist, die meisten Benutzer jedoch motiviert sind, mit den SAST-Tools zu interagieren, um sie zu konfigurieren und an das Zielprogramm anzupassen. Auf dieser Grundlage schlagen wir SWAN vor, einen vollautomatischen maschinellen Lernansatz zur Ableitung sicherheitsrelevanter Methoden aus dem Zielprogramm, die für die Spezifikation von Taint-Flows erforderlich sind. Unsere experimentellen Ergebnisse zeigen, dass SWAN eine hohe Präzision und Wiedererkennung für eine Reihe von populären Sicherheitslücken erreicht. Seine Grenzen können durch SWAN_{ASSIST}, einen halbautomatischen Ansatz für aktives maschinelles Lernen, verbessert werden, der das Feedback des Benutzers einbezieht, um die Ergebnisse von SWAN zu verbessern. Schließlich schlagen wir *fluentTQL* vor, die erste entwicklerorientierte, domänenspezifische Sprache zur Spezifikation von Taint-Flows, um die Taint-Analyse mit neuen Sicherheitsschwachstellen zu konfigurieren. In einer Nutzerstudie hat sich *fluentTQL* im Vergleich zu einer kommerziellen State-of-the-Art-DSL, CODEQL, als exzellent benutzbar erwiesen. *fluentTQL* ist unabhängig von der zugrundeliegenden Taint-Analyse und wird als Teil von SECUCHECK implementiert, unserem Taint-Analyse-Tool, das in mehreren integrierten Entwicklungsumgebungen läuft.

Die erzielten Ergebnisse bringen das Design und den Einsatz von Taint-Analyse-Tools voran. Wir zeigen, dass durch maschinelles Lernen und benutzerzentriertes Design die Anpassung der Werkzeuge für die Endbenutzer verbessert werden kann.

Acknowledgments

This thesis would not have been completed without many people I worked with or supported me during my Ph.D. Hence, I would like to acknowledge them.

First, I would like to thank my advisor Eric Bodden whom I met in 2016 when he moved to Paderborn. He introduced me to the area of program analysis and helped me understand the concepts of static code analysis and its applications in security. Furthermore, he provided valuable feedback on every research challenge I worked on. He was always approachable when I needed feedback or to hear a different opinion.

I want to thank Matthias Meyer, who gave me the opportunity to work at Fraunhofer IEM and supported my research along many diverse projects I worked on. Special thanks to Matthias Becker, who helped me in deciding on to pursue a Ph.D at Paderborn University. He became my direct supervisor, and in the last year of my Ph.D he promoted me to a team manager. During my time at Fraunhofer IEM, I worked with wonderful colleagues from Fraunhofer IEM and the Heinz Nixdorf Institut, with whom I enjoyed many discussions and collaborations. I particularly want to thank Lisa Nguyen Quang Do, Johannes Späth, Linghui Luo, and Ben Hermann. Lisa gave me plenty of feedback during the first years I explored the research area and attempted to identify the thesis research problem. She contributed to SWAN and SWAN_{ASSIST}, which are the results of this thesis. With Johannes, I shared an office for some time; therefore, we had plenty of discussions on data-flow analysis. Johannes gave valuable feedback to *fluentTQL*, which is another contribution of this thesis. With Linghui, we cooperated on a few research projects and exchanged many ideas in the area of taint analysis. She provided excellent feedback to the empirical studies I conducted. Ben introduced me to the Software Campus program, where he and Eric supported my application. I was selected for this program and led my research project aligned with my Ph.D, where SECUCHECK, the tooling for *fluentTQL* was developed. Many thanks to Tanja Tornede for helpful feedback on the machine-learning domain.

I thank my co-authors, especially of the thesis-related publications: Eric Bodden, Lisa Nguyen Quang Do, Johannes Späth, Oshando Johnson, Ingo Budde, Ranjith Krishnamurthy, Stefan Dziwok, Thorsten Koch, and Sven Merschjohan. Further thanks to the students I supervised: Oshando Johnson, Ranjith Krishnamurthy, Tobias Petrasch, and Abdul Rehman Tareen.

During my Ph.D, I gained international and industrial experience through several projects. Thanks to Eric Bodden, Matthias Meyer, and Matthias Becker for making these opportunities possible. I completed an internship with Amazon, where I worked on a challenging project and experienced the application of static analysis on a large scale. I want to thank my mentor Liana Hadarean for the great discussions and feedback she provided.

Finally, I am very thankful to my parents, who always supported me, especially for my education, even when that was economically challenging. I want to thank my partner as well, who supported me during the entire Ph.D time in any possible way.

Contents

1	Introduction	1
1.1	Motivating Example	1
1.2	Problem Statement	2
1.3	Contributions	4
1.4	Overview	7
2	Background	9
2.1	Security Vulnerabilities	9
2.1.1	Ranking Lists	10
2.1.2	Automatic Detection of Security Vulnerabilities	10
2.2	Taint Analysis	11
2.2.1	Data-flow Analysis	12
2.2.2	Security-relevant Methods	13
2.2.3	Quality of Taint Analysis Results	14
2.2.4	Typestate Analysis	15
2.3	Machine-learning	16
2.3.1	AutoML	17
2.3.2	Active Machine-learning	17
2.4	Domain-specific languages	17
3	Using SAST Tools in Practice	19
3.1	Related Work	19
3.1.1	Usability of static analysis	19
3.1.2	Studies on adaption of security tools	20
3.1.3	Taint analysis results and comparison	21
3.2	Survey and Interviews	22
3.2.1	Study Design	23
3.2.2	Results	25
3.2.3	Ethical considerations	30
3.2.4	Threats to Validity	31
3.3	User Study	32
3.3.1	Study Design	33
3.3.2	Results	38
3.3.3	Threats to Validity	46

4	Detecting Security-Relevant Methods	47
4.1	Requirements	47
4.2	Related Work	49
4.3	Two-phase Classification Model	50
4.4	FRCODE: Code Features	51
4.5	Classifiers	52
4.6	SWANFRAME: General Framework for Creating Machine-learning Pipelines for SRM Prediction	53
4.7	FRDOC_M: Implementing Features Based on Doc Comments	55
4.8	FRDOC_A: Automated Features Based on Doc Comments	62
4.9	Pipelines	62
4.10	Evaluation	63
4.10.1	Comparison (RQ7)	63
4.10.2	Real-world Applications (RQ8)	65
4.10.3	Utilizing doc comments (RQ9)	66
4.10.4	Automatic vs. manual features based on doc comments (RQ10)	66
4.10.5	Hybrid feature representations (RQ11)	70
4.10.6	Optimal classifier (RQ12)	71
4.11	Threats to Validity	72
5	Active Learning of Security Relevant Methods	75
5.1	Related Work	75
5.2	Approach	75
5.3	Tool	76
5.4	Suggesting Methods	78
5.5	Evaluation	79
5.5.1	Manual Training	79
5.5.2	Usability	82
6	<i>fluent</i>TQL	85
6.1	Requirements	85
6.1.1	Selection of Sensitive Methods	86
6.1.2	Selection of In- and Out-Values	86
6.1.3	Composition of Taint-Flows	87
6.1.4	Detailed Error Message	87
6.1.5	Integration into Developer’s Workflow	87
6.1.6	Independence of Concrete Taint Analysis	88
6.2	Related Work	88
6.2.1	Graph-based approaches	89
6.2.2	Typestate approaches.	89
6.2.3	Other approaches.	89
6.3	Design	90
6.3.1	Concrete Syntax	90
6.3.2	Abstract Syntax	91
6.4	Semantics	93
6.5	Implementation	98
6.6	SECUCHECK	99
6.6.1	Architecture	99
6.6.2	UI Features	99
6.7	Evaluation	102

6.7.1	Study Design	102
6.7.2	Usability (RQ15)	104
6.7.3	Comparison (RQ16)	105
6.7.4	Expressiveness (RQ17)	106
6.7.5	Analyzing Java/Android Applications (RQ18)	107
6.7.6	Threats to Validity	109
7	Conclusion	111
	Bibliography	114
7.1	Supplementary matherial for Chapter 3	127
7.1.1	Survey Questions	127
7.1.2	Interview Questions	130

Introduction

In our highly digitalized world, software is being used daily by many users storing and sharing sensitive data, e.g. personal information or digital business assets. Securing users' data is becoming a functional requirement in many software development companies. In many domains, this requirement is enforced by national and international regulations, such as GDPR¹. Moreover, software development is becoming highly automated. To stay competitive, companies use automated tools for security testing. One such group of tools is Static Application Security Testing (SAST) tools, e.g., CheckMarx [Che20a], LGTM/Semmlle [Git21b], SonarQube [Son21], and CodeSonar [Gra21]. These tools analyze the program without executing it [NNH99] and can reason about possible security flows, i.e., data-flows related to security vulnerabilities. At its core, many SAST tools use a data-flow analysis technique called *taint analysis*. Taint analysis tracks data-flow information among program statements starting from statements that create sensitive information, such as user inputs, named *sources* that reach program statements performing security-critical tasks, such as database insertion, called *sinks*. A generic taint analysis tool requires user-defined source/sink specifications with data-flow propagation rules to find different security vulnerabilities, like SQL injection (SQLi) [Mit21c] and Cross-site Scripting (XSS) [Mit21b]. In this thesis, we propose *methods and tools for the users of taint analysis* that enable automation and improved creation and maintenance of security vulnerability specifications.

1.1 Motivating Example

To demonstrate the application of taint analysis in SAST context, we use the program in Listing 1.1 containing a code snippet of an HTTP-handler method from a given Java enterprise application. The method *doGet* receives the HTTP request object, reads the provided user's ID stored as a parameter value in the request object, queries the SQL database for the user data, and constructs a URL link which is sent to the client to be redirected. In Line 3, the value of the request parameter enters the program as a return value of the method call *getParameter* and should be considered as untrusted data. A taint analysis declares this statement as *source*, triggering the start of the taint tracking. If the analysis is unaware of the encoder *encodeForSQL* in Line 4, it will follow the data-flow through Lines 4 and 6, reporting a false warning of type SQLi [Mit21c] vulnerability at Line 7, where the taint reaches the sink, the method call *execute*

¹<https://gdpr-info.eu/>

Query. Following the taint via Line 8, it reaches a second sink, the method call *sendRedirect* where an actual vulnerability is reported of type Open Redirect [Mit21d].

```

1  protected void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
2      try {
3          String userId = request.getParameter('userId');
4          userId = ESAPI.encoder().encodeForSQL(new MySQLCodec(), userId);
5          Statement st = conn.createStatement();
6          String query = "SELECT * FROM User WHERE userId='" + userId + "'";
7          ResultSet res = st.executeQuery(query);
8          String url = "https://" + userId + "." + res.getString(1) +
                ".company.com";
9          response.sendRedirect(url);
10     } catch (Exception e) { ... }
11 }

```

Listing 1.1: Potential SQL injection (from l.3 to l.7) and open redirect (from l.3 to l.9).

1.2 Problem Statement

All security vulnerabilities that are detectable by a taint analysis, such as SQLi and Open Redirect from the previous example, are called *taint-style* vulnerabilities. Numerous known security vulnerabilities are taint-style. In OWASP top-10 from 2017 [OWA21], five (A1, A3, A4, A7, and A8) out of ten vulnerabilities are taint-style. To cover as many security vulnerabilities as possible, most SAST tools consist of a taint analysis that runs on a transformed simplified form of the target program. As shown in Figure 1.1, the taint analysis is designed in a generic manner, i.e., it is not specific to any security vulnerability. Apart from the target program the taint analysis accepts a set of security vulnerability specifications as an input. These specifications guide the taint analysis to detect taint-flows that are reported as potential security vulnerabilities. Such a generic design allows a *separation of concerns* [SVC06]. On the one hand, static analysis experts can easily maintain and extend the taint analysis independently from the security domain. Their concern on the programming language design can be easily addressed within the taint analysis, e.g., when new abstractions are introduced to the supported language, such as lambda expression in Java version 8². On the other hand, security experts do not need to know the internals of the taint analysis. Their concern is how to express new security vulnerabilities into new specifications for the taint analysis.

Another stakeholder, having one of the central roles in this thesis as a user of the SAST tools, is the software developer. The software developers are neither experts in the static analysis nor in security. However, they are the experts of the target program and work with the results reported from the SAST tools. Hence, the quality of the results is highly relevant. We identify the first challenge of this thesis.

Challenge 1 Understand how software developers use the SAST tools in practice, focusing on taint-style security vulnerabilities.

A generic taint analysis is aware of the data-flow concepts for propagating the taint-flow within the program's simplified form, including particular statements such as sources and sinks. In Line 3 from Listing 1.1, knowing that the return value of *getParameter* is tainted, the analysis propagates the taint to the left-hand side of the assignment statements by tainting

²<https://www.oracle.com/java/technologies/javase/8-whats-new.html>

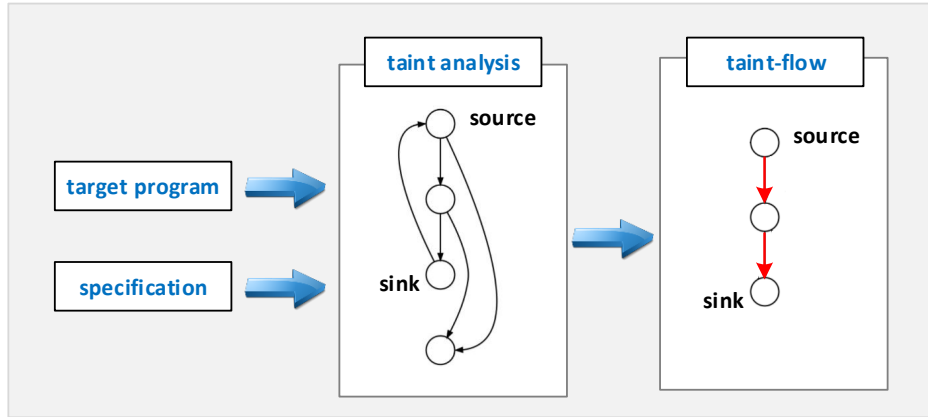


Figure 1.1: Overview of a generic taint analysis

the local variable *userId*. However, the generic taint analysis is unaware of any security-related information, for instance, *security-relevant methods* (SRM) that include source, sinks, and sanitizers. In the previous example, for only reporting the actual vulnerability Open Redirect, the taint analysis should be aware that:

- the return value of *getParameter* is tainted for both SQLi and Open Redirect
- if the second parameter of *encodeForSQL* is tainted, then the return value is not tainted anymore only for SQLi, making it a vulnerability-specific sanitizer
- the first parameter of *executeQuery* is a SQLi-specific sink
- the first parameter of *sendRedirect* is an Open Redirect-specific sink

To cover multiple vulnerability types, the taint analysis accepts specifications for security vulnerabilities created manually by mainly security experts. These specifications can be of different forms - from simple hard-coded lists of sources and sinks [ARF⁺14] to basic pattern-based entries in XML/JSON format [Gmb, Spo] or domain-specific languages (DSLs) with different expressivity targeting different types of users [Git21a, Che20b]. Creating and maintaining the vulnerability specifications **requires expertise in the security domain and a good understanding of the specification language**. However, even when created by security experts, the results still contain false warnings, as recent empirical studies show [CB16, JSMHB13, SJMH⁺15].

Creating complete specifications for known vulnerabilities is practically impossible due to the vast number of software components used in real-world applications. SAST tools provide only limited support for relevant libraries and frameworks. An analysis that is unaware of the *ESAPI* library³ used to encode the SQL query in Listing 1.1 will not report a false warning to the user. Compared to technical false warnings caused by over-approximations in the data-flow analysis (e.g., call graph construction or points-to analysis), these are contextual false warnings and can be avoided by adapting the vulnerability specifications to the target program, which often is the codebase in which the software developer currently works. Running a taint analysis on real-world applications **requires codebase-adaptations to the specifications**. Motivated by the need for automation of extracting security-relevant information from arbitrary codebases, we identify the second challenge.

³<https://owasp.org/www-project-enterprise-security-api/>

Challenge 2 Automate the extraction of security-relevant information, e.g., sources and sinks, from an arbitrary codebase that is needed in the security vulnerability specifications.

The users of the SAST tools have diverse backgrounds and expertise. One of the most studied profiles of users has been software developers [SNQDMH20, CB16, SJMH⁺15, JSMHB13, SAE⁺18, NQDWA20, WZW⁺15]. They have reported that SAST tools should be well integrated into the development workflow and detect relevant results with as few false positives and false negatives as possible. As seen earlier in the example from Listing 1.1, the information about the *ESAPI* library might be missing in the existing specifications. The software developers who know the codebase very well can easily adapt the specifications. For this task, domain-specific languages (DSLs) are provided by some SAST tools, e.g., CODEQL is the DSL of the LGTM/Semmler tool [Git21b]. To our knowledge, the existing DSLs are designed for static analysis experts and require domain knowledge that most software developers do not have. Hence, the last challenge of this thesis targets this issue.

Challenge 3 Improve the usability of the taint analyses for software developers to enable them in adapting the security vulnerability specifications.

Finally, this thesis addresses the problem of creating and adapting security vulnerability specifications used by taint analyses on real-world codebases. The usage scenario in which we address this problem is when the taint analysis runs within the IDE. Based on the identified challenges, we formulate the following hypothesis, addressed by the contributions introduced in the following section.

The existing taint analyses will detect more relevant security vulnerabilities, i.e., fewer false positives and false negatives for the users when they are adapted to the codebase and the users' needs.

1.3 Contributions

Figure 1.2 gives an overview of the contributions of this thesis. To better understand how taint analysis and, in general, SAST tools are used in practice (addressing **Challenge 1**), we conducted an empirical study (**Contribution 1**) consisting of an online survey, interviews, and a user study with semi-structured interviews. Even though our primary stakeholders are software developers, as shown in the upper part of Figure 1.2, the empirical study also targets other relevant stakeholders, such as static analysis experts and security experts, to address the viewpoints of other users of SAST tools.

Each technique in our user study addresses different aspects. *First*, we conducted an extensive study consisting of an online survey and interviews. The study, which focuses on the secure software development practices in German companies, included a section on SAST tools to understand how different stakeholders use the tools and their expectations. Through the online survey, we collected valid responses from 256 participants, of which 155 completed the section on SAST tools. The interviews were conducted with 17 experts (product owners, managers, and architects) from German companies to understand better the internal regulations and policies for using SAST tools. *Second*, we conducted a between-subjects design experiment [CGK12] with 40 software developers, followed by semi-structured interviews to find out what configuration parameters enable them to resolve taint-style vulnerabilities more effectively. The study shows that users who engage in the tool configuration are more effective than those using only the default configuration.

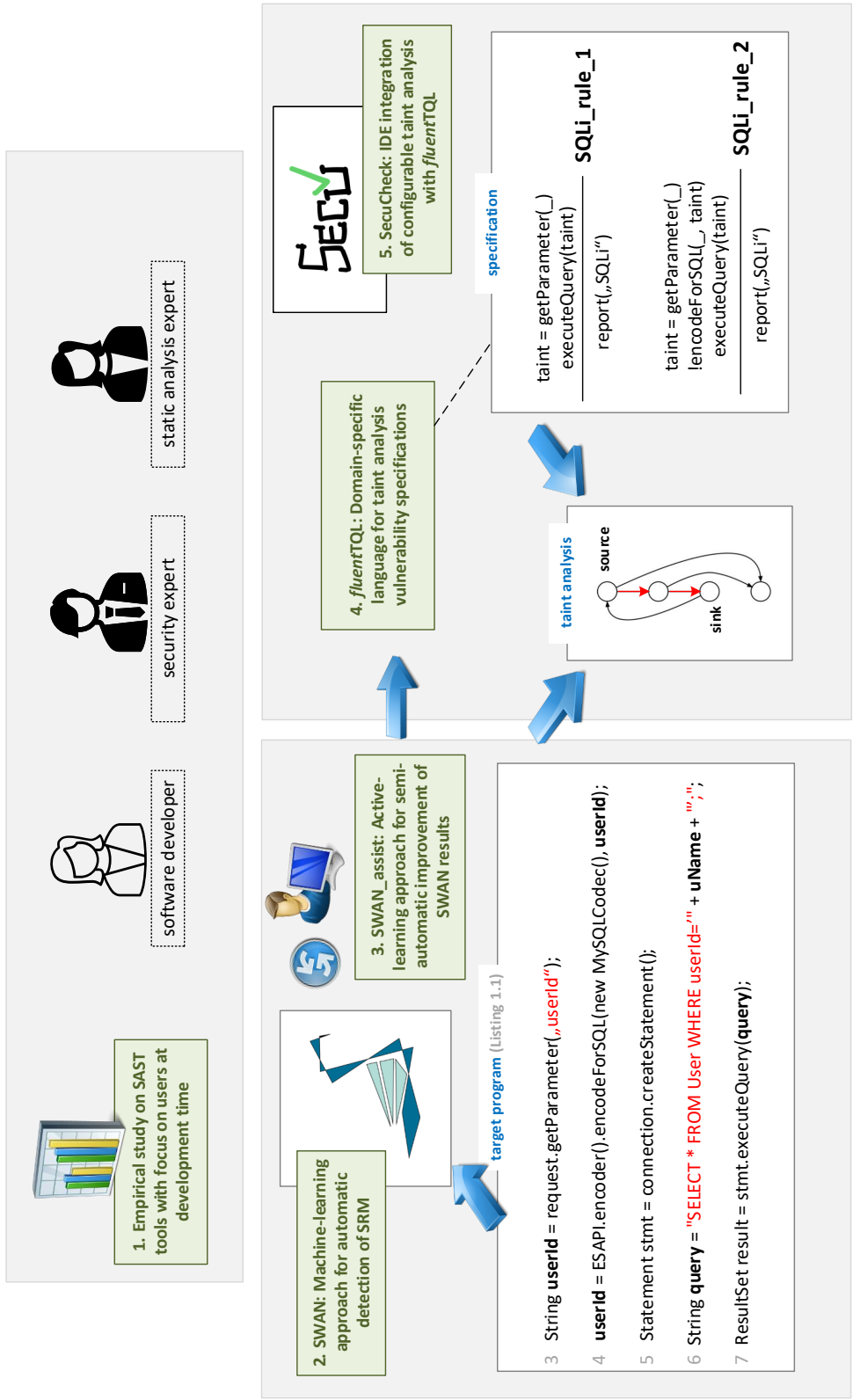


Figure 1.2: Overview of the thesis contributions (green boxes highlight the five contributions)

Through the empirical study, we learned that users, in particular software developers, are willing to interact with the SAST tools to get better results for their codebases while better understanding how the analysis works. This study result motivates the need for new methods and tools that enable the users to adapt the taint analysis to their needs and context.

The following contribution is a machine-learning (ML) approach for fully automatic detection of SRM, named SWAN⁴ (**Contribution 2**). SWAN extends the existing approach SuSi [ARB13] which classifies Android methods into potential sources and sinks based on source code information such as method signature information and intra-procedural data-flow information. Compared to Susi, SWAN has (1) an extended list of ML-features, (2) is general to any Java codebase, (3) apart from sources and sinks, additionally classifies methods into validators such as sanitizers and authentication methods, and (4) performs classification of the SRM into specific vulnerability types. In our motivating example, SWAN will classify the method *getParameter* into a potential source for SQLi and Open Redirect, the method *encodeForSQL* into a potential sanitizer for SQLi, the method *executeQuery* into a potential sink for SQLi, and the method *sendRedirect* into a potential sink for Open Redirect.

While SWAN reaches high precision and recall for SRM classification, the classification into security vulnerabilities is more context-dependent and shows lower precision and recall. This is addressed by SWAN_{ASSIST} (**Contribution 3**). SWAN_{ASSIST} is an active ML-approach that fully engages the user in training the classifier with a small sample of data. It uses SWAN in multiple iterations where users can use a tool in the IDE to label new training data or re-label an existing data that is falsely labeled. SWAN_{ASSIST} tool is accompanied by SWAN_{SUGGEST}, an algorithm that selects non-labeled data from the codebase for the user to label manually, which will significantly impact the overall classifier. Using SWAN_{SUGGEST} compared to a random selection of methods, users need less manual labeling to reach higher precision and recall. Both SWAN and SWAN_{ASSIST} address **Challenge 2**.

The SRM detected by SWAN and SWAN_{ASSIST} are helpful only when correctly used by the generic taint analysis. General propagation rules, such as "*return value of the source is always tainted*" can be used and may lead to some results, such as data leaks in Android with FlowDroid [ARF⁺14], but SAST tools require more customizations as they tend to support multiple vulnerabilities. Hence, SRM are combined with propagation rules to form different vulnerability specifications. While many existing SAST tools have a DSL for the specifications, existing studies [SNQDMH20], including our empirical study, show that these DSLs are designed for experts and are rarely used by software developers due to their complexity. To address this, we propose *fluentTQL*, an internal Java DSL for specification of rules for taint-style vulnerabilities (**Contribution 4**). *fluentTQL* has operational semantics independent of the underlying taint analysis. To demonstrate its versatility, we instantiate *fluentTQL* in two existing taint analysis solvers, Boomerang [SAB19] and FlowDroid [ARF⁺14].

SECUCHECK is the final contribution, a taint analysis tool allowing developers to use *fluentTQL* specifications on top of the Boomerang solver or the FlowDroid solver (**Contribution 5**). Using the Magpie Bridge [LDB19] framework, SECUCHECK can be used among many existing IDEs and provides a user interface for further customizations. An initial version of the tool was used in the user study introduced within the first contribution earlier. Using the customizations of entry points and security specifications, we use SECUCHECK to identify the runtime factors impacting the Boomerang solver. *fluentTQL* and SECUCHECK address **Challenge 3**.

⁴Security methods for WeAkNess detection

1.4 Overview

While this thesis is best to be read chronologically, the readers may refer only to specific chapters or sections. The main contributions are detailed in Chapters 3 - 6. Each chapter is self-contained and provides evaluation evidence for the concepts presented. The survey study and the user study are covered in Chapter 3. The internal details of the ML approach SWAN are discussed in Chapter 4. In the following Chapter 5, the readers can gain further details on $\text{SWAN}_{\text{ASSIST}}$ and $\text{SWAN}_{\text{SUGGEST}}$. The last two contributions *fluentTQL* and SECUCHECK are covered in Chapter 6. Each of these main chapters discusses and compares relevant existing work. For background information and fundamentals, the readers can refer to Chapter 2. Final remarks with summary and conclusions are to be found in Chapter 7.

Background

This chapter provides background information on the main concepts used in the following chapters and should help readers unfamiliar with any of the related areas to familiarize themselves with the basic terms and easily understand the contributions of this thesis. First, Section 2.1 introduces the concepts from the application security domain and discusses existing methods for detecting security vulnerabilities. Then, we discuss the relevant terms from static code analysis, in particular, the area of data-flow analysis (in Section 2.2), and relate them to the application of detecting security vulnerabilities. Next, in Section 2.3, we introduce the relevant terms from machine-learning. Finally, we introduce the area of domain-specific languages in Section 2.4.

2.1 Security Vulnerabilities

According to ISO/IEC 27000:2016¹, a security vulnerability is a weakness of an asset that can be exploited by one or more threats. Typical assets in software development are users' personal data, companies' intellectual property data, etc. When the integrity, confidentiality, or availability of the assets are affected by a given event, this is classified as a security incident. In application security, a security incident occurs when the software contains a known or unknown security vulnerability, which is a poor code design that an attacker may exploit.

Known security vulnerabilities are well-documented by recognized organizations. The Common Weakness Enumeration (CWE) database² maintained by the MITRE non-profit organization³, stores known security vulnerabilities under unique identifiers. For example, CWE89 refers to the popular *SQL injection* (SQLi) security vulnerability. This CWE defines the behavior of the design flow, i.e., the program allows untrusted user input to be directly passed to an SQL query statement and executed, causing a potential SQLi. Hence, the CWE defines a type of security vulnerability. The CWEs are organized in a hierarchical structure, where one more concretely defined CWE is a subtype of another more general CWE, e.g., CWE89 is a child of CWE943, which defines a design flow where untrusted user input is part of a query to a data store, such as a database. In this thesis, we use the terms CWE and (security) vulnerability type interchangeably.

The concrete instances of the vulnerability types are known as Common Vulnerabilities and Exposures (CVE), and similarly to the CWEs, are well-documented⁴. For example, the

¹<https://www.iso.org/standard/66435.html>

²<https://cwe.mitre.org/index.html>

³<https://www.mitre.org/>

⁴<https://www.cve.org/>

code in Listing 1.1 is a CVE of type CWE89. This thesis uses the terms CVE and (security) vulnerability interchangeably.

2.1.1 Ranking Lists

Among the security community, the security vulnerabilities are periodically ranked based on the current popularity and importance. There are two ranking lists often used within the security community. First, the OWASP organization publishes the OWASP top ten list of CWEs [OWA21]. Second, the SANS institute publishes the top 25 CWEs [Mit21a]. We used these lists to select the relevant CWEs that our contributions address. Moreover, we evaluated how many of the CWEs on these lists can be detected using taint analysis. Table 2.1 shows the CWEs from the top 25 list from 2019 and our evaluation to express them as a taint analysis problem. Similarly, in the OWASP top 10, most vulnerability types can be detected with taint analysis. These vulnerabilities are known as taint-style vulnerabilities. Due to the high number of popular vulnerabilities being taint-style, many SAST tools are based on a taint analysis, e.g., FORTIFY [Mic20], LGTM [Git21b], CHECKMARX [Che20a], Xanitizer [Gmb], etc.

Table 2.1: List of the SANS top 25 CWE [Mit21a] from 2019

CWE	Description	Taint analysis
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	yes
CWE-79	Improper Neutralization of Input During Web Page Generation	yes
CWE-20	Improper Input Validation	yes
CWE-200	Information Exposure	yes
CWE-125	Out-of-bounds Read	no
CWE-89	Improper Neutralization of Special Elements used in an SQL Command	yes
CWE-416	Use After Free	no
CWE-190	Integer Overflow or Wraparound	no
CWE-352	Cross-Site Request Forgery (CSRF)	yes*
CWE-22	Improper Limitation of a Pathname to a Restricted Directory	yes
CWE-78	Improper Neutralization of Special Elements used in an OS Command	yes
CWE-787	Out-of-bounds Write	yes
CWE-287	Improper Authentication	yes*
CWE-476	NULL Pointer Dereference	yes
CWE-732	Incorrect Permission Assignment for Critical Resource	no
CWE-434	Unrestricted Upload of File with Dangerous Type	yes
CWE-611	Improper Restriction of XML External Entity Reference	yes
CWE-94	Improper Control of Generation of Code	yes
CWE-798	Use of Hard-coded Credentials	yes
CWE-400	Uncontrolled Resource Consumption	no
CWE-772	Missing Release of Resource after Effective Lifetime	no
CWE-426	Untrusted Search Path	yes
CWE-502	Deserialization of Untrusted Data	yes
CWE-269	Improper Privilege Management	no
CWE-295	Improper Certificate Validation	no

* this is specific to the programming language and frameworks

2.1.2 Automatic Detection of Security Vulnerabilities

Program analysis has been used to detect security vulnerabilities automatically. There are two main areas of program analysis: static and dynamic. The former technique inspects the code without executing it and is known as *white-box* testing. The latter technique runs the program with given inputs, inspects the output without looking into the code and is known as *black-box*

testing. The advantage of dynamic analysis is that the findings are always true positive, and the analysis terminates in a reasonable time when the inputs are chosen carefully. However, choosing the right inputs is a challenging task and is currently an ongoing research topic. Even with random inputs, dynamic analyses, primarily testing, is the most used method for quality assurance. The disadvantage of dynamic analyses is that there is no guarantee that all possible program execution paths have been tested. Therefore, dynamic analyses are insufficient for specific systems, such as security-critical or safety-critical systems. The disadvantage of dynamic analyses is the advantage of static analyses. They can guarantee that all possible program executions satisfy or do not satisfy a given property. Nevertheless, sound static analyses are computationally expensive, and for real-world programs, often impractical due to long-running times. To reduce the running times, many analyses introduce sound approximations, which may introduce false positive results, i.e., the reported finding can never happen when the program is executed.

In the following, we give a brief overview of the static analysis techniques known from the literature that have different strategies to approximate the runtime behavior. On the side of sound and complete techniques, there is model checking [BK08], abstract interpretation [CC77, Zan02], and symbolic execution [Kin76]. Model checking requires a system model as a state machine, and it uses temporal logic (e.g., LTL and CTL) to verify if a given property holds for a given state of the state machine. In practice, this technique is expensive as it requires a state machine model of the program that can be created by highly skilled experts only. Therefore, it is used only in safety-critical systems such as aerospace software. Symbolic execution statically interprets the program using artificially created symbols instead of actual input values. Then, the expressions with the symbols and the program variables are solved to find the exact values. Due to path explosion, this technique does not scale well on large programs. Static analysis with sound approximation is known as abstract interpretation. Its goal is to reason about the program semantics.

Data-flow analysis is a static analysis technique in which the program is represented as a control-flow graph. At each node, representing a program statement, the analysis computes the data facts that hold before and after the node. For each type of node, the analysis writer implements a so-called *flow function* which defines how the facts before the statement are transformed into facts after the statement. This computation runs until a fixed point is reached. Data-flow analyses can introduce over- and under-approximations. The former causes false positives, while the latter introduce false negatives. In Section 2.2.1, we cover the basics of data-flow analysis. Next, we introduce taint analysis as a type of data-flow analysis.

2.2 Taint Analysis

Taint analysis is a type of data-flow analysis that tracks tainted data through the program statements. Tainted data is carried through the program variables. In Listing 1.1, we can use a taint analysis to detect the SQLi vulnerability. The return value of the method call *getParameter* is untrusted and, therefore, should be considered tainted data. The goal of the analysis is to check if there is a data-flow path between the variable *userID*, caring the tainted data, and any potential method call that gets an SQL statement and executes it, such as the method call *executeQuery*. An analysis writer is responsible for implementing the rules for the propagation of all possible types of statements found in the program. For example, in line 6, Listing 1.1, the propagation rule should define that if the right-hand side of the assignment statement carries tainted data, then the variable on the left-hand side should be tainted after that statement is analyzed. The statement that creates the taint is called a *source*. The taint analysis should report a finding to the user if tainted data reaches a predefined statement, called a *sink*. A

generic taint analysis can have different applications in the way sources and sinks are chosen. As seen in the previous section, most vulnerabilities in Table 2.1 can be detected statically with taint analysis when the correct sources and sinks are selected.

There are several existing taint analyses from the research community. Particularly popular are the taint analysis tools for analyzing Android apps [ARF⁺14, WROR18, GdP⁺15]. These tools aim at detecting potential data leaks (CWE200) within the app. Since Android is a framework that can execute apps developed by other developers, it is essential to check if the app does not process any sensitive information from the device (location, images, device ID, etc.) maliciously. Hence, marketplaces for Android apps perform static checks before the apps are accepted and distributed through the marketplace. The sources and sinks in this application are the standard APIs from the Android platform that enable the apps to process the information from the device. Typical sources are APIs that provide the calendar entries, contacts, location, or device IDs, whereas sinks are APIs for storing data into shared preferences, sending messages, logging data to files, or Bluetooth communication.

Similarly, web applications, in practice, are developed within frameworks, such as Spring [Spr] and Struts [Apaf], or JavaEE [Ora]. Developers implement the application’s business logic in isolated controllers executed in a predefined lifecycle by the framework. Hence, these controllers should be checked against any potential security vulnerabilities. Compared to Android, the number of potential sources and sinks is more significant since, in the web domain, developers use many third-party libraries, which may contain new sources and sinks. Few research tools have been published in this area as well [TPF⁺09, AFK⁺20].

2.2.1 Data-flow Analysis

Data-flow analysis is a technique for gathering information about possible values of given points in the program. Typically, the program’s control-flow graph is used for this kind of analysis. This graph models each program statement as a node. The edges model the transition of the possible program execution during runtime. The analysis collects the information about possible values before and after each node in the graph. This information is modeled as a so-called *data fact*. The analysis writer defines how each type of statement propagates the data facts by defining equations, named *flow functions*. Once the flow functions are defined, all possible inputs and outputs of each node can be calculated. To ensure stability and termination, the analysis writers apply a *fixed-point algorithm*. Due to its simplicity, we use the *monotone framework* [KU77, Kil73] and the code from Listing 1.1 as an example to demonstrate a taint analysis. For this, we first define the data-facts domain D , containing the values the analysis can track. For taint analysis, these are the program variables. In Figure 2.1, these are shown in curly brackets between each node. The flow function f maps values from D to D by defining the rule of how each node takes the in-set data facts and produces the out-set data facts. For example, the flow function of the node l3 encodes that this is a source statement, and the return value of `getParameter` creates a taint that needs to be added to the out-set data facts. In Figure 2.1, the call to `encodeForSQL` is not modeled as a sanitizer as the fact `userID` still holds after l4. The analysis requires an initial set of data facts as in-set for the program’s entry point. In our example, this is the empty set. Finally, for nodes with multiple incoming edges, the analysis needs a join operator that now defines the in-set data facts of each edge that will be propagated. When this is all defined, the monotone framework uses the *meet over all paths (MOP)* approach to compute a stable set of facts at each program point. This is a sound approach, but it is undecidable in the case of loops; hence, approximations are required. For example, the *maximal fixed point (MFP)* approach can be used. MOP merges the results at the end, while MFP merges the results at each meet point of the graph. Hence, MFP is a subset of MOP, which is still sound but less precise. Subsection 2.2.3 discusses the soundness, precision, and other metrics relevant

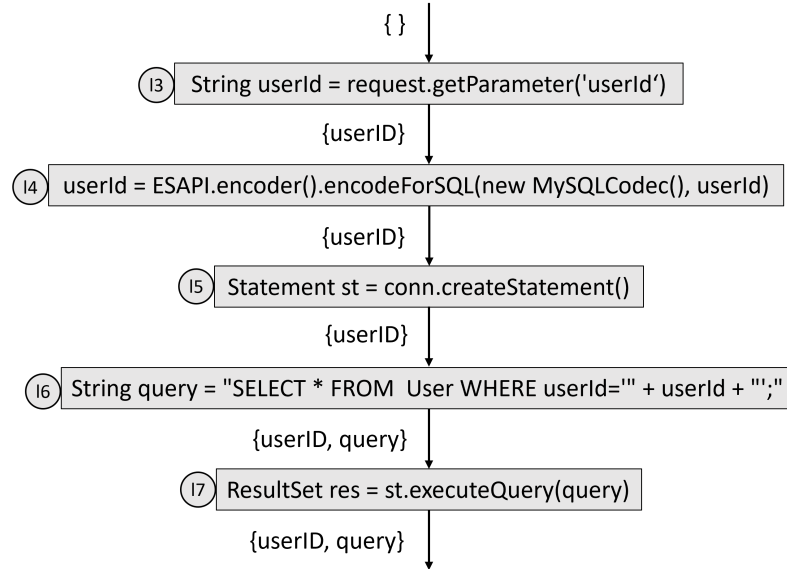


Figure 2.1: Control-flow graph and data facts for the example code in Listing 1.1 lines 3-7

to the quality of the taint analysis results.

An important property for the design of the analysis is whether the problem being solved is distributive, i.e., the results are the same regardless if the join operator is applied before or after the flow function, $\forall x, y \in D, f(x \sqcup y) = f(x) \sqcup f(y)$. For many non-distributive problems, the monotone framework assumes monotonicity, i.e., $\forall x, y \in D, f(x \sqcup y) \sqsubseteq f(x) \sqcup f(y)$. For distributive problems such as taint analysis, MFP is equal to MOP. This allows us to compute MOP via MFP, because, in general, MOP is uncomputable.

There are more frameworks for distributive problems that are used in the scientific community, such as the inter-procedural, finite, distributive subset problems (IFDS) [RHS95], inter-procedural distributive environment problems (IDE) [SRH96], and the synchronized pushdown systems (SPDS) [SAB19]. FlowDroid [ARF⁺14], for example, is a popular taint analysis for Android and uses the IFDS framework. Chapter 6 explains how we designed a taint analysis, SECUCHECK, which uses SPDS.

Intermediate representation. Another important decision for analysis writers is the actual code representation used for the control-flow graph. Even though for taint analysis, the results are helpful for the end user only when they are represented on the source code level, due to the large number of instructions which are also complex, the analyses work on a lower level language, named intermediate representation (IR). The IRs for data-flow analyses are chosen to be simple, with a small number of instructions and often 3-address based. Figure 2.2 shows how the Java code in line 3 from Listing 1.1 is represented in the IR for the data-flow analysis. We see that the single node of a function call and assignment statement is broken into two IR instructions by introducing an additional variable. For our implementation in Chapter 6, we used the Soot framework [LBLH11] and the Jimple IR. A challenging task for many tools is how to transform the IR results of the analysis back to the original source code and display it to the user.

2.2.2 Security-relevant Methods

When using data-flow analyses, such as taint analysis, one needs to select relevant method calls or, in some cases, specific program statements that are relevant for the analysis. For taint

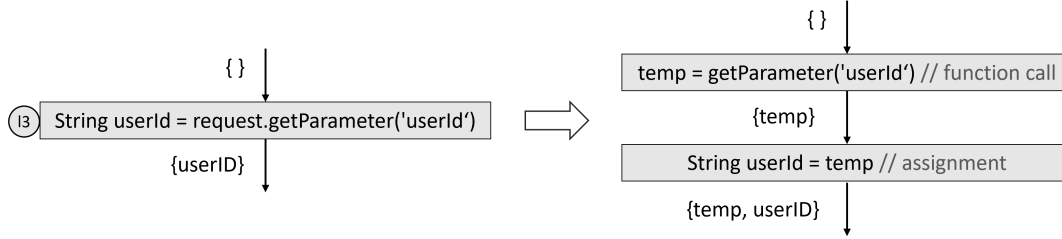


Figure 2.2: IR for line 3 of the example code in Listing 1.1

analysis, one generally needs sources and sinks, but other methods can be used, e.g., sanitizers. When the analysis is used to detect different security vulnerabilities, then these methods are specific to the vulnerabilities. Therefore, we use the term *security-relevant methods* (SRM) in this thesis. The following list summarizes the types of SRM that are used within this thesis.

- **source** - these method call creates the taint that the analysis starts to track.
- **sink** - if a taint reaches this method call, the analysis reports a finding to the user.
- **sanitizer** - at these method calls, the existing taint is killed and is not propagated further.
- **authentication method** - similar to a sanitizer, a taint that models the authentication state of the program sets the state to authenticated. See Subsection 2.2.4 for further details on these SRM.
- **validator** - we refer to sanitizers and authentication methods as validators.
- **entry point** - these methods initiate the start of the analysis and they are often not security-specific. They may be used by the call graph algorithms or directly by the analysis.
- **propagator** - these are not security-specific methods. They are used to model the propagation behavior of methods for which the analysis does not have the code, such as a library or framework code.
- **required propagator** - these are method calls that do not influence the analysis, but are security-specific and need to be in the path between the source and the sink to report the finding. See Chapter 6 for further examples.

In most of the vulnerabilities shown in Table 2.1, the SRM are method calls. There are a few exceptions. For example, in CWE798, i.e., using hard-coded credentials, the source is not actually a method call. In this case, the analysis can model all constant variables as sources and track whether these variables are passed to a relevant sink.

2.2.3 Quality of Taint Analysis Results

When considering static analyses in general, two characteristics are often discussed, i.e., *soundness* and *completeness*. Sound analyses produce findings that hold for all possible executions of the program, i.e., if the analysis says that a specific property holds, then this is true for all executions of that program. On the other hand, if there is a property that is true for all executions, then a complete analysis will produce such findings. In practice, for a whole-program analysis of real-world programs, one analysis cannot achieve soundness and completeness. Hence, for practical applications of static analysis, the term *soundy* was introduced [LSS⁺15]. Soundy analyses

are sound by certain assumptions. For example, many static analyses are sound by ignoring the Java reflection.

In the scientific community, the standard metrics used to compare different approaches for static analysis are *precision*, *recall*, and the runtime of the analysis, often referred to as *scalability*. For precision and recall, we need to measure the number of true positives *TPs* (findings reported by the analysis which are actual issues), false positives *FPs* (findings reported by the analysis which are not actual issues), and false negatives *FNs* (actual issues which the analysis was unable to find). The precision can then be calculated with $TPs / (TPs + FPs)$, whereas the recall with $TPs / (TPs + FNs)$. Based on this, for good precision, the analysis needs to reduce the number of FPs, and for good recall, a low number of FNs is required. Generally, there is a tradeoff between precision, recall, and runtime. One can achieve two of these with high value with the best effort. However, for specific applications and certain tricks in the analysis design, Bodden reported [Bod18] how one does not need to sacrifice one of the three properties.

When benchmarking a taint analysis, one requires that the given programs have documented information of the expected findings. The OWASP Benchmark [Ben21] is the most extensive project containing small programs with or without findings that challenge the tools regarding different aspects of the analysis. Mainly these aspects focus on different language features or corner cases for the alias or callgraph algorithms. Similarly, there are other benchmarks such as DroidBench [Enga] (a set of crafted Android apps covering different capabilities of the Android framework), TaintBench [LPP⁺21] (a set of real-world malware Android apps with documented findings), PointerBench [Engb] (a set of small Java programs to test the capabilities of alias algorithms). Based on these programs, one can calculate precision and recall and compare different taint analysis tools.

While a benchmark with all expected findings is needed for the recall, one can also use an arbitrary program and then classify the findings as true or false positives to test the precision. This is the typical scenario of how end users, such as software developers, use the tools. When classifying one finding as a false positive, there are two general groups, *technical FPs*, and *contextual FPs*. The analysis produces the technical FPs due to the approximations in the design of the analysis. Typically, the analysis writers will approximate specific language features or, to gain scalability, they would select a less precise callgraph algorithm or alias algorithm. These approximations can also lead to FNs. The contextual FPs are caused by the nature of the taint analysis application. This depends on which context the analysis is applied and how the user expects the findings to be. For detecting security vulnerabilities, often, the selection of SRM leads to contextual FPs.

The usability of the taint analysis tools is essential when we involve the end users in the studies. To test usability, the researchers use empirical methods from the software engineering community to design experiments with a population of selected participants. As usability metrics, the *system usability scale* (SUS) [Bro13] and the *net promoter score* (NPS) [Rei03] are often used.

2.2.4 Typestate Analysis

The security vulnerabilities related to access-control vulnerabilities such as CWE269, CWE732, CWE287, or *use after free* CWE416 from Table 2.1 can not be detected by taint analysis or only in some specific cases in CWE286. A more general form of taint analysis can be designed for these cases, called typestate analysis. A typestate of a given object specifies the operations that may be performed on this object at a given point in the program [SY86]. The typestate analysis models a state machine of the valid states of an object of a given type, and with that,

it can detect incorrect order of method calls. This technique detects API⁵ misuses, such as cryptographic APIs with the CogniCrypt approach [KSA⁺18].

In the case of access-control vulnerabilities, one needs to track the program's state and not a single object of a given type. Hence, one specifies the state machine with the relevant SRM, particularly the authentication methods used to track the entire program state. AuthCheck [PPSB20] is an example of how this technique can be applied to the Java Spring framework.

2.3 Machine-learning

Machine-learning is a method for predicting the outcome of given events without explicitly being programmed. It incorporates probabilistic and statistical methods to make predictions based on provided data. There are a few branches of machine-learning, of which most known are supervised and unsupervised learning. In supervised learning, there is a portion of data with a known outcome, and the task is to predict the outcome of new data based on a model created from the data with the known outcome. In unsupervised learning, all the data has an unknown outcome, and the task is to detect specific data patterns that help predict the outcome. In this thesis, we used supervised learning algorithms in Chapter 4; in the following, we only focus on this branch.

In supervised learning, the provided data has a portion of labeled data. The task is to predict the label of new data with an unknown label. For example, a data point can be a method in a given program. If we consider the code in Listing 1.1, for a taint analysis we are interested which of the methods called within the code of *doGet* are potential sources. Hence, the task of the supervised learning is to predict for each method *getParameter*, *encodeForSQL*, *createStatement*, *executeQuery*, *getString*, and *sendRedirect*, whether is source. In this example, the label is then "source". To predict this, we need data with known labels, methods for which we know are sources, and not sources. Based on this, there are different *classifiers* that we can build. A classifier is a model that is later used to predict the labels of new data. Different classifiers, such as decision trees, support vector machines, neural networks, and many more, can be used. The data used to create the classifiers is known as *training data*. To validate the classifier's results, researchers and practitioners typically divide the labeled data into two sets, one used as training data and the other known as *testing data*. Since the testing data has known labels, those are used to compare against the predicted labels and calculate the precision and recall. Researchers often use the k-fold validation method [Sto74] with k=10 to divide the data into k subsets and average the precision and recall. When the classifier is then used in production, the data on which the prediction is applied is known as a *validation set*.

To build the classifiers, we use the characteristics of the data. These characteristics are known as *features*. For example, in the case of classifying methods as sources or not, we can use the information we know about those methods, such as the method name and signature, the code of the method, documentation of the method, etc. The feature is a single dimension that describes the data. For example, a feature can be "*does the method return a string value?*". The value of this feature on a given data point is yes or no, which is an example of a binary feature. Using such features, classifiers can be created. The labels that the classifier can predict are also known as *classes*.

⁵application programming interface

2.3.1 AutoML

There are simple classifiers such as decision trees and more complex ones like neural networks, requiring many parameters to be selected. For non-experts in the area of machine-learning, this can be a challenging task. The choice of the classifier and its configuration of parameters can be defined as an exploration problem. Therefore, the area of *automated machine learning* (AutoML) has evolved. AutoML is the process of automating the tasks of applying machine learning to real-world problems. One task of AutoML is to find the optimal machine-learning pipeline for a given application. A popular AutoML implementations are ML-plan [MWH18] and Auto-Weka [KTH⁺19]. We applied Auto-Weka in Chapter 4.

2.3.2 Active Machine-learning

In applications where the labeled data is small, selecting the data to be used for the classifier’s training is crucial for the quality of the prediction. One branch of machine-learning that addresses this problem is *active machine-learning* [Ang88]. In active machine-learning design, the algorithm queries the user interactively to label data for the training set. These data are samples from the pool of unlabeled data. Inspired by this approach, we designed one of our contributions presented in Chapter 5.

2.4 Domain-specific languages

In contrast to general-purpose languages (GPLs), such as Java, Python, and C/C++, domain-specific languages (DSLs) are specialized to a particular domain. They allow the users to model the concepts from a given domain more manageable than GPLs. For example, HTML⁶ is a popular DSL for creating web resources. As with any programming language, a DSL is defined through its abstract and concrete syntax and static and dynamic semantics [SVC06]. The abstract syntax defines the terms (entities) of the language and their relationship. The concrete syntax defines how the DSL users specify the terms from the abstract syntax in a textual or graphical way. Its static semantics defines additional language constraints that cannot be expressed in the abstract syntax. Finally, the dynamic semantic formally models all possible executions of a given DSL, i.e., the runtime behavior. DSL designers can use different methods to define the runtime behavior, such as translational or operational semantics.

DSLs can be internal or external. Internal DSLs are part of an existing programming language, usually a GPL. They use a host language’s infrastructure (lexer, parser, type checker, etc.) to embed the abstract syntax. Hence, they have the same concrete syntax as the host language. On the other hand, external DSL have their infrastructure and unique concrete syntax.

DSLs are widely used among static analysis tools to allow the users to define their custom rules and analyses. In particular, many popular SAST tools are shipped with their DSL to allow experts to define custom rules for new security vulnerabilities that are not covered by the standard ruleset or to adapt it and improve the precision and recall of the tool. For example, CODEQL is popular DSL among security experts. It is well-integrated with GitHub, and the existing ruleset is publicly available. Other commercial tools, such as CHECKMARX and its DSL CxQL, provide similar support to their clients.

⁶hyper-text markup language

Using SAST Tools in Practice

Static application security testing (SAST) tools, as a branch of static analysis tools, appeared in the early 2000s. In recent years, their popularity increased significantly, as many commercial (CHECKMARX [Che20a], FORTIFY [Mic20], Veracode [Ver20], SonarQube [Son21], LGTM [Git21b], DeepCode [Sny21]) and open source (FindBugs [oM21], Infer [Fac21], CogniCrypt [KSA⁺18]) tools became available. As discussed in the previous chapter (Section 2.1), when configured with proper SRM, a taint analysis can detect large amounts of security vulnerabilities. Therefore, many SAST tools have a taint analysis as a core component.

In practice, the users of SAST tools are diverse, from software developers and quality teams to security experts and technical leads. Furthermore, these tools are used in different contexts, such as integrated development environments (IDEs), continuous integration pipelines, or management dashboards. Therefore, researchers have been empirically studying different aspects of static analysis tools and SAST tools, primarily focusing on usability. In this chapter, we first discuss the most relevant existing studies (Section 3.1). Then, we present our empirical study and its results (Section 3.3) that motivate the contributions presented in the following chapters.

3.1 Related Work

With the increased number of new open-source and commercial tools on the market, researchers have studied different aspects, such as usability, incorporation into users' workflow, quality of results, etc. In the following, we discuss existing studies on static analysis tools. We categorize the studies into three groups: studies that focus on the usability of static analysis tools (Subsection 3.1.1), studies targeting the security aspect (Subsection 3.1.2), and studies on the quality of the results (Subsection 3.1.3). Tables 3.1–3.3 summarize all studies by stating their goal, methodology, and scale in terms of the number of participants, companies involved, and/or size of the analyzed data.

3.1.1 Usability of static analysis

Within this group of studies, we identify two subgroups: studies that focus on usability and on the adaption and integration of the tools within the companies' processes.

Nguyen et al. [NQDWA20] performed a survey with developers from the German company *Software AG* and analyzed the usage data of CHECKMARX [Che20a] for two of the company's projects. They identified the needs and motivations that developers have while using static analysis tools. Based on that, they provide recommendations for new features and research

ideas for future consideration. A similar study was conducted earlier at *Microsoft* by Christakis et al. [CB16]. They surveyed and interviewed developers and analyzed data from live site incidents, which are highly critical bugs handled by the on-call engineers. They identified usability issues and functionalities that program analyses should provide for better usability. Similar results were also reported in the study by Johnson et al. [JSMHB13], in which they performed semi-structured interviews with 16 developers from a single company and four graduate students. Vassaleo et al. [VPP⁺18] studied the different contexts in which static tools are used by developers, i.e., development environment, review, and continuous integration. Moreover, they studied the configuration options for these contexts and found that most developers use the same configuration among all environments (IDE, continuous integration, or review). In comparison, our user study (Section 3.3) targets only the IDE. They performed a survey with 42 participants. To confirm their findings, they interviewed eleven developers from six companies.

Next, we discuss two studies that target the integration of static analysis tools into the development workflow and processes. Sadowski et al. [SvGJ⁺15] proposed the Tricoder platform that *Google* uses to integrate different program analysis tools in one system and improve the user experience. They explain the requirements and how the system was deployed. They collected usage data from the deployed system to confirm the design decisions of the platform. In a follow-up publication, Sadowski et al. [SAE⁺18] shared the lessons learned from the Tricoder platform. Zampetti et al. [ZSO⁺17] studied how static analysis tools are integrated into the pipeline in open-source Java projects from GitHub. They used repository mining techniques.

Table 3.1: Related studies with focus on usability: goal, methodology, and scale.

Study	Goal	Methodology	Scale
Nguyen et al. [NQDWA20]	identify developers' goals and motivations for using static analysis tools	a survey and an analysis of company's usage data of CheckMarx	87 participants and data from two internal projects at <i>Software AG</i>
Christakis et al. [CB16]	identify practitioners' needs from program analysis	a survey, interviews with group managers, and an analysis of live site incidents	375 participants and 256 live site incidents reports from 17 services at <i>Microsoft</i>
Johnson et al. [JSMHB13]	identify developers' usability issues with static analysis tools	semi-structured interviews	16 developers from single company and 4 graduate students
Vassallo et al. [VPP ⁺ 18]	study the developers' usage context of static analysis tools	a survey and semi-structured interviews	42 participants through open invitation and 11 interviewees from six companies
Sadowski et al. [SvGJ ⁺ 15]	provide a set of principles for building and integrating program analysis tools in practice	case study of Tricoder as a platform for program analysis tools	Tricoder usage data at <i>Google</i>
Zampetti et al. [ZSO ⁺ 17]	study the CI and usage of static analysis tools	mining repository techniques	20 open source GitHub projects

3.1.2 Studies on adaption of security tools

Next, we discuss studies focusing on detecting security vulnerabilities with static analysis tools. Thomas et al. [TLC⁺16] experimented with 13 developers in which the participants were given the task of using a tool that reports security vulnerabilities in the IDE. After interacting with the tool, the participants were interviewed. In a similar study, Smith et al. [SJM⁺15] invited ten developers from the same project to solve four tasks using an extended version of FindBugs. First, the participants were asked to orally explain their thoughts, which the authors used to

formulate questions. Then, they used the card-sorting method to come to relevant conclusions. Smith et al. [SNQDMH20] evaluated the usability of the user interfaces of four SAST tools. They used heuristic walkthroughs and an experiment followed by an interview with twelve participants. In a survey study with multiple stages, Witschey et al. [WZW⁺15] identified factors that can predict developers’ adoption of security tools. Patnaik et al. [PHR19] have mined 2,491 Stack Overflow questions to identify developers’ usability issues when using cryptography libraries.

A study on the security processes during software development was conducted by Thomas et al. [TTCL18], who interviewed 32 security experts to understand the use of different tools, including static analysis tools. This study gives a general overview of some security practices that are applied in the industry.

Table 3.2: Related studies with focus on security: goal, methodology, and scale.

Study	Goal	Methodology	Scale
Thomas et al. [TLC ⁺ 16]	study the perceptions and actions taken by developers when they interact with static analysis tool in the IDE	experiment with an interactive tool in the IDE and an interview	13 participants from multiple companies
Smith et al. [?]	study the information need of developers while using static analysis tool for security vulnerabilities	an experiment and card sorting	10 developers working on the same project
Smith et al. [SNQDMH20]	study the user interface of 4 tools and propose areas for improvements	heuristic walkthroughs, an experiment and interviews	12 participants
Witschey et al. [WZW ⁺ 15]	quantify the relative importance of factors that predict security tool adoption	a multi-staged survey	119 participants from 14 companies and 61 participants from 5 mailing lists
Patnaik et al. [PHR19]	identify usability issues of cryptography libraries used by developers	mining techniques	2,491 Stack Overflow questions
Thomas et al. [TTCL18]	understand the security processes for application development	semi-structured interviews with security experts	phone interviews with 32 experts recruited with snowball technique from different companies

3.1.3 Taint analysis results and comparison

Several previous studies focus on the quality of the analysis results in terms of recall, precision, or runtime, reported by taint analysis tools. Luo et al. [LBS19] performed a quantitative and qualitative analysis of the taint flows reported by FlowDroid [ARF⁺14] by analyzing 2,000 Android apps. They identified that selecting sources and sinks was one of the main factors for imprecision. Qui et al. [QWR18] compared the three Android taint analysis tools, FlowDroid, Amandroid [WROR18], and DroidSafe [GdP⁺15]. They ran all tools under the same configuration in order to gain a fair comparison of the tools’ capabilities. Their work on finding a common configuration among the three tools to make a fair comparison provides valuable insights into the importance of the configuration for the quality of the findings. Habib et al. [HP18] investigated the quality of the findings from three static analysis tools Spotbugs, Infer, and Error Prone. They used the real-world Java applications from Defects4J with 594 known bugs and inspected the findings automatically and manually. They find that the tools detect only 4,5% of the bugs, and the types of findings they report are complementary. Next, Zhang et al. [ZGSN17] proposed an interactive approach for resolving the findings from static analysis tools. They experimented with data race analysis to evaluate their approach and show a 74% reduction in the false pos-

itive rate. They used eight real-world Java applications and a set of questions collected from Java developers they hired from UpWork. Finally, Lee et al. [LLK⁺17] proposed a clustering algorithm for static analysis findings. They compared several algorithms on the buffer overflow findings from 14 C packages and showed a 45% reduction in the false positive rate.

Table 3.3: Related studies with focus on the results quality: goal, methodology, and scale.

Study	Goal	Methodology	Scale
Luo et al. [LBS19]	identify important factors for imprecision in FlowDroid	a case study with manual inspection	2000 analysed apps and 146 manually inspected
Qui et al. [QWR18]	compare the results of Android taint analysis tools and identify strength and weaknesses	analysis and inspection of the results by three tools (FlowDroid, Amandroid, and DroidSafe)	collection of microbenchmarks
Habib et al. [HP18]	compare the results of three static analysis tools	automatic and manual inspection of the bugs reported from three tools	collection of 15 Java application (Defects4J) with known bugs
Zhang et al. [ZGSN17]	reduce false positives by proposing an interactive approach for resolving findings from static analysis	experiment with datarace analysis	evaluated on 8 Java projects and data collected from Java developers hired from UpWork
Lee et al. [LLK ⁺ 17]	reduce false positives via clustering algorithms	experiment with buffer overflow findings	evaluated on the findings from the Sparrow static analyzer on 14 C packages

3.2 Survey and Interviews

We next present the first part of our empirical study. We followed a triangulation methodology consisting of an online survey and expert interviews [Sea99]. The activities were part of a more extensive study within the research project *AppSecure.NRW*.¹ The project’s goal is to understand and evaluate the overall security activities in software development among German companies. This thesis only presents the results of SAST tools. The results, not a contribution to this thesis, are available as a white paper.²

This part of the study answers the following research questions:

- RQ1** To what extent are SAST tools used in practice among software development teams in Germany?
- RQ2** How are SAST tools used in practice, and what are the problems faced by the users?
- RQ3** What are typical culture and processes for using SAST tools and other security checks in German companies?

The survey and the interviews provide valuable data to answer different aspects of the research questions. We explain this in detail in the following subsection. The study targeted software development teams in Germany; therefore, the survey and interviews were conducted in German. Participation in the study was voluntary and without any compensation.

¹<https://www.appsecure.nrw/>

²<https://arxiv.org/abs/2108.11752>

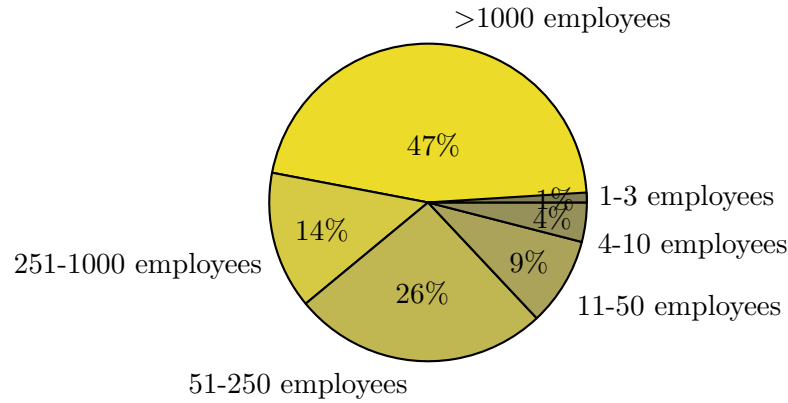


Figure 3.1: How many employees does your company have? (N=256)

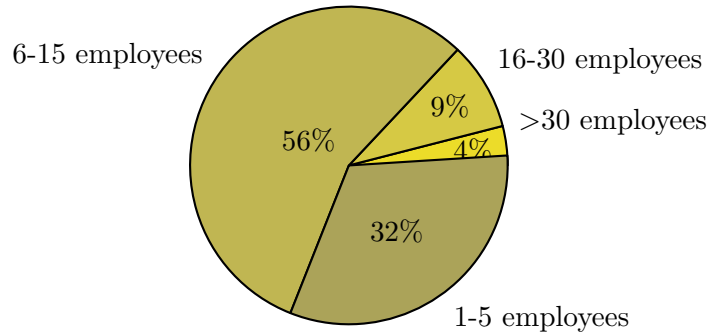


Figure 3.2: How big is your team? (N=256)

3.2.1 Study Design

Survey. To understand the usage of SAST tools and the culture and processes throughout the development process, we invited participants from all roles involved in the software development, including developers, architects, product owners, and executives. We used three ways to gather participants. First, we used our direct contacts from the industry and asked them to invite their teams internally. Second, we created posts on our institution's social media channels and website. Third, the survey was promoted by the media of the publishing house *Heise*³ and among the networks *Bitkom*,⁴ *it's OWL*,⁵ and *innosent OWL*⁶. In total, we received responses from 350 participants. We excluded all responses that were (1) incomplete, (2) answered in an unrealistically short time, or (3) not from Germany. After this filtering, we gathered 256 responses. If we consider that there are roughly 900.000 software developers in Germany⁷, we get a margin of error of only 6%⁸, which has a confidence level of 95% or a Z-value of 1.96, making our study representative of the population of software developers.

Of all participants, 47% are from large companies with more than 1000 employees, and the rest are from small- and medium-sized companies, see Figure 3.1. The size of most of the teams

³<https://www.heise.de/>

⁴<https://www.bitkom.org/>

⁵<https://www.its-owl.de/home/>

⁶<https://www.innosent-owl.de/>

⁷<https://www.daxx.com/de/blog/entwicklungstrends/anzahl-an-softwareentwicklern-deutschland-weltweit-usa>

⁸<https://www.surveymonkey.com/mp/margin-of-error-calculator/>

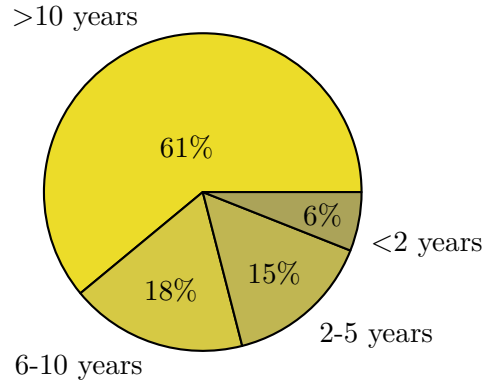


Figure 3.3: How many years of experience in software development do you have? (N=256)

is 6-15 persons (56%), and about one-third are small teams of 1-5 persons (32%), see Figure 3.2. 61% of the participants have long experience of more than ten years in software development, see Figure 3.3. Of all participants, 80% have the role of *software developer*, followed by 12% *executive*, 10% *other*, 8% *product owner*, 6% *project manager*, 5% *data protection officer*, and 4% *security analyst*, where multiple answers were allowed. Finally, the participants come from 37 different industry sectors, such as automotive, electrical, chemical, insurance, internet, research, transportation, arts, sports, law, tourism, etc.

We conducted the survey using the online tool *Survey Monkey* [Mon21]. The tool allowed us to use different links for each company that we invited and one link that was shared publicly. Among all links, we received a considerable number of responses from one large company, 59 responses from 256. This company remains anonymous, and we will refer to it with *ABC*. Hence, in our results, we can compare the trends among multiple companies of mixed size and a single large company. The survey was open for six weeks. On average, the participants needed 25 minutes to complete the questionnaire, measured based on the session duration per participant collected by Survey Monkey.

We followed the guidelines for opinion surveys by Kitchenham et al. [KP08]. Initially, we conducted a literature search to identify relevant related work (Section 3.1). None of the existing studies provided a survey instrument (i.e., a questionnaire) that could have been reused. Hence, we created a new questionnaire for a cross-sectional survey. Five researchers were involved in creating and selecting the questions in a top-down process, starting from the research questions and breaking them into more concrete questions. The questionnaire was reviewed by three more researchers and then modified based on the feedback. We conducted a test phase that included internal and external tests. We performed two internal tests with students from our research group to verify the clarity of the questions and measure the time needed to complete the questionnaire. After that, we performed three external tests with professionals involved in software development from the industry.

The final questionnaire with 42 questions is available within our artifact⁹, and those relevant for this chapter are listed in the Appendix 7.1.1. The questionnaire consists of seven parts:

- 1 Questions for all roles (general questions about security)
- 2 Questions related to requirements
- 3 Questions related to design

⁹Link to our Open Science Framework repository containing the artifact with our study materials and collected data: <https://osf.io/k37c9/>

- 4 Questions related to implementation and testing
- 5 Questions related to SAST tools
- 6 Questions related to software operation and maintenance
- 7 Meta-data questions

Parts 1 and 7 were answered by all participants (256). Parts 2 to 6 were optional. The most relevant data for SAST tools was in parts 4 (answered by 220 participants) and 5 (answered by 114 participants).

Interviews. In addition to the survey, we performed 17 interviews with product owners and executives from German companies. The intention was to collect additional qualitative data from these roles as we could not get high number of participants with those roles in the survey. Moreover, these roles give helpful information about the culture and processes in their companies. Four of the interviewees were our previously known contacts. The rest were selected through convenience sampling [Giv21]. We invited several randomly chosen companies from our region who also participated in the survey and used the *first come, first served* principle to conduct the interviews with persons that volunteered to participate. Seven interviewees were product owners (PO), six were managers/executives (ME), and the remaining four had both roles. All experts have been involved in software development during their professional experience.

Two researchers performed each interview. One researcher was the moderator asking the questions and the other researcher wrote a protocol and, in rare cases, asked questions. Additionally, an audio recording of all sessions was made. After the interviews, the recordings were automatically transcribed and used to extend the protocols created during the interview. On average, each interview took 45 minutes. The interviews were conducted during the second half of 2019. For the evaluation of the interviews, we used the codebook method [RBW0], where three researchers manually annotated all transcripts.

We applied a similar process as the survey (Subsection 3.2.1) to design the questionnaire for the interviews used as a guide during the interviews. We created two versions for each role, PO and ME, which differ only in a few questions. The experts who had both roles were asked all questions.

3.2.2 Results

The study reached a broad range of companies. Participants from multiple companies completed our survey. Fifty-nine responses (23%) from the survey are from the company (*ABC*).

With N , we denote the number of responses collected for each question. Note that many participants answered different survey sections due to the optional questions. Hence, in the following, we report the percentage and the absolute numbers. To find correlations between two questions, we use the Cramer v value [Cra16] (where values are between 0 and 1, with values over 0.25 having a strong correlation) and, for statistical significance, the p -value. The numbering of the questions is based on the list in the Appendix 7.1.1.

Use of static analysis tools (RQ1). Among the survey participants, there is a heterogeneous use of IDEs and programming languages. The top three used IDEs are IntelliJ IDEA (60%), Eclipse (53%), and Visual Studio Code (36%) (Figure 3.5). The top three used programming languages are Java (76%), JavaScript/TypeScript (45%), and SQL (34%), see Figure 3.6. There is a correlation that those developers who use Java also use IntelliJ IDEA (Q13-Q12, Cramer

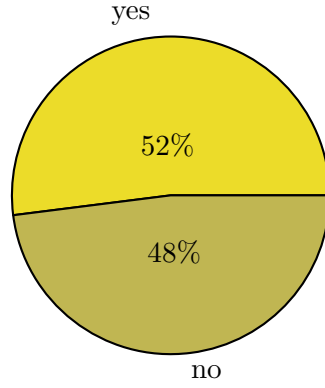


Figure 3.4: Do you use static code analysis tools? (N=220)

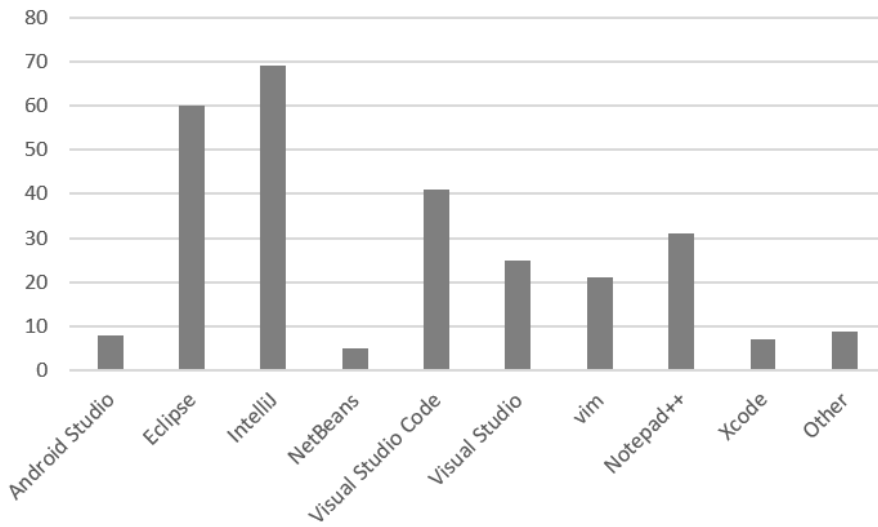


Figure 3.5: Which IDEs are used among the survey participants? (N=114, Y-axis denotes number of responses)

$V=0.537$ $p=0$, $N=123$) and develop web applications (Q13- Q30, Cramer $V=0.385$ $p=0.0001$, $N=114$).

In total, 114 (51.8%) out of 220 responses said they use static analysis tools within their teams. When asked which tools they use (Q16), there were 57 unique tools named, of which only four tools were named in at least ten responses, i.e., SonarQube (50 responses), Findbugs (19), Checkstyle (13), and OWASP Dependency-Check (10). These tools are freely available or at least have a free version. They mostly perform simple pattern-based matching techniques to detect issues. SonarQube additionally has a taint analysis as more complex analysis. Other tools with more sophisticated dataflow analyses that the survey participants mentioned are: Checkmarx, Fortify, Klockwork, and Coverity. Only two participants noted that they use internally developed tools.

Furthermore, the participants were asked to prioritize, according to their preference, where the warnings from the tools should be reported. From the possible options: (1) within the IDE, (2) on an internal website, and (3) in the ticket system, 160 out of 239 responses selected the option (1) with the highest priority.

To compare the results within a single company with multiple differently-sized companies, we extracted the results from the company *ABC*. 27 out of 59 responses (45.8%) said that they

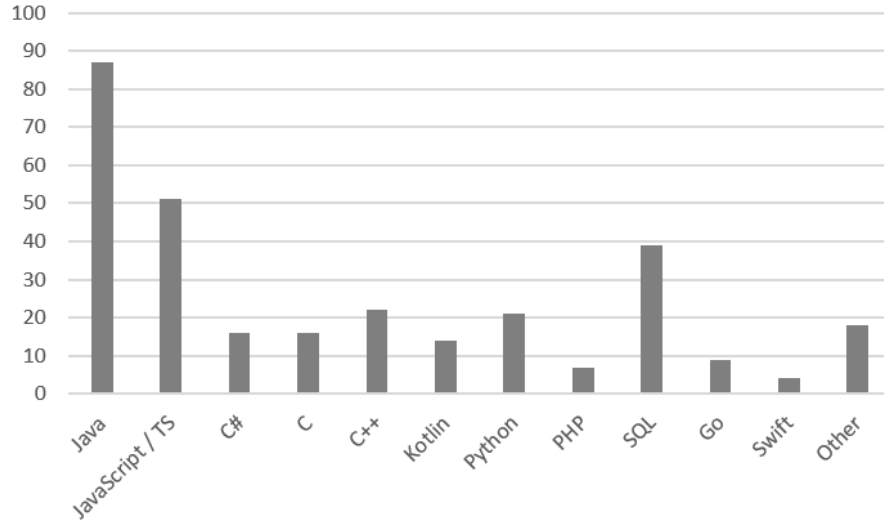


Figure 3.6: Which programming languages are used among the survey participants? (N=114, Y-axis denotes number of responses)

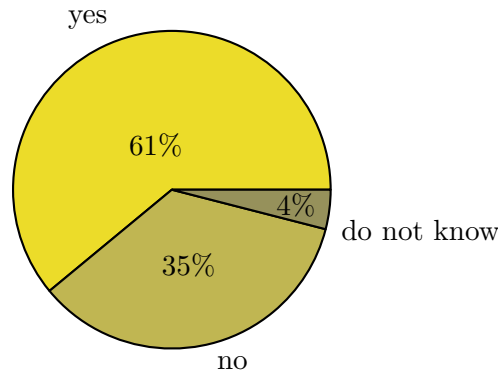


Figure 3.7: Do you use tools to automatically check security properties in code? (N=221)

use static analysis tools, which is lower than the 51.8% among all companies, including *ABC* or 54% excluding *ABC*.

German companies have a diversity in the use of IDEs and programming languages. Moreover, only about half of the teams use static analysis tools, of which the most popular are SonarQube, Findbugs, Checkstyle, and OWASP Dependency-Check.

Problems and Tools Configuration (RQ2). Previous studies [CB16] have reported that the existing tools are not fast enough to be used in development time. The participants in our study perceive the current situation differently. Ninety-three out of 114 respondents (82%) think the tools they use are fast (Q18.1). This shows improvement in recent years due to new research results and engineering efforts visible to the users. However, the number of false warnings from the static analysis tools remains high. Sixty-five out of 114 participants (57%) have indicated this (Q18.2). In particular, those who answered that the number of false warnings is high also said that they program in the C language (Q18.2-Q13, Cramer $V=0.305$ $p=0.0205$, $N=112$).

Based on their expertise and the warnings from the tools, most participants (79 out of

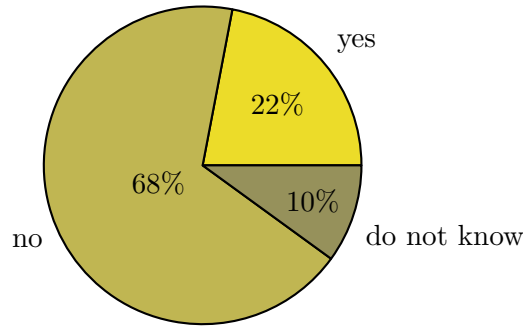


Figure 3.8: Is security automatically checked during operation? (N=134)

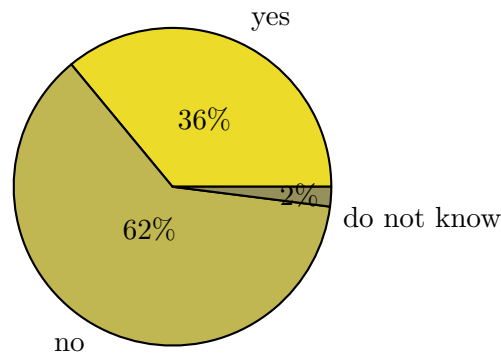


Figure 3.9: Are there automatic security checks after a release is built? (N=134)

114 responses, 70%) stated that they could confirm which of the warnings were true positives (Q18.3). The participants think that the tools find real issues regularly. Seventy percent (80 out of 114 responses) of the participants confirmed this statement (Q18.4). Finally, 81% (92 out of 114) from the participants said that the messages reported from the warnings helped them to fix the issues found in the code (Q18.5).

When it comes to the configuration of the tools, we asked the participants to what extent they are willing to change, add, or remove the rules used by the tools to find different security issues. Sixty percent (69 out of 114 responses) are willing to define their own custom rules to be used by the tool (Q18.6). However, only 35% (40 out of 114 responses) have experience defining custom rules for the tools. From them, many have answered that they have the role of *Security Analyst* (Cramer $V=0.377$ $p=0.0026$, $N=108$). Those participants who are willing to define custom rules are in teams that have testing responsibilities (Q18.8-Q2, Cramer $V=0.313$ $p=0.0163$, $N=114$). Eighty-three percent (95 out of 114 responses) of the participants are willing to provide feedback to the tools in terms of marking false positives to get better results in future runs of the tools (Q18.7).

When observing the results from *ABC*, there is a slightly higher willingness by the participants with 63% (17 out of 27 responses) to configure or with 89% (24 out of 27 responses) to provide feedback to the tools. For the questions on the quality of the results, there are only minor differences (under 3 %) except for one: 78% (21 out of 27 responses) of the participants from *ABC* think that the tools find real issues regularly, compared to 68% (59 out of 87 responses) to the rest.

Our results show that users consider static code analysis tools fast enough but still report high number of false warnings. The messages of the warnings are helpful for most of the users to fix the issues. Most users are willing to invest time in adapting the rules of the tools or provide feedback for improved future results.

Culture and processes for using static analysis tools (RQ3). The most popular tools listed by the participants are free tools. The most popular commercial tools listed by the participants are CHECKMARX and FORTIFY, listed only 5 and 4 times, respectively (Q17). We asked the product owners and the managers/executives their opinion on open and free tools. Fourteen have answered this question, allowing their software developers to use free and open source SAST tools. Even most encourage the teams to use open and free SAST tools. When asked whether there is a budget for commercial SAST tools, only one participant said there is no budget for that purpose and only free and open source tools are possible. Ten participants said there is a budget only when there are requests, whereas six said there is a dedicated budget for this purpose. One participant commented, *"These tools are a good investment"*. Most interviewees said that it is seldom that developers request new tools.

Even though in most German companies there is a budget for SAST tools, the free SAST tools are more popular among the developers than the commercial tools.

The participants from *ABC* have different opinions than the rest regarding the availability of tools for secure software development. Fifty-nine percent (16 out of 27 responses) from *ABC* think that they have the right amount of tools, whereas only 38% (33 out of 87 responses) in the other companies.

In particular, our study targeted the security aspect of the processes. We asked for each phase: requirements (Figure 3.11, Q4.1), design (Figure 3.12, Q6.1), and implementation/testing (Figure 3.13, Q10.1), whether the security is considered. During the design and implementation/testing phases, three-quarters of the participants answered that security is considered. During the requirements phase, security is less considered. When we asked all participants if they use tools to check the security properties of their code (Q10.3), 52% answered positive (77 out of 221 responses) and 44% negative (134 out of 221 responses), see Figure 3.7. Moreover, among the participants that use SAST tools, only 17% (11 out of 63 responses) answered that they perform an official security review before each release (Q22.1), while 80% do not perform, and 3% do not know. Similarly, only 36% (38 out of 134 responses) said that they perform automatic security checks after the release (Q22.2), while 62% do not do and 2% do not know, see Figure 3.9. In addition, only 29% (23 out of 134 responses) said that they perform automatic security checks on the software while in operation (Q22.3), while 68% do not and 3% do not know, see Figure 3.8.

Also, there is a correlation showing that the teams in which the security requirements are checked by the developer who implemented the code need better tools to accomplish their tasks (Q11-Q15.2, Cramer $V=0.364$ $p=0.0444$, $N=67$) and that these checks are done during development time (Q14-Q10.3, Cramer $V=0.376$ $p=0.00007$, $N=123$). The participants who answered that they perform security checks before the release also have a dedicated security team (Q22.2-Q11, Cramer $V=0.455$ $p=0.000001$, $N=123$) and/or hire an external company for security (Q22.2-Q11, Cramer $V=0.358$ $p=0.0001$, $N=123$).

Finally, many teams have responsibilities in multiple phases of the development. Figure 3.10 shows the responses from 114 participants. Only 29% of the participants (63 out of 98 responses) have a clear process to verify the implementation of the design concerning security (Q10.4),

whereas 64% do not (28 out of 98 responses). Eighteen percent of the participants (21 out of 114 responses) said that nobody is responsible for checking the specified security requirements of the software (Q11). The programmer does this for 66% (75 out of 114 responses). Only for 32% (36 out of 114 responses), this is performed by a security team, or 18% (20 out of 114 responses) from an external company.

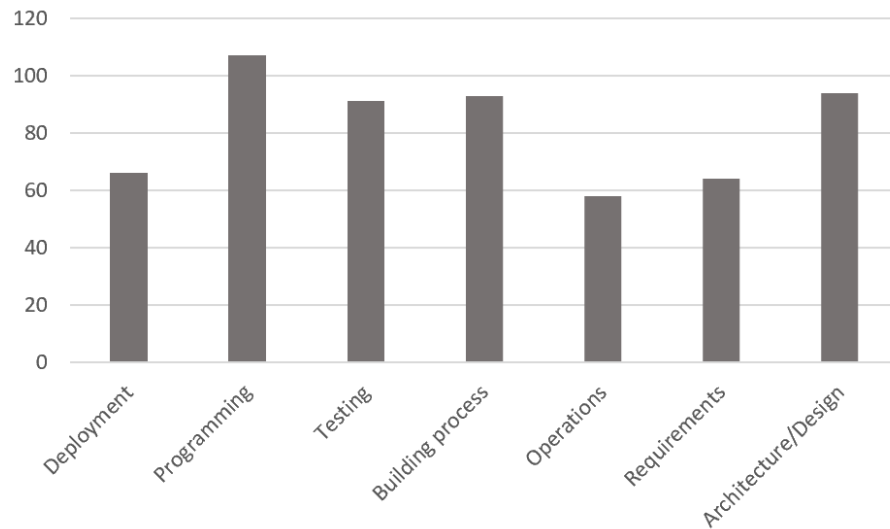


Figure 3.10: In which activities is your team involved? N=114 (Y-axis is the number of responses)

3.2.3 Ethical considerations

Participation in the study was voluntary. The participants in the interviews signed a consent form. For most questions, we provided an option for participants not wanting to give any details (i.e., “I don’t know”). In addition, we aligned our study to the data protection laws in Germany and the EU. Multiple researchers reviewed the questions at our institution, including one expert on professional trainings and surveys, the head of the department, the data protection officer, and one of the directors.

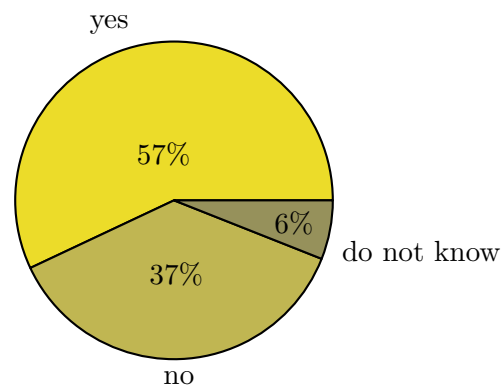


Figure 3.11: Does your team perform any activities related to the requirements phase? (N=134)

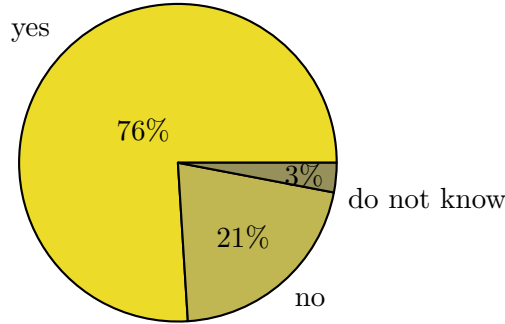


Figure 3.12: Does your team perform any activities related to the design phase? (N=134)

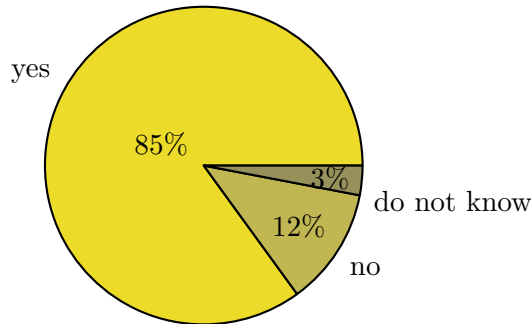


Figure 3.13: Does your team perform any activities related to the implementation and testing phase? (N=134)

3.2.4 Threats to Validity

According to Runeson et al. [RHRR12], we discuss construct validity, external validity, and reliability.

Construct Validity The set of questions used in the survey and the interviews are the outcome of several workshops in which software security researchers and practitioners with medium to high-security expertise participated. A possible threat is the level of expertise that these persons have which influences the quality of the questions. Moreover, we avoided the possibility that the interviewers asked wrong or irritating questions as we - the researchers that conducted this study - held the interviews ourselves. For gaining a representative survey, we made sure that the developers were selected as randomly as possible. Most developers of our survey were invited by a news article from the German publishing house Heise, and the minority were developers of our industry partners. Our industry partners have small to large companies in all branches of IT. All interview participants came from partners of Fraunhofer IEM. Thus, they were not a random set of product owners and managers in Germany. However, these companies differ significantly in size, domain, and business model. Moreover, we knew only four of the 17 interviewed persons upfront - thus, they were not influenced by us or our project AppSecure.nrw. We pre-tested our survey and interviews with people from our target group to identify whether our questions were understandable and interpreted as we intended and made changes due to this pre-test.

External Validity As our study was conducted only with companies from Germany, it might be the case that our results do not apply to companies outside of Germany as they have other standards and laws that they have to consider. However, several international standards like

ISO IEC 27001 exist.

Even though we made a pre-test to analyze whether our questions were interpreted correctly, it might be the case that the survey participants misinterpreted our questions. In our semi-structured interviews, this threat is reduced because our interviewees always had the chance to ask whether they understood our questions correctly. Our survey represents the intended target group as 256 software developers or similar roles have answered it. Nearly half of the participants came from the industry partners in our research project AppSecure.NRW. However, all these participants were voluntarily invited by their managers and not by us directly. Compared to the survey participants, our number of interviews might not be sufficient to extract a representative result as we only interviewed 17 experts that are product owner, manager, or both.

Reliability The questions of our study (survey and interviews) are based on our experience concerning secure software engineering. Other researchers might ask these questions differently. This is especially the case for our semi-structured interviews, as the follow-up questions depend on the interviewer. Moreover, we might have made mistakes while analyzing the interviews, especially when matching the interview answers to our pre-defined classes and when interpreting the complete encoding of our interviews. Also, we might have made mistakes while analyzing the survey, especially when analyzing the free text fields or drawing conclusions. To prevent all these mistakes, we peer-reviewed all our work. Additionally, we contribute all our data and materials as an open artifact for future reproductions. To minimize human errors while analyzing our survey, we used as many automated tools as possible, e.g., we used Survey Monkey to collect and export the survey data automatically and recorded all interviews and used a reliable voice-to-text software for the transcription. With a self-created script, we automatically processed all raw data wherever possible.

3.3 User Study

Most issues reported by developers can be handled by choosing the correct configuration for the target program being analyzed. To enable this, tool vendors provide a wide range of configuration options. For example, developers can select rules or write custom ones using a domain-specific language, set different thresholds for the analysis engine, or select the target program’s scope.

As learned from the survey results, most participants are willing to configure their tools. They are willing to label findings as correct or incorrect and even modify or specify their own custom rules for the tool they use. Qui et al. [QWR18] and Mordahl et al. [MW21] reported the impact of the configuration on the results. While these studies executed and compared different tools to reason about the configuration impact, we decided to involve the actual users of the tools and, in a controlled experiment, analyze how users configure a given tool. We study the configuration options of SAST tools *integrated into an IDE*. Within an IDE, users often have a specific context in which they work, e.g., the code last written or edited or a vulnerability relevant to a specific component of the project for which the user is responsible. When this is the case, the SAST tool can be configured to provide the results quickly by focusing on such a limited scope [NQDAL⁺17].

This user study investigates the configuration options that impact precision, recall, and analysis time. We omit other options such as filtering and prioritization of warnings, as they are more relevant when a project is analyzed as a whole. This is often the case when the analysis time is not that critical, e.g., during nightly builds. Instead, we investigate two specific options: (1) selection of security rules and (2) selection of analysis scope. We chose these options based on the insights from previous studies [NQDWA20, CB16], showing that users find them highly relevant.

Through the user study, we answer the following research questions:

RQ4 Can users resolve findings in SAST tools more effectively by configuring the tools’ analysis scope and rule selection?

RQ5 What strategies do users of static analysis tools in the IDE use to resolve the findings?

RQ6 Which configuration options in the existing tools do users find useful?

The dataset and all materials we created and used for this research are available as an artifact via the link <https://research-sast-config.github.io/>.

3.3.1 Study Design

To counter-act possible learning effects by study participants, we applied a between-subjects study design [CGK12] consisting of a lab study and a semi-structured interview to find out to what extent the configuration options in SAST tools are helpful to end users.

We recruited the participants through our contacts in seven companies with software development departments. We had a single point of contact that internally distributed our invitation to potential participants, either developers or security experts. As the number of security experts was insufficient, we additionally invited Ph.D. candidates doing research in security or program analysis. Previous studies have shown that graduate students are valid proxies for such studies [NDGS20, NDTS18, NDG⁺19, NDT⁺17]. The participation was voluntary and without compensation. In total, we had 40 participants (P01-P40), 24 from industry, and 16 Ph.D. candidates from academia. Twenty-three participants have mainly software development responsibilities, while 17 are security or program-analysis experts. Table 3.4 lists the profile of each participant. The columns *From* and *Role* were collected prior to the session with the participant, based on the information provided by the point of contact. The columns *Coding* and *Security* originate from a self-assessment by the participant collected during the session. The column *Study Type* is the allocation of the two types of experiments, which we explain in the following section. This allocation was done randomly, separately for each group, industry and academic participants. Lastly, we asked for a review of the questions we used during the user study in terms of study design and ethical opinion. Reviews were done by our team leader and one senior external researcher with experience in empirical studies for usable security.

Usage scenario. The study focuses on users of SAST tools during development time. Our selected tool, SECUCHECK [PKB21], runs within the IDE in the background while the user can review or edit the code before checking it into the remote repository.

The user has a fixed time to use the tool and decide which of the findings from the tool are true positives and which are false positives. To make this scenario as realistic as possible, we provide each participant with a relatively small project with meaningful logic and a complete Java application (see below). Each participant was given time to familiarize herself well with the code before performing the tasks. The moderator explicitly asked the participants whether they were familiar enough with the code in order to be able to resolve the findings from SECUCHECK.

Each participant was given a time of 15 minutes to run the tool, look into the findings, and resolve them. The participant was asked to resolve the finding by stating if the finding is a true positive or false positive or if the participant cannot decide due to missing expertise or any other reason. We randomly divided the participants into two groups. One group, the treatment group, was able to use the configuration page of our tool freely and run the tool multiple times with different configurations. They could use two configuration options, i.e., selection of security

Table 3.4: Participants' profile.

	From	Role	Coding experience (years)	Security experience	Study Group
P01	industry	developer	3-5	beginner	control
P02	industry	developer	6-9	beginner	treatment
P03	industry	developer	10+	knowledgeable	control
P04	industry	developer	6-9	knowledgeable	control
P05	industry	developer	10+	knowledgeable	control
P06	industry	developer	10+	beginner	treatment
P07	industry	expert	10+	knowledgeable	treatment
P08	industry	developer	10+	knowledgeable	control
P09	industry	expert	10+	knowledgeable	control
P10	industry	developer	6-9	knowledgeable	treatment
P11	industry	developer	3-5	beginner	control
P12	industry	developer	10+	beginner	control
P13	industry	developer	6-9	beginner	treatment
P14	industry	developer	6-9	knowledgeable	control
P15	industry	developer	10+	beginner	control
P16	industry	developer	6-9	beginner	treatment
P17	academia	developer	6-9	knowledgeable	control
P18	academia	expert	6-9	knowledgeable	treatment
P19	industry	developer	10+	knowledgeable	treatment
P20	academia	developer	6-9	beginner	control
P21	academia	expert	6-9	beginner	treatment
P22	academia	expert	3-5	beginner	treatment
P23	academia	expert	10+	knowledgeable	control
P24	academia	expert	3-5	beginner	treatment
P25	academia	expert	10+	beginner	treatment
P26	academia	expert	10+	expert	treatment
P27	industry	developer	3-5	beginner	treatment
P28	academia	expert	6-9	beginner	control
P29	academia	expert	6-9	knowledgeable	control
P30	academia	expert	6-9	beginner	treatment
P31	industry	developer	10+	knowledgeable	treatment
P32	industry	expert	10+	beginner	control
P33	academia	developer	6-9	expert	control
P34	industry	expert	10+	knowledgeable	treatment
P35	industry	expert	3-5	expert	treatment
P36	academia	expert	3-5	knowledgeable	control
P37	industry	developer	6-9	knowledgeable	control
P38	academia	expert	10+	beginner	treatment
P39	academia	developer	3-5	beginner	control
P40	industry	developer	3-5	knowledgeable	treatment

rules and selection of classes to be analyzed. Thus, depending on the selection chosen by the participant, not all possible findings were shown by the tool at once.

The control group used only the default configuration, in which all security rules and all classes were selected. Hence, the control group ran the tool only once and saw all possible findings simultaneously. Our experience with SAST tools is that most default configurations will include all available security rules and will analyze the entire project. To simplify the task, we only used taint-style vulnerabilities in our user study. Through the program statements, a taint analysis tracks untrusted (tainted) values from so-called source statements (such as user input values from Http-request object) to security-relevant operations, so-called sink statements (such as writing to file or database). We chose taint analysis because it can be used to detect most [PDB19] of the top 25 popular vulnerabilities by the SANS institute [Mit21a].

The session was organized as follows. It starts with short introductions and clarification of what the study is about and what data is collected. Then, the moderator performs the first two parts of the interview (part one is general meta-questions and part two is general questions on SAST experience). Before giving the task, the moderator explains the main concepts of taint analysis and how it works to find an SQL injection as an example. The moderator also makes a walkthrough with SECUCHECK to demonstrate its features. Then, the participant is provided with the Eclipse IDE and the project under analysis in the workspace. The participant is given time to familiarize herself with the project after which the task is explained and given 15 minutes to work. After the task, the moderator starts with the second two parts of the interview: part three is a discussion about the task, the strategies, and feedback on the tool, and part four is the questionnaire for the configuration options in FORTIFY and CHECKMARX.

SAST tool Previous studies have shown that users prefer SAST tools that run within their IDE [NQDWA20], using native features such as syntax highlighting, error view, hover messages, etc. Such integration enables the users to quickly locate and understand the code’s findings. For our user study, we selected the existing tool SECUCHECK [PKB21] (see Chapter 6 for more details), which is implemented with the Magpie Bridge [LDB19] framework that enables native integration within the IDE. Due to the underlying Language server protocol [Mic21] used by Magpie Bridge, SECUCHECK can be used in multiple IDEs.

For this study, we used the Eclipse client, in which the analysis results are shown in the error view with click navigation to the relevant file. The standard markers on the sidebar from the editor mark relevant statements (sources and sinks) for each finding. For the configuration, SECUCHECK has a custom page that is shown in the web browser via HTTP. Using this page, the participants can select the security rules and the classes to be analyzed via check boxes and trigger the analysis by clicking a button. Figure 3.14 shows a screenshot of the view of the IDE and the project given to the participants, while Figure 3.15 shows a screenshot of the configuration page. None of the participants have used SECUCHECK before the user study.

Target project and built-in vulnerabilities We considered several criteria for selecting the target project used in the user study. First, the project should be realistic such that participants can see clear functionalities and business logic implemented in the code. Second, it should be relatively small so that participants can understand the code in the limited time they are given and be comfortable resolving potential security vulnerabilities. Third, there should be multiple security vulnerabilities that SECUCHECK will report, including true and false positives. We used an existing Java Spring¹⁰ project to showcase different security vulnerabilities. The project implements a simple task management tool where users create, delete, and edit tasks stored in MySQL database. The application uses the Spring MVC architecture. It consists of 35 classes

¹⁰<https://spring.io/>

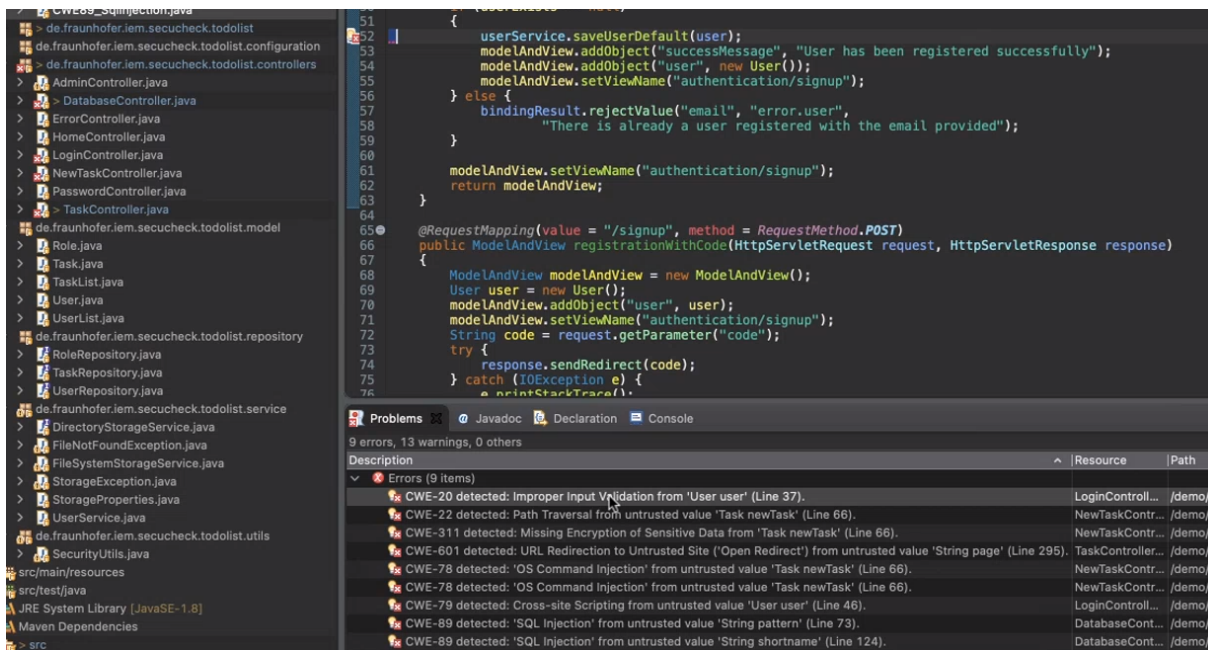


Figure 3.14: IDE view of the workspace in which participants worked on the given task. The bottom view shows the findings that the tool reports in default configuration.

Welcome to MagpieBridge Configuration

Configuration

clanJW: java

☒ FluentSQL Specification files

- ☒ SimpleSQLInjectionSpec.java
- ☒ SQLiWithPreparedStatementsSpec.java
- ☐ ServletSQLInjectionSpec.java
- ☒ CommandInjectionSpec.java

☐ Select java files for entry points

- ☐ COCPlayers.java
- ☒ COCPlayersTest.java
- ☐ COCServerConnectionException.java
- ☒ Clan.java
- ☐ ClanJWException.java
- ☐ ClanSearch.java
- ☐ ClanSearchFactory.java
- ☐ ClanSearchTest.java
- ☐ ClanTest.java
- ☐ DarkElixirSpell.java

Actions

clanJW: java

Run Analysis

Figure 3.15: Configuration page used in the tool for the user study

and nine findings that SECUCHECK reports. The nine findings correspond to seven unique, real vulnerabilities and two false positives. Table 3.5 lists the findings reported by the tool when all rules and classes are selected, showing the vulnerabilities type (common weakness enumeration - CWE¹¹), the name of the class in which the vulnerability is located, and whether it is a true or false positive. The taint analysis in SECUCHECK uses rules specified in Java fluent interface. These files were also available to the participant for inspection. The participant was not required to make any changes to the rules or to the code.

Table 3.5: Findings reported by the SAST tool on the target project used in the user study.

ID	CWE	Name	Location	TP/FP
F01	20	Improper input validation	LoginController.java	TP
F02	22	Path traversal	NewTaskController.java	TP
F03	311	Missing encryption	NewTaskController.java	TP
F04	601	Open redirect	TaskController.java	TP
F05	78	OS command injection	NewTaskController.java	FP
F06	78	OS command injection	NewTaskController.java	TP
F07	79	Cross-site scripting	LoginController.java	TP
F08	89	SQL injection	DatabaseController.java	TP
F09	89	SQL injection	DatabaseController.java	FP

Semi-structured interview The interview consisted of four parts, (1) meta-questions, (2) questions on the experience with SAST tools, (3) discussion and feedback, and (4) a questionnaire. In part one, we asked three questions (**Q1-3**) about previous coding experience and security expertise. Part two comprised 11 questions (**Q4-14**) about previous experience with SAST tools, e.g., when and how tools are used, who configures the tools, which tools are used, and any company-related culture and processes for using SAST tools. This part was asked before the task to give the participants more context and recall their experience with SAST tools. The data collected in this part helps us understand the background of our population. Part three consisted of 9 questions (**Q15-23**). The moderator asked about the experience with the task and the tool to collect feedback. Additionally, the treatment group participants were asked to explain their strategies when using the configuration page. Finally, part four included two questions (**Q24-25**) that listed the configuration options available in two commercial SAST tools, FORTIFY and CHECKMARX and asked the participants to label each option if it is *understandable*, and whether it is *useful* for their role. The questionnaire only listed the option names as they appear in the tools. The participants were encouraged to ask if they needed an explanation for that option, in which case we provided them with a further description based on the official documentation of the tools. We selected all relevant options for precision, recall, or analysis time. We used the official documentation of the tools. The names of the tools were not revealed. **Q24** contains 18 options from FORTIFY and **Q25** contains 11 from CHECKMARX.

Calculating effectiveness We explain how we calculate the effectiveness in resolving the findings reported by SECUCHECK. In a limited time of 15 minutes, each participant was asked to use the tool, observe the findings, and, based on the code, assess if each finding is *true positive*, *false positive*, or *cannot assess*. The maximum number of findings that the tool reports and each participant can resolve is nine. We count the ratio of correctly resolved findings out of the total number of findings that the participant resolved. We refer to this as *effectiveness in resolving the findings* (do the right thing). We prioritize quality over quantity. We do not look into the efficiency, i.e., more number of (correct) resolutions in less time, because security

¹¹<https://cwe.mitre.org/data/index.html>

is not a property where users should compete, nor feel pressure for quantity. In other words, to do the task effectively, we care about the quality of each resolution. When the participant resolved X findings of which Y were resolved correctly as true or false positive, we report the effectiveness as the ratio Y/X , expressed in percent. This way, we may get high effectiveness for participants who resolved almost no issue. However, due to the nature of the task, we believe that incorrect resolutions are worse than a high number of resolutions. Hence, leaving this to the experts is better than making a wrong resolution that may lead to vulnerabilities not being fixed. Moreover, with our study design, calculating efficiency (resolving more findings in the given time) would give an advantage to the control group since these participants see all possible findings at the beginning, while the treatment group may choose configurations that will not show all findings in the application.

Data collection The user study was conducted between December 2020 and January 2021. After the initial contact with each participant via e-mail, which provided basic information about the study and the data we planned to collect, we arranged a virtual session over Microsoft Teams. On average, the sessions took 75 minutes. The participants did not need to prepare or install any software. All required tools were prepared by the moderator who shared the screen and, when needed, gave control to the participant to perform the task. All sessions were recorded for post-processing. We invited a researcher with experience conducting user studies to moderate all sessions. To adapt and verify our design, we performed four test runs with students from our group.

As an IDE, we used Eclipse, where we installed the SECUCHECK tool running as a MagpieBridge Server [LDB19]. For the semi-structured interview, we used *Google Forms*, where the moderator collected answers. Most questions were of closed type. The answers of the few open questions (Q16-19, 22-23), which were in part three, were collected in the post-processing phase.

3.3.2 Results

Resolving findings in configurable tools effectively (RQ4) We asked each participant during the task to clearly state which finding is a *true positive* or *false positive*, or if the participant cannot answer (*do not know*). Table 3.6 summarizes the data per finding and for each study group. The treatment group, which consisted of participants allowed to configure the tool, resolved 76% of SECUCHECK’s findings correctly, while the control group only resolved 61% correctly (effectiveness as described in Subsection 3.3.1).

	Control group			Treatment group		
	Correct	Incorrect	DoNotKnow	Correct	Incorrect	DoNotKnow
F01	12	5	2	16	1	0
F02	10	4	5	12	3	0
F03	8	7	3	9	3	2
F04	14	0	2	13	0	0
F05	8	7	1	9	4	0
F06	5	8	1	9	2	1
F07	11	3	1	8	3	3
F08	15	1	0	17	1	0
F09	6	7	0	9	7	1
Sum	89	42	15	102	24	7

Table 3.6: Number of resolved findings per study group and per finding (F01-F09). T = true (correctly resolved), F = false (incorrectly resolved), DN = do not know.

Participants using the configuration page resolved the findings reported by SECUCHECK more effectively than those using a single default configuration. However, using the configuration page requires more engagement with the code and/or with security vulnerabilities.

Strategies users have when configuring static analysis tools (RQ5) To answer RQ5, we collected the usage data of the configuration page from the participants in the treatment group. Additionally, we gathered qualitative data from the answers in the third part of the interview. We asked two questions about the strategies each participant used. First, *What was your strategy when you used the configuration page?* (Q22), and second, *Would you use the same strategy if this was your own project?* (Q23). The moderator asked further questions based on the answers to gain more details. Among all answers, we identified four different strategies that the participants named. During the task, all participants used one or two of these strategies, i.e., some of them decided to change their strategy at some point. The first strategy, *AllEntriesSubsetRules*, is to use all possible entry points and select only a subset of the vulnerability rules for each run of the analysis. Most participants who used this strategy selected a single vulnerability rule per configuration. When asked why they decided to do this, there were only two reasons: first, to avoid **being overwhelmed by a high number of findings**, and second, to **avoid long running times** of the analysis if everything is selected. One participant said, *"I get overwhelmed when I get too many results to resolve"*, and another said, *"I was not aware how long would it take to check the vulnerability so selected incrementally"*.

The second strategy, *PairingEntryAndRule*, is to select a combination of entry points and vulnerability rules. When asked why one participant said, *"I tried to match a vulnerability with an entry point that makes sense based on the name"*. The third strategy, *SelectAll*, is to select everything. All participants mentioned that the main reason was to ensure they did not miss any vulnerability. The last strategy, *AllRulesSubsetEntries*, is to select all vulnerability rules but make different configurations by selecting a subset of the entry points.

Table 3.7 shows the strategies that each participant used or mentioned during the interview. The participants applied one or two strategies while solving the given task. With 13 participants using it, *AllEntriesSubsetRules* was the most used strategy. With six participants, the second-most used strategy was *SelectAll*. Based on the interview, we learned that the participants did not use *AllRulesSubsetEntries* during the task as most of them said they were not familiar with the code given and therefore did not know how to choose which classes would be relevant as entry points. However, when asked if that was their own project whether they would use a different strategy, *AllRulesSubsetEntries* was chosen by most participants, with 13 answers, followed by *PairingEntryAndRule* with nine answers.

On average, the participants re-ran the tool, i.e., selected different configurations, 3.4 times (with $\sigma = 1.79$), with maximally six times done by four participants. Three participants ran the tool only once (P26, P31, and P38), and they all used *SelectAll*. In doing so, these three participants performed the task the same as the control group, i.e., did not make use the configuration option. Interestingly, they resolved 24 findings in total and correctly resolved 14 of them, yielding 58%, which is the lowest in the group, and conforming to the result of RQ1. Participant P31 would always use *SelectAll*, saying, *"All at a time, all selected! Even if the analysis was much slower, I would leave it run over the night, and get everything. But I would like to have a lazy loading. I should see the relevant findings for the file that I open and not the rest"*.

Table 3.7: Strategies for using the configuration page.

	Strategy used during the task	Strategy would use on own project	Number of configurations submitted	Inspected the rules
P02	<i>PairingEntryAndRule</i> then <i>SelectAll</i>	<i>PairingEntryAndRule</i> then <i>AllRulesSubsetEntries</i>	6	no
P06	<i>AllEntriesSubsetRules</i>	<i>AllRulesSubsetEntries</i>	6	no
P07	<i>PairingEntryAndRule</i> then <i>SelectAll</i>	<i>PairingEntryAndRule</i> then <i>SelectAll</i>	2	no
P10	<i>PairingEntryAndRule</i> then <i>AllEntriesSubsetRules</i>	<i>PairingEntryAndRule</i> then <i>AllRulesSubsetEntries</i>	2	yes
P13	<i>AllEntriesSubsetRules</i>	<i>AllEntriesSubsetRules</i> then <i>AllRulesSubsetEntries</i>	4	no
P16	<i>AllEntriesSubsetRules</i>	<i>AllEntriesSubsetRules</i> then <i>PairingEntryAndRule</i> <i>AllRulesSubsetEntries</i>	6	yes
P18	<i>AllEntriesSubsetRules</i>	<i>AllRulesSubsetEntries</i> then <i>PairingEntryAndRule</i>	2	no
P19	<i>AllEntriesSubsetRules</i>	<i>AllEntriesSubsetRules</i> then <i>SelectAll</i>	3	yes
P21	<i>AllEntriesSubsetRules</i> then <i>SelectAll</i>	<i>AllRulesSubsetEntries</i>	2	yes
P22	<i>AllEntriesSubsetRules</i> then <i>SelectAll</i>	<i>AllRulesSubsetEntries</i>	2	yes
P24	<i>AllEntriesSubsetRules</i>	<i>AllEntriesSubsetRules</i> then <i>AllRulesSubsetEntries</i>	3	yes
P25	<i>AllEntriesSubsetRules</i>	<i>PairingEntryAndRule</i>	5	yes
P26	<i>SelectAll</i>	<i>AllRulesSubsetEntries</i>	1	yes
P27	<i>AllEntriesSubsetRules</i> then <i>SelectAll</i>	<i>AllRulesSubsetEntries</i>	4	yes
P30	<i>AllEntriesSubsetRules</i>	<i>AllRulesSubsetEntries</i> then <i>PairingEntryAndRule</i>	5	yes
P31	<i>SelectAll</i>	<i>SelectAll</i>	1	yes
P33	<i>AllEntriesSubsetRules</i>	<i>AllRulesSubsetEntries</i> then <i>PairingEntryAndRule</i>	4	yes
P34	<i>AllEntriesSubsetRules</i>	<i>PairingEntryAndRule</i> then <i>SelectAll</i>	3	yes
P38	<i>SelectAll</i>	<i>AllEntriesSubsetRules</i> then <i>AllRulesSubsetEntries</i>	1	yes
P40	<i>AllEntriesSubsetRules</i> then <i>SelectAll</i>	<i>PairingEntryAndRule</i> then <i>SelectAll</i>	6	no

Most participants randomly selected all entry points and ran the configuration individually for each vulnerability rule. When asked if they would use the same strategy if they used the tool on their project, most of them would use the selection of entry points to know where to look for particular vulnerabilities. To avoid missing findings, a few participants would only use the default configuration, where all entry points and vulnerability rules are selected. However, these participants were comparatively ineffective.

Configuration options in commercial tools (RQ6) To answer **RQ6**, we use the data collected in part four of the questionnaire, i.e., **Q24-25**, where the participant was asked to evaluate the understandability and the usefulness of the configuration options in FORTIFY and CHECKMARX. Figure 3.16 shows a screenshot of how these options were presented to the participant. For the options, we used the exact formulation as defined in the user documentation of the tools. We screened all configuration options these tools provide and selected the options that impact the precision, recall, or analysis time. In total, we selected 18 options from FORTIFY (F1-F18) and 11 options from CHECKMARX (C1-C11).

Figure 3.17 shows four boxplots, two for FORTIFY and two for CHECKMARX. Each FORTIFY option is a single data point in the first two boxplots (blue and yellow). Each CHECKMARX option is a single data point in the second two boxplots (green and red). The first boxplot (blue) shows the understandability of the FORTIFY options in percent. The mean is 73.53 ($\sigma=19.69$), i.e., 73.53% of the options provided to the users are found to be understandable by the participants. There are four outliers in the upper part, F7 (*noDefaultRules*), F9 (*rules*), F11 (*dataflowMaxFunctionTimeMinutes*), and F12 (*maxFunctionVisits*), which are options that are found understandable by most participants (over 85%). There are also four outliers in the lower part, F6 (*noDefaultIssueRules*), F10 (*enableInterproceduralConstantResolution*), F13 (*maxTaintDefForVar*), and F14 (*maxTaintDefForVarAbort*), which are options that were found least understandable (under 63%). The second boxplot (yellow) shows the understandability and usefulness of the FORTIFY options in percent. The mean is 78.12 ($\sigma=13.4$). There are four outliers in the upper part (over 85%), F1 (*filter*), F2 (*disableSourceBundling*), F4 (*analyzers*), and F9 (*rules*), and four in the lower part (under 70%), F3 (*disableLanguage*), F8 (*noDefaultSinkRules*), F10 (*enableInterproceduralConstantResolution*), and F14 (*maxTaintDefForVarAbort*). The third boxplot (green) shows the understandability of the CHECKMARX options in percent. The mean is 73.86 ($\sigma=17.98$). There are three outliers in the upper part, C2 (*maxQueryTime*), C6 (*scanBinaries*), and C11 (*maxQueryTimePer100K*), which are options that are found understandable by most participants (over 88%). There are also three outliers in the lower part, C2 (*maxQueryTime*), C3 (*useRoslynParser*), and C11 (*maxQueryTimePer100K*), which are options that were found least understandable (under 60%). The fourth boxplot (red) shows the understandability and usefulness of the CHECKMARX options in percent. The mean is 78.85 ($\sigma=21.15$). There are three outliers in the upper part (over 95%), C1 (*excludePath*), C5 (*maxQueryTime*), and C6 (*scanBinaries*), and three in the lower part (under 66%), C4 (*languageThreshold*), C9 (*maxFileSizeKb*), and C10 (*maxPathLength*). There are no significant differences when comparing the mean and standard deviation between both tools.

Finally, we categorize the options into three categories. Category (1) includes options that are related to the analysis scope (F1-3, C1, C4-7), Category (2) includes options that impact the approximations done by the solver or set thresholds for analysis time (F4-5, F10-18, C2-3, C8-11), and Category (3) includes options that are related to rule selection (F6-9). The most understandable category is (2), with an average score of 32.5 but with the least percentage of usefulness 70%. Category (1) has an average score of 29.25 and most usefulness with 85%,

Section 4: Tool 2 Configuration Options

Read the configuration options available in a commercial SAST tools (Tool 2) and evaluate if they are understandable and useful. These are options used to configure different properties of the SAST tools, similar to the configurations performed for SecuCheck.

[C1] EXCLUDE_PATH

Semicolon separated list of file names to exclude from the scan (e.g. file1;file2;file3). Include only file names, not paths.

[C2] MAX_QUERY_TIME

Defines part of a formula to calculate the maximum execution time allowed for a single query. After the set time, the query execution is terminated, the result is empty and the log indicates that its execution failed.

[C3] USE_ROSLYN_PARSER

Enable the use of Roslyn parser to scan C# files.

[C4] LANGUAGE_THRESHOLD

Sub-setting of MULTI_LANGUAGE_MODE. The minimal percentage of complete number of files required to scan a language. Should be set to 0.0 (and MULTI_LANGUAGE_MODE=2) to match the Portal_s Multi-language mode. See MULTI_LANGUAGE_MODE parameter for more details.

[C5] MULTI_LANGUAGE_MODE

Defines which languages the application should scan. 1 = One Primary Language, 2 = All Languages, 3 = Matching Sets, 4 = Selected Languages.

C1-C5 Options

	Understandable	Useful
C1	<input type="checkbox"/>	<input type="checkbox"/>
C2	<input type="checkbox"/>	<input type="checkbox"/>
C3	<input type="checkbox"/>	<input type="checkbox"/>
C4	<input type="checkbox"/>	<input type="checkbox"/>
C5	<input type="checkbox"/>	<input type="checkbox"/>

Figure 3.16: Screenshot of **Q25** of the questionnaire

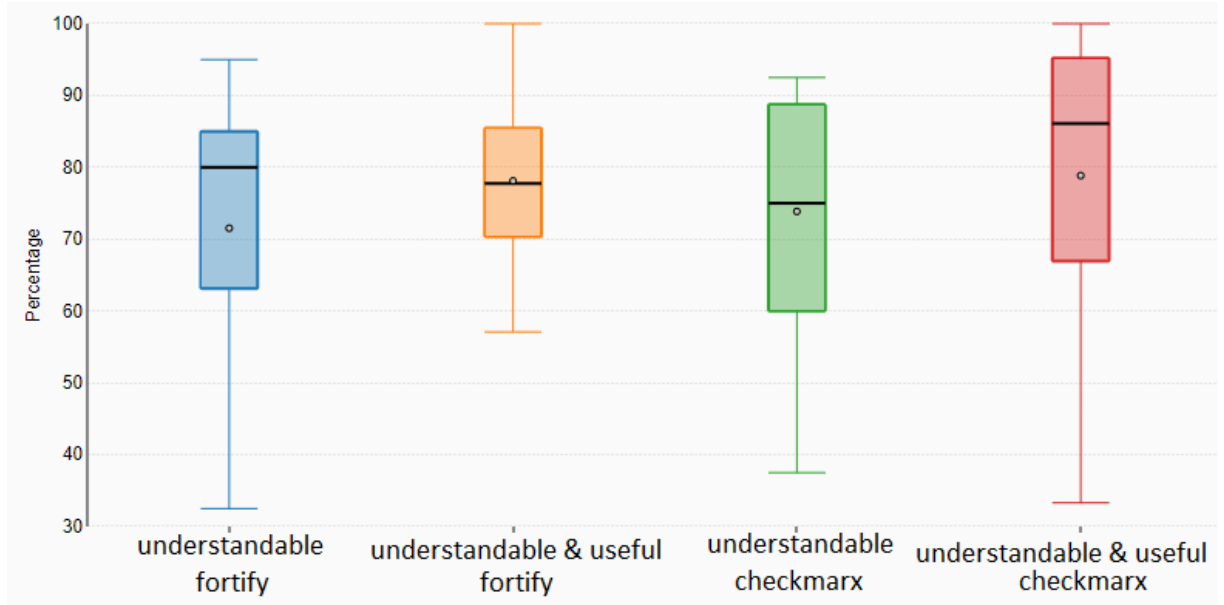


Figure 3.17: BoxPlots for percentage of participants that marked each option as *understandable* and *useful* for each tool, FORTIFY (blue and yellow) and CHECKMARX (green and red)

whereas (3) has a score of 28 and 74% usefulness.

A quarter of the participants find the provided options from the selected commercial SAST tools to be not understandable. The other three quarters find the options to be, on average, 78% useful. The most useful are the options related to the analysis scope.

Recommendations for building future SAST tools. After the participants had experienced the tool, in part three of the interview, we also asked questions about the experience with the tool. In particular, we asked what features of the tool the participants liked and disliked and what other features they would like to have when working on a similar tool within the IDE. Moreover, in part two, we asked questions about their previous experience with SAST tools. In the following, we present the results. Based on these results and the results from the previous sections, we propose a list of recommendations for future SAST tools.

Twenty-four of the 40 participants said they use SAST tools in their everyday workflow. Out of these, 17 use SAST tools within the IDE. The tools that participants named are shown in Figure 3.18. When asked if they configure those tools, 30 participants said they do not, while only ten said they do. Only eight participants said they used domain-specific language to configure the tools. During the task, we also made the vulnerability detection rules available to the users and let them know they were available. Then we observed how many of them opened the files and inspected them. As seen in Table 3.7 (page 40), most of them did. However, many said that they would not write the rules on their own, but they like to have them available to help them decide if the findings reported by the tool are true or false positives. For example, one participant checked the rule to verify if the tool was aware of potential sanitizers. Another participant said she prefers writing the rules and never uses default rules, as only then is she sure that the tool is doing the right job. Twenty-seven participants said there are culture and processes by the company for using SAST tools. Twenty-two participants said they are allowed to configure the tools, while 13 answered they are not.

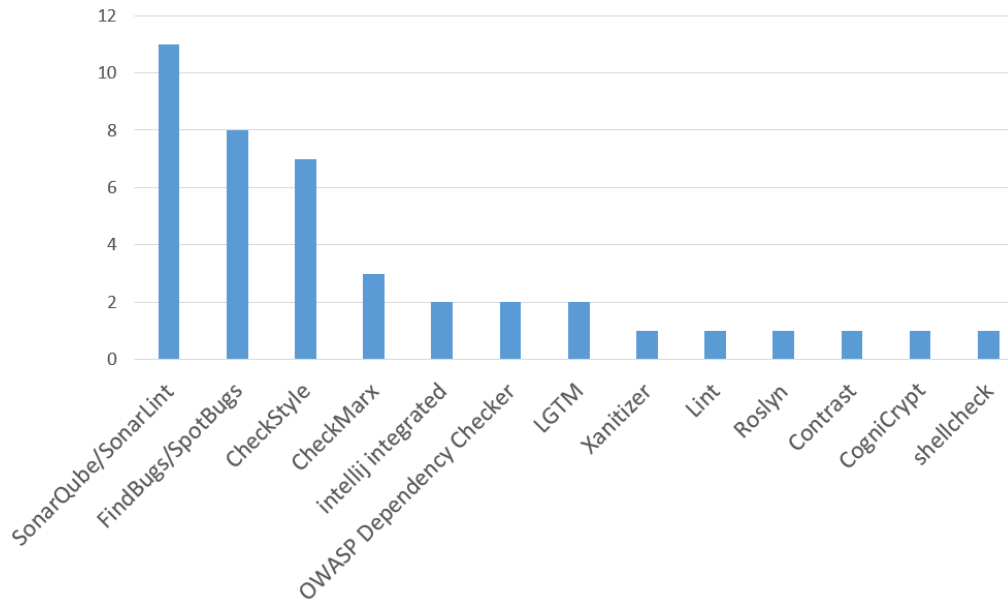


Figure 3.18: Tools that participants use or used in the past and number of participants that named each tool.

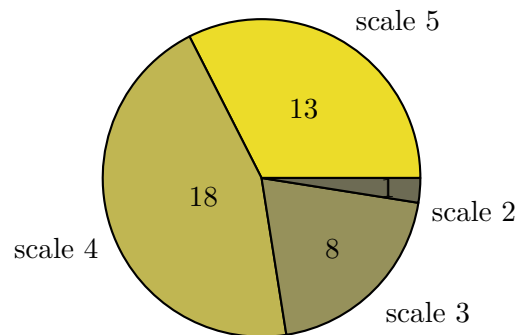


Figure 3.19: How relevant is the following statement on a scale from 1 (least relevant) to 5 (most relevant): *The issues reported by the tool should be relevant for me (the context I am currently focused on)?*

Recommendation 1: SAST tools should provide the rules of the analysis to the users for inspections or modification.

In part three of the interview, we asked the participants to name what they liked, disliked, and missed about the tool they used to perform the given task. Most of the participants liked the integration of the analysis within the standard features of the IDE, such as error view list, error markers, clicking links to the findings, etc. The most disliked feature of the tool was that the configuration page was in the web-browser and not within the IDE. This was perceived as a usability issue due to context-switching. We contacted the authors of MagpieBridge, which we used to build our tool, and they extended the framework to allow the page to be opened within the IDE if the respective IDE supports this. Additionally, we pointed out a few other usability issues and cooperated with them to make the framework support more UI elements for the configuration page. Among the features that the participants missed, there were two which were mentioned frequently. First, the participants wanted to better visualization of the

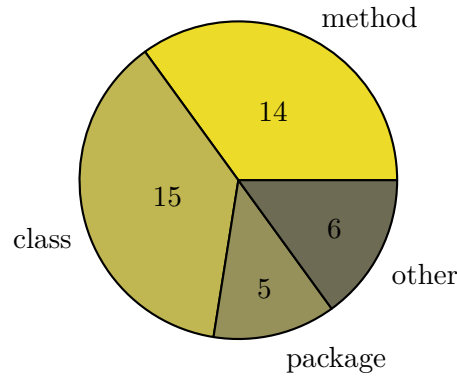


Figure 3.20: On what level would you prefer the entry points selection option to be?

data-flow path between the source and the sink. Second, participants said our tool had a limited description of the vulnerability reported in the error view. They prefer to see a feature where the description can be expanded to show further details, examples, and possible fixes or proposals validation libraries.

Recommendation 2: The SAST tools should have full integration within the IDE, and when reporting, they should have a rich explainability mechanism for data-flow paths and educational information for the vulnerabilities that are reported.

Previously, we learned that many participants would use the entry points selection if they worked on their project (Table 3.7). Additionally, in part two of the interview, we asked the participants to rank the following statement *"The issues reported by the tool should be relevant for me, i.e., the context I am currently working on."*. Figure 3.19 shows the distribution of the responses showing that for most participants, the reported issues should be relevant to the context. We also asked the participants about the granularity level of the entry points. In our study, we chose entry points to be on the class level. As answers, we offered method level, class level, package level, or others. Figure 3.20 shows the distributions of the answers. Under *Other*, we received two answers "all", two answers combination of two of the given options, one answer "annotations", and one answer "hierarchical starting at package level". Based on this data, we make the next recommendation.

Recommendation 3: The SAST tools in the IDE should provide options for the analysis scope in which the users can select which parts of the project should be analyzed. Popular choices include selection of methods or classes as entry points of the analysis.

Finally, we refer to the results from our user study, where we learned that the participants from the treatment group were able to resolve the findings more effectively than participants from the control group. We learned from the data that most participants used the vulnerability rules as the primary selection criterion. They did not use the entry points as they were not very familiar with the code. This is a realistic scenario where security or quality assurance teams are performing the analysis for code they have not written themselves. We also observed that all participants started with resolving the *SQL injection* [Mit21c] findings. This is one of the most popular vulnerabilities nowadays. This shows that the users of SAST tools will probably focus on the vulnerabilities they know and are more likely to solve.

Recommendation 4: The tools should enable users to selectively enable or disable the vulnerability rules.

3.3.3 Threats to Validity

Construct validity. A possible threat to the validity of the user study relates to its setup. All participants performed the study remotely by sharing voice, video, and screen. During the task, the moderator noted the outcome of the findings that the respective participant resolved. To mitigate the risk of human mistakes, the first author watched all videos in the post-study processing phase to collect and confirm the results. As a result, we had a 100% inter-rater reliability score. Additionally, we collected notes of each session from which we created an artifact from our study.

Internal validity. To avoid any random answers to the questionnaire, we asked the participant to share the screen and give further comments to some questions while guided by the moderator in an interview style. To confirm the clarity of the questions and the timing of our study, we ran four tests with students from our research department.

External validity. Participation in the study was voluntary and without compensation. We asked our contacts from the industry to invite their software developers and additionally, we invited researchers, and students from Paderborn University. The invitation explained that the user would evaluate the configuration capabilities of SAST tools. This information makes it more likely that the participants have some interest in security and static analysis. However, this might make our population biased towards SAST tools.

We consciously chose a study design that would yield high internal validity at the necessary cost of limiting external validity [SSA15]. This is limited by the tool used for the study and the choice of an example project. The tool is limited to taint analysis, while other SAST tools also include other types of analyses. However, the most popular vulnerabilities are of taint-style, and the core of most SAST tools is a taint analysis [PDB19]. The project we used is artificially created, but includes different components that modern web applications would have. The vulnerabilities within the application are inserted based on existing vulnerabilities that we found in the OWASP benchmark [Ben21] and OWASP Webgoat [OWA]. We decided to use our own tool as this gave us control over what features to include and exclude.

The fact that several participants noted that the analysis was fast compared to what they expect from a SAST tool is due to the reason that our example project is relatively small compared to most real-world projects in the industry. Since this may impact the strategies that the participants used and discussed in Q22, we additionally asked in Q23, i.e., how they would have used the tool if that was their project where the analysis time would be an essential factor. This question allowed us to gain further insights relating to more real-world situations, strengthening external validity.

Lists of security-relevant methods (SRM) are generally created manually by the analysis writers and, in larger companies, are often refined by dedicated security teams through manual work. This is important, as even lists used in commercial tools can be incomplete and thus, can cause the analyses to miss vulnerabilities or to signal false positives. This chapter presents SWAN (Security methods for WeAkNess detection), an approach that directly aids analysis users in detecting SRM in their code and the libraries they use. Compared to earlier work, SWAN detects two additional types of SRM: validators and authentication methods. This allows the analyses to detect more types of vulnerabilities. When compared to earlier work, SWAN shows higher precision. In addition, SWAN provides more granularity in the SRM lists, as it can differentiate between different vulnerability types in terms of CWEs (Common Weakness Enumerations).

4.1 Requirements

To detect SANS 25 [Mit21a] problems, data-flow analyses need to be aware of critical method calls in the program that influence the computation of the analysis: the SRM. In our case, those are *sources*, i.e., methods that create the data that should be tracked (e.g., `getParameter()` at line 3), *sinks*, i.e., methods at which the analysis should raise an alarm (e.g., `executeQuery()` at line 7 and `sendRedirect()` at line 9), *validators*, i.e., methods at which the data becomes safe and should no longer be tracked (e.g., `encodeForSQL()` at line 4). In addition, *authentication methods* change the program's state from safe to unsafe or vice-versa (needed for CWE306 and CWE862). Supporting the SANS top 25 thus yields the following requirement for SRM lists.

R1 SRM should differentiate between sources, sinks, and validators.

When analyzing a program, the choice of SRM can heavily influence the outcomes of the analysis. For example, configuring an analysis with `executeQuery()` as a sink could make it detect SQL injections. Configuring the same analysis with `sendRedirect()` could make it detect Open Redirects. This is further illustrated in Table 4.1, where we detail the types of methods considered as sources, sinks, and validation methods for each of the CWEs.

For example, in CWE306 (Missing Authentication for Critical Function), methods requiring prior authentication are considered sinks since the analysis should report an error as soon as they are reached without proper authentication. Authentication methods are thus considered

Table 4.1: List of the SANS top 25 CWE [Mit21a] from 2009, and a description of the SRM required to detect them using data-flow analysis. CWE120 and CWE131 cannot happen in Java: exceptions are triggered before the issues are exploitable. CWE863 and CWE307 happen on the conceptual level, and should be detected at design time instead.

CWE	Description	Source	Validator	Sink
89	SQL Injection	External	Code sanitization	SQL execution
78	OS Command Injection	External	Code sanitization	Execution
120	Classic Buffer Overflow	N/A	N/A	N/A
79	Cross-site Scripting	External	Code sanitization	Save / Execution / Print
306	Missing Authentication for Critical Function	Entry point	Authentication	Critical function
862	Missing Authorization	Entry point	Authentication	Critical function
798	Use of Hard-coded Credentials	New string	Anonymization	Send / Save / Execute
311	Missing Encryption of Sensitive Data	External	Encryption	Execution
434	Unrestricted Upload of File with Dangerous Type	External / New string	Type check	Load
807	Reliance on Untrusted Inputs in a Security Decision	External	Code sanitization	Security function
250	Execution with Unnecessary Privileges	External / New object	Object sanitization	Execution function
352	Cross-Site Request Forgery	External	Code sanitization	Save / Execution / Print
22	Path Traversal	External	Path sanitization	File operation
494	Download of Code Without Integrity Check	External / New string	Integrity check	Load
863	Incorrect Authorization	Entry point	Incorrect authorization	Critical function
829	Inclusion of Functionality from Untrusted Control Sphere	External / New string	Input sanitization	Load
732	Incorrect Permission Assignment for Critical Resource	External / New object	Object sanitization	Execution function
676	Use of Potentially Dangerous Function	External / New object	Object sanitization	Dangerous function
327	Use of a Broken or Risky Cryptographic Algorithm	External / New object	Object sanitization	Cryptographic API
131	Incorrect Calculation of Buffer Size	N/A	N/A	N/A
307	Improper Restriction of Excessive Authentication Attempts	N/A	N/A	N/A
601	Open Redirect	External	URL sanitization	URL access
134	Uncontrolled Format String	External	Execution with format	Execution without format
190	Integer Overflow or Wraparound	External / New integer	Value test	Operation on integer
759	Use of a One-Way Hash without a Salt	Hashing function	Hashing function with salt	Hashing function

validators, unlike in CWE89 (SQL Injection), where validators are typically String sanitizers. This shows that SRM are vulnerability-type specific. An analysis configured with the wrong sets of SRM can easily cause false positives and negatives, yielding what practitioners tend to call a “bad signal”. To improve the signal-to-noise ratio and aid in categorization of the analysis warnings, it is therefore essential to relate the SRM to each CWE.

R2 SRM lists should be specific to each CWE.

In past work, SRM have been extracted from particular libraries and frameworks. SuSi, for example, lists all sources and sinks from the Android framework [ARB13]. However, this overlooks SRM in other third-party libraries and in the source code itself. For example, external SRM like `encodeForSQL()` in Listing 1.1 or custom methods defined in the source code will be overlooked by an off-the-shelf SRM list. When analyzing a program, it is thus essential to consider all libraries and frameworks it uses and its methods as potential SRM.

R3 SRM lists should be specific to the code base.

The Java Spring framework contains more than 30,000 methods. Considering that a reasonably-sized program uses multiple such libraries, it is infeasible to create a complete list of SRM manually. Therefore, it is necessary to compute SRM automatically.

R4 The detection of SRM should be automated, but **R5** should also involve the code developer.

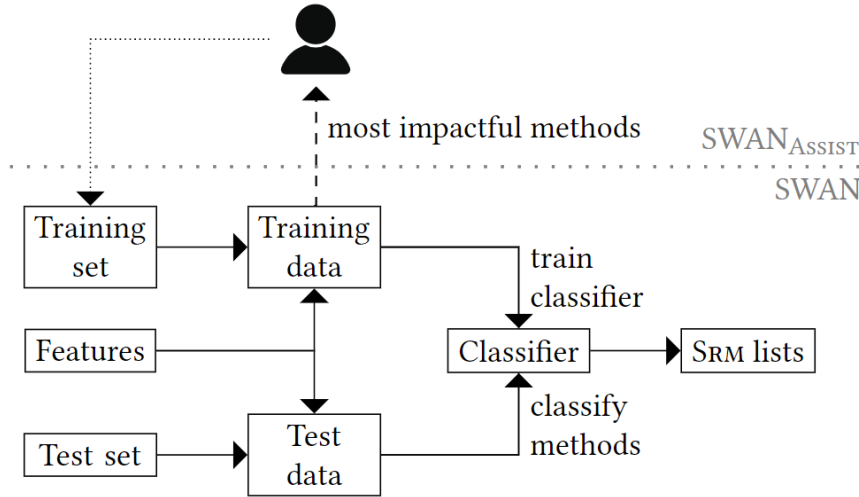
4.2 Related Work

SuSi [ARB13] is a machine-learning approach for sources and sinks in the Android framework. It uses 26 feature types and runs two iterations of machine-learning to classify methods as sources, sinks, or neither, and then, in different Android-specific classes such as Bluetooth, browser, etc. SWAN extends SuSi to detect sanitizers and authentication methods on top of sources and sinks. It also allows for classifications into CWE sub-classes. Unlike SuSi, which is specific to Android, SWAN generalizes SuSi to Java applications. In addition, SWAN introduces SWAN_{ASSIST}, which interleaves the code developer with the SRM detection task. This allows SWAN to generate SRM more specific to the analyzed code base.

Many static analyses use SRM to configure their analyses. For example, in the domain of Android applications, the SRM are typically computed using SuSi-like approaches. This makes those analyses [ARF⁺14, MAS⁺17, DAL⁺17, MG18] susceptible to SuSi’s weaknesses. For instance, those approaches do not consider sanitizers. SWAN and SWAN_{ASSIST} support sanitizers, and include user feedback in order to refine the list and reduce the number of false positives.

Sas et al. [SBF18] introduce the need for generalizing the detection of SRM for general Java libraries and the classification in CWE classes. They extend SuSi, modifying its features to achieve the former goal. However, unlike SWAN, they do not address the latter. Similar to SuSi, Sas et al.’s approach detects sources and sinks offline. SWAN can additionally recognize sanitizers and authentication methods and classify SRM by CWEs.

Like SWAN, Merlin [LNRB09] also detects SRM automatically. It uses probabilistic inference to detect specifications for taint-style analyses of string-based vulnerabilities. It models information flow paths in a propagation graph using probabilistic constraints. However, the resulting SRM are specific to the application of Merlin, i.e., string-based vulnerabilities, and Merlin does not provide support to classify them in sub-types such as CWEs.

Figure 4.1: Overview of SWAN and SWAN_{ASSIST}

SinkFinder [BLH⁺20] is another approach based on machine-learning using a support vector machine to detect a pair of sinks. SinkFinder first takes one pair of well-known interesting functions as the initial seed to infer enough positive and negative training samples using sub-word word embeddings. As it is implemented for C and C++ programs, an example for a seed is the pair of functions for allocating and freeing memory (kmalloc and kfree). Using this seed, the approach can identify other pairs, such as open and close functions for a database. Due to the nature of the problem SinkFinder addresses, it works only for method pairs. This is a limitation, because many security vulnerabilities require single sink, e.g., SQL injection.

4.3 Two-phase Classification Model

SWAN runs the automated classification shown in the lower part of Figure 4.1 twice: in the first iteration, it classifies all methods of the analyzed program and libraries into general SRM classes (**R1**): *sources* (So), *sinks* (Si), *sanitizers* (Sa), one of the three types of *authentication methods* detailed below, or *none* of the above. The second iteration discards the methods marked with *none*, and classifies the remaining SRM into the individual CWEs (**R2**). This is done to avoid classifying non-SRM methods as CWE-relevant.

In the first iteration, SWAN runs a set of four classifications, one for each type of SRM. Since those four sets are not disjoint (e.g., `getContent()` can be both a source and a sink), the classifications are run independently. For sources, sinks, and sanitizers, the classifications are binary e.g., for the sources, each method is classified in one of the two classes: *source* or *not source*. In the case of authentication methods, SRM are typically distributed between four disjoint classes: *auth-safe-state* (Ass), *auth-unsafe-state* (Aus), *auth-no-change* (Anc), and *none*. The first one refers to authentication methods that elevate the privileges of the user, e.g., login methods. The second contains methods that lower those privileges (e.g., logout methods). The third category marks methods that do not change the state of the program (e.g., `isAuthenticated()`).

Although exceptions are not rare, in most cases seen in our data sets, Ass and Aus tend to be disjoint. In addition, the two types of authentication methods are semantically very similar. As a result, running three different binary classifications yields a significantly lower precision and recall than a single classification with both classes. Anc was thus introduced to reduce the

number of such methods classified into *Ass* and *Aus*.

In the second iteration of the classification, the training set is kept the same, but the methods that were not classified in the SRM classes are removed from the test set. Each CWE classification is done via binary classifier. Currently, SWAN supports seven CWEs: CWE78, CWE79, CWE89, CWE306, CWE601, CWE862, and CWE863.

After running SWAN on the code shown in Listing 1.1, `getParameter()` is classified as a source for injection vulnerabilities (CWE78 and CWE89) and open redirect (CWE601), `executeQuery()` is classified as a sink for CWE89, `sendRedirect()` is classified as a sink for CWE601 and `encodeForSQL()` is found to be a sanitizer for CWE89.

4.4 FRcode: Code Features

To help the machine-learning algorithm classify the methods into different classes, SWAN uses a set of binary features that evaluate specific properties of the methods. For example, the feature instance `methodClassContainsOAuth` is more likely to indicate an authentication method than any other type of SRM. As a first phase of the learning, SWAN constructs a feature matrix by computing a true/false result for each feature instance on each method of the training set. We name this matrix a feature representation FRCODE. FRCODE is used to learn which combination of features best characterizes the classes. Finally, this knowledge is used to classify the methods of a new validation set.

We have identified 25 feature types, instantiated as 206 concrete features into SWAN's FRCODE. We call *feature types* generic features such as `methodClassContains` and *feature instances* their concrete instances (e.g., `methodClassContainsOAuth`). Table 4.2 shows the list of feature types in SWAN and their number of concrete instances. Overall, 15 feature types, and only 18 feature instances of SWAN, are derived from SuSI, where ten feature types and 188 feature instances have been added to complete the approach and make it compliant with **R1–R5**. To ensure a good selection of the new feature instances, we manually selected SRM methods from the Spring framework and created feature instances that comply with the methods' characteristics.

Compared to SuSI, SWAN contains more general features. For instance, SWAN does not contain Android-specific features such as **Required Permission**. On the other hand, SWAN contains more features based on *method and class names* such as **F03**, **F04**, **F10**, **F14**, **F15**, or **F16**. This is due to the Java naming conventions followed in major libraries, which make functionalities explanatory through naming. Those features are beneficial for the classification in CWEs, as both method/class names and CWEs are human-defined concepts and match their descriptions. For example, a call to a database library is made, or when the method is called "query", this can likely denote an SQL injection (CWE89).

SWAN features also support *access control to methods*: SRM are more likely to be publicly accessible, so whether the method is public, private, protected, contained in an anonymous class, or an inner class is covered in **F01**, **F02**, and **F08**, and are used to differentiate between potential SRM and other methods. SWAN also dedicates features to *parameters and return types* like **F21**, **F23**, or **F25**, which can help differentiate between different types of SRM (e.g., void methods are less likely to be sources), and between different types of CWE (e.g., Open redirect CWE601) most likely take Strings or URLs as inputs.

Other features in SWAN aim at *removing false positives*, e.g., **F11**, which helps distinguish constructors from sources, since they both return potentially sensitive data. *Data-flow-specific features* (e.g., **F19**, **F20**, **F24**) also serve this purpose, refining the classifications with more information such as whether a parameter flows to the return value (potentially indicating a sanitizer) or if a parameter flows to a method call (denoting a potential sink).

Table 4.2: Feature types of SWAN, and their total number of instances (#I) used within all classifications in SWAN.

Feature	#I	Feature	#I
F01 IsImplicitMethod	1	F14 MethodNameStartsWith	9
F02 AnonymousClass	1	F15 MethodNameEquals	3
F03 ClassContainsName	36	F16 MethodNameContains	46
F04 ClassEndsWithName	3	F17 ReturnsConstant	1
F05 ClassModifier	3	F18 ParamContainsTypeOrName	11
F06 HasParameters	1	F19 ParaFlowsToReturn	1
F07 HasReturnType	1	F20 ParamToSink	13
F08 InnerClass	1	F21 ParamTypeMatchesReturnType	1
F09 InvocationClassName	10	F22 ReturnTypeContainsName	6
F10 InvocationName	39	F23 ReturnType	5
F11 IsConstructor	1	F24 SourceToReturn	7
F12 IsRealSetter	1	F25 VoidOnMethod	1
F13 MethodModifier	4		
<i>Total</i>			<i>206</i>

SWAN’s features further aim to recognize sanitizers and authentication methods. For example, some instances of **F14** are dedicated to sanitizer detection: `MethodNameContainsSanit`, or `MethodNameContainsReplac`. Similarly, **F19** finds methods that transform a parameter into a return value. In combination with `ParameterContainsTypeString`, the instance of **F18** applied to Strings, and this covers the most typical type of sanitizer, which replaces sensitive data or strips dangerous characters in a String. Feature instances have also been created with the three types of authentication methods in mind. Authentication methods are mainly determined through their names or the names of their declaring classes, so they are targeted through instances of **F03** and **F10**, and **F14** such as `methodNameContainsLogin`, or `methodClassContainsOAuth`.

The training set in SWAN contains 235 Java methods collected from 10 popular Java frameworks: Spring [Spr], jsoup [jso], Google Auth [Goob], Pebble [Tem], jguard [JGu], WebGoat [OWA], and four Apache frameworks [Apah, Apag, Apaa, Apab]. We put particular care in ensuring that the methods were chosen so that each of the 206 feature instances of SWAN had at least a positive and a negative example making each example relevant for the machine-learning algorithm.

4.5 Classifiers

SWAN uses the Soot [ARB17] program analysis framework to obtain FRCODE. As its machine-learning module, it uses the SVM learner from the WEKA [WFHP16] library. The training set is a JSON file containing the 235 Java methods mentioned above, annotated with SRM types and CWEs. SWAN accepts a Java program or library as its test set and runs the two-phase classification, yielding lists of classified test SRM.

WEKA contains different classifiers: linear, probabilistic, tree-based, rule-based, etc. We have evaluated seven of them to determine which one would be most appropriate to use in SWAN: Support Vector Machine (SVM), Bayes Net, Naive Bayes, Logistic Regression, C4.5, Decision Stump, and Ripper. We have run a set of ten 10-fold cross-validations [Sto74] for each of the classifiers on the training set. The median precision and recall are shown in Table 4.3. We see that SVM yields the best precision and recall in all cases, classifying on average with 90% of the methods correctly. Naive Bayes also yield good results, and Decision Stump has the lowest

precision, with 62,5% for the CWE79. As a result, we chose SVM as the default classifier for SWAN.

4.6 SWANframe: General Framework for Creating Machine-learning Pipelines for SRM Prediction

SWAN’s architecture has three inputs (methods, labels, and a program) and one output (predicted methods). Internally, it consists of two components, one for feature extraction and another one for model selection. The first input is a list of preselected methods. The second input is the methods’ labels for each class present in the models. Additionally, when the models are used with new data, the third input is the program containing new unlabeled methods. When the pipeline runs with the methods from the program, it adds labels to them (predicted methods). The previous work SUSI followed the same architecture. However, SWAN and SUSI use different training set. The feature extraction is also different as both approaches have different features. Finally, they both follow a two-phase classification model, which was manually selected as seen in the previous section, but internally there are different classifiers. In the following sections, we explore even more variations of this architecture. Therefore, we abstract this as general framework for creating a machine-learning pipeline of models for predicting SRM called SWANFRAME.

Figure 4.2 shows an overview of SWANFRAME. Both SWAN and SUSI are one concrete instance (pipeline) of this framework. When the framework is applied in training mode, the input methods from the program and labels are combined with the features from the *Feature Extraction* component to create the feature representation. Then, the *Classification* component makes the predictions. The pipeline can be used in the testing mode with a new program. Then, the methods from the program are outputted as *Predicted Methods*. SWANFRAME defines the classes the legend as shown in the inner part of *Classification* in Figure 4.2.

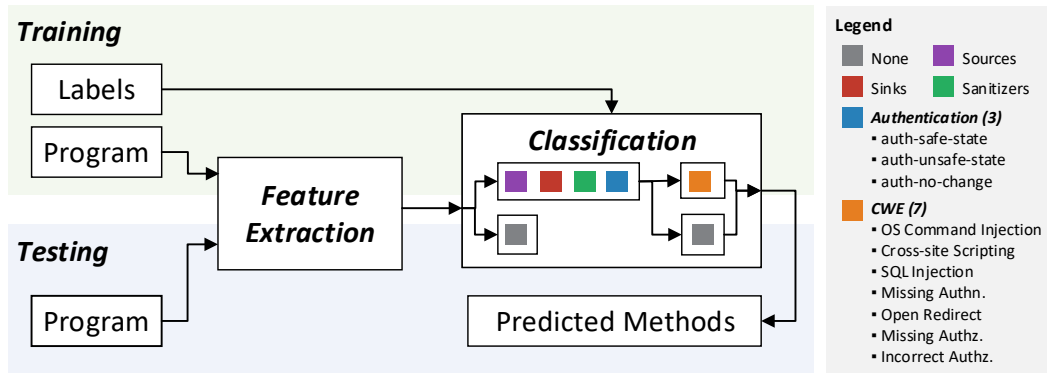


Figure 4.2: Overview of the general framework SWANFRAME for creating ML-based pipelines to classify methods into SRM and CWE classes.

The classification of SRM and CWEs is a multi-label classification problem. Each method from the program (e.g., application or library) can be assigned one or many labels, or none at all.

Let \mathcal{X} denote the instance space consisting of program information. In general, the program could be code, documentation, both, or something different. In order to make the program understandable for standard ML classifiers, each instance is represented by a feature representation

Table 4.3: Precision (P) and recall (R) of the 10-cross fold validation for all classifiers averaged over 10 iterations in %.

	So		Si		Sa		Auth		CWE78		CWE79		CWE89		CWE306		CWE601		CWE862		CWE863		Average	
	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R
SVM	93.8	94	87.8	87.9	97.9	97.9	94.7	94.8	86.5	89.6	77.9	79.9	86.3	90.2	93.4	93.1	85.3	86.3	85.4	85.7	88.8	90	90	90
BayesNet	94.5	94.6	87.8	88	97.5	97.2	92.9	92.4	89.2	90.4	78.2	78.5	89.4	91	93.6	93.2	82.4	85.3	87.1	87.3	88	88.3	89	90
NaiveBayes	94.5	93.5	86.9	86.8	96.6	96.4	89.4	90.5	88.1	89.9	78	78.6	88.5	90.6	93.1	92.6	82.5	85.5	86.9	87.1	89.1	89	88.5	89.1
LogReg	94.3	94.1	78	78.3	95.2	95.3	94.4	94.2	87.7	89.9	63	79.1	86.5	89.6	93.6	93.4	84	86.2	84.4	85.4	88	88.2	87.5	88.5
C4.5	94.5	94.6	82.4	83	97.4	97.5	93.4	93.8	85.3	89.1	81.6	80.3	86.9	90.2	93.6	93.2	85.6	86.1	86.5	86.6	87.5	87.7	88.6	89.3
Stump	90.4	89.8	66.4	71.5	88.9	89.6	78	86.8	87.2	90.2	62.5	77	82.6	90.3	87.5	84.7	86.3	85.7	84.6	80.4	86.6	83.8	81.9	84.5
Ripper	92.8	93	82.9	83.3	97.4	97.5	89.2	90	86.7	90	70	76.4	85.4	90.4	92.3	91.6	77.5	85	84.5	84.2	87	86.8	86	88

generated by feature map $f : \mathcal{X} \rightarrow \mathbb{R}^F$. The finite set of class labels is denoted by $\mathcal{L} = \mathcal{L}_S \cup \mathcal{L}_C$, i.e., the set of SRM and CWEs.

The goal is to learn a classifier $h : \mathcal{X} \rightarrow 2^{\mathcal{L}}$ mapping an instance $x \in \mathcal{X}$, i.e., a feature representation $f(x)$ of a program, to the correct set of class labels $L \subseteq \mathcal{L}$.

To do so, SWAN is decomposed into a two-phase classification process

$$h(x) = h_S(f(x)) \cup h_C(f(x)) \quad . \quad (4.1)$$

In the first phase $h_S : \mathcal{X} \rightarrow 2^{\mathcal{L}_S}$ assigns any number of SRM class labels

$$\mathcal{L}_S = \{source, sink, sanitizer, authentication\} \quad (4.2)$$

to a given instance $x \in \mathcal{X}$. If the program is classified to be no SRM, i.e., h_S predicts \emptyset , it is assigned *NONE* in the first phase and therefore excluded for further investigation in the second phase.

In second phase $h_C : \mathcal{X} \rightarrow 2^{\mathcal{L}_C}$ assigns any number of the following CWE class labels

$$\mathcal{L}_C = \{CWE078, CWE079, CWE089, CWE306, \\ CWE601, CWE862, CWE863\} \quad (4.3)$$

to the instance $x \in \mathcal{X}$. Again, if the program is classified to be no CWE, i.e., h_C predicts \emptyset , it is assigned *NONE*

Training data of the form

$$\mathcal{D}_{train} = \{(x_i, y_i)\}_{i=1}^N \subset \mathcal{X} \times \mathcal{L} \quad (4.4)$$

is given, where each instance $x_i \in \mathcal{X}$ represents a program, and $L_i \in 2^{\mathcal{L}}$ denotes the ground truth set of labels assigned to x_i . The classifier h is meant to generalize well beyond the training data to also perform well on unseen data. To do so, unseen test data \mathcal{D}_{test} of the same form than \mathcal{D}_{train} is needed to compute the empirical performance of h , aiming to maximizing the performance w.r.t. a given performance measure for \mathcal{D}_{test} . To this end, both classifiers h_S and h_C have to perform well.

In the following sections, we propose new types of features that can be used to create new feature representations and with that new pipelines of SWANFRAME. Additionally, we apply an automatic process for model selection and evaluate it. For this, we use Auto-WEKA, an AutoML extension of WEKA [WFHP16], which identifies machine-learning algorithms and hyperparameter settings appropriate to their applications. Auto-WEKA splits the provided dataset into a training and test set and returns the optimal classifier found in the given exploration time.

4.7 FRdoc_m: Implementing Features Based on Doc Comments

In Java, in-line software documentation, called documentation comments (doc comments), can be added to the source code by delimiting comments with `/** ... */` to describe the classes, fields, methods, constructors, or package declarations they precede. The tool that processes these comments is called Javadoc.

We create new type of feature representation FRDOC_M based on doc comments information. The *Feature Extraction* component of FRDOC_M is shown in Figure 4.3.

FRDOC_M extracts doc comments from source code, processes them with Natural Language Processing, and instantiates new features. In the training phase, the source and JAR files for the methods in the training set are retrieved. The source JAR files are processed using

two doclets, namely **Javadoc Coverage Doclet** and the **SSLDoclet**. The **Javadoc Coverage Doclet** calculates the documentation coverage for the source code in the source JAR files, useful for filtering methods with no or insufficient documentation. The **SSLDoclet** extracts doc comments from the source code and exports them to XML files. The exported doc comments are processed with either CoreNLP (manual features) for FRDOC_M or DL4J (automatic features with word embedding) for FRDOC_A (Section 4.8). In the testing phase, a list of Maven identifiers (JAR files retrieved using Jeka) or JAR files can be provided as *Program*. Similar to the training phase, the feature extraction module provides the FRDOC_M as input for the classification.

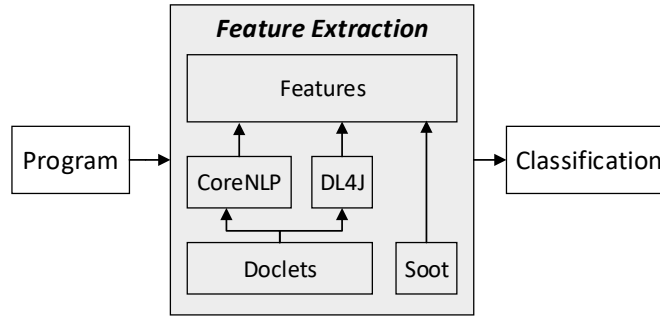


Figure 4.3: Components of FRDOC_M machine learning approach to train and evaluate models that classify Java methods into CWE classes based on source code and software documentation features.

Although Javadoc’s standard export format is HTML, the Javadoc API can be used to create Java programs that specify the output format of content exported by the Javadoc tool called doclets. HTML is not ideal for natural language (NLP) processing analysis. NLP requires a more versatile format such as XML. Therefore, we use a doclet that exports documentation found in Java source code using an XML schema optimized for NLP. The **SSLDoclet** exports syntactically and semantically models source code information, maintains relationships found in the source code, and represents the documentation using various XML tags, attributes, and elements. After running the **SSLDoclet** on a program’s source code, XML files containing the class and method doc comments for each Java class are exported. During the doc comment extraction, another doclet reports on the documentation coverage of the program as a useful metric that can be used to enrich the training set with new relevant methods.

However, checking the presence of text in a doc comment is insufficient for the NLP because the presence of method or class doc comment does not necessarily imply that it is useful. For example, the `com.mysql.jdbc.Field` class from the `mysql:mysql-connector-java:3.1.7` library has 21 methods with “DOCUMENT ME!” as their doc comments. Developers sometimes write the method/class name or write the name in sentence form to ensure that documentation is present.

For the new feature representation based on doc comments, we created a new training set, including 153 out of SWAN’s 235 existing training examples in SWAN with proper doc comments. In addition to these methods, 129 were taken from the list of methods used by a security plugin called Early Vulnerability Detector, developed by Sampaio [Sam14]. The list contained sources, sinks, sanitizers, and examples for CWE078, CWE079 and CWE089. Forty-two methods were taken from the SRM list used by the Find Security Bugs Plugin [AFP20]. In addition, for validation purposes, the SRM list for Java created by OWASP was used [VdSCC⁺08]. Using these lists, 171 new methods were added to the training set. Table 4.4 shows the distribution of

the 153 existing and 171 new methods. Given that a method can belong to multiple categories (for example, a method that is a source and a sink), the sum of the available labels can exceed the number of methods.

Category	Number of Examples		
	Existing	New	Total
Sources	27	61	88
Sinks	49	62	111
Sanitizers	21	50	71
Auth-no-change	5	0	5
Auth-unsafe-state	9	0	9
Auth-safe-State	33	0	33
CWE078	21	3	24
CWE079	37	16	53
CWE089	18	7	25
CWE306	55	0	55
CWE601	26	0	26
CWE862	47	0	47
CWE863	42	0	42
<i>Unique Methods</i>	<i>153</i>	<i>171</i>	<i>324</i>

Table 4.4: Distribution of the 153 existing and 171 new methods in the dataset that have method doc comments containing at least one sentence or phrase. Considering that the 324 unique methods can belong to multiple classes, the total number examples exceeds the count of the unique methods.

When the `SSLDoclet` exports doc comments, they contain the block and inline tags described in Table 4.5 and HTML characters such as `<code>...</code>`. For example, the doc comment “Writes the {@link ResponseHeader header} to the output stream.” contains the link in-line tag. Before annotating the doc comments with a Natural Language Processing library, these tags would be removed; however, counting the number of tags used in a doc comment can be useful in establishing relationships between SRM classes and the number of tags. The set of features based on the preprocessed doc comments shown in Table 4.6 were implemented to utilize this information. The four features, U1–U4, count the occurrences of the code ({@code text}), link ({@link package.class#member label}) or ({@linkplain package.class#member label}), deprecated (@deprecated deprecated-text) and see (@see reference) tags. These features use regular expressions to identify if the tags are used in the doc comments. When counting code tags, the U1 CodeTagCount feature additionally uses a regular expression to identify `<code>...</code>` patterns, another method that can also be used to enclose code in doc comments.

Tag	Description
@author	Specifies the person(s) that made significant contributions to the design or implementation of the documented object.
@version text	Current software version release number.
@param name descr	Name and description for method's or constructor's parameters.
@return descr	Description for the method's or constructor's return type.
@exception name descr @throws name descr	Provides the type and description of the exception that the method may throw.
@see reference	A link or text that references another part of the software.
@since text	Software release version number where the code was contributed.
@deprecated text	Informs the user that the API was deprecated.

Table 4.5: Block tags used in Javadoc

Table 4.6: Manual features that count block and in-line tags in pre-processed method and class doc comments.

Key	Description
U1	Number of code tags.
U2	Number of deprecated tags.
U3	Number of link tags.
U4	Number of see tags.

After removing in-line and block tags, all HTML characters such as `<code>...</code>`, code examples, and symbols (for example, hashtags #), the doc comments are annotated using CoreNLP's default pipeline. The features based on the annotated doc comments are categorized as follows: part of speech counters, data flow evaluators, word counters, and statistical features. The list of features and the categories they belong are outlined in Table 4.7.

A subset of the features in Table 4.7 is based on implementations of feature engineering strategies used by an open source framework called Featuretools that performs automated feature engineering by transforming datasets into feature matrices [KV15]. The framework offers numerous functions called primitives that aggregate and transform data in order to create features for machine-learning. Among the primitives described in Featuretools' API reference are the following natural language processing primitives: PartOfSpeechCount, TitleWordCount, StopwordCount, MeanCharactersPerWord, and DiversityScore. Implementations of these primitives are part of the manual features based on the annotated doc comments.

Table 4.7: Manual features based on doc comments that are annotated using CoreNLP.

Category	Key	Description
POS Count	F1	Number of adjectives.
	F2	Number of adverbs.
	F3	Number of conjunctions.
	F4	Number of nouns.
	F5	Number of prepositions.
	F6	Number of pronouns.
	F7	Number of punctuation marks.
	F8	Number of verbs.
Text Statistics	F9	Number of title words.
	F10	Number of stop words.
	F11	Number of unique words divided by total number of words.
	F12	Average number of characters for each word.
	F13	Nmber of distinct lemmas.
	F14	Number of characters.
	F15	Number of digits.
	F16	Number of sentences.
Word Count	F17	Average length of sentences.
	F18	Number of incomplete code words.
	F19	Weighted count of auth-safe-state words.
	F20	Weighted count of auth-unsafe-state words.
	F21	Weighted count of auth-no-change words.
	F22	Weighted count of command injection words.
	F23	Weighted count of cross site scripting words.
	F24	Weighted count of incorrect authentication words.
	F25	Weighted count of missing authentication words.
	F26	Weighted count of missing authorization words.
	F27	Weighted count of open redirect words.
	F28	Weighted count of sanitizer words.
Data Flow	F29	Weighted count of sink words.
	F30	Weighted count of source words.
	F31	Weighted count of SQL injections words.
	F32	Data flow using an auth-safe-state preposition.
	F33	Data flow using an auth-unsafe-state preposition.
	F34	Data flow using an auth-no-change preposition.
	F34	Data flow using a sanitizer data flow preposition.
	F35	Data flow using a sink data flow preposition.
	F36	Data flow through a source preposition.

POS Count The first category of features, POS Count, is based on the implementation of the Featuretools *PartOfSpeechCount* primitive which counts the occurrences of different parts of speeches. Accordingly, eight features are implemented that count the following part of speech tags in the doc comments: adjective, adverb, conjunction, noun, verb, preposition, pronoun, and punctuation marks. Counting the part of speech tags assigned to words in the doc comments can help establish relationships between the number of a particular part of speech and a particular SRM class. For example, the doc comments on methods and classes that perform authentication or authorization tasks are likely to contain numerous adjectives that describe the authentication

or authorization state such as “disconnected”, “connected”, “granted”, or “denied”. Counting these words can therefore assist in the classification of CWEs related to authentication and authorization, namely: CWE306 Missing Authentication, CWE862 Missing Authorisation, and CWE-863 Incorrect Authorisation.

Text Statistics The second category of features, text statistics, contains implementations of the remaining primitives mentioned above. Feature **F9** counts the number of words that start with capital letters, title words, in the doc comments. The next feature, **F10**, computes the number of stopwords in the doc comment. Stop words are often referred to as function words which are mostly articles (“the”, “a”, “this”, etc.), prepositions (“at”, “for”, “of”, etc.), pronouns (“he”, “it”, “them”, etc.) and verb articles (“am”, “be”, “was”, etc.). CoreNLP’s list of stopwords is used in **F10** feature and other features that evaluate stopwords. Feature **F11** evaluates the complexity of the doc comment by dividing the sum of unique words by the total number of words in the doc comment. This feature ignores stopwords and punctuation marks. The last feature in this category, based on the Featuretools primitive **F12**, calculates the average number of characters for each word in the doc comments. For this feature, punctuations are not considered.

In the text statistics category, there are also features that count the number of unique lemmas **F13**, characters **F14**, digits **F15**, and sentences **F16** within the doc comments. Feature **F16** calculates the average sentence length by dividing the sentence length by the number of sentences. These statistical features and the ones previously discussed using the logical structure of the doc comments to extract quantitative information for classification.

The last feature in the text statistics category, **F18**, evaluates if words that convey incomplete code or implementation, such as “fixme”, “todo”, “backdoor”, “trick”, etc., are used in the doc comments. This list of words was created by Van der Stock et al. and these words may point to possible software vulnerabilities as they imply incomplete or fault-prone implementations [VdSCC⁺08]. Such implementations are likely to be vulnerable to software attacks because a developer may not have followed best practices when developing, or the solution may not be foolproof.

Word Count The previously discussed features use structural information of the doc comments to classify the methods. However, more helpful information can be extracted using the content or semantics of the doc comments. When classifying methods according to software vulnerability classes, helpful clues can be derived based on the words used in the doc comments. Domain experts classifying methods based on doc comments would base their selections significantly on what verbs and nouns appear in a doc comment. Specific words infer the possibility of certain vulnerabilities. Moreover, the frequency of the words and their function in the sentence plays a pivotal role. For example, the words “database” and “insert” are more readily associated with SQL Injection in comparison to the words “browser” and “redirect” which are more likely associated to Cross-site scripting.

The verbs in a doc comment convey useful information about possible data operations performed by a method. They describe the action applied to an object or the state of the object. For example, words such as “insert”, “query” and “select” can be readily associated with CWE89 SQL Injection, while verbs such as “redirect”, “send” or “request” are more aligned with CWE601 Open Redirect. The nouns in a doc comment are the objects to which the method applies its data operations. Nouns such as “file”, “network”, “hardware”, “cookie”, “email”, “internet”, “printer”, “server”, or “string” provide contextual information that can be insightful for classification. Therefore, the phrase “inserts string” implies data insertion that could be relevant for SQL Injection, while the phrase “redirects to provided string” is more relevant for the Open

Redirect vulnerability.

Based on this observation, a vocabulary of verbs and nouns for each CWE and SRM class was created using words used in FRCODE. Other words were extracted by generating a list of verbs and nouns used in the Java Runtime library and assigning the words to the applicable vocabularies. The vocabularies were also augmented with words from the Computer Science Word List (CSWL), a technical vocabulary of computer science words [Min13]. The vocabularies contain the lemmatized versions of words to ensure that even if a word is morphed because of pluralization or tense, it could still be recognized after using CoreNLP’s lemmatization annotator.

Using the predefined vocabularies of nouns and verbs for the CWE and SRM classes, the features F19-F31 count the number of times the words from the vocabularies appear in the doc comments. The features do not merely count how many words from a particular vocabulary are in the sentence. Instead, the dependency graph, which captures the semantic meaning of the sentence, is used to calculate the weighted count for each feature. The depth of a word in the dependency graph implies the importance of the word in ascertaining the meaning of the sentence. The verb in the sentence is predominantly the root of the graph, and the nouns or noun phrases would be its children. With this insight, the features were created to model the sentences’ semantic meaning. Using this approach, the most essential words in a sentence would impact the final sum more.

As an example, Figure 4.4 shows the dependency graph for the doc comment of the `void writeResponseHeader(RequestData, int)` method. The root of the dependency graph is “writes” (depth = 0), which has two children “header” and “.” (depth = 1). The node “header” has three children “the”, “response”, and “stream” (depth = 2). The remaining words, “to”, “the” and “output” are children of “stream” and have a depth of 3. Each depth of the dependency graph is assigned a weight such that: $\text{depth}(0) = 100$, $\text{depth}(1) = 30$, $\text{depth}(2) = 10$, and $\text{depth}(\geq 3) = 2$. Evaluating the example sentence with the **F28** feature would result in 100 (writes) + 30 (header) + 10 (stream) + 2 (output) = 142 given that the lemmas “write”, “header”, “stream” and “output” belong to the sink vocabulary. However, applying the **F26** feature returns 30 (header) + 10 (response) = 40 given that only the words “response” and “header” are present in the open redirect vocabulary. Based on the doc comment, the method would be more relevant for a sink than an open redirect vulnerability, and the weighted count captures this expectation.

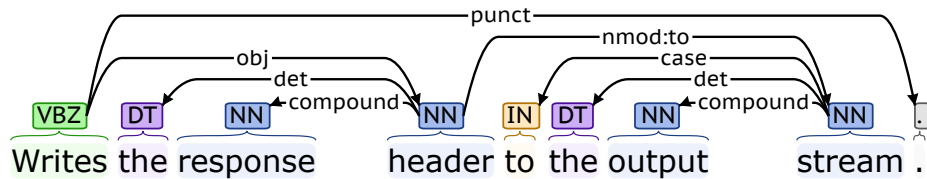


Figure 4.4: Dependency graph for the doc comment of the `void writeResponseHeader(RequestData, int)` method.

The word count category of features uses these principles to represent the semantics of the doc comments. For example, doc comments that contain a certain set of words are more likely to be related to a particular CWE than doc comments that do not. Capturing These features evaluate the nouns and verbs in the sentences and assign a weight based on the depth of the word in the dependency graph.

Data Flow The fourth category of features, data flow, evaluates if there are data flows in the sentence based on the prepositions used. Prepositions are placed before pronouns, nouns

or noun phrases and convey information about direction (“to”, “into”, “on”, etc.), place (“at”, “inside”, “in”, etc.), location (“in”, “at”, “on”, etc.), and spatial relationships (“from”, “out of”, “through”, etc.) [Cen]. In doc comments, these words can describe what data operations (for example, writing, reading, deleting, etc.) are being done by a method or in a class. They can also indicate where the data is located (for example, “in a file”, “on the server”, etc.). Using the context implied by the nouns and pronouns in the doc comments, the preposition “from” could imply a source operation, while “to” might imply a sink operation.

For example, the method doc comment for `void writeResponseHeader(RequestData, int)` is “Writes the response header to the output stream”. In this doc comment, a data flow is implied by the use of the preposition “to” with the nouns “response header” is written to the “output stream”. Given these clues, this method would be classified as a sink. As seen with this example, the implied data flow depends on the nouns or pronouns used in the sentence, and the data flow features also check what nouns or pronouns are used with the preposition. For example, using the prepositions “to” and “from” with the word database implies two different data operations. “To” in this context would likely imply a sink operation, while “from” might be a source operation. Identifying possible data flows implied in the sentence can help to detect possible vulnerabilities that a method or class may be prone to.

4.8 FRdoc_a: Automated Features Based on Doc Comments

In this section, we describe the feature extraction approach that analyses the doc comments with automatic feature engineering. It is represented as fixed-length vectors, which are used as the FRDOC_A feature representation. A common fixed-length vector representation for text is the bag-of-words approach which is a simple, efficient, and accurate approach often used in text classifications scenarios [Har54]. However, the major disadvantage with bag-of-words is that the ordering of words is lost, which means that sentences that have the same words but different meanings would have the same representation. An improvement to the bag-of-words method is word-embeddings, in which the semantic and syntactic information of the text is represented as a vector of real numbers based on the relationship of words in the text [Hel19].

Paragraph Vector is an algorithm that creates word-embeddings as a fixed-length vector representation of a variable-length text such as phrases, sentences, paragraphs, and documents [LM14]. Paragraph vectors use a machine learning to learn the vector representation for the given text while considering the text’s semantics. Mikolov et al. showed that paragraph vectors outperformed bag-of-words models by 30% [LM14]. This is accomplished by concatenating the word vectors from a paragraph into a paragraph vector by predicting what the next word should be in a given context. After the training completes, paragraph vectors for the text are inferred based on the word vectors that were calculated. These vectors can then be used as features for machine learning instead of, or in addition to, bag-of-words.

FRDOC_A uses the paragraph vector algorithm from the Deeplearning4J (DL4J) library, a framework of learning libraries and tools written for Java [Tea17]. We use the algorithm initialized with all method and class doc comments and configured with the default parameters recommended in the algorithm’s documentation [Tea17]. After fitting the paragraph vector algorithm, the vector representations for the doc comments can be queried and a 100 dimension vector is obtained for each method in the dataset.

4.9 Pipelines

Based on the general framework SWANFRAME we proposed in Section 4.6, and the possible feature representations, we created and evaluated the pipelines shown in Table 4.8.

Table 4.8: SWANFRAME instances used to answer the research questions

Pipeline	Feature representation	Classification
Pipe-1	FRCODE	manual
Pipe-2	FRDOC_M	manual
Pipe-3	FRDOC_A	manual
Pipe-4	FRCODE+DOC_M	manual
Pipe-5	FRCODE+DOC_A	manual
Pipe-6	FRCODE+DOC_M	automatic

4.10 Evaluation

In this section, we answer the following research questions:

- **RQ7** How does SWAN (Pipe-1) compare to existing approaches?
- **RQ8** What is SWAN (Pipe-1) precision on real-world libraries?
- **RQ9** Is it possible to use CoreNLP features in FRDOC_M (Pipe-2) and achieve a performance comparable to the existing approaches based on code information (Pipe-1)?
- **RQ10** How does FRDOC_A (Pipe-3), the automatic word-embedding feature representation technique for doc comments, compares to the manual variant FRDOC_M (Pipe-2)?
- **RQ11** How does the combined feature representations FRCODE+DOC_M (Pipe-4) and FRCODE+DOC_A (Pipe-5) compare to individual feature representations?
- **RQ12** What is the optimal classifier for each class when using FRCODE+DOC_M (Pipe-6)?

To answer the research questions **RQ9**, **RQ10**, **RQ11**, and **RQ12**, we used the extended training dataset from SWAN. We run a 10-fold cross-validation on the training dataset and recorded the precision, recall, and F1 measure (harmonic mean of the precision and recall). The values we report in the following are averaged over ten runs of the cross-validation with WEKA. Due to readability, we report the F1 measure only, whereas our artifact contains the precision and recall as well. When running the pipelines with manual classification, i.e., Pipe-1 to Pipe-5, we set up our tool to run seven classifiers from WEKA in their default configuration. The classifiers are NaiveBayes, SMO (support vector machine algorithm), DecisionStump, BayesNet, JRip, Logistic, and J48 (decision tree algorithm). The experiments with Pipe-6, where we use Auto-WEKA, we explored the configuration space for three hours using F1 as internal metric for performance. We used Lenovo Thinkpad T14 Gen 1 with 16 GB RAM, AMD Ryzen Pro 4750U CPU, and 8 Cores.

4.10.1 Comparison (RQ7)

We know of the following three approaches to have open-sourced their SRM: SuSi [ARB13], Sas et al.’s approach [SBF18], and JoanAudit [TSBB17]. In the evaluation for this research question we use the pipeline Pipe-1 (Table 4.8).

SuSi and Sas et al. We compare the lists of sources and sinks from SuSi [ARB13] and its extension by Sas et al. [SBF18] to the lists of sources and sinks generated by SWAN on the Android framework (version 4.2). The number of sources and sinks detected by the three approaches is shown in Figure 4.5. SWAN reports a total of 25,085 sources and 13,798 sinks,

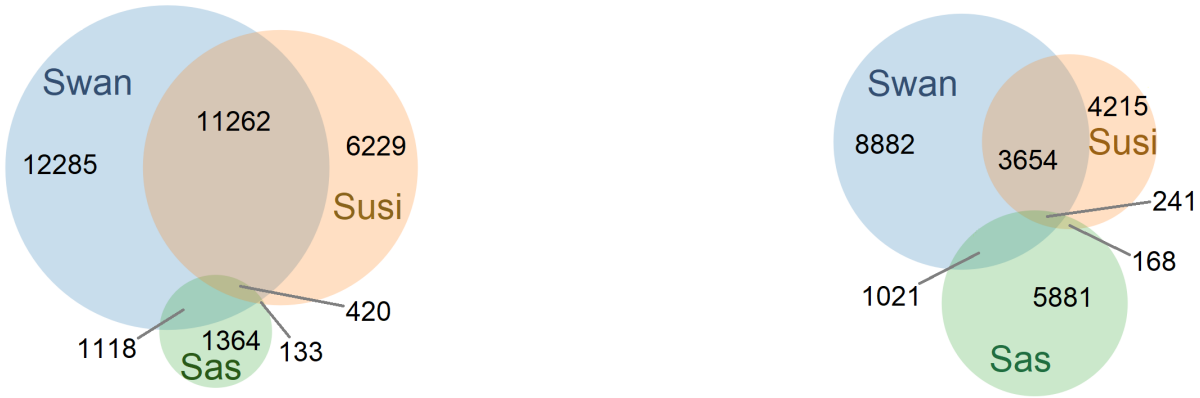


Figure 4.5: Two venn diagrams showing the number of methods detected by the three approaches SuSi, Sas et al., and SWAN in the Android framework. Each venn diagram contains three sets and their intersection. The diagram on the left side shows the number of sources. The diagram on the right side shows the number of sinks.

SuSi 18,044 sources and 8,278 sinks, and the tool by Sas et al., 3,035 sources and 7,311 sinks. SWAN reports more SRM than the other two approaches, which we attribute to two reasons after a manual investigation. First, SWAN’s features target a broader range of vulnerabilities than SuSi’s and Sas et al.’s data privacy focus. Second, SuSi reports methods from abstract classes and interfaces, SWAN reports their concrete implementations, which allows for better precision. Sas et al. are stricter and only report warnings belonging to certain classes: *database*, *gui*, *file*, *web*, *xml*, and *io*. Unlike SWAN and SuSi, Sas et al. report more sinks than sources. This is due to the larger number of sink features than source features contained in their approach. Both SWAN and SuSi contain enough features and training instances to overcome this.

To compare the precision of the three approaches, we randomly selected 50 sources and 50 sinks in the lists produced by the three tools and manually classified them. The selected methods of each tool were labeled by a different researcher, two of the authors, and one external researcher. SWAN shows a precision of 0.99 for sources and 0.92 for sinks (confirming our findings of **RQ1**), whereas SuSi yields respective precisions of 0.96 and 0.88, and Sas et al.’s tool has 0.88 and 0.88, respectively.

JoanAudit. The authors of JoanAudit have manually created lists of 177 SRM classified in five injection vulnerabilities for taint analysis, including sources, sinks, and validators. JoanAudit’s SRM are taken from various Java applications, two of which are in common with SWAN: Spring and Apache Commons. Applying to the Spring framework, SWAN can detect two of the three methods listed in JoanAudit, the third being an interface method of which SWAN reports the concrete implementations. On Apache, SWAN detects seven of the ten JoanAudit methods. Two of the missing three are related to the XML injection vulnerability, which is not yet included in the classification of SWAN. This indicates that SWAN can be used to create lists of SRM whose quality is comparable to hand-crafted lists such as JoanAudit’s.

SWAN yields a higher precision for sources and sinks than SuSi and Sas et al.’s approach. It can detect SRM with a quality comparable to hand-crafted lists.

4.10.2 Real-world Applications (RQ8)

We ran SWAN (PIP1) on a benchmark of twelve popular Java libraries. The benchmark applications were selected to be real-world, open-source Java programs that contain at least 500 methods and have evidence of maintenance and development over a recent time (i.e., at least two years and five contributors). They are composed of two frameworks from the mobile domain (Android v4.2 [Gooa] and Apache Cordova v2.4 [Apac]), eight web frameworks (Apache Lucene v6.6.5 [Apad], Apache Stratos v4.0 [Apae], Apache Struts v1.2.4 [Apaf], Dropwizard v1.3 [Dro], Eclipse Jetty v9.2 [Ecla], GWT v2.8.2 [GWT], Spark v2.7.2 [Jav], and Spring v4.3.9 [Spr]), one framework from the home automation domain (Eclipse SmartHome v0.9 [Eclb]), and one utility framework (Apache Commons¹ v19 [Apab]).

Table 4.9 presents the number of SRM detected by SWAN in each Java library. Over the 290,791 methods of the twelve frameworks, SWAN classified 74,603 of them as SRM. We see that many methods are classified as sources and sinks. This is due to the broad definition of sources and sinks, as they should allow an analysis to detect any type of SANS top 25 sources and sinks. However, restricting the SRM to particular CWEs significantly reduces the number of methods to consider (e.g., from 20.39% source SRM to under 1% source/sink/validator/authentication methods for all CWE-specific SRM), and therefore decreases the complexity of the analyses that use them.

Because of the high number of reported SRM, we did not manually verify the complete classification. For each framework, we randomly selected 50 methods from each category of SRM and CWE (or fewer if the number of methods detected by SWAN was lower) and manually verified their classification. The verification was done by two of the authors and one external researcher. Each person verified one-third of the selected methods. The resulting precision of SWAN for each category is presented in Table 4.10.

Over the different classes, SWAN yields a precision of 0.826 for the SRM classes and of 0.677 for the CWE classes. SWAN is most precise (0.91) when detecting sources. Misclassifications for this category are mostly due to the presence of getter methods in plain old Java objects, which share similarities with source methods (e.g., returning a String). This can be improved by training the model with more counter-examples in the training set. SWAN is least precise for CWE862 (0.574), particularly on Spark (0), which is based only on three methods detected making the value an outlier to the dataset. Even though CWE862 and CWE863 are similar, making their SRM overlapping, the precision of CWE863 is better as there are more examples available and more specific. Any authorization information available such as credentials and tokens, are considered in CWE863, but not in CWE862 and the frameworks generally have more related methods.

Other misclassifications cannot be improved by modifying the training set. For example, the Spring method `Connection.getConnection()` has a different behaviour when overwritten in its subclasses: in `SingleConnectionFactory`, it is an authentication method of type `Ass`, and in `ContextHolder` it does not perform an authentication behavior. This information cannot be inferred from the source code, as those two methods are too similar. The differentiation information can be found in the API documentation of the methods. We conclude that SWAN could be improved by adding features that go beyond the code.

Over the twelve libraries, SWAN yields a precision of 0.76. While it is naturally lower than the 0.9 found with the 10-fold cross-validation (Table 4.3), it shows that the generalization of the approach to Java projects is still able to classify SRM with a good precision. The low standard deviation ($\sigma = 0.075$) denotes the stability of SWAN's precision over the different Java projects.

SWAN is most precise on *Eclipse Smarthome*, which is explained by the fact that the library

¹This also includes Apache XML-Xalan, XML-Xerces, XML-Rcp, HttpComponents, and Oltu-OAuth2

is aimed at home automation and does not contain the web CWEs that SWAN currently supports. Therefore, SWAN could only detect sources and sinks, for which it is strongest. One of the libraries for which SWAN performs the weakest is *Android*, with low precision for authentication methods and methods for CWE306, CWE862, and CWE863. This is due to the keywords used in SWAN's features (e.g., `dis/connect`), which overlap with domain-specific methods (e.g., Wi-Fi connection, Bluetooth, or NFC adapter). On our training set, such methods are typically used for authentication, which is not the case for Android. We can conclude that despite its good precision, SWAN still needs more domain-specific information, motivating the need for user input and of SWAN_{ASSIST}.

Over 12 Java libraries, SWAN yields a precision of 0.76. It is more precise for detecting SRM types (0.826) than for CWEs (0.677). SWAN can be improved by adding non-source code-specific features, a complete training set, and domain-specific information. The latter two can be provided by the code developer using SWAN_{ASSIST}.

4.10.3 Utilizing doc comments (RQ9)

To answer whether the feature representation based on doc comments achieve comparable results to previous approaches based on source code information, we use the results from Pipe-1 and Pipe-2. Pipe-1 uses the features from SWAN with the new training dataset to build the feature representation FRCODE. Pipe-2 uses only the new feature representation based on doc comments created manually, FRDOC_M. The F1 measures for each class (SRM and CWE) and each classifier are shown in Table 4.11 for Pipe-1 and Table 4.12 for Pipe-2. Comparing the highest values (i.e., bold valued in the tables) for each class, we can see that FRCODE is in most cases with higher value than FRDOC_M, e.g., for sources 0.8 compared to 0.67. There is an exception for CWE079, where FRDOC_M has F1 value of 0.49 for J48 and FRCODE has 0.28 for BayesNet.

We can observe that for different classes, different classifiers show the best F1 measures. Decision Stump classifier does not reach the highest value for any class in Pipe-1, whereas Naive-Bayes for Pipe-2.

For most classes, FRCODE outperforms FRDOC_M. Hence, using only feature representation based on doc comments does not improve the state-of-the-art approaches based on source code information, such as SWAN and SuSi.

4.10.4 Automatic vs. manual features based on doc comments (RQ10)

To compare the performance based on the F1 measure between the manually and automatically created feature representations based on doc comments, i.e., FRDOC_M and FRDOC_A, we use the results from Pipe-2 and Pipe-3. Table 4.12 shows the F1 measure values per class for FRDOC_M and Table 4.13 for FRDOC_A. In the SRM classes, we observe that FRDOC_M outperforms FRDOC_A. However, for CWE classes, we cannot conclude which feature representation is better. In general, both approaches perform with lower F1 measures, especially for CWE078 and CWE601 with values lower than 0.5.

In terms of classifiers, in Pipe-3, we can observe that DecisionStump, Logistic, and JRip do not show the best performance for any class. Naive Bayes and SMO perform the best among most classes.

	#M	#SRM	So	Si	Sa	Ass	Aus	Anc	78	79	89	306	601	862	863
Android	128,783	39,165	25,085	13,798	503	503	136	288	188	158	334	1,151	229	1,016	109
Apache Commons	24,654	9,129	6,200	2,905	39	81	24	22	126	557	9	110	72	104	35
Apache Cordova	717	273	147	123	3	9	0	1	7	2	0	10	0	2	0
Apache Lucene	3,240	717	491	221	5	0	0	0	0	0	0	0	0	0	0
Apache Stratos	50,724	19,596	12,774	6,602	214	191	44	92	145	26	24	327	107	334	131
Apache Struts	2,670	1,315	752	560	6	0	0	0	360	4	0	0	3	0	0
Dropwizard	659	280	139	137	0	5	0	2	0	0	0	7	0	6	5
Eclipse Jetty	1,157	650	371	255	4	22	4	25	0	20	24	49	27	45	28
Eclipse SmartHome	934	261	185	76	0	0	0	0	0	0	0	0	0	0	0
GW7	44,970	8,093	5,785	2,255	117	7	1	2	41	268	5	10	57	0	0
Spark	884	142	96	22	1	24	0	0	0	0	0	24	6	3	4
Spring	31,369	12,622	7,275	5,138	72	339	36	134	125	301	785	504	233	441	157
Median %	100	31.72	20.39	11.04	0.33	0.41	0.08	0.19	0.34	0.46	0.41	0.75	0.25	0.67	0.16

Table 4.9: Total number of methods (#M), and number of SRM detected by SWAN per category.

	#MV	So	Si	Sa	Ass	Aus	Anc	78	79	89	306	601	862	863	Median
Android	650	0.98	0.9	0.98	0.62	0.8	0.66	0.96	0.78	0.62	0.52	0.5	0.52	0.62	0.727
Apache Commons	529	0.88	0.78	0.9	0.74	0.792	0.727	0.9	0.54	0.694	0.75	0.56	0.7	0.743	0.747
Apache Cordova	134	0.88	0.9	1	0.556	N/A	1	1	1	N/A	0.52	N/A	1	N/A	0.888
Apache Lucene	105	0.94	0.68	0.8	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.81
Apache Stratos	594	0.9	0.82	0.78	0.64	1	0.64	0.68	0.769	0.792	0.68	0.56	0.5	0.7	0.721
Apache Struts	163	0.94	0.88	1	N/A	N/A	N/A	1	0.54	N/A	N/A	1	N/A	N/A	0.804
Dropwizard	125	0.88	0.66	N/A	0.8	N/A	0.5	N/A	N/A	N/A	0.8	N/A	0.667	0.8	0.76
Eclipse Jetty	348	0.88	0.68	1	0.773	0.75	0.8	N/A	0.75	0.708	0.714	0.704	0.533	0.714	0.724
Eclipse Smarthome	100	0.96	0.96	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.96
JWT	367	0.92	0.8	0.82	0.857	0	1	0.439	0.6	1	0.8	0.72	N/A	N/A	0.734
Spark	134	0.82	0.955	1	0.875	N/A	N/A	N/A	N/A	N/A	0.917	0.833	0	0.75	0.851
Spring	636	0.9	0.82	0.82	0.68	0.861	0.8	0.9	0.66	0.889	0.82	0.6	0.62	0.8	0.785
Median		0.91	0.82	0.864	0.7	0.862	0.718	0.798	0.648	0.71	0.723	0.608	0.574	0.716	

Table 4.10: Number of manually verified methods (#MV), and precision of SWAN for each category on twelve Java libraries. N/A marks categories for which SWAN detected no methods.

Table 4.11: F1 measure of Pipe-1 for different classifiers per class based on 10-fold cross-validation averaged over ten runs (the highest values in each row are highlighted)

Class	NaiveBayes	SMO	DecisionStump	BayesNet	JRip	Logistic	J48
Sources	0.75	0.8	0.46	0.75	0.54	0.68	0.65
Sinks	0.74	0.76	0	0.79	0.66	0.77	0.66
Sanitizers	0.87	0.91	0.74	0.89	0.87	0.91	0.93
Auth-unsafe	0.05	0.43	0	0.36	0.66	0.49	0.58
Auth-no-cng	0	0.51	0	0	0.04	0.56	0.24
Auth-safe	0.78	0.95	0.83	0.81	0.88	0.84	0.88
CWE078	0.31	0.33	0.29	0.46	0.27	0.37	0.23
CWE079	0.26	0.15	0	0.28	0	0.22	0.03
CWE089	0.32	0.44	0.19	0.4	0.4	0.49	0.33
CWE306	0.79	0.81	0.75	0.84	0.8	0.85	0.84
CWE601	0.16	0	0	0.16	0	0.05	0
CWE862	0.71	0.71	0.47	0.79	0.68	0.75	0.71
CWE863	0.73	0.71	0.57	0.77	0.79	0.72	0.71

Table 4.12: F1 measure of Pipe-2 for different classifiers per class based on 10-fold cross-validation averaged over ten runs (the highest values in each row are highlighted)

Class	NaiveBayes	SMO	DecisionStump	BayesNet	JRip	Logistic	J48
Sources	0.52	0.67	0.57	0.58	0.57	0.61	0.64
Sinks	0.66	0.68	0.62	0.61	0.62	0.69	0.71
Sanitizers	0.54	0.86	0.84	0.79	0.79	0.76	0.8
Auth-unsafe	0.15	0	0	0.36	0.06	0.08	0.09
Auth-no-cng	0.04	0	0	0.25	0	0.03	0.07
Auth-safe	0.63	0.5	0.66	0.72	0.79	0.62	0.7
CWE078	0.21	0	0	0	0.37	0.26	0.3
CWE079	0.35	0.2	0.06	0.24	0.47	0.39	0.49
CWE089	0.28	0	0.07	0	0.34	0.29	0.35
CWE306	0.63	0.53	0.78	0.69	0.7	0.57	0.67
CWE601	0.17	0	0	0	0.18	0.26	0.21
CWE862	0.73	0.68	0.76	0.78	0.77	0.62	0.74
CWE863	0.73	0.53	0.77	0.74	0.71	0.61	0.7

The manual feature representation based on doc comments outperforms the automatic approach for the SRM classes, whereas for the CWE classes the F1 measure is comparable and, therefore, no approach can be declared as better.

Table 4.13: F1 measure of Pipe-3 for different classifiers per class based on 10-fold cross-validation averaged over ten runs (the highest values in each row are highlighted)

Class	NaiveBayes	SMO	DecisionStump	BayesNet	JRip	Logistic	J48
Sources	0.46	0.61	0.27	0.49	0.49	0.61	0.58
Sinks	0.61	0.63	0.38	0.55	0.54	0.6	0.57
Sanitizers	0.8	0.79	0.31	0.75	0.57	0.76	0.6
Auth-unsafe	0.18	0.18	0	0.18	0.08	0.16	0.06
Auth-no-cng	0	0.05	0	0.05	0	0	0
Auth-safe	0.78	0.81	0.66	0.76	0.66	0.64	0.59
CWE078	0.34	0.22	0	0.03	0.12	0.22	0.14
CWE079	0.42	0.46	0.04	0.38	0.42	0.43	0.47
CWE089	0.63	0.39	0	0.29	0.26	0.33	0.36
CWE306	0.6	0.61	0.43	0.61	0.52	0.51	0.52
CWE601	0.32	0.16	0	0.03	0.04	0.29	0.21
CWE862	0.72	0.81	0.32	0.71	0.54	0.64	0.58
CWE863	0.76	0.62	0.4	0.64	0.61	0.62	0.57

4.10.5 Hybrid feature representations (RQ11)

To answer this research question, we investigate the hybrid approaches FRCODE+DOC_M and FRCODE+DOC_A. FRCODE+DOC_M (Pipe-4) combines FRDOC_M and FRCODE, whereas FRCODE+DOC_A (Pipe-5) combines FRDOC_A and FRCODE. Table 4.14 shows the F1 measures for Pipe-4 and Table 4.15 for Pipe-5. When comparing the values of the best classifiers per class between Pipe-4 and Pipe-5 there is no significant difference. For example, for Sources, SMO was the best classifier in both pipelines, in Pipe-4 with 0.81 and in Pipe-5 with 0.85. For eight classes Pipe-4, got higher values, whereas Pipe-5 for five. Among the classifiers, Decision Stump has the lowest values in both pipelines.

When observing the F1 values of Pipe-4 and Pipe-5, compared to the previous pipelines, Pipe-1, Pipe-2, and Pipe-3, the hybrid approaches FRCODE+DOC_M and FRCODE+DOC_A outperform the approaches with individual feature representations in all classes.

The hybrid approach feature representations from source code and doc comments information shows to perform better than the individual feature representations, i.e., FRCODE+DOC_M and FRCODE+DOC_A outperform FRCODE and FRDOC_M.

Table 4.14: F1 measure of Pipe-4 for different classifiers per class based on 10-fold cross-validation averaged over ten runs (the highest values in each row are highlighted)

Class	NaiveBayes	SMO	DecisionStump	BayesNet	JRip	Logistic	J48
Sources	0.55	0.81	0.54	0.72	0.66	0.67	0.69
Sinks	0.69	0.84	0.62	0.8	0.76	0.7	0.74
Sanitizers	0.73	0.91	0.84	0.88	0.83	0.88	0.9
Auth-unsafe	0.31	0.3	0	0.66	0.69	0.5	0.6
Auth-no-cng	0	0.29	0	0.28	0.04	0.48	0.15
Auth-safe	0.67	0.92	0.68	0.78	0.83	0.85	0.87
CWE078	0.22	0.37	0.12	0.43	0.48	0.42	0.42
CWE079	0.35	0.45	0.04	0.43	0.39	0.46	0.53
CWE089	0.29	0.49	0.15	0.4	0.48	0.48	0.52
CWE306	0.7	0.8	0.75	0.75	0.78	0.74	0.84
CWE601	0.16	0.02	0	0.16	0.14	0.24	0.2
CWE862	0.75	0.81	0.76	0.8	0.79	0.79	0.76
CWE863	0.76	0.8	0.77	0.78	0.75	0.72	0.77

Table 4.15: F1 measure of Pipe-5 for different classifiers per class based on 10-fold cross-validation averaged over ten runs (the highest values in each row are highlighted)

Class	NaiveBayes	SMO	DecisionStump	BayesNet	JRip	Logistic	J48
Sources	0.53	0.85	0.46	0.62	0.62	0.76	0.69
Sinks	0.68	0.82	0.03	0.75	0.57	0.7	0.72
Sanitizers	0.86	0.9	0.74	0.85	0.79	0.86	0.93
Auth-unsafe	0.18	0.32	0.0	0.38	0.46	0.53	0.6
Auth-no-cng	0.0	0.29	0.0	0.14	0.5	0.47	0.24
Auth-safe	0.83	0.87	0.83	0.84	0.87	0.82	0.84
CWE078	0.4	0.4	0.08	0.43	0.37	0.46	0.37
CWE079	0.44	0.53	0.04	0.45	0.43	0.48	0.4
CWE089	0.66	0.64	0.0	0.44	0.27	0.48	0.36
CWE306	0.65	0.73	0.75	0.73	0.72	0.62	0.79
CWE601	0.34	0.21	0.0	0.18	0.09	0.29	0.14
CWE862	0.75	0.82	0.38	0.76	0.61	0.67	0.73
CWE863	0.78	0.77	0.58	0.7	0.58	0.68	0.72

4.10.6 Optimal classifier (RQ12)

SWANFRAME integrates the Auto-WEKA tool, which is an implementation of the AutoML approach. In three hours exploration, we got the results for each class as shown in Table 4.16.

In this experiment, we used FRCODE+DOC_M (Pipe-6), as it showed the best results among the manual variants that we tested previously. Considering the F measure, Pipe-6 found nearly the perfect classifiers in SWANFRAME. For Sanitizers, CWE089, and CWE862, it found the optimal classifiers reaching perfect accuracy of 1. The lowest values were reached for CWE078 with 0.51 and CWE601 with 0.69. These two classes performed badly in all other pipelines.

Furthermore, AutoWEKA found few classifiers as optimal which we manually did not test with. Logistic Model Tree (LMT) was selected for Sources, AdaBoostM1 for all authentication classes, Locally weighted learning (LWL) for CWE078, CWE089, and CWE601, SimpleLogistic for CWE306 and CWE863, and MultilayerPerceptron for CWE862. From those that we manually selected only BayesNet for Sinks and CWE079 and SMO for Sanitizers were selected as optimal.

Using the automatic selection of classifiers with AutoML, SWANFRAME finds the optimal classifiers for each class, which significantly improves the performance of the F1 measure when compared to the manual approach.

Table 4.16: Best classifiers per class based on F1 measure of Pipe-6 after running Auto-WEKA for three hours

Class	Classifier	F-Measure
Sources	LMT	0,9
Sinks	BayesNet	0,95
Sanitizers	SMO	1
Auth-unsafe	AdaBoostM1	0,86
Auth-no-cng	AdaBoostM1	0,84
Auth-safe	AdaBoostM1	0,98
CWE078	LWL	0,51
CWE079	BayesNet	0,94
CWE089	LWL	1
CWE306	SimpleLogistic	0,86
CWE601	LWL	0,69
CWE862	MultilayerPerceptron	1
CWE863	SimpleLogistic	0,82

4.11 Threats to Validity

According to Runeson et al. [RHRR12], we discuss internal, external, and construct validity.

Internal validity The results shown in our 10-fold cross-validation **RQ9-12** depend on a random variable that is used to create the folds during the 10-fold cross-validation. This means the values of the F measure may have different values with each iteration. To mitigate this, we reported an average value over ten runs.

External validity The main threat of our evaluation is that the results from **RQ9-12** may apply to only the labeled dataset we used. To mitigate this we used a well-established technique the 10-fold cross-validation. However, the conclusions made do not have statistical significance. Nevertheless, the experiment in **RQ8** with manual checks showed that SWAN achieves comparable results also for new datasets.

Construct validity The training datasets were constructed by the author and one more researcher, and reviewed by a third researcher. Selecting methods and labeling them with SRM and CWE information requires knowledge in security vulnerabilities. All researchers completed this task to best of their knowledge. They are all involved in application security research for several years.

Active Learning of Security Relevant Methods

This chapter introduces $\text{SWAN}_{\text{ASSIST}}$, an extension of SWAN for integrating developer feedback in the training set to improve the precision of the SRM detection by adapting to the codebase.

5.1 Related Work

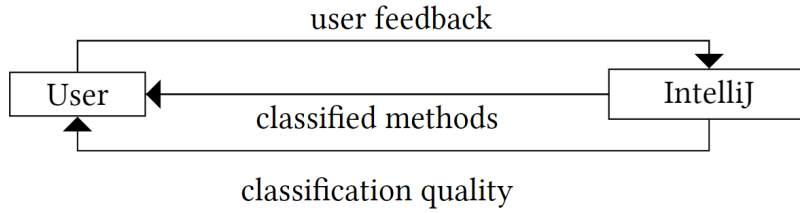
Past approaches have included post-processing of the static analysis results or have incorporated developers' feedback to refine static analysis results. In the first group, Fry et al. [FW13] use machine-learning to cluster analysis warnings into similarly actionable warning groups. Heckman et al. [HW09] propose a process for building false positive mitigation models to classify static analysis alerts as actionable items for the user. In such approaches, the machine-learning step is used offline, i.e., after the static analysis is run, and before the results are shown to the developer. They do not include developer feedback.

The second group includes developer feedback during or after the static analysis. For example, Aletheia [TGPA14] filters the results it displays to the user by learning the needs of developers. It shows the developer a portion of the warnings and asks them to classify them. Based on this information, it instantiates features for the machine-learning algorithms. This classification is used as a filter in the UI. Lucia et al.'s approach [LLJB12] uses incremental machine-learning to detect false positives in real-time. The system first presents a few findings from a tool in its default ordering. Users then either classify them as true or false positive. Based on the feedback, the system automatically and iteratively refines a classification model and re-sorts the rest of the findings. Similarly, Nguyen Quang Do et al. proposed Cheetah [DAL⁺17], an incremental approach for displaying the tools's findings as soon as they are generated. Cheetah does not incorporate any feedback from the user.

5.2 Approach

Because SWAN is designed for general Java applications, running on one particular program may not be sufficiently precise to correctly classify all methods in the codebase. In particular, this is the case in the classification of some CWEs, where SWAN achieves lower precision, such as CWE862 with an average precision of 0.574 in Table 4.10 (Page 68). To improve its precision, we have extended SWAN to query the code developer for their knowledge of the codebase.

SWAN is extended with the component $\text{SWAN}_{\text{ASSIST}}$ (as shown in the upper part of Figure 4.1), which allows developers to edit SWAN's training set directly in their Integrated Devel-

Figure 5.1: Active machine-learning in $\text{SWAN}_{\text{ASSIST}}$

opment Environment (IDE). The developer can add or remove methods of the training set or change the classification of a method. The new training set is then fed to SWAN for another classification iteration. To continuously refine the list of SRM, $\text{SWAN}_{\text{ASSIST}}$ uses the active machine-learning approach, by integrating the user. Typical users are software developers seeking to configure a static analyzer or members of security teams actively using and configuring SAST tools.


In this particular instance of active learning, $\text{SWAN}_{\text{ASSIST}}$ integrates the developer in the loop. It runs SWAN at each iteration by changing the training set. This system allows developers to further adapt the classification of the methods in their codebase after the original run of SWAN by improving the training set. Since the user is involved in the process, $\text{SWAN}_{\text{ASSIST}}$ is a tool-assisted approach.

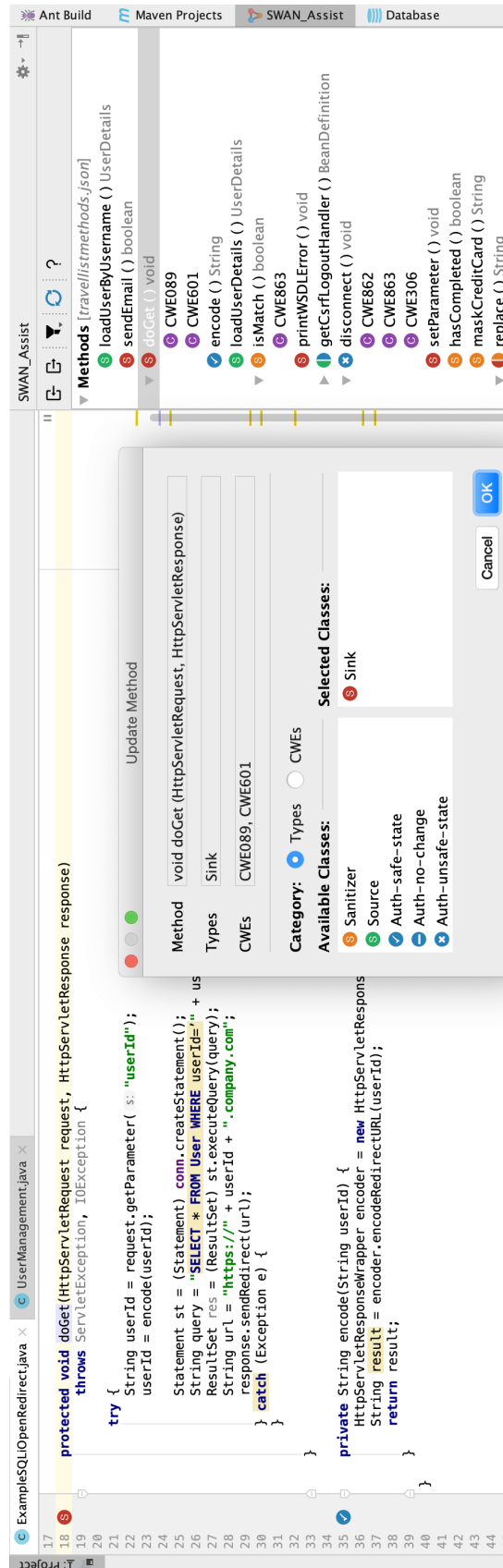
To help the developer identify methods that are most useful to the classification, $\text{SWAN}_{\text{ASSIST}}$ generates a list of methods that—if classified—would yield the most impact on the next run of SWAN, based on the feature matrix generated for the training set. Overall, $\text{SWAN}_{\text{ASSIST}}$ uses the automated mechanism of SWAN to detect SRM and enhances it with developer-based information to improve the precision of the SRM detection.

5.3 Tool

We have implemented $\text{SWAN}_{\text{ASSIST}}$ as a plug-in component for the IntelliJ IDEA IDE [Jet]. $\text{SWAN}_{\text{ASSIST}}$ provides an interface for editing the SRM lists and executing SWAN, updating the SRM classification on demand. Figure 5.2 presents $\text{SWAN}_{\text{ASSIST}}$ ’s Graphical User Interface (GUI), which we detail below.

SWAN’s training set is shown on the rightmost view of the GUI, called the $\text{SWAN}_{\text{ASSIST}}$ view. Methods in this view can be filtered by classification class or by file. The pop-up dialog in the center allows the developer to edit the training set. It is accessible through the $\text{SWAN}_{\text{ASSIST}}$ view or the context menu when a method in the code editor is selected. The developer can add or remove classes for the method with this dialog. Likewise, methods can be added to the training set through the context menu, and removed through the context menu or using the $\text{SWAN}_{\text{ASSIST}}$ view.

$\text{SWAN}_{\text{ASSIST}}$ also allows the developer to re-run the classification by clicking on the  icon in the toolbar of the $\text{SWAN}_{\text{ASSIST}}$ view. This update configures the inputs for SWAN, runs it in the background, and updates the list of SRM. This is shown as the dotted edge in the upper part of Figure 4.1 (Page 50). The methods that were just removed are displayed in gray, at the bottom of the list, and can be returned into the training set by using the *restore* functionality from the context menu. Otherwise, they are removed from the list on the next run. We have made the re-running of SWAN manual because of its running times. Re-computing the SRM for smaller libraries containing about a thousand methods (e.g., Eclipse Smarthome) takes under

Figure 5.2: Graphical User Interface (GUI) of SWAN_{ASSIST}

one minute. Larger libraries with thousands of methods (e.g., Android) can take up to a few minutes.

5.4 Suggesting Methods

To help developers classify methods more efficiently, the `SWANSUGGEST` module provides them suggestions of methods from the codebase. The suggested methods are likely to have the most impact on the classification (dashed edge in Figure 4.1). We developed two strategies for suggesting methods, *FeatureBasedSuggester* and *ModelChangeSuggester*.

Algorithm 1 presents the *FeatureBasedSuggester* strategy: it selects a pair of methods that will be the most impactful in the next classification round. The pair of methods is calculated by iterating over all method pairs. For all features of SWAN, if the method pairs have different values in the feature matrix (i.e., they are a pair of example/counter-example for that feature), the weight of that pair increases. In the end, the pair with the best weight is returned to the user. This is repeated until all features are covered, which is monitored by the global *features* set. When this point is reached, *features* is emptied and the loop starts again until all methods are classified, or until the user decides to stop.

Algorithm 1 Choosing the most impactful pair of methods with *FeatureBasedSuggester*

```

1: alreadySuggested  $\leftarrow \emptyset$ 
2: features  $\leftarrow \emptyset$  ▷ Keep covered features globally.
3: function SUGGEST(Boolean[Methods][Features] testSet)
4:   if features = swanFeatures() then
5:     features  $\leftarrow \emptyset$  ▷ Reinitialize coverage.
6:     (Method m1, Method m2)  $\leftarrow \emptyset$  ▷ Most impactful pair.
7:     for Method m1' in methods do
8:       for Method m2' in methods do
9:         if alreadySuggested.contains(m1', m2') then
10:          continue
11:         for Feature f in swanFeatures() do ▷ Add to the pair's impact if they have
           opposite evaluations.
12:           if testSet[m1'][f]  $\neq$  testSet[m2'][f] then
13:             updateFeaturesAndWeight(m1', m2')
14:             (m1, m2)  $\leftarrow \max((m1, m2), (m1', m2'))$ 
15:   features  $\leftarrow \text{features} \setminus (m1, m2).features$ 
16:   alreadySuggested = alreadySuggested  $\cup \{m1, m2\}$ 
   return (m1, m2)

```

The weight added to a pair for a particular feature depends on the feature: some features in SWAN are more likely to be impactful than others. We have determined this by evaluating the impact of the individual SWAN features through a One-At-a-Time (OAT) analysis. In this analysis, we ran ten 10-fold cross-validations on SWAN's training set per class (four SRM and seven CWE classes), disabling one feature instance at a time. For each run of the SVM classifier, we marked the F-measure (harmonic mean of precision and recall) averaged over all repetitions with randomly distributed folds. We used the F-measure to rank the offsets to obtain the feature weights with which we initialize. This rank of features is calculated only once and later reused for each project where the *FeatureBasedSuggester* is used. This is because the OAT analysis is based on the training set, which is used to build the classifier.

FeatureBasedSuggester strategy has quadratic complexity. More complex strategies could be used to suggest methods with a better impact, considering several iterations at once. The ideal solution can be reduced to a knapsack problem over all combinations of features, running in an exponential complexity. Since SWAN_{ASSIST} is designed to run in the IDE, we privileged the faster running method to satisfy the need for responsiveness.

Since the next strategy *ModelChangeSuggester* has higher complexity than *FeatureBasedSuggester* resulting in much longer running-times, and both show similar results in our evaluation in Section 5.5; we chose the *FeatureBasedSuggester* to be the default strategy for SWAN_{ASSIST}. We refer to *FeatureBasedSuggester* as SWAN_{SUGGEST}.

Algorithm 2 presents the *ModelChangeSuggester* strategy: it selects a pair of methods that make the most significant change to the model based on the F measure over 10-fold validation compared to all other method pairs. When a pair is selected, the models of all possible pairs of unlabeled methods are computed. Finally, the pair with the most significant change in precision is selected. Due to the complexity of this strategy to compute the models of all possible pairs of unlabeled methods, it is not part of the SWAN_{ASSIST} tool as the running times are long, making the tool unusable in the GUI context.

Algorithm 2 Choosing the most impactful pair of methods with *ModelChangeSuggester*

```

1: function SUGGEST(Set<Methods> testSet, Set<Method> trainSet)
2:   maxPair  $\leftarrow \emptyset$ 
3:   maxImpact  $\leftarrow 0$ 
4:   allUnlabeled  $\leftarrow \text{removeAlreadyLabeledMethods}(\text{testSet})$ 
5:   for method1  $\in$  allUnlabeled do
6:     for method2  $\in$  allUnlabeled do
7:       if method1  $\neq$  method2 then
8:         trainSet.add(method1)
9:         trainSet.add(method2)
10:        model  $\leftarrow \text{createModel}(\text{trainSet})$ 
11:        impact  $\leftarrow \text{run10foldCrossValidation}(\text{trainSet}, \text{model})$  ▷ F-measure as
        impact
12:        if impact  $>$  maxImpact then
13:          maxImpact  $\leftarrow$  impact
14:          maxPair  $\leftarrow (\text{method1}, \text{method2})$ 
15:          trainSet.remove(method1)
16:          trainSet.remove(method2)
17:   trainSet.add(maxPair)
18:   return maxPair

```

5.5 Evaluation

- **RQ13** How much manual training does SWAN_{ASSIST} require until it reaches *optimal* precision?
- **RQ14** How usable is the tool SWAN_{ASSIST}?

5.5.1 Manual Training

To answer **RQ13**, we next seek to evaluate how well SWAN_{ASSIST} helps improve the classifiers' precision, particularly how much manual training these improvements require. We consider both

strategies, *FeatureBasedSuggester* and *ModelChangeSuggester*, and compare them with a random selection. To evaluate the precision of the active machine-learning approach, we selected a (1) well-maintained, (2) open-source project that contains (3) a high number of SRM, and (4) fewer than 2,000 methods so we could classify all of them manually. We used the GitHub mining tool BOA [DNRN13] and selected the Gene Expression Atlas (GXA) [Ins] application, a popular tool in the domain of bioinformatics maintained by the European Bioinformatics Institute (EMBL-EBI) [EE]. We chose GXA since it showed lower precision with the base SWAN (e.g., the precision for sources is 0.75) compared to other libraries we evaluated in the previous chapter (see Table 4.10 with an average precision for sources of 0.91). This allows us to showcase the potential of the active machine-learning approach in the worst case compared to an application that already has a good precision, to begin with.

We manually classified and labeled the 1,663 methods of GXA with one or more of the following classes: sources, sinks, CWE89, and none. Two hundred eighty-six methods were identified as sources, 183 as sinks, and 29 as relevant to CWE89, and we consider this our ground truth. The methods were labeled twice, first by the author of this thesis and then by one external researcher. The Cohen’s Kappa value for sources is 0.605, 0.725 for sinks, and 0.919 for CWE89, which are above the significant agreement threshold of 0.6 [LK77]. For the cases with disagreement, they were discussed between the raters to reach an agreement.

To evaluate how different suggesting strategies help improve the results of SWAN, we compare the resulting SRM lists when feeding SWAN_{ASSIST} randomly selected method pairs, and when using *FeatureBasedSuggester* and *ModelChangeSuggester* to select those pairs.

Initially, we create two equal sets of methods. We use one for validation to calculate the precision of the resulting model with new data. The other set is used as testing set to feed the training set iteratively. We first run SWAN with its initial training set and calculate the precision of source, sinks, and CWE89 on the validation set. Then, we add a new method pair from the testing set to the training set and continue until we run out of methods. For every 410 iterations, we report the classification’s precision of sources in Figure 5.3, sinks in Figure 5.4, and CWE89 in Figure 5.5. The precision shown for the random suggester is averaged over ten runs.

We see the evolution of the precision for the random suggester is steady linear. This shows that the suggester does not help the classification: the precision increases naturally as a new pair is added to the training set. On the other hand, there is a quick increase in precision at the beginning when *FeatureBasedSuggester* and *ModelChangeSuggester* are used. Both strategies are efficient in selecting the methods with the most impact first. This maximizes the impact of the classification and minimizes the developer’s work to tune SWAN to their codebase.

When *FeatureBasedSuggester* and *ModelChangeSuggester* are compared, *ModelChangeSuggester* has more fluctuations (bigger spikes in the graph) and, on average, has lower values than *FeatureBasedSuggester*. In the case of sources, the precision reaches 0.8 at iteration 31 for *FeatureBasedSuggester* (from 0.75 at iteration 1), making 60 methods labeled (4% of the total number of methods in the application). For *FeatureBasedSuggester* this precision is reached at iteration 42. Afterward, the growth slows down, reaching a precision of 0.9 after 91 iterations. *ModelChangeSuggester* reaches this precision at iteration 103. In the random case, the growth is slower, reaching a precision of 0.8 at iteration 68 and 0.9 at iteration 249. For sinks, the precision of 0.8 is reached at iteration 10 for *FeatureBasedSuggester* and iteration 25 for *ModelChangeSuggester*. Precision 0.9 is reached at iteration 54 by *FeatureBasedSuggester* and iteration 76 by *ModelChangeSuggester*. *FeatureBasedSuggester* requires the developer to label less than 1% of the total number of methods in the application to reach a precision of 0.9. For the random suggester, this precision is only reached at iteration 234.

We see a similar trend in the case of CWE89. Here, we can note that *FeatureBasedSug-*

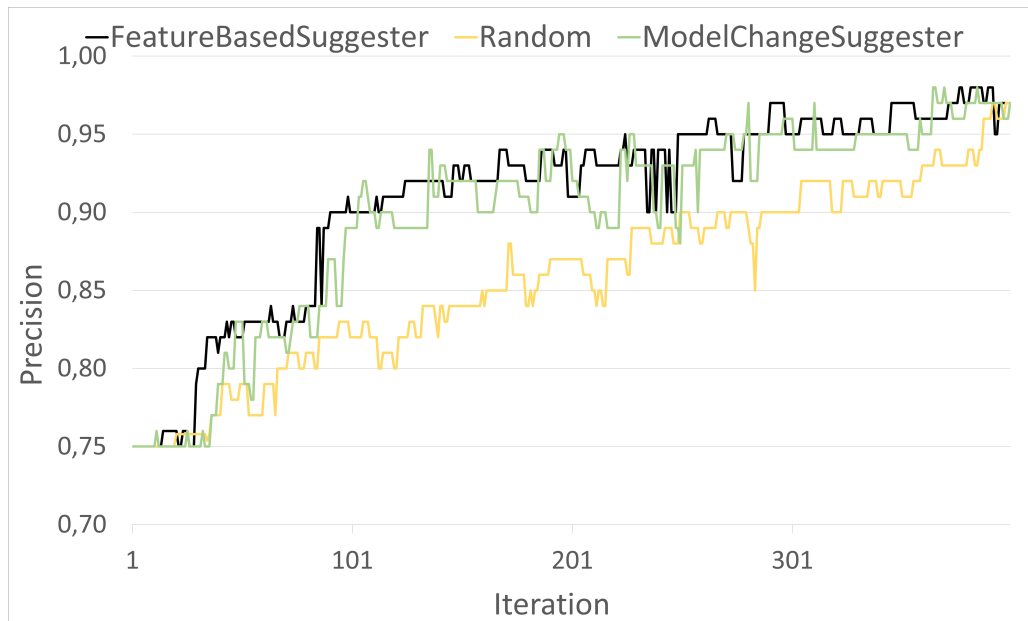


Figure 5.3: Precision of the sources over 401 iterations of SWAN by adding methods with *FeatureBasedSuggester*, *ModelChangeSuggester*, and with random selection

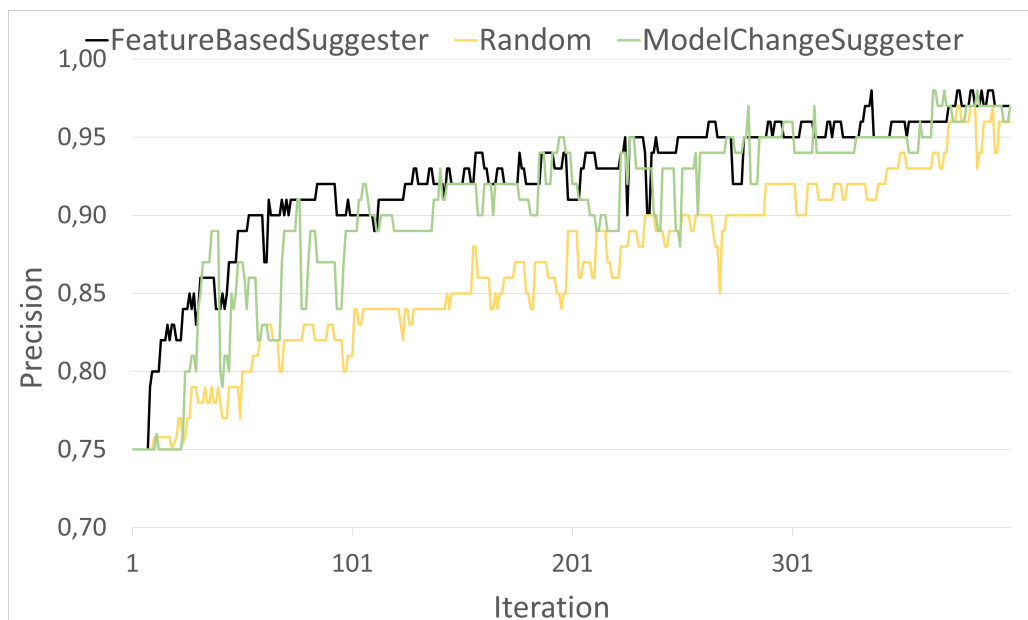


Figure 5.4: Precision of the sinks over 401 iterations of SWAN by adding methods with *FeatureBasedSuggester*, *ModelChangeSuggester*, and with random selection

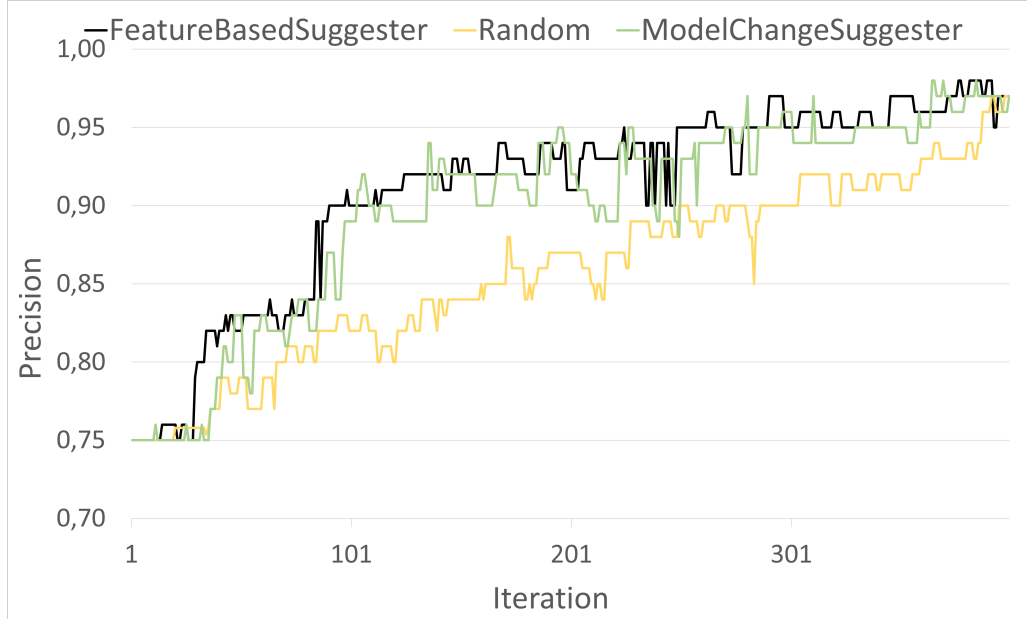


Figure 5.5: Precision of the CWE89 over 401 iterations of SWAN by adding methods with *FeatureBasedSuggester*, *ModelChangeSuggester*, and with random selection

gester and *ModelChangeSuggester* behave very similarly. Precision 0.8 is reached faster by *FeatureBasedSuggester*.

In all three cases, we note regular drops in the precision. Further investigation reveals that those drops occur when a problematic method is added to the training set. For example, method `void uk.ac.ebi.gxa.utils.EfvTree.put(uk.ac.ebi.gxa.utils.EfvTree)` is expected by the classifier to be a sink method, since it does not return anything, contains “put” in its name, and accepts an argument. However, the method is a simple accessor method and does not constitute a sink. Such methods pollute the training set and make the classification less precise until enough methods are added to compensate for the uncertainty. This issue can be mitigated by improving SWAN’s features or through smarter handling of the problematic methods and when to add them to the training set to minimize pollution. The presence of such methods also shows one more reason why a user-guided approach such as $\text{SWAN}_{\text{ASSIST}}$ is useful, particularly in the presence of imperfect training sets.

Using $\text{SWAN}_{\text{SUGGEST}}$ on GXA yields high precision significantly faster than with a random selection of methods.

5.5.2 Usability

To evaluate the usability of the $\text{SWAN}_{\text{ASSIST}}$ plugin (**RQ14**), we ran a 45-minutes user study with 22 participants who had experience with static analysis tools. The study included a 20-minutes cognitive walkthrough where participants performed 19 tasks from the following categories: executing the analysis, updating method classifications, and managing the method list. The participants were then asked to provide feedback on the best and worst features, possible improvements, and the plugin’s usefulness. After the interview, participants responded digitally to the System Usability Scale (SUS) ten-item questionnaire [Bro13], which assesses product usability. In addition to the SUS questions, the Net Promoter Score (NPS) question [Rei03] that evaluates how likely participants are to recommend the plugin to others was also included.

The average SUS score was 82.39, which corresponds to the 90-95th percentile range (scores better than 90-95% of the scores in the database) and an adjective rating [BKM09] of "excellent". The NPS score was 50, which indicates that most of the participants are promoters (59%) who would recommend the plugin to others.

The participants identified the following as the best features: code analysis and an initial list of security-relevant methods (59%); SRM icon designs and their visibility in the editor (50%); navigating from plugin tool window to method declaration (45%). Most participants considered the initial list of SRM to be most useful as it shows *"what could be possible security lapses and leaks"* in the code and this is a *"good headstart"*. The plugin's *"very straightforward and minimalistic"* design allows users to *"easily make out sources, sinks, and sanitizers based on the icons"*. Some participants commended the *"tight integration between [editor] and the plugin window"* which allows users *"to easily check if the method's classification is correct or not"*. The plugin *"is integrated into my workflow as a developer"* and *"most users would figure out how to use the plugin quickly"*. The tool's use of machine-learning *"tries to make [developers] more productive, and it does the work for [developers]"*. These reactions to the plugin indicate that most users are satisfied with its design and consider its features useful in creating and maintaining a SRM list.

The participants' feedback suggested improvements in the following areas: user experience, workflow, and IDE integration. As a result, the following changes have already been implemented to the tool which is available as an open-source project on GitHub¹:

- Standard menu icons, consistent with IntelliJ UI guidelines [Int] relating to design and colour palette, are now used. These icons are less ambiguous and meet user expectations.
- SRM icons were redesigned using more fitting colours and graphics to make categories easier to identify.
- List of reported classes/methods is now sorted for easier navigation and finding of methods
- Tool window action bar updated such that more popular buttons appear first in the menu.
- Dialog and menu labels updated to ensure that features/settings are better explained and less ambiguous

These improvements will enhance the plugin's usability and user experience as they meet user expectations and follow recommended design guidelines. Other suggestions relating to the plugin's workflow and integration have been recorded as GitHub Issues and will be worked on for future releases.

¹<https://github.com/secure-software-engineering/swan>

This chapter presents *fluent*TQL, a Java-internal domain-specific language (DSL) for specifying taint-flows. First, we identify requirements that motivated the design of this new DSL. Second, we discuss related existing DSLs. Afterward, we present the design and the semantics of *fluent*TQL, followed by implementation details and evaluation.

6.1 Requirements

Listing 6.1 shows an excerpt of the Java code of an HTTP handler. The method *doGet* is called upon a GET-request from a web browser when a user changes the password by providing the username, the old password, and the new password. The method calls a helper method *changePassword* shown in Listing 6.2, which verifies the user and changes the database. The code in *doGet* contains a potential cross-site scripting vulnerability (XSS) [Mit21b]. The username value from the request in the variable *uName* is added to the created HTML page for the response object to inform the user if the password was changed successfully (line 16). There is no sanitization check if the value contains any malicious behavior before it is added to the generated HTML page.

```

12 protected void doGet(@RequestParam("user")String uName, HttpServletRequest
    request, HttpServletResponse response) {
13     String oldPass = request.getParameter('oldKey');
14     String newPass = request.getParameter('newKey');
15     if (changePassword(uName, oldPass, newPass))
16         response.getWriter().append('<html>... Password changed for user' +
            uName + '...</html>');
17     else
18         response.getWriter().append('<html>...
19         Wrong credentials....</html>');
20 }

```

Listing 6.1: Java code with potential XSS vulnerability (from line 1 to line 5)

The code in the helper method *changePassword* contains a potential NoSQL injection vulnerability (NoSQLi) [Mit20b]. A single atomic action performs the user authentication and a password change in line 31, in which two database documents (*filter* and *set*), one with *\$where* clause and one with *\$set* clause, are executed. To report the taint-flow precisely, both values should be marked as tainted. We explain both XSS and NoSQLi vulnerabilities throughout this section.

```

21     protected boolean changePassword(String uName, String oldPass,
22                                     String newPass) {
23         MongoClient myMongoClient = new MongoClient("localhost",
24                                                     8990);
25         MongoDBDatabase credDB = myMongoClient.getDatabase('CREddb');
26         MongoCollection<Document> credCollection =
27             credDB.getCollection('CRED', Document.class);
28         BasicDBObject filter = new BasicDBObject();
29         filter.put('$where', '(username == \"' + uName + '\" ) \&
30             (password == \"' + oldPass + '\" )');
31         BasicDBObject newPassDoc = new BasicDBObject();
32         newPassDoc.put('password', newPass);
33         BasicDBObject set = new BasicDBObject();
34         set.put('$set', newPassDoc);
35         UpdateResult res = credCollection.updateOne(filter, set);
36         return (res.getMatchedCount() == 1);
37     }

```

Listing 6.2: Potential NoSQLi vulnerability (lines 1-3 to line 20)

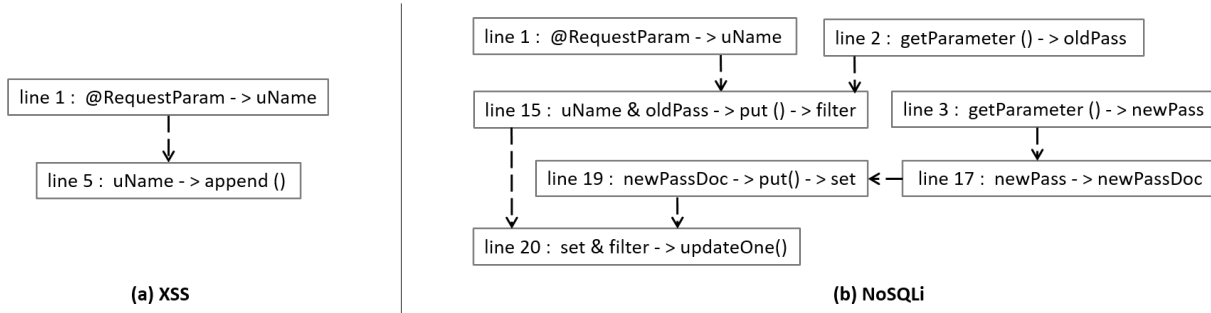


Figure 6.1: Data-flow graphs for (a) XSS and (b) NoSQLi vulnerabilities from Listing 6.1 and Listing 6.2

6.1.1 Selection of Sensitive Methods

To detect such vulnerabilities using a taint analysis, one must configure the analysis with any security-relevant methods (SRM), such as *sources*, *sinks*, and *sanitizers*.

Consider the example of the XSS vulnerability in Listing 6.1. Here, untrusted data flows from the parameter *uName* of the method *doGet* to the sink in line 16, where method *append()* is called with a string value of a request. Figure 6.1 (a) shows the data-flow graph extracted from the code. To fix this vulnerability, a software developer should apply a *sanitizer* such as *encodeHTML()* to clear potential malicious inputs from the variable *uName* before appending the contents to the HTML string. This leads to our first requirement:

R1: The DSL must allow one to express the following security-relevant methods (SRM): *source*, *sanitizer*, *propagator*, and *sink*.

6.1.2 Selection of In- and Out-Values

Apart from selecting the call sites, the actual values flowing in or out of the methods (return values, parameters, and receiver) must be selected. For example, for the source of the XSS vulnerability in Listing 6.1, the developer must select the argument value of the first parameter

of the method *doGet*. At the call to the sink of the vulnerability, the developer needs to provide the possibility to select a parameter of a called method.

R2: *The DSL must allow one to express the data-flow propagation of each SRM to a granularity of a single argument, a return value, and a receiver.*

6.1.3 Composition of Taint-Flows

The presented XSS vulnerability is detected by a “single-step taint analysis”. It is relatively easy to detect, even manually. However, many real-world taint-analysis problems comprise a sequence of multiple events. For example, consider the NoSQL injection vulnerability in Listing 6.2 and its data-flow graph in Figure 6.1 (b).

The NoSQLi vulnerability occurs in line 31 when the method *updateOne* is called under the condition that the Mongo database has a record with the username and the old password that matches the values coming from the request object (*uName* in line 12 and *oldPass* in line 14). The *filter* value contains the document that checks the existing password for the given username by calling the method *put* in line 26 with a *\$where*-clause. The value of *set* contains the document that sets the new password by calling the method *put* in line 18 with a *\$set*-clause. When the method *put* is called in line 26 and line 28, the *uName* and *oldPass* taint the *filter*, whereas the *newPass* taints the *set*. For the taint-flow to be complete, *both* calls to the method *put* must occur before the *set* and *filter* flow to the sink *updateOne()* in line 31. Thus, we desired a feature to compose complex queries consisting of multiple single-step taint analyses.

R3: *The DSL must allow one to express complex multi-step taint-flow queries.*

6.1.4 Detailed Error Message

When findings are reported, the analysis tool usually provides a description to the user to help understand the vulnerability. The study by Christakis et al. [CB16] showed that software developers struggle to understand those descriptions. For different vulnerabilities and types of data-flow, the DSL shall present the results of the taint analysis with fine-grained error messages that help developers quickly identify and fix the vulnerability. The user that specifies the taint-flow should be able to define a custom error message that can be reported at different locations.

R4: *The DSL must allow one to specify error messages for each type of finding.*

6.1.5 Integration into Developer’s Workflow

Empirical studies show that software developers need static analysis tools integrated into their workflow [CB16, JSMHB13]. Most software developers use integrated development environments (IDEs) and prefer static analyses directly integrated into the IDE. The results of the analysis should be shown within the IDE, preferably visible near the editor for the code. Therefore, a DSL designed for software developers should be integrated with appropriate tooling and usability in this workflow.

R5: *The DSL must integrate well with the software developers’ workflow.*

6.1.6 Independence of Concrete Taint Analysis

Software developers desire to reuse taint-flow specifications for static and dynamic taint analyses. Moreover, some analysis tools are only part of the continuous integration, whereas others can be integrated into different workflows, e.g., the IDE. To enable reusability of the specifications among different tools, the DSL semantics must therefore be independent of any concrete static or dynamic analysis. Thus, any limitations due to the approximations of the underlying solver are transferred to the results reported by *fluentTQL*.

R6: *The specified taint-flow queries can be reused among existing taint analysis tools, i.e., the DSL is independent of the underlying taint analysis.*

Due to its specific structure, the NoSQLi vulnerability from the example in this section, can not be detected by default with the existing tools. Such complex taint-flows require the user to specify a custom query. *fluentTQL* introduced in the following section aims to provide a usable and easy approach for mainly software developers specifying custom queries. The existing DSLs are designed for experts who understand data-flow analysis, which most developers do not have. Moreover, based on our evaluation of the existing DSLs in the following section, indicates that none of them completely fulfills all requirements.

6.2 Related Work

With few exceptions, such as DroidSafe [GdP⁺15], which has hard-coded SRM, the SRM of the existing static analyses can be customized by the user to some extent. In this section, we discuss the related approaches summarized in Table 6.1, which shows the design principles of each approach and the level of fulfillment of the requirements from Section 6.1.

Table 6.1: List of related approaches, their design characteristic, and their support of the requirements in Section 6.1. ● - fulfilled, ◐ - partially fulfilled, ○ - not fulfilled

	Approach	declarative	general-like	object-oriented	constraints	pattern-based	annotations	R1	R2	R3	R4	R5	R6
Graph-based	CxQL [Che20a]		X	X				●	◐	◐	●	●	○
	CODEQL [Git21b]	X	X	X				●	◐	◐	●	●	○
	PIDGIN [JWMC15]			X	X			●	●	◐	○	○	○
	IncA [SEV16]				X	X		◐	◐	○	○	◐	○
Typestate	CrySL [KSA ⁺ 19]	X				X		●	●	◐	○	◐	○
	PQL [MLL05]	X			X			●	●	◐	○	○	○
Other	CheckersF. [DDE ⁺ 11]						X	◐	●	○	●	○	○
	Apposcopy [FADA14]	X					X	◐	●	◐	○	○	●
	Athena [LS11]	X			X	X		●	●	○	○	○	○
	AQL [PBW18]				X	X		◐	●	○	○	●	○
	WAFL [SAP ⁺ 11]		X	X				◐	◐	◐	○	○	○
	Saluki [GvTB18]	X			X	X		◐	●	◐	○	○	○
	<i>fluentTQL</i>					X		●	●	●	●	●	●

6.2.1 Graph-based approaches

In this category, we group DSLs allowing users to explicitly manipulate graph structures to specify code patterns.

CxQL, the DSL of the tool Checkmarx [Che20a], is a general-purpose-like and object-oriented language. It supports a wide range of SRM (**R1**). The flow propagation is done implicitly via the graph patterns (**R2**). As a commercial tool, it provides a well-integrated workflow for the users (**R4** and **R5**). Since CxQL can express a broad range of graph properties, it is hard to integrate it into a generic taint analysis (**R6**) as it is bound to the tool’s core analysis.

CODEQL has good integration in the developers’ workflow (**R5**) with plugins for popular IDEs and a web-based interface. It is a declarative language with support for predicates and object-oriented design. The DSL can express a wide range of properties similar to CxQL.

PIDGIN [JWMC15] follows the object-oriented design. It is not designed for taint analysis; therefore, it is hard to integrate it in a generic way (**R6**). PIDGIN does not provide tool integrations for software developers.

IncA [SEV16] is a DSL specifying the rules for incremental execution of static analyses. Compared to the other DSL in this category, it is the least expressive and targets a particular domain. The SRM and flow propagation can be expressed through the graph patterns (**R1**, **R2**). User-defined messages are not supported (**R4**).

6.2.2 Typestate approaches.

This category consists of two DSLs, i.e., CrySL and PQL, designed for typestate analysis and detecting incorrect API usage.

CrySL [KSA⁺19] enables cryptography experts to specify the crypto API’s correct usage, making it unsuitable for generic taint analysis (**R6**). The DSL is declarative with a mechanism based on predicates and constraints. It has full support for SRM and flow propagation (**R1**, **R2**). The tool support is maintained.

PQL [MLL05] is declarative DSL with specifications comparable to CrySL. Compared to *fluent*TQL, the PQL’s syntax significantly differs from regular Java syntax, making it challenging for non-experts. PQL does not ship with available tooling support for the users (**R6**).

6.2.3 Other approaches.

The approach used in CheckersFramework [DDE⁺11] is based on the annotations *@tainted* and *@untainted*, which developers can use to annotate their code to mark custom SRM. The annotation specification requires additional manual work, which is even infeasible in the case of legacy code (**R5**). The CheckersFramework allows to configure the messages that are reported (**R4**).

Apposcopy [FADA14] is an Android-specific taint analysis (**R6**) with a Datalog-based DSL for data-flow and control-flow predicates. The sources, sinks, and propagators are specified in the form of annotations (**R1**, **R2**). We were not able to find tooling support (**R5**) for the language.

Athena [LS11] is a declarative DSL based on patterns and constraints with explicit support for SRM and flow propagations (**R1**, **R2**). Athena does not support user-defined messages and is tightly coupled to a generator of analysis configurations.

AQL [PBW18] is an Android-specific (**R6**) querying language for taint flow results from different taint analysis tools. It supports specifications of sources and sinks (**R1**) and flow propagations (**R2**) and provides workflow with tool support for developers to query taint results from multiple tools (**R5**).

WAFL is the DSL of the F4F approach [SAP⁺11] and is a general-purpose-like language with an object-oriented design that allows the specification of reflective behavior in frameworks so that static analyses can propagate the flow. WAFL partially supports SRM and flow propagation (**R1**, **R2**). Its main purpose is the modeling of frameworks.

Finally, Saluki [GvTB18] is a declarative DSL where method patterns can be specified. SRM and flow propagation are supported (**R1**, **R2**), but complex flows are not (**R3**).

6.3 Design

Next, we define the domain-specific language *fluentTQL* through its abstract and concrete syntax [SVC06].

6.3.1 Concrete Syntax

As a concrete syntax for *fluentTQL*, we use a Java fluent-interface syntax. Since Java is one of the most popular programming languages, software developers can learn the DSL with little effort. Moreover, in interviews with nine software developers [PKB21]¹, we asked what concrete syntax they would prefer if given the choice of (1) a fluent interface, (2) a graphical syntax, or (3) a textual syntax for taint-flow queries, six participants chose the fluent interface, and only two chose the graphical and one the textual syntax.

In the following, we explain the concrete syntax by specifying the *fluentTQL* queries for detecting the XSS and NoSQLi code in Listing 6.1 and Listing 6.2. The specification is presented in Listing 6.3, where lines 34–42 contain the SRM declaration and lines 43–49 contain the taint-flow queries.

```

34      Method source1 = new Method("String
      getParameter(String)").out().return();
35      Method source2 = new Method("void doGet(String, HttpServletRequest,
      HttpServletResponse)").out().param(0);
36      MethodSet sources = new MethodSet().add(source1).add(source2);
37      Method sanitizer = new Method("String encodeHTML
      (String)").in().param(0).out().return();
38      Method reqPropagator1 = new Method("BasicDBObject put(String,
      String)").in().param(1).out().thisObject();
39      Method reqPropagator2 = new Method("DBObject put(String,
      DBObject)").in().param(1).out().thisObject();
40      MethodSet reqPropagatorsPut = new MethodSet().
      add(reqPropagator1).add(reqPropagator2);
41      Method sinkXss = new Method("PrintWriter
      append(CharSequence)").in().param(0);
42      Method sinkNoSql = new Method("FindIterable
      updateOne(BasicDBObject,
      BasicDBObject)").in().param(0).param(1);
43      TaintFlowQuery xss = new
      TaintFlowQuery().from(source1).notThrough(sanitizer)
      .to(sinkXss).report("Reflective XSS
      vulnerability.").at(Location.SOURCE);
44      TaintFlowQuery noSqlI1 = new
      TaintFlowQuery().from(source1).through(
      reqPropagatorsPut).to(sinkNoSql).report(
      "No-SQL-Injection.").at(Location.SINK);
45      TaintFlowQuery noSqlI2 = new TaintFlowQuery();
46      noSqlI2.from(source1).through(reqPropagator1).to(
      sinkNoSql).and().from(source2).through(

```

¹https://github.com/secucheck/secucheck.github.io/blob/master/docs/SecuCheck_Interviews_Results.pdf


```

47      reqPropagator1).to(sinkNoSql).and().from(source1).
48      through(reqPropagator2).to(sinkNoSql).report(
49      "No-SQL-Injection vulnerability with multiple
        taint-flows").at(Location.SOURCEANDSINK);

```

Listing 6.3: *fluentTQL* specification for XSS and NoSQLi in Listings 6.1 and 6.2. To simplify, fully qualified method names are omitted.

In the code, there are two potential sources. One source is the return value of the *getParameter()* method which in Listing 6.3 is specified in line 34. The first argument to the constructor of *Method()* takes a method signature as a String argument. Next, using the fluent interface of *fluentTQL*, we append *out()* indicating that the method generates a sensitive data-flow. Eventually, by appending *return()*, we select the return value as the generated out-value. The other source is the first parameter of the *doGet()* method (line 35), indicated by *out()* and *param(0)*.

The fluent interface of *fluentTQL* allows calling *out()* or *in()* on a *Method* object. After *out()* there has to be at least one more call to *return()*, *thisObject()* and/or one or more calls to *param(int)* with the integer referring to the parameter index of the out-value. After *in()* there must be a call to *thisObject()* and/or one or more calls to *param(int)*.

Both sources in line 34 and line 35 are potential sources for SQLi and XSS, i.e., they are not specific to the vulnerability type. Thus, they are grouped into a *MethodSet* object (line 36). Afterward, the method *encodeHTML* is specified as sanitizer, which is only relevant to the XSS vulnerability. The method *put()* is a propagator (i.e., only propagates the taint) but a required one because it has to be called between the source and the sink for this specific vulnerability. It can be called with two different parameter types. Hence, it is specified twice (lines 38 and 39). They are grouped in the method set *reqPropagatorsPut*. Finally, the sinks are specified (lines 41 and 42). They are specific to each vulnerability type.

The taint-flow query for XSS is specified in line 43 where the class *TaintFlowQuery* is instantiated after which *from(...)*, *to(...)*, and *report(...)* are called. The sanitizer is also specified for the XSS taint-flow query by calling the method *notThrough(...)*. Each of these methods expects an object of type *Method* or *MethodSet*.

In the end, there is a call to *at(Location.SOURCE)*, which is optional and expresses where the report message should be shown in the code. *Location* is an enumeration with values *SOURCE*, *SINK*, and *SOURCEANDSINK*. The taint-flow query can be read as follows: *If there is a taint-flow from the source source1 not propagating through the sanitizer and reaching any of the sinkXss, then report a finding with "Reflective XSS vulnerability" at the source location.*

For the NoSQLi vulnerability, there are two taint-flow queries in Listing 6.3, in lines 44–49. The object *noSQLi1* will report a finding with a message "No-SQL-injection vulnerability" for the source *getParameter*, defined with *source1*, propagating through any required propagator from the set *reqPropagatorsPut* reaching the *sinkNoSql*. If applied to the code example from Listing 6.1 and Listing 6.2, two traces found will be found, which will be reported as separate findings. The taint-flow from the first parameter of *doGet* carrying the username will be missed. To detect this taint-flow, one can also use the method set *sources* instead of the single method *source1*. On the other hand, a taint analysis specified as defined though *noSQLi2* will report a single finding: For this specification, the three single taint-flows are joined by a call to *and()*, which means all separate taint-flows need to occur individually.

6.3.2 Abstract Syntax

We discuss the abstract syntax through the meta-model shown in Figure 6.2. The DSL has a root node (class *RootNode*) containing all objects. An object of this class represents a single instance of the DSL that can contain multiple top-level elements. For example, the abstract class

TopLevelElement is a superclass of the main concepts in *fluentTQL*, i.e., the classes *Method* and *TaintFlowQuery*.

Methods The class *Method* represents a reference to a method from the analyzed code. It contains information about the method signature and the data-flow propagation when that method is called in a given context (conforming to **R1** and **R2**). This is expressed through the references to *InputDeclaration* and *OutputDeclaration*. A *Method* object has to have one or both *InputDeclaration* or *OutputDeclaration* references. An *InputDeclaration* contains an in-value (abstract class *Input*), whereas *OutputDeclaration* contains an out-value (abstract class *Output*). In-values can be a parameter of a method call (class *Parameter*) or a receiver of the method (class *ThisObject*). Out-values can be a parameter, a receiver, or a return value (class *Return*). In-values flow into the method call, and out-values flow out of the method call.

The class *Method*, in combination with the classes *InputDeclaration* and *OutputDeclaration*, can model sources, sinks, and sanitizers (**R1**). Source is a combination of *Method* and *OutputDeclaration* specifying which values become tainted through a method call. Sink is an instance of *Method* and *InputDeclaration* specifying which values must be tainted for the sink to be considered “reached”. Sanitizer is a combination of a *Method* and *InputDeclaration*, specifying which tainted value flowing in the method call will get untainted.

Required propagators Required propagators are method calls that have to be on the path between a source and a sink for a given vulnerability to be present. For instance, the method *put()* in the running example from Section 6.1 has to be on the taint-flow trace from the source to the sink. It only propagates the taint from the in-value to the out-value. In *fluentTQL*, a required propagator is modeled as a combination of *Method*, *InputDeclaration*, and *OutputDeclaration*. This model allows propagating out-values once an in-value reaches a method. The analyses aware of these methods know how to propagate the data-flow without analyzing them, for example for improving scalability or handling calls for which the source code is unavailable.

Taint-flow queries The class *TaintFlowQuery* represents a taint-flow query. It contains all the information one needs to trigger a taint analysis. It contains one or more *TaintFlow* objects and a user-defined message (**R4**). The class *TaintFlow* has four references to the class *FlowParticipant*. The *from* reference defines the set of sources, the *through* reference defines the required propagators, the *notThrough* reference defines the sanitizers, and the *to* reference defines the sinks. For any valid *TaintFlow*, there should be at least one source and one sink. A *FlowParticipant* is either a *Method* or a *MethodSet*, i.e., a collection of methods. Similarly, the *QueriesSet* is a collection of taint-flow queries.

Imports and reuse The root node can contain imports from other models defined in other locations. This is modeled via the class *Import*, which allows references of methods and taint-flow queries from different files. The classes *Import*, *MethodSet*, and *QueriesSet* are provided for maintenance, reusability, and structure of *fluentTQL* specifications, enabling software developers to define categories of methods and taint-flow queries and share them (**R5**). As Java internal DSL, the users of *fluentTQL* get all advantages of Java compared to any external DSL or XML/JSON-based DSL often used in the existing tools. From Java, users can reuse existing abstractions such as packaging, modules, and object-oriented design to improve the maintenance, readability, and accessibility of the rules.

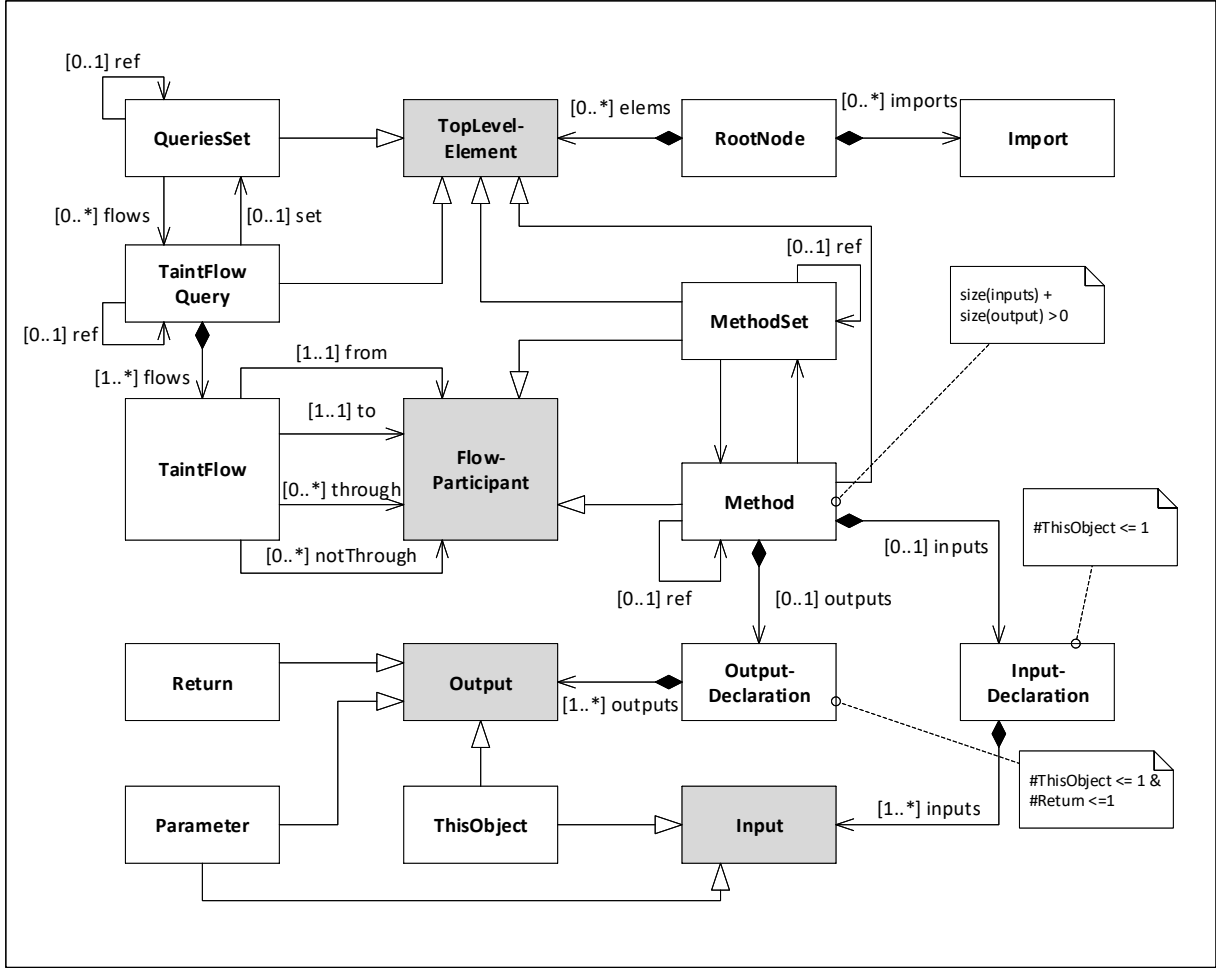


Figure 6.2: *fluent*TQL meta-model (UML class diagram, gray-filled classes are abstract). The constraints of the cardinalities of the classes are shown as messages, since the semantics of UML class diagram can not express all of them.

6.4 Semantics

A taint-flow query, an instance of the class *TaintFlowQuery*, is a *fluent*TQL specification that describes which traces of the program should be returned as findings to the user when a given taint analysis is triggered with that taint-flow query. In the following, we define the relevant terms and how *fluent*TQL refers to them.

We denote M as the set of all method signatures where a signature includes the fully qualified method name, parameter types, and a return type. A sensitive value is a type definition with information about the direction of propagation (in- or out-) and location (return, receiver, or parameter index). Hence, in- and out-values are sensitive values with in- and out-propagation.

Definition 6.1. A sensitive method is a tuple (m, SV) , where $m \in M$ and SV is a set of sensitive values. SV contains subset SV_{in} for in-values and subset SV_{out} for out-values.

Definition 6.2. A taint analysis specification TAS consists of the tuple $(Sources, Sanitizers, RequiredPropagators, Sinks)$, where

1. Sources is a set of sensitive methods (m, SV_{out}) for which SV contains at least one out-value, $(SV_{out} \neq \emptyset)$,

2. *Sanitizers and Sinks* are sets of sensitive methods (m, SV_{in}) for which SV contains at least one in-value, $(SV_{in} \neq \emptyset)$, and
3. *RequiredPropagators* is a set of sensitive methods (m, SV_{in}, SV_{out}) containing at least one in-value and one out-value $(SV_{in} \neq \emptyset, SV_{out} \neq \emptyset)$.

Given a taint analysis specification TAS , some black-box taint analysis T , and a program P , we assume the execution of T returns a set of traces for the data-flow, i.e., $T(P, TAS) = \{t_1 \dots, t_n\}$ where each t_i is a data-flow trace. A trace is a sequence of program statements, i.e., $t_i = s_i^1 s_i^2 \dots s_i^n$. For each individual trace t_i it holds that

- the first statement is a source statement, $s_i^1 \in Sources$,
- the last statement is a sink, $s_i^n \in Sinks$
- none of the statement s_i^j is a sanitizer, $s_i^j \notin Sanitizers$, and
- if *RequiredPropagators* is non empty, there exists exactly one element from *RequiredPropagators* that appears at statement s_i^j , where $j \in \{1, \dots, n\}$.

Note that in the case the analysis T is a dynamic taint analysis, the set of traces is a singleton set, while static analyses, which simulate all possible executions, may generate multiple traces.

Example: A TAS can detect rudimentary data-flows modeled with the class *TaintFlow* from Figure 6.2, such as the XSS vulnerability in Listing 6.1. The *TaintFlowQuery xss* in line 43 specifies a *TaintFlow* with

```

sources - {(getParameter(String),returnOUT), (doGet(String, ...), 0OUT)}
sanitizers - {(encodeHTML(String),0IN)}
r. propagators - {}
sinks - {(append(CharSequence),0IN)}
```

fluentTQL allows one to specify these sets with respective syntax elements **from(...)**, **notThrough(...)**, and **to(...)**. Running a taint analysis with the *fluentTQL* specification for *xss* on the code in Listing 6.1 returns the single trace consisting of the two statements² $t_1 = 12\ 16$.

Additionally, the syntax element **through(...)** allows specifying the set of *RequiredPropagators*.

For instance, the taint-flow query *noSQLi1* in line 44 specifies a non-empty set *RequiredPropagators*.

```

sources - {(getParameter(String),returnOUT)}
sanitizers - {}
r. propagators - {(put(String, String),1IN,returnOUT), (put(String, BasicDBObject), 1IN,returnOUT)}
sinks - {(updateOne(BasicDBObject, BasicDBObject),0IN,1IN),returnOUT}
```

The result of the taint-flow query *noSQLi1* is

$$Trace_{noSQLi1} = 13\ 15\ 21\ 26\ 31, 14\ 15\ 21\ 28\ 30\ 31$$

and consists of two traces. Each of these traces is reported as a separate finding to the user. A “simple” finding is a single trace with a single message. For instance, the findings of *noSQLi1* are $Findings_{noSQLi1} = \{F_{noSQLi1}^1, F_{noSQLi1}^2\}$, where

² We use line numbers from Listing 6.1 and Listing 6.2 to represent traces.

$$F_{noSQLi1}^1 = (\{13 \ 15 \ 21 \ 26 \ 31\}, \text{"No-SQL-Injection vulnerability."})$$

$$F_{noSQLi1}^2 = (\{14 \ 15 \ 21 \ 28 \ 30 \ 31\}, \text{"No-SQL-Injection vulnerability."})$$

Yet, for more complex queries one can use the *and()* operator, which combines findings over individual traces to a single finding over multiple traces.

Combining taint-flow queries: The *and()* operator allows one to merge multiple TAS as a single query. This is through an object of type *TaintFlowQuery* (from Figure 6.2) that contains multiple objects of type *TaintFlow*. The operator computes the cross-product of the traces of the individual TAS. For example, the taint-flow query *noSQLi2* in line 45 defines three TAS specifications:

```

sources - {(getParameter(String),returnOUT)}
sanitizers - {}
r. propagators - {(put(String, String),1IN),returnOUT}
sinks - {(updateOne(BasicDBObject, BasicDBObject),0IN,1IN)}

sources - {(put(String, String),returnOUT)}
sanitizers - {}
r. propagators - {}
sinks - {(updateOne(BasicDBObject, BasicDBObject),0IN,1IN)}

sources - {(getParameter(String),returnOUT)}
sanitizers - {}
r. propagators - {(put(String, BasicDBObject),1IN,returnOUT)}
sinks - {(updateOne(BasicDBObject, BasicDBObject),0IN,1IN)}
    
```

The first one returns the trace “13 15 21 26 31”, the second one returns the trace “12 15 21 26 31”, and the last one returns the trace “14 15 21 28 30 31”. Yet, the result of the query *noSQLi2* will be a single finding, $Findings_{noSQLi2} = \{F_{noSQLi2}^1\}$, where

$$F_{noSQLi2}^1 = (\{13 \ 15 \ 21 \ 26 \ 31, \ 12 \ 15 \ 21 \ 26 \ 31, \ 14 \ 15 \ 21 \ 28 \ 30 \ 31\}, \text{"No-SQL-Injection vulnerability with multiple taint-flows."}).$$

Calculating traces: By its definition, *fluent*TQL has precise runtime semantics. However, when applied in a static context, the traces need to be approximated by the underlying data-flow engine. Thus, reported traces of different tool implementations can differ.

To explain the precise runtime semantics for trace construction, we define a taint analysis core language similar to previous works [SAB10, Liv12]. Though simple, the core language covers relevant statements that can be mapped one-to-one with Java statements. The statements are listed in Table 6.2. A program of the language contains a sequence of statements with line numbers. For simplicity, we decided to exclude method calls from the language. These can be compiled to the language by storing the memory address of the return statement and transferring the control flow. This rule is not applied to the four statements in Table 6.2, which are special method calls.

We denote variables with x and y , a field of an object with f , and an i -th index of an array with $a[i]$. We model all memory locations through a shadow heap: The shadow-heap value for a memory location v is *true* if the value is tainted and *false* otherwise. The execution context Σ has the parameters listed in Table 6.3. $\Sigma.\Delta[x]$ stores the current taint value of variable x . We write $\Sigma \vdash x \rightarrow v$ to extract that value into v . Similarly, notations like $src(x) \vdash (m, SV)$ extract the method m with its sensitive values SV , when a method call src is matched. Additionally,

Table 6.2: Statements of the core language for constructing *fluentTQL* traces

Statement	Description
$\text{src}(x)$	call to a sensitive method $(m, SV_{out}) \in \text{Sources}$ with sensitive parameter x
$\text{snk}(x)$	call to a sensitive method $(m, SV_{in}) \in \text{Sinks}$ with leaked parameter x
$\text{san}(x)$	call to a sensitive method $(m, SV_{in}) \in \text{Sanitizers}$ sanitizing parameter x
$\text{rpr}(x)$	call to a sensitive method $(m, SV_{in}, SV_{out}) \in \text{RequiredPropagators}$
$x = y$	assignment
$x = y.f$	field load
$x.f = y$	field store
$x = a[i]$	read from array at index i
$a[i] = x$	write to array at index i
skip	skip and continue

Σ stores all traces $t \in T$ that will be created during the execution. t is a sequence of statements (which we here denote by line numbers).

Table 6.3: Parameters used within the core language for constructing *fluentTQL* traces

Parameter	Description
Δ	match a variable, a field, an array element or a sensitive value to its taint value
λ	match a given statement to its line number
θ	returns the set of all traces created

Figure 6.3 shows *fluentTQL*'s semantics through inference rules. We use a syntax akin to the one used by Schwartz et al. [SAB10]. The semantics essentially define a regular dynamic taint analysis which collects un-sanitized traces from sources to sinks as a side-effect. For instance, given the statement $x = y$, the ASSIGN rule's computation comprises four parts. First, $\Sigma \vdash y \rightarrow v$ evaluates and extracts the taint value v for variable y . Due to the assignment, the rule updates the taint value of x with v . The rule then extracts each trace in $t \in \theta$ and adds the current statement, identified by its line number. The rules for load/store and array accesses are equivalent. The rule SOURCE creates a new trace and taints the out-value; the rule SINK gracefully terminates a trace by untainting the sensitive value. The rule SANITIZER also discontinues the tracing. The rule PROPAGATOR taints the out-value if the in-value is tainted. SKIP advances to the following statement, whereas SEQ enables the progression of the semantics covering the recursive case. The semantics must enforce one aspect that we found hard to capture with inference rules: for such *fluentTQL* specifications that define required propagators, the taint analysis must ensure to report only such traces that contain all required propagators. Finally, the notion of a user-defined message is skipped in the formal semantics due to simplicity, but we explain it in the following through our example.

Report message: As seen in the previous examples, the queries specified in *fluentTQL* contain a user-defined message added to each finding (**R4**). In the concrete syntax, the mandatory syntax element `report(...)` takes the string message as an argument. Optionally, the user may specify the location for the reporting message by using the syntax element `at(...)`. As an argument, the enumeration *Location* can be used, which contains three elements *SOURCE*, *SINK*, and *SOURCEANDSINK*. *SOURCE* and *SINK* define that the reporting message should be shown at the source and the sink location respectively. For *SOURCEANDSINK* the message should be shown at the source and sink locations in the code. The reporting message is shown for each trace individually if the finding has multiple traces. For example, for *noSQLi2*, the error message will be shown at each source and sink location, i.e., lines 1, 2, 3, and 20, because *SOURCEANDSINK* is used in the query specification (Listing 6.3, line 49). Tools can use this

$$\begin{array}{c}
 \frac{\Sigma \vdash y \rightarrow v \quad \Sigma.\Delta' = \Sigma.\Delta[x \leftarrow v] \quad t \in \Sigma.\theta \quad \Sigma.\theta' = \Sigma.\theta[t \leftarrow \langle t, \lambda[x = y] \rangle]}{(\Sigma, x = y) \rightsquigarrow (\Sigma', \text{skip})} \quad (\text{ASSIGN}) \\
 \\
 \frac{\Sigma \vdash y.f \rightarrow v \quad \Sigma.\Delta' = \Sigma.\Delta[x \leftarrow v] \quad t \in \Sigma.\theta \quad \Sigma.\theta' = \Sigma.\theta[t \leftarrow \langle t, \lambda[x = y.f] \rangle]}{(\Sigma, x = y.f) \rightsquigarrow (\Sigma', \text{skip})} \quad (\text{LOAD}) \\
 \\
 \frac{\Sigma \vdash y \rightarrow v \quad \Sigma.\Delta' = \Sigma.\Delta[x.f \leftarrow v] \quad t \in \Sigma.\theta \quad \Sigma.\theta' = \Sigma.\theta[t \leftarrow \langle t, \lambda[x.f = y] \rangle]}{(\Sigma, x.f = y) \rightsquigarrow (\Sigma', \text{skip})} \quad (\text{STORE}) \\
 \\
 \frac{\Sigma \vdash a[i] \rightarrow v \quad \Sigma.\Delta' = \Sigma.\Delta[x \leftarrow v] \quad t \in \Sigma.\theta \quad \Sigma.\theta' = \Sigma.\theta[t \leftarrow \langle t, \lambda[x = a[i]] \rangle]}{(\Sigma, x = a[i]) \rightsquigarrow (\Sigma', \text{skip})} \quad (\text{READ-ARRAY}) \\
 \\
 \frac{\Sigma \vdash x \rightarrow v \quad \Sigma.\Delta' = \Sigma.\Delta[a[i] \leftarrow v] \quad t \in \Sigma.\theta \quad \Sigma.\theta' = \Sigma.\theta[t \leftarrow \langle t, \lambda[a[i] = x] \rangle]}{(\Sigma, a[i] = x) \rightsquigarrow (\Sigma', \text{skip})} \quad (\text{WRITE-ARRAY}) \\
 \\
 \frac{\text{src}(x) \vdash (m, SV_{out}) \quad sv \in SV_{out} \quad \Sigma.\Delta' = \Sigma.\Delta[sv = \text{true}] \quad \Sigma.\theta' = \Sigma.\theta \cup \{\lambda[\text{src}(x)]\}}{(\Sigma, \text{src}(x)) \rightsquigarrow (\Sigma', \text{skip})} \quad (\text{SOURCE}) \\
 \\
 \frac{sv \in SV_{in} \quad \Sigma \vdash sv \rightarrow \text{true} \quad \text{snk}(x) \vdash (m, SV_{in}) \quad t \in \Sigma.\theta \quad \Sigma.\theta' = \Sigma.\theta[t \leftarrow \langle t, \lambda[\text{snk}(x)] \rangle]}{(\Sigma, \text{snk}(x)) \rightsquigarrow (\Sigma', \text{skip})} \quad (\text{SINK}) \\
 \\
 \frac{\text{san}(x) \vdash (m, SV_{in}) \quad sv \in SV_{in} \quad \Sigma.\Delta' = \Sigma.\Delta[sv = \text{false}]}{(\Sigma, \text{san}(x)) \rightsquigarrow (\Sigma', \text{skip})} \quad (\text{SANITIZER}) \\
 \\
 \frac{sv_2 \in SV_{out} \quad \Sigma \vdash sv_1 \rightarrow \text{true} \quad \text{rpr}(x) \vdash (m, SV_{in}, SV_{out}) \quad sv_1 \in SV_{in} \quad \Sigma.\Delta' = \Sigma.\Delta[sv_2 = \text{true}] \quad \Sigma.\theta' = \Sigma.\theta[t \leftarrow \langle t, \lambda[\text{rpr}(x)] \rangle]}{(\Sigma, \text{rpr}(x)) \rightsquigarrow (\Sigma', \text{skip})} \quad (\text{PROPAGATOR}) \\
 \\
 \frac{}{(\Sigma, \text{skip}; S) \rightsquigarrow (\Sigma, S)} \quad (\text{SKIP}) \qquad \frac{(\Sigma, S_1) \rightsquigarrow (\Sigma', S'_1)}{(\Sigma, S_1; S_2) \rightsquigarrow (\Sigma', S'_1; S_2)} \quad (\text{SEQ}) \\
 \\
 \frac{}{\Sigma \vdash \text{v} \rightarrow \text{false}} \quad (\text{FALSE})
 \end{array}$$

 Figure 6.3: Inference rules of the operational semantics of the traces construction in *fluent*TQL

information for visualization purposes. E.g., an IDE plug-in may display error markers in the editor at the source location, the sink location, or both.

Usability versus expressiveness: *fluentTQL* is a DSL for users without deep expertise in the static analysis, as most software developers. Its purpose is to enable users to specify a custom taint analysis for their codebase and detect many popular security vulnerabilities. Hence, the usability and simplicity of the language are the primary aim. A trade-off to this design decision is the lower expressiveness compared to some existing DSLs such as *CODEQL*. *fluentTQL* does not provide the users a fine-grained manipulation of the abstract syntax tree. Program analysis experts use such expressive DSLs. Our evaluation shows that the most popular security vulnerabilities can be expressed in *fluentTQL*. On the side of expressiveness, *fluentTQL* supports **R3**, which is only partially supported by other DSLs. Complex multi-step taint-flow queries are particularly relevant for stored versions of SQLi and XSS vulnerabilities.

6.5 Implementation

We implemented *fluentTQL* as an internal Java DSL which can be easily used in any Java project by implementing the interface *FluentTQLSpecification*. Hence, any Java editor can be used to write and edit *fluentTQL* queries.

fluentTQL is implemented as a standard Java library using the builder pattern to allow method chaining as a user interface. All queries need to be implemented within a class that implements the interface *FluentTQLSpecification*. Using the Java classloader, the classes are located and the queries correctly loaded and provided as input to the analysis.

To instantiate *fluentTQL* with concrete analyses, we first implemented a taint analysis built on top of the Boomerang solver [SAB19], an efficient and precise context-, flow-, and field-sensitive data-flow engine with demand-driven pointer analysis. Boomerang provides an API to query all traces from given seeds. The API of the seed is expressible to cover the *fluentTQL* semantics of the sensitive methods. However, the basic API of Boomerang does not support sanitizers or required propagators. To support the sanitizers, we transformed the bodies of the sanitizers to empty, a terminal case of the Boomerang data propagation solver. To support required propagators, we break the TAS specification into multiple TAS specifications containing only sources and sinks. A TAS with a required propagator is broken to two TAS where the first one has the original source and the required propagator as sink, whereas the second one has the required propagator as source and the original sink as sink. Boomerang returns the traces of the individual TAS, and our implementation merges them.

Moreover, we instantiated *fluentTQL* with the existing taint analysis of FlowDroid [ARF⁺14]. This, however, was not possible without limitations. Specifically, the default component for defining sources and sinks in FlowDroid is limited and supports only return as out-value of sources and parameter index as in-value of sinks. This can be extended by adding a new implementation of the *SourceSinkManager*, which we left as future work. Furthermore, Sanitizers by default are not supported, but we applied the same solution as in our Boomerang implementation. In contrast, required propagators are not supported and require either extension of the taint analysis or post-processing of the findings, which we also consider as future work.

Finally, both instances of *fluentTQL* have some limitations in how the traces are constructed and reported. Since *fluentTQL* has precise runtime semantics, it is expected that static analysis engines like Boomerang and FlowDroid will approximate. In particular, both engines will unsoundly underapproximate the constructed traces. For example, both apply different strategies for merging conditional paths of the program. Thus, these limitations are part of our implementation, too.

6.6 SecuCheck

SECUCHECK is the tool that integrates *fluentTQL* with the available taint analyses using the Magpie Bridge [LDB19] framework. The tool addresses the following requirements:

- R1** *Workflow integration*: Software developers reported that the tools should be well integrated within their daily used development environments (IDEs) to develop new applications. They should appear as part of the IDEs and only provide the necessary findings reported from the analysis using the standard IDE features, such as error view, editor markers, and syntax highlighting.
- R2** *Configurable tools*: One of the known weaknesses of static analysis, including taint analysis, is the reporting of false findings, which causes usability issues [CB16]. One approach to improve this is by configuring the rules of the analyses through domain-specific languages (DSLs). This allows the specification of custom rules for company-specific contexts. Even though many tools provide such DSLs, their stakeholders are static analysis experts. Therefore, software developers need developer-centric DSLs.
- R3** *Explainability*: The messages of the findings shown to the users should be understandable. The tools should provide additional information about the findings when needed. Referring to **R2**, the DSL should also be understandable for software developers.
- R4** *Fast results*: Taint analysis can run long on real-world applications measured in minutes and even hours, which is not practical in the IDEs. Hence, a taint analysis running in the IDE should provide means to analyze only parts of the code relevant to the user in the current context a few minutes or seconds.

SECUCHECK is open source under <https://github.com/secure-software-engineering/secucheck>. A video is available under <https://www.youtube.com/watch?v=3ivgsib0mXo>.

6.6.1 Architecture

Figure 6.6 shows the internal components of SECUCHECK and their interaction with the external components. The components in orange are directly accessible to the users through provided interfaces. The SECUCHECK-core analysis runs the main analysis process. It uses SOOT to generate the Jimple format from the bytecode being analyzed and calls the Boomerang or the FlowDroid APIs to run one of the solvers. SECUCHECK-Magpie integrates the SECUCHECK-core into Magpie Bridge. An alternative way to run SECUCHECK is through the command line tool SECUCHECK-cmd. The *fluentTQL*-DSL is a DSL for specifying taint-flows queries for the analysis. *fluentTQL*-classloader uses the JCL-core to load the taint-flow specifications into the JVM. The maven-plugin-api provides APIs for running tools as a Maven plugin. This is used by *fluentTQL*-maven-plugin to run a semantic check of the *fluentTQL* specifications. Finally, the *fluentTQL2English* transforms the *fluentTQL* specifications into English sentences to provide the user with a more detailed description of the queries (**R3**). The components SECUCHECK-Magpie and SECUCHECK-cmd use *fluentTQL2English* to display it in the error message (**R1**).

6.6.2 UI Features

In the following, we discuss the three primary user interface features SECUCHECK provides to the users of multiple IDEs clients.



Figure 6.4: Configuration first page

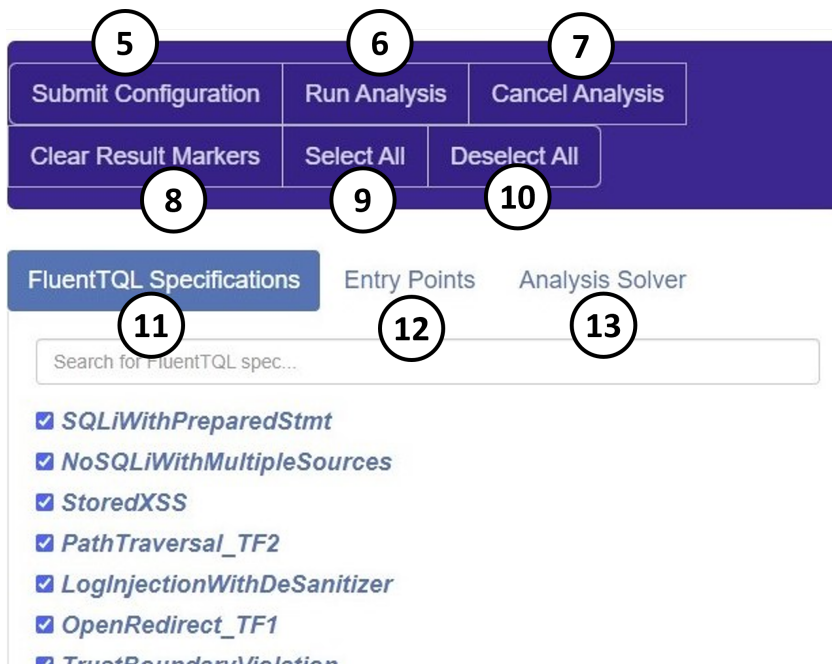


Figure 6.5: Configuration second page

Configuration page For managing the analysis, SECUCHECK has two configuration pages (**R2**), created with the Bootstrap 3.3.5 framework³. This is supported through the Magpie Bridge server using the HTTP protocol. When the project in the IDE opens, SECUCHECK will create the first configuration page, as shown in Figure 6.4. The project name is shown on the top ①. Two tabs ② and ④ are available on this page. ① is for setting the path of the external jar with *fluentTQL* specification ③. ④ is for customizing the view of the queries on the next page. When the first page is submitted, the second one will automatically appear (Figure 6.5). This page shows six buttons for submitting a configuration ⑤, triggering the analysis ⑥, canceling already started analysis ⑦, clearing the results from the previous analysis in the IDE ⑧, selecting all elements from the list ⑨, and deselecting all elements from the list ⑩. The page has three lists of elements, one in each tab. ⑪ shows a list of all taint-flow queries (**R2**)

³<https://bootstrapdocs.com/v3.3.5/docs/getting-started/>

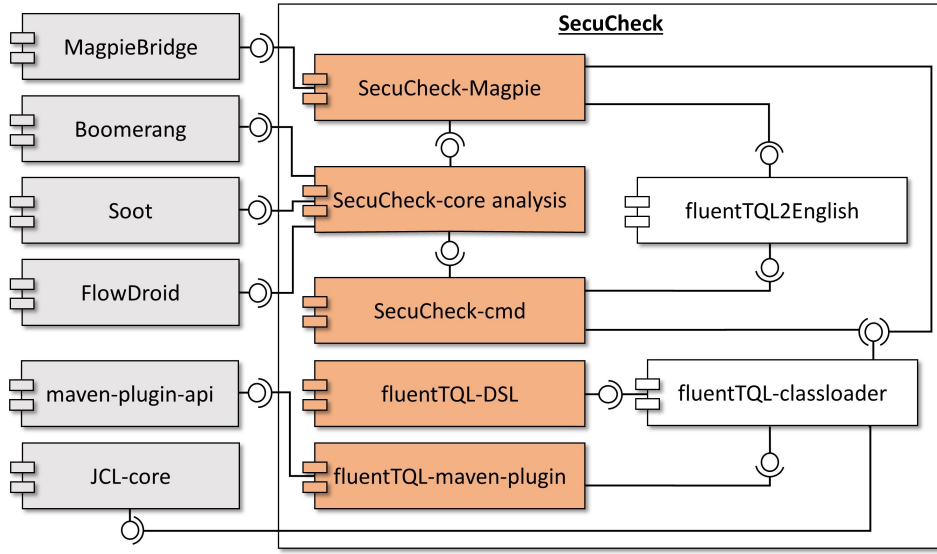


Figure 6.6: SECUCHECK architecture.

that are available. (12) lists all classes from the codebase that can be selected as entry points for the call graph construction (R2). With (13) the user can select the solver, Boomerang, or FlowDroid. These selections allow the user to run the analysis for a specific context and get fast results (R4).

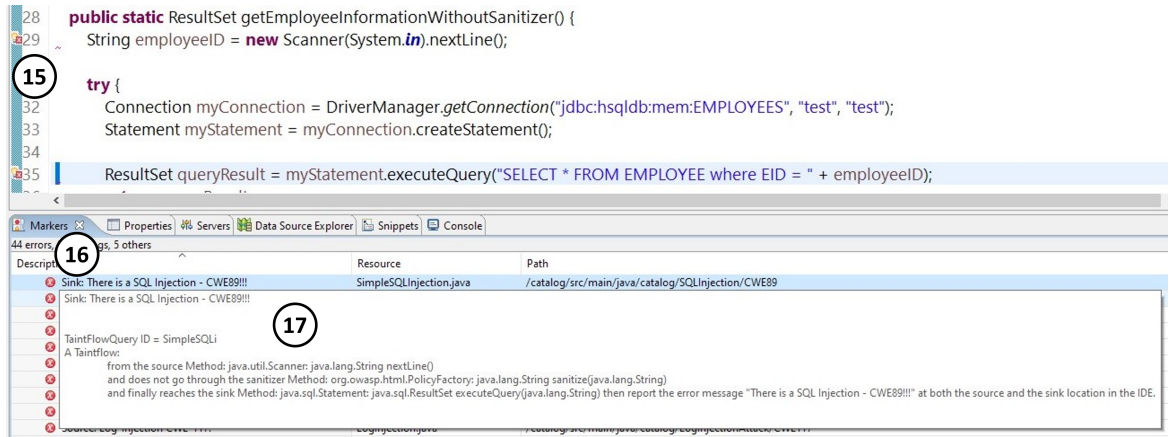


Figure 6.7: Eclipse IDE view.

IDE standard UI features Figure 6.7 shows a screenshot of the Eclipse as an example IDE that indicates the standard editor features (R1) that SECUCHECK uses to display the analysis results. The results are listed in the standard error view (3). Error markers are shown on the side of the editor (2). A hover over the error item or the marker shows a more detailed description of the found taint-flow (14). This message is an English translation of the *fluentTQL* specification for the found taint-flow. We explain this translation in the following.

Explainability of the findings To improve the explainability of the result messages in SECUCHECK, we implemented *fluentTQL2Eng*, a translator to English sentences. *fluentTQL2Eng*

parses a taint-flow query object and visits each field. It maps each field to a predefined phrase recursively. Within the sentence, it adds information about the found taint-flow by the analysis, such as the source and sink locations. The final sentence is provided to SECUCHECK-Magpie which maps the message to the corresponding findings. An example is shown in the yellow message box (14) in Figure 6.7.

6.7 Evaluation

We evaluated the usability of *fluentTQL* by conducting a comparative user study between *fluentTQL* and CODEQL. We chose CODEQL because it is part of LGTM, a state-of-the-art security tool, which has, in our perspective, very good tool support, and the query specifications are open-source. There is also an Eclipse plugin, a web console for queries, and integration with GitHub, a popular versioning system among developers. Additionally, we evaluated the applicability of *fluentTQL* by specifying queries for a different set of applications: a catalog of eleven Java programs, each demonstrating different security vulnerability, the deliberately insecure application OWASP WebGoat aims to teach developers about relevant security vulnerabilities, an insecure version of the Spring Demo application PetClinic, and randomly selected five real-world Android apps with known malicious taint-flows part of TaintBench[LPP⁺21]. All selected applications have known expected taint-flows that can be used to evaluate how the analysis performs in finding real vulnerabilities. We answer the following research questions:

- **RQ15** How usable is *fluentTQL* for software developers?
- **RQ16** How does *fluentTQL* compare to CODEQL for specifying taint-flow queries for taint-style security vulnerabilities?
- **RQ17** Are *fluentTQL* syntax elements sufficient to express queries for popular taint-style security vulnerabilities?
- **RQ18** Can *fluentTQL* express and detect the known security vulnerabilities Java/Android applications?

To answer the research questions, we use corresponding metrics. For **RQ15**, we use the System Usability Scale and Net Promoter Score. The same metrics are also used in **RQ16** to compare both DSLs. Additionally, we measure the time needed for the participants to complete the given tasks. We count only the solutions which are complete queries. The partial solutions are not counted due to the nature of the task. In a similar realistic scenario, incomplete queries will not return results from the tools. For **RQ17**, we evaluate how each *fluentTQL* construct contributes in specifying the most popular Java security vulnerabilities. Moreover, we identify security vulnerabilities for which *fluentTQL* can not express the required constructs. Finally, for **RQ18**, we count how many expected taint-flows in the selected applications are found when *fluentTQL* runs with adequate queries.

The following subsection explains our methodology for the user study used to answer **RQ15** and **RQ16**. The following subsections discuss the results of each research question individually. Finally, we discuss threats to validity.

6.7.1 Study Design

Setup: The user study was conducted over teleconferences where each participant shared the screen. Each study took, on average, 80 minutes. The session was recorded for post-processing purposes. We invited 35 software developers to participate in the study, of which 26 accepted the

invitation, referred to as P01-P26. We invited professional developers via our contacts from the industry as well as researchers and master-level students. Additionally, we asked three students to participate in a test session, which helped us to estimate the time and adjust the difficulty of the tasks.

Due to the limited number of participants, we chose a within-subjects design. Hence, each participant worked in Eclipse with available tool support for both DSLs. The *fluentTQL* implementation used the more versatile instantiation based on Boomerang. To avoid bias, we referred to the DSLs as DSL-1 and DSL-2. Initially, the participants received a project with all files needed for the practical part. Then, the moderator introduced taint analysis and showed a Java code example with an SQL injection vulnerability [Mit21c] to ensure that the participant understands the required concepts such as source, sanitizer, required propagator, and sink. Then, the exercises for DSL-1 and DSL-2 followed.

Each exercise consisted of a tutorial and a task. The tutorial for each DSL was based on the SQL injection vulnerability. Then, the participants had ten minutes to write a specification in the same DSL for a new vulnerability explained by the moderator. We chose the vulnerability types open redirect [Mit21d] for *fluentTQL* and cross-site scripting [Mit21b] for CODEQL. We selected an example with the same pattern in the form of source-sanitizer-sink for either type. This ensures that writing a specification for each vulnerability is equally hard, i.e., the effort is the same regardless of the vulnerability.

We provided a Java code example for each vulnerability type as a reference. The participant was allowed to use any of the files provided that included the Java classes and the files with example specifications of *fluentTQL* and CODEQL. For each task, we additionally provided a file with a skeleton code in which the participant wrote the solution. During the tasks, the participants were allowed to ask questions for clarification.

After the tasks, we let the participants fill out a web form. The moderator guided the participant in the discussion and collected the data for the questionnaire.

Questionnaire: In total, the questionnaire asked 28 questions, of which two were of open type and optional (Q26 and Q27). The complete list of questions is part of our artifact. Each of the questions asks for feedback for each DSL by the participant. Of the 26 mandatory questions of closed type, four are informational, 20 are related to the System-Usability-Scale (SUS) [Bro13], and two are related to the Net Promoter Score (NPS) [Rei03]. The SUS value is a usability metric derived from ten simple questions in a predefined format. The SUS-related questions (Q4-Q23) are the same ten questions per DSL with answering options on an agreement scale from one to five. SUS expresses the usability of a single DSL. Hence, for comparison, we use the same questions for each DSL. The NPS metric expresses how likely the participant would recommend something to a colleague. NPS identifies so-called promoters and detractors among the participants to calculate a value. The NPS-related questions (Q24-Q25) ask for the likelihood of DSL1 being recommended over DSL2 for specifying taint-flow queries and vice versa. The informational questions ask about participant coding experience (Q1), security expertise (Q2), willingness to learn new DSLs (Q3), and preferred way of learning new languages (Q28).

Participants: The study population of 26 participants is larger than the size of related studies that have been performed earlier, e.g., 10 in [SJM⁺19], 12 in [SNQDMH20], and 22 in [NB20]. We chose participants with diverse backgrounds. Ten of them are professional developers, six are computer science students on the master-level, and ten are researchers in computer science. The participants have different experiences in programming. Twelve of the participants have 10+, nine have 6-10, four have 3-5, and one has 1-2 years of programming experience. They rated their experience with security vulnerabilities. Three consider themselves beginners, 16 have

basic knowledge, five regularly inform themselves about the topic, and two consider themselves as experts.

6.7.2 Usability (RQ15)

fluentTQL was positively received by the participants of our user study. It received an excellent System Usability Score of 80,77 (with $p=0,0094 < 0,05$) on a scale from 0 to 100, where 68 is considered average usability and 100 is imaginary perfect. For the given task, 20 out of 26 participants finished with a correct solution in 10 minutes (on average 472 seconds, with $\sigma=99,05$). Table 6.4 shows the exact time in seconds for each participant. In the open questions (Q26-Q27), many participants gave additional feedback on what they liked and what they would improve in *fluentTQL*. Many participants said that they could learn the language quickly; one of them said "*with simple tutorial, I can learn it (fluentTQL) even without an expert. (...) it was very intuitive*" and other said, "*I didn't have to learn a lot*". Few participants mentioned that they like that the queries are compact and have the right level of abstraction.

Table 6.4: List of participants: coding experience and position, time in solving each task and DSL used in the first task (X means the participant did not task in 10 minutes)

	Coding (years)	Position	Security experience	<i>fluentTQL</i> (seconds)	CODEQL (seconds)	1st DSL
P01	3-5	developer	basic	554	X	<i>fluentTQL</i>
P02	>10	developer	basic	499	X	<i>fluentTQL</i>
P03	6-10	student	expert	482	588	<i>fluentTQL</i>
P04	>10	researcher	basic	560	590	CODEQL
P05	>10	researcher	basic	X	591	<i>fluentTQL</i>
P06	>10	researcher	advanced	544	562	<i>fluentTQL</i>
P07	3-5	researcher	basic	X	595	<i>fluentTQL</i>
P08	>10	student	advanced	449	495	CODEQL
P09	6-10	student	basic	X	587	CODEQL
P10	>10	researcher	basic	545	567	CODEQL
P11	1-2	researcher	beginner	558	585	CODEQL
P12	6-10	researcher	basic	X	X	<i>fluentTQL</i>
P13	3-5	researcher	beginner	473	541	CODEQL
P14	6-10	researcher	basic	305	434	CODEQL
P15	6-10	researcher	basic	571	X	<i>fluentTQL</i>
P16	6-10	student	beginner	412	558	CODEQL
P17	>10	developer	basic	X	X	<i>fluentTQL</i>
P18	>10	developer	basic	328	600	CODEQL
P19	6-10	developer	basic	594	X	<i>fluentTQL</i>
P20	6-10	developer	expert	375	492	CODEQL
P21	6-10	student	basic	455	467	CODEQL
P22	>10	developer	advanced	X	X	CODEQL
P23	>10	developer	advanced	507	600	<i>fluentTQL</i>
P24	>10	developer	advanced	206	425	CODEQL
P25	3-5	student	basic	531	X	<i>fluentTQL</i>
P26	>10	developer	basic	492	X	<i>fluentTQL</i>

We noted a few points that many participants disliked. Most dislike that the method signatures are specified as a string value. One participant said "*method calls are prone to typos or cumbersome to create*". We have already added a check in the editor to inform the users if their string is an invalid method signature. We support Java and Soot signatures. We even plan to add suggestions for existing methods from the workspace to the code completion feature of

the editor. Some participants gave suggestions for improving the names of some keywords. For example, the class *ThisObject*, which in *fluent*TQL is called with *thisObject()*, was earlier called *This* and confused many participants with the *this* keyword in Java.

*fluent*TQL as a new DSL is found to be very usable. The participants of our user study gave a score of 80,77 on the System Usability Score system.

6.7.3 Comparison (RQ16)

In terms of usability, with a SUS value of 38,56 CODEQL is perceived with bad usability (with $p=8,37e-36 < 0,05$). On the questions of how likely will the participant recommend one DSL over the other, for the task they were given (Q24-Q25), *fluent*TQL over CODEQL has a Net Promoter Score value of 30,77, whereas CODEQL over *fluent*TQL has a value of -86,96, where on the scale from -100 to 100, positive values are considered good. It follows that for specifying taint-flow queries, participants would more likely recommend *fluent*TQL over CODEQL.

```

50         class XSSConfig extends TaintTracking2::Configuration {
51             XSSConfig() { this = "XSSConfig" }
52             override predicate isSource(DataFlow::Node source) { source
                    instanceof RemoteFlowSource }
53             override predicate isSink(DataFlow::Node sink) { sink
                    instanceof XssSink }
54             override predicate isSanitizer(DataFlow::Node node) {
55                 node.getType() instanceof NumericType or
                    node.getType() instanceof BooleanType}
56         }
57         from DataFlow2::PathNode source, DataFlow2::PathNode sink,
            XSSConfig conf
58         where conf.hasFlowPath(source, sink)
59         select sink.getNode(), source, sink, "Cross-site scripting due to
            $$.", source.getNode(), "user-provided value"
60     }
```

Listing 6.4: CODEQL specification for XSS

To compare both languages, let us consider the CODEQL example for XSS in Listing 6.4. This is a solution for the task given to the participants. The query (lines 57- 59) consists of three sections, *from*, *where*, and *select*. In the *from* section, the user defines objects from predefined or self-defined classes. In the *where* section, constraints are defined that may also contain calls to predicates. In the *select* section, the results of the query are defined. For taint analysis, CODEQL provides a module. The class *XSSConfig* extends from the configuration class for taint analysis, where the sources, sanitizers, and sinks are defined. Additionally, the classes *RemoteFlowSource* and *XssSink* are provided and can be used to detect sources and sinks for XSS. The stub code with relevant imports given to each participant contained information that these classes exist and can be used. A user who needs other SRM that the provided classes cannot detect will need to write a new implementation. Note that the provided classes *RemoteFlowSource* and *XssSink* will match more sources and sinks than the *fluent*TQL query solution. To have an equivalent query in *fluent*TQL, the participants would have to write additional code for the *isSource* (Line 52) and *isSink* (Line 53) methods instead of using the provided classes.

Few participants mentioned the amount of code they would need to write in CODEQL is large. One participant said, "*...way too much code to get to the actual thing that needs to be written.*".

Furthermore, we observed how each participant performed in solving the tasks. The task with CODEQL was solved by 17 participants, compared to 20 with *fluent*TQL. Fourteen participants

solved both tasks. We measured the time each participant needed for each task, which is given in Table 6.4. On average participants solved the task with CODEQL in 546 seconds ($\sigma = 57,89$), which is by 13,4% slower than with *fluentTQL* with $p=0,01$ ($p < 0,05$). Concerning the time and the other four independent variables, i.e., *1st DSL*, *Coding*, *Position*, and *Security experience*, the p-value of the chi-square tests is much smaller than the threshold of 0,05. An exception is only the variable *Position* for the tasks with CODEQL with $p=0,097$ and a minimal effect (Cramer’s V is 0,043), where the researchers are slightly slower than the developers and the students.

Finally, we look into the outcome of each task. Using the ANOVA [?] test, we find out that there is no difference between the means of the outcomes of the first task and the second task ($p=0,131 > 0,05$).

While CODEQL is a more expressive DSL for multiple types of static analyses, *fluentTQL* is more preferred among software developers due to its user-friendliness. CODEQL scored a bad SUS value of 38,56. On the NPS system, *fluentTQL* is preferred over CODEQL with a score of 30,77, whereas CODEQL is preferred over *fluentTQL* with a negative value of 86,96. When specifying a taint-flow for given known vulnerability, the participants in our user study were 13,4% faster when using *fluentTQL* compared to CODEQL.

6.7.4 Expressiveness (RQ17)

To evaluate whether *fluentTQL* syntax elements are sufficient to express popular Java taint-style vulnerabilities, we created a catalog with Java code examples accompanied by *fluentTQL* specifications. The catalog contains eleven types of security vulnerabilities (Table 6.5). Each Java code example has a variant with and without sanitization. The catalog demonstrates different language syntax elements of *fluentTQL* and how they can be used for specifying vulnerabilities. The Java examples and the SRM are manually collected from several sources, including the Mitre and OWASP [OWA21] databases, OWASP benchmark project [Ben21], and other publicly available SRM lists [ARB13, Bro13, PDB19, TSBB17].

Table 6.5: List of vulnerability types implemented in the *fluentTQL* catalog (so - sources, sa - sanitizers, rp - required propagators, si - sinks)

Vulnerability type	flows	so	sa	rp	si	SRM
SQL injection [Mit21c]	3	13	3	6	10	32
XPath [Mit20a]	1	12	1	0	12	25
Command injection [Mit20c]	1	12	1	1	1	15
XML injection [Mit20h]	1	12	1	0	4	17
LDAP injection [Mit20d]	1	12	1	0	8	21
Cross-site scripting [Mit21b]	2	13	1	1	3	18
Open redirect [Mit21d]	2	13	1	0	2	16
NoSQL injection [Mit20b]	2	5	2	3	2	12
Trust boundary violation [Mit20g]	1	12	1	0	1	15
Path traversal [Mit20f]	2	12	1	1	2	16
Log injection [Mit20e]	2	12	1	1	4	18
Total (unique):	18	46	14	13	49	122

Many taint-style vulnerabilities from the Mitre and OWASP databases can be modeled with single taint flow queries. Nevertheless, we found examples such as the *noSQLi2* query in Listing 6.3, where the `and()` operator is needed.

Taint flows that require multiple intermediate source-sinks steps were necessary for the specification of many taint flows, i.e., the feature of multi-step taint analysis is ubiquitous. For example, the OWASP Benchmark test 00001⁴ contains Path Traversal vulnerability [Mit20f]. A *File* object is constructed using a *String* parameter as the file’s location. If the *String* is user-controllable, i.e., tainted, and the *File* object is passed to a *FileInputStream* constructor, a path traversal vulnerability occurs. The file constructor, in this case, is a required propagator that ensures the order of SRM calls.

When it comes to the SRM specifications, we observed that most of the sources have a *Return* object as an out-value. For sinks, most of the in-values are *Parameter* objects.

We also inspected the vulnerability types (Common Weakness Enumerations - CWEs) in the SANS-25 list [Mit21a]. 17 of 25 vulnerability types can be expressed as taint-style. 13 of those can be modeled in *fluentTQL*.

The remaining four CWEs are CWE-119, CWE-787, CWE-476, and CWE-798. CWE-119 and CWE-787 are related to buffer overflows, which do not apply to Java. The CWE-476 cannot be expressed because the potential sources are **new**-expressions, which cannot currently be modeled. Also, constant values cannot currently be modeled as potential sources, which is needed for CWE-798 where these values should be detected as hard-coded credentials. Extending *fluentTQL* to support **new**-expressions and constant values is possible in the abstract syntax by modeling them with a new class that extends the class *FlowParticipant*. The semantics need to be extended to define how these values will be detected and the appropriate concrete syntax.

Even though our implementation of *fluentTQL* is bound to Java only, *fluentTQL* can also express taint-style vulnerabilities in other languages. The only requirement to specify a query for other languages is that the sources, sanitizers, and sinks are defined as method calls. Like Java, *fluentTQL* can be adapted to work for C/C++, C#, other JVM-hosted languages and cover many taint-style vulnerabilities. In languages such as JavaScript, the coverage of vulnerabilities is smaller since the sources and sinks are often not method calls.

fluentTQL can express all taint-style vulnerabilities in which the key constructs of the taint-flows are method calls. Our implementation shows that at least 11 types of security vulnerabilities can be specified with *fluentTQL*. These are the most popular security vulnerabilities for Java. Theoretically, one can express many more.

6.7.5 Analyzing Java/Android Applications (RQ18)

To answer **RQ18**, we ran *fluentTQL* queries on two Java applications, OWASP WebGoat and PetClinic, and seven Android applications from TaintBench [LPP⁺21].

The OWASP WebGoat is a deliberately insecure application that teaches developers about relevant security vulnerabilities. As a Java Spring application⁵, it is popular in the community and has been used for evaluating static analyses [AFK⁺20]. We used this application to evaluate the applicability of *fluentTQL* in a real-world scenario, including specifying taint-flow queries and running our Boomerang-based and FlowDroid-based taint analysis.

We chose to work with the SQL injection as an example since it has the most taint-flows in WebGoat. We documented all 17 SQL injection taint-flows in OWASP WebGoat and used them as ground truth. This was manually done by following the directions of the lessons present in WebGoat and inspect the source code.

⁴<https://github.com/OWASP/Benchmark/blob/master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00001.java>

⁵<https://spring.io/>

Next, we specified the sensitive methods, which include 17 sources, one sanitizer, two required propagators, and two sinks. We only needed to create two taint-flow queries to be able to cover all types of taint-flows. The Boomerang-based implementation was able to detect all 17 taint-flows. The official FlowDroid implementation (we used version 2.8) could not find any taint-flow in WebGoat. So, we investigated and found that FlowDroid defines only the return values of the sources as taints. For all taint-flows in WebGoat, the taints are the parameters of the sources. Hence, we adapted FlowDroid to support this, and after doing so, the FlowDroid implementation detected 13 taint-flows. Those that were missed are the types that contain a required propagator, which is currently not supported by FlowDroid.

For the second Java application, PetClinic, we followed the same steps as for OWASP WebGoat. We identified and documented five taint-flows of type hibernate injection and two taint-flows of type cross-site request forgery. In this application, all taint-flows were detected by our implementation with Boomerang and our updated version of FlowDroid. Table 6.6 shows summary of the Java applications.

TaintBench is a collection of real-world Android apps that contain malicious behavior in the form of taint-flows. These apps have well-documented information about the expected taint-flows and should help analyses writers evaluate their tools rigorously and fairly. Table 6.7 summarizes the findings of running our *fluentTQL* implementation with Boomerang as well as with FlowDroid. Out of 25 expected taint-flows among all apps, the Boomerang-based implementation found 18, whereas FlowDroid-based implementation found 13. We manually inspected those that were not found and identified two causes which are due to the existing solvers and not the inability of *fluentTQL* to express them. The first cause is the inability to analyze taint-flows through different threads in the code. Due to the implicit data-flow behavior of the threads, the existing call graph algorithms have limitations in modeling this correctly. This second cause is that the existing data-flow analyses do not analyze the expressions within path constraints. In the case of our experiments, we found that the call of the source method is within the condition of an IF statement, which is not analyzed by Boomerang nor by FlowDroid.

The runtime values reported in Table 6.6 and Table 6.7 are the average values over ten runs on a system with Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz, 16 GB RAM with Win-10 OS.

```

61         private void loadClass(Context context) {
62             ...
63             try {
64                 InputStream is = getAssets().open("ds");
65                 int len = is.available();
66                 byte[] encrypedData = new byte[len];
67                 is.read(encrypedData, 0, len);
68                 byte[] rawdata = new DesUtils(
69                     DesUtils.STRING_KEY).decrypt(encrypedData);
70                 FileOutputStream fos = new
71                     FileOutputStream(sourcePathName);
72                 fos.write(rawdata);
73                 fos.close();
74             } catch (Exception e) {
75                 e.printStackTrace();
76             }
77             try {
78                 Object[] argsObj = new Object[]{sourcePathName,
79                     outputPathName, Integer.valueOf(0)};
80                 DexFile dx = (DexFile) Class.forName(
81                     "dalvik.system.DexFile").getMethod("loadDex",
82                     new Class[]{String.class, String.class,
83                         Integer.TYPE}).invoke(null, argsObj);
84                 ...
85             } catch (Exception e2) {

```

```

80         }
81     }

```

Listing 6.5: Malicious taint-flow through a file in the *dsencrypt* app from TaintBench

Detecting taint-flows through files. The code in Listing 6.5 shows the *loadClass* method from the app *dsencrypt*⁶, which contains the malicious taint-flow. It reads an encrypted zip file from the asset folder (source in Line 64), decrypts it, and extracts class.dex which contains malicious code (intermediate statements in the trace are lines 67, 68, and 69). The malicious code is called via reflection (sink in Line 77). As reported in work by Luo et al.[LPP⁺21], these kinds of taint-flows going through files, databases, etc., can not be detected by the existing Android taint analysis tools. However, with *fluent*TQL, we can now model and detect these taint-flows using the *and()* operator.

Table 6.6: Overview of the evaluated Java projects. Flows/B/F is number of expected taint-flows (vulnerability instances) and those found by Boomerang and FlowDroid, CWE is number of common weakness enumerations (vulnerability types), Runtime is average over ten runs.

Project	#Classes	#Flows/B/F	#CWE	#Runtime(s) B/F
Catalog	36	27/27/25	11	52.8/43.7
PetClinic	42	4/4/4	1	10.9/14.4
WebGoat	35	17/17/13	1	30.3/36.7

Table 6.7: Overview of the evaluated Android apps from TaintBench. Flows/B/F is number of expected taint-flows (vulnerability instances) and those found by Boomerang and FlowDroid, Runtime is average over ten runs.

App	#Classes	#Flows/B/F	#Runtime(s) B/F
blackfish	338	13/11/11	18.6/29.8
beita_com_beita_contact	379	3/1/1	11.2/25.5
phospy	236	2/2/0	8.6/11.5
repane	5	1/1/0	3/4.8
dsencrypt	4	1/1/1	10.2/4.9
fakeappstore	402	3/2/0	23/16.5
fakemart	868	2/0/0	27.3/34.9

Our Boomerang-based implementation of *fluent*TQL can detect all expected taint-flows in the Java Spring applications: OWASP WebGoat and the PetClinic. Among seven real-world Android apps with malicious taint-flows, *fluent*TQL can detect 18 out of 25 expected taint-flows. Those that can not be detected are complex threads modeling and not considering path conditions.

6.7.6 Threats to Validity

External validity. Participation in the study was voluntary. We asked our contacts in industry to invite their software developers. The invitation mentioned that the study would try

⁶https://github.com/TaintBench/dsencrypt_samp/blob/master/src/main/java/com/kbstar/kb/android/star/ProxyApp.java

to compare two domain-specific languages for static analysis. This information makes it more likely that the participants are interested in the design of programming languages and/or static analysis. Hence, there is a threat of having a subject not representative of the entire population of software developers.

Internal validity. Apart from professional software developers, we invited researchers and master-level students from the university. Previous work has shown that graduate students are valid proxies for software developers in such studies [NDGS20, NDTS18, NDG⁺19, NDT⁺17]. Also, our results confirm that there is no significant correlation between the position of the participant and the performance in the task, thus also confirming that—for such studies—researchers and master-level students have coding knowledge comparable to professional developers.

Moreover, the format of within-subjects study design has its limitations. For example, as both tasks were the same, but for a different DSL and context (vulnerability example), participants may have been influenced by the first task, known as carryover effects when solving the second task. To deal with this, we applied randomization of the order of the tasks.

Construct validity. Another threat to validity is the fairness of the tasks. Both DSLs are not equally expressible. This means one may need more or less time to learn a new DSL. To address this, we took into consideration the following points. First, we used vulnerabilities that have the same taint-flow pattern. The Java code shown as an example for each task had the same complexity. Second, for each task, we provided a stub code for the solution. In the case of CODEQL, which is a more expressible DSL than *fluent*TQL and has support for taint analysis and other analyses, too, we asked the participants to focus only on the taint analysis module. Finally, we had three test sessions to adjust the tasks and define what exact information the participants will need to solve each task in under ten minutes. Similarly, a possible threat to validity comes from the design of our study to use the open redirect vulnerability with *fluent*TQL and XSS for CODEQL for all participants without switching among half of the participants. To mitigate the threat, we have selected the code examples used in the tasks to have the same structure, i.e., the taint was in both cases created by a call to an HTTP request object, and only the sink method differs for each vulnerability. Additionally, while explaining each task, we also explained the vulnerability. While the participants were performing the task, we encouraged them to share their thoughts verbally. After processing the recorded material, we found that none of the participants struggled with understanding the vulnerability.

Conclusion

The application of static analysis in detecting security vulnerabilities has been used in many cases in the research community. These results have initiated the development of open-source as well as commercial tools. With this development, the branch of static application security testing (SAST) tools has evolved from the pool of static analysis tools. However, designing a tool that is sound, precise, scalable, and at the same time usable for the intended user is a challenging task. Each of these properties can be addressed in a specific way through customizations of the analysis in the given context, which requires highly skilled experts in multiple domains, i.e., static analysis, security, and software engineering.

The scope of this thesis was limited to a specific data-flow analysis technique, namely, taint analysis. This type of analysis is most versatile in detecting different security vulnerabilities. It can detect many popular security vulnerabilities such as SQL injection, cross-site scripting, log injection, etc. Taint analysis is the core analysis of many SAST tools on the market, such as CHECKMARX, and FORTIFY, and proper adaptation to the context can be very useful. As primary users, we targeted software developers. We studied how to improve the adaptation of taint analysis tools within their workflow at development time, with the intention to early detect security vulnerabilities before they land on the central repository or even work in production.

Initially, as the first contribution of this thesis, we performed empirical studies to uncover the current state of the usage of SAST tools in practice. We conducted a study among German companies of different sizes to find that about half of the participants do not use SAST tools. The study confirmed the results of a few earlier studies that false warnings remain a relevant issue for users. The participants reported that the configuration of the tools is complex, but most are willing to provide feedback to the tool and improve the findings. Contrary to earlier studies that reported that static analyses are too slow, the participants in our study perceive that the tools they have used are fast enough to be used in their development workflow. Moreover, we performed a user study to investigate how effective users can resolve the findings of a given taint analysis tool. The study showed that the participants who adapted the configuration were more effective in resolving the findings than those who only used the default configuration.

The findings of our studies motivated the need to improve how the users adapt taint analysis tools for detecting security vulnerabilities. We addressed two challenges: (1) *how to automate the extraction of security-relevant methods (SRM) from arbitrary codebases* and (2) *how to improve the usability of the configuration of the taint analysis tools for software developers*.

The second contribution, SWAN, enables the users to use machine-learning algorithms to extract SRM from their codebase, which their taint analysis can use. This is a fully automatic approach. Compared to earlier approaches, SWAN supports additional SRM types such as

validators, and it is the first approach that further classifies the SRM into security vulnerability categories. The results show high precision and recall, but the accuracy still leaves room for improvement in some instances. The third contribution, SWAN_{ASSIST}, improves the results of SWAN by adding a manual step where the user provides feedback to the classifier used in SWAN. This is an active machine-learning approach. We additionally developed a strategy that proposes the most impactful data to the user that needs to be manually labeled. Our experiments show that the classifier can adapt fast to the codebase and make better predictions with low manual effort.

The last contribution addresses the usability of the configuration of taint analysis. We propose *fluentTQL*, a new domain-specific language (DSL) that can express taint-style vulnerabilities. It is the first developer-centric DSL. Compared to a state-of-the-art DSL, CODEQL, based on our usability user study, *fluentTQL* is more usable. Moreover, the semantics of the language is independent of the underlying data-flow analysis, and hence we show that *fluentTQL* can be instantiated to different data-flow analysis solvers. Along with *fluentTQL*, we developed the open-source taint analysis tool, SECUCHECK, as a plug-in that can run in multiple IDEs.

While the contributions of this thesis have moved forward the state-of-the-art adapting taint analysis tools by their users, there are several areas for future exploration. We list possible research directions for the future:

Regular studies. The studies we conducted indicated the current state of use of the tools, mainly in one given region, Germany. However, similar studies may be conducted in other regions as well. Moreover, our studies confirmed some results of previous studies but also showed that some previous results are now changing. This shows that the area of SAST tools is evolving, and repeating these studies in the future can give different results.

Studies for evaluating feedback and configuration features of SAST tools. *To what extent can the users provide feedback to the tools or configure them, and how effective is this in detecting security vulnerabilities?* This question should be explored in more depth. User-controlled experiments can be used to answer this. Some tools already provide features for providing feedback, e.g., CHECKMARX enables the user to label the findings as false positives.

Extracting more types of SRM. With SWAN, we have shown that validators, such as sanitizers and authentication methods, can be extracted through classification. To cover more vulnerabilities types, such as required propagators, cryptographic misuses, or other types of API misuses, we need other types of SRM. Further, in SWAN, we showed that the SRM could be classified into seven CWEs. This certainly can be extended to more CWEs. However, it is unknown how well these classifiers work and the challenges of implementing them. Whether the code and doc comments information is sufficient for particular CWEs is yet to be researched.

Extracting SRM with more information. The existing machine-learning approaches such as SWAN, classify the methods into SRM as a whole. However, as seen in *fluentTQL*, the specification of taint-flows requires further information about the taints, which is the relevant part of the method call. In taint analysis, for example, we need to know the actual taint created by the source method, such as the return value or the object on which the source method is called. Currently, this information still needs to be specified manually. FlowDroid [ARF⁺14], for example, assumes that sources always taint the return value, and sinks should report a finding if any of the parameters is tainted. However, this is insufficient as it can produce false results for many vulnerability types. Hence, one can explore whether the relevant party of the SRM can also be learned automatically.

Extracting complete taint-flows. *fluentTQL* is designed with reusability in mind. For one team that regularly uses a given set of libraries or frameworks, many taint-flow queries can be reused in multiple projects, and only minor adaptations are needed. However, for a new codebase with many unknown libraries, creating the taint-flow queries can still be a tedious task. A future research topic is to explore whether we can apply machine-learning to learn the complete taint-flows based on previously known examples.

Usable Typestate DSLs. *fluentTQL* showed that in terms of usability, it outperforms the state-of-the-art DSL CODEQL. However, the expressiveness of *fluentTQL* compared to CODEQL is limited. In the future, we plan to extend *fluentTQL* with new constructs that can support other types of analyses, such as typestate analysis or value analysis to express even more types of vulnerabilities. Nevertheless, it is unknown if these extensions will impact its usability. Therefore, one may also consider whether a new, more expressive DSL for software developers is needed or only extensions of *fluentTQL* are sufficient.

Bibliography

- [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 *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 794–807, New York, NY, USA, 2020. Association for Computing Machinery.
- [AFP20] Philippe Arteau, David Formánek, and Tomáš Polešovský. Find-sec-bugs resources. <https://github.com/find-sec-bugs/find-sec-bugs/tree/master/findsecbugs-plugin/src/main/resources/injection-sinks>, August 2020.
- [Ang88] Dana Angluin. Queries and concept learning. *Mach. Learn.*, 2(4):319–342, apr 1988.
- [Aaaa] Apache. Abdera. <https://abdera.apache.org/>.
- [Apab] Apache. Apache commons. <https://commons.apache.org/>.
- [Apac] Apache. Apache cordova. <https://cordova.apache.org/>.
- [Apad] Apache. Apache lucene. <http://lucene.apache.org/>.
- [Apae] Apache. Apache stratos. <http://stratos.apache.org/>.
- [Apaf] Apache. Apache struts. <https://struts.apache.org/>.
- [Apag] Apache. Roller. <http://roller.apache.org/>.
- [Apah] Apache. Tomcat. <http://tomcat.apache.org/>.
- [ARB13] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Susi: A tool for the fully automated classification and categorization of android sources and sinks. In *Network and Distributed System Security Symposium 2013*, NDSS’13, 2013.
- [ARB17] S. Arzt, S. Rasthofer, and E. Bodden. The Soot-based toolchain for analyzing android apps. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, MOBILESoft ’17, pages 13–24, Piscataway, NJ, USA, 2017. IEEE Press.
- [ARF⁺14] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Outeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of*

the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

- [Ben21] OWASP Benchmark. Owasp. <https://owasp.org/www-project-benchmark/>, 2021. Online; accessed January 2021.
- [BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
- [BKM09] Aaron Bangor, Philip Kortum, and James Miller. Determining what individual sus scores mean: Adding an adjective rating scale. *J. Usability Studies*, 4(3):114–123, May 2009.
- [BLH⁺20] Pan Bian, Bin Liang, Jianjun Huang, Wenchang Shi, Xidong Wang, and Jian Zhang. *SinkFinder: Harvesting Hundreds of Unknown Interesting Function Pairs with Just One Seed*, page 1101–1113. Association for Computing Machinery, New York, NY, USA, 2020.
- [Bod18] Eric Bodden. The secret sauce in efficient and precise static analysis: The beauty of distributive, summary-based static analyses (and how to master them). In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ISSTA '18, page 85–93, New York, NY, USA, 2018. Association for Computing Machinery.
- [Bro13] John Brooke. Sus: A retrospective. *J. Usability Studies*, 8(2):29–40, February 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, pages 332–343, New York, NY, USA, 2016. ACM.
- [CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fix-points. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.
- [Cen] Walden University Writing Center. Grammar: Prepositions. <https://academicguides.waldenu.edu/writingcenter/grammar/prepositions>. Accessed: 17.08.2020.
- [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.
- [Che20a] Checkmarx. Checkmarx. <https://www.checkmarx.com/>, 2020. Online; accessed January 2020.
- [Che20b] Checkmarx. Cxql, checkmarx query language. 2020. Online; accessed March 2020.
- [Cra16] Harald Cramér. *Mathematical Methods of Statistics (PMS-9)*. Princeton University Press, 2016.

- [DAL⁺17] L. Nguyen Quang Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill. Cheetah: Just-in-time taint analysis for android apps. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 39–42, Piscataway, NJ, USA, 2017. IEEE Press.
- [DDE⁺11] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE11, pages 681–690, New York, NY, USA, 2011. Association for Computing Machinery.
- [DNRN13] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE'13, pages 422–431, 2013.
- [Dro] Dropwizard. Dropwizard. <https://www.dropwizard.io/>.
- [Ecla] Eclipse. Jetty. <https://www.eclipse.org/jetty/>.
- [Eclb] Eclipse. Smarthome. <https://www.eclipse.org/smarthome/>.
- [EE] European Bioinformatics Institute (EMBL-EBI). Embl-ebi home page. <https://www.ebi.ac.uk/>. Online; accessed 10 December 2018.
- [Enga] Secure Software Engineering. Droidbench. <https://github.com/secure-software-engineering/DroidBench>.
- [Engb] Secure Software Engineering. Pointerbench. <https://github.com/secure-software-engineering/PointerBench>.
- [Fac21] Facebook. Infer. 2021. Online; accessed January 2021.
- [FADA14] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 576–587, New York, NY, USA, 2014. Association for Computing Machinery.
- [FW13] Z. P. Fry and Westley. Clustering static analysis defect reports to reduce maintenance costs. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 282–291, Oct 2013.
- [GdP⁺15] Michael Gordon, Kim deokhwan, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow analysis of android applications in droid-safe. In *Network and Distributed System Security Symposium 2015*, 01 2015.
- [Git21a] Semmler Inc GitHub. Code ql. <https://semmler.com/codeql>, 2021. Online; accessed May 2021.
- [Git21b] Semmler Inc. Github. Lgtm. <http://lgtm.com/>, 2021. Online; accessed May 2021.
- [Giv21] Lisa M. Given. Convenience sample. In *In The SAGE encyclopedia of qualitative research methods*, volume 1, pages 125–125. SAGE Publications, Inc., 2021.

- [Gmb] RIGS IT GmbH. Xanitizer. <https://www.xanitizer.com/xanitizer/>. Online; accessed January 2021.
- [Gooa] Google. Android api 4.2. <https://developer.android.com/about/versions/android-4.2>.
- [Goob] Google. Google auth java. <https://github.com/googleapis/google-auth-library-java>.
- [Gra21] Grammatech. Codesonar. <https://www.grammatech.com/products/codesonar>, 2021. Online; accessed January 2021.
- [GvTB18] Ivan Gotovchits, Rijnard van Tonder, and David Brumley. Saluki: finding taint-style vulnerabilities with static property checking. In *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2018.
- [GWT] GWT. Gwt. <http://www.gwtproject.org/>.
- [Har54] Zellig S. Harris. Distributional structure. *WORD*, 10(2-3):146–162, 1954.
- [Hel19] J. Hellrich. *Word Embeddings: Reliability & Semantic Change*. Dissertations in Artificial Intelligence. IOS Press, 2019.
- [HP18] Andrew Habib and Michael Pradel. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 317–328, New York, NY, USA, 2018. Association for Computing Machinery.
- [HW09] S. Heckman and L. Williams. A model building process for identifying actionable static analysis alerts. In *2009 International Conference on Software Testing Verification and Validation*, pages 161–170, April 2009.
- [Ins] European Bioinformatics Institute. Gene expression atlas. <https://github.com/gxa/gxa>.
- [Int] IntelliJ. IntelliJ platform ui guidelines. <https://jetbrains.github.io/ui/>.
- [Jav] Spark Java. Spark. <http://sparkjava.com/>.
- [Jet] JetBrains. IntelliJ home page. <https://www.jetbrains.com/idea/>. Online; accessed 17 October 2018.
- [JGu] JGuard. Jguard. <http://jguard.net/>.
- [JSMHB13] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don’t software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE ’13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press.
- [jsou] jsoup. jsoup. <https://jsoup.org/>.
- [JWMC15] Andrew Johnson, Lucas Wayne, Scott Moore, and Stephen Chong. Exploring and enforcing security guarantees via program dependence graphs. *SIGPLAN Not.*, 50(6):291–302, June 2015.

- [Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, page 194–206, New York, NY, USA, 1973. Association for Computing Machinery.
- [Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, jul 1976.
- [KP08] Barbara Kitchenham and Shari Pfleeger. *Personal Opinion Surveys*, pages 63–92. 01 2008.
- [KSA⁺18] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 10:1–10:27, 2018.
- [KSA⁺19] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.
- [KTH⁺19] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. *Auto-WEKA: Automatic Model Selection and Hyperparameter Optimization in WEKA*, pages 81–95. Springer International Publishing, Cham, 2019.
- [KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7(3):305–317, sep 1977.
- [KV15] James Max Kanter and Kalyan Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. In *2015 IEEE International Conference on Data Science and Advanced Analytics, DSAA 2015, Paris, France, October 19-21, 2015*, pages 1–10. IEEE, 2015.
- [LBLH11] Patrick Lam, Eric Bodden, Ondrej Lhotak, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, Oktober 2011.
- [LBS19] L. Luo, E. Bodden, and J. Späth. A qualitative analysis of android taint-analysis results. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 102–114, 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)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:25, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [Liv12] Benjamin Livshits. Dynamic taint tracking in managed runtimes. Technical report, Microsoft Research, 2012.
- [LK77] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

- [LLJB12] Lucia, D. Lo, L. Jiang, and A. Budi. Active refinement of clone anomaly reports. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 397–407, June 2012.
- [LLK⁺17] Woosuk Lee, Wonchan Lee, Dongok Kang, Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Sound non-statistical clustering of static analysis alarms. *ACM Trans. Program. Lang. Syst.*, 39(4), August 2017.
- [LM14] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. *CoRR*, abs/1405.4053, 2014.
- [LNRB09] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. *SIGPLAN Not.*, 44(6):75–86, June 2009.
- [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.
- [LS11] Wei Le and Mary Lou Soffa. Generating analyses for detecting faults in path segments. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA11*, pages 320–330, New York, NY, USA, 2011. Association for Computing Machinery.
- [LSS⁺15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. 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, jan 2015.
- [MAS⁺17] B. Mathis, V. Avdiienko, E. O. Soremekun, M. Böhme, and A. Zeller. Detecting information flow by mutating input data. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, pages 263–273, Piscataway, NJ, USA, 2017. IEEE Press.
- [MG18] A. Mendoza and G. Gu. Mobile application web api reconnaissance: Web-to-mobile inconsistencies amp; vulnerabilities. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 756–769, May 2018.
- [Mic20] Microfocus. Fortify. <https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview>, 2020. Online; accessed January 2020.
- [Mic21] Microsoft. Language server protocol. <https://microsoft.github.io/language-server-protocol/>, 2021. Online; accessed May 2021.
- [Min13] Daniel E Minshall. A computer science word list. *Unpublished MA dissertation, University of Swansea. Available at DE Minshall*, 2013.
- [Mit20a] Common Weakness Enumeration Mitre. Improper neutralization of data within xpath expressions. <https://cwe.mitre.org/data/definitions/643.html>, 2020. Online; accessed January 2021.
- [Mit20b] Common Weakness Enumeration Mitre. Improper neutralization of special elements in data query logic. <https://cwe.mitre.org/data/definitions/943.html>, 2020. Online; accessed January 2021.

- [Mit20c] Common Weakness Enumeration Mitre. Improper neutralization of special elements used in a command. <https://cwe.mitre.org/data/definitions/77.html>, 2020. Online; accessed January 2021.
- [Mit20d] Common Weakness Enumeration Mitre. Improper neutralization of special elements used in an ldap query. <https://cwe.mitre.org/data/definitions/90.html>, 2020. Online; accessed January 2021.
- [Mit20e] Common Weakness Enumeration Mitre. Improper output neutralization for logs. <https://cwe.mitre.org/data/definitions/117.html>, 2020. Online; accessed January 2021.
- [Mit20f] Common Weakness Enumeration Mitre. Relative path traversal. <https://cwe.mitre.org/data/definitions/23.html>, 2020. Online; accessed January 2021.
- [Mit20g] Common Weakness Enumeration Mitre. Trust boundary violation. <https://cwe.mitre.org/data/definitions/501.html>, 2020. Online; accessed January 2021.
- [Mit20h] Common Weakness Enumeration Mitre. Xml injection. <https://cwe.mitre.org/data/definitions/91.html>, 2020. Online; accessed January 2021.
- [Mit21a] Common Weakness Enumeration Mitre. 2011 cwe/sans top 25 most dangerous software errors. <http://cwe.mitre.org/top25/>, 2021. Online; accessed January 2021.
- [Mit21b] Common Weakness Enumeration Mitre. Improper neutralization of input during web page generation. <https://cwe.mitre.org/data/definitions/79.html>, 2021. Online; accessed January 2021.
- [Mit21c] Common Weakness Enumeration Mitre. Improper neutralization of special elements used in an sql command. <https://cwe.mitre.org/data/definitions/89.html>, 2021. Online; accessed January 2021.
- [Mit21d] Common Weakness Enumeration Mitre. Url redirection to untrusted site ('open redirect'). <https://cwe.mitre.org/data/definitions/601.html>, 2021. Online; accessed January 2021.
- [MLL05] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: A program query language. *SIGPLAN Not.*, 40(10):365–383, October 2005.
- [Mon21] Survey Monkey. Survey online tool. 2021. Online; accessed January 2021.
- [MW21] Austin Mordahl and Shiyi Wei. The impact of tool configuration spaces on the evaluation of configurable taint analysis for android. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 466–477, New York, NY, USA, 2021. Association for Computing Machinery.
- [MWH18] Felix Mohr, Marcel Wever, and Eyke Hüllermeier. Ml-plan: Automated machine learning via hierarchical planning. *Machine Learning*, 107(8-10):1495–1515, 2018.
- [NB20] L. Nguyen Quang Do and E. Bodden. Explaining static analysis with rule graphs. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

- [NDG⁺19] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, Emanuel von Zezschwitz, and Matthew Smith. If you want, i can store the encrypted password: A password-storage field study with freelance developers. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI 19, pages 1–12, New York, NY, USA, 2019. Association for Computing Machinery.
- [NDGS20] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, and Matthew Smith. On conducting security developer studies with cs students: Examining a password-storage study with cs students, freelancers, and company developers. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI 20, pages 1–13, New York, NY, USA, 2020. Association for Computing Machinery.
- [NDT⁺17] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. Why do developers get password storage wrong? a qualitative usability study. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS 17, pages 311–328, New York, NY, USA, 2017. Association for Computing Machinery.
- [NDTS18] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, and Matthew Smith. Deception task design in developer password studies: Exploring a student sample. In *Proceedings of the Fourteenth USENIX Conference on Usable Privacy and Security*, SOUPS 18, pages 297–313, USA, 2018. USENIX Association.
- [NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, Heidelberg, 1999.
- [NQDAL⁺17] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. Just-in-time static analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 307–317, New York, NY, USA, 2017. ACM.
- [NQDWA20] Lisa Nguyen Quang Do, James R. Wright, and Karim Ali. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. In *Proceedings of the Sixteenth Symposium on Usable Privacy and Security*, 2020.
- [oM21] University of Maryland. Findbugs. 2021. Online; accessed January 2021.
- [Ora] Oracle. Javaee. <https://www.oracle.com/java/technologies/java-ee-glance.html>.
- [OWA] OWASP. Webgoat. <https://github.com/WebGoat/WebGoat>.
- [OWA21] OWASP. Top 10 security vulnerabilities. <https://owasp.org/www-project-top-ten/>, 2021. Online; accessed May 2021.
- [PBW18] Felix Pauck, Eric Bodden, and Heike Wehrheim. Do android taint analysis tools keep their promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 331–341, New York, NY, USA, 2018. Association for Computing Machinery.

- [PDB19] Goran Piskachev, Nguyen Quang Do, and Eric Bodden. Codebase-adaptive detection of security-relevant methods. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 181–191, New York, NY, USA, 2019. Association for Computing Machinery.
- [PHR19] Nikhil Patnaik, Joseph Hallett, and Awais Rashid. Usability smells: An analysis of developers’ struggle with crypto libraries. In *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*, Santa Clara, CA, August 2019. USENIX Association.
- [PKB21] Goran Piskachev, Ranjith Krishnamurthy, and Eric Bodden. Secucheck: Engineering configurable taint analysis for software developers. In *21st IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2021)*, Luxembourg, September 2021. IEEE.
- [PPSB20] Goran Piskachev, Tobias Petrasch, Johannes Späth, and Eric Bodden. Authcheck: Program-state analysis for access-control vulnerabilities. In *Formal Methods. FM 2019 International Workshops*, pages 557–572, Cham, 2020. Springer International Publishing.
- [QWR18] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 176–186, New York, NY, USA, 2018. Association for Computing Machinery.
- [RBW0] Victoria Reyes, Elizabeth Bogumil, and Levin Elias Welch. The living codebook: Documenting the process of qualitative data analysis. *Sociological Methods & Research*, 0(0):0049124120986185, 0.
- [Rei03] Frederick F Reichheld. The one number you need to grow. *Harvard business review*, 81(12):46–55, 2003.
- [RHRR12] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley Publishing, 1st edition, 2012.
- [RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’95, page 49–61, New York, NY, USA, 1995. Association for Computing Machinery.
- [SAB10] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE Symposium on Security and Privacy*, pages 317–331, 2010.
- [SAB19] Johannes Späth, Karim Ali, and Eric Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, 3(POPL):48:1–48:29, January 2019.
- [SAE⁺18] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4):58–66, March 2018.

- [Sam14] L. Sampaio. Which methods should be considered “sources”, “sinks” or “sanitization”? <https://thecodemaster.net/methods-considered-sources-sinks-sanitization/>, August 2014. Accessed 05.03.2020.
- [SAP⁺11] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4f: Taint analysis of framework-based web applications. *SIG-PLAN Not.*, 46(10):1053–1068, October 2011.
- [SBF18] D. Sas, M. Bessi, and F. A. Fontana. [research paper] automatic detection of sources and sinks in arbitrary java libraries. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 103–112, Sept 2018.
- [Sea99] C.B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999.
- [SEV16] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. Inca: A dsl for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, pages 320–331, New York, NY, USA, 2016. Association for Computing Machinery.
- [SJM^H+15] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, pages 248–259, New York, NY, USA, 2015. ACM.
- [SJM^H+19] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. How developers diagnose potential security vulnerabilities with a static analysis tool. *IEEE Trans. Softw. Eng.*, 45(9):877–897, September 2019.
- [SNQDMH20] Justin Smith, Lisa Nguyen Quang Do, and Emerson Murphy-Hill. Why can’t johnny fix vulnerabilities: A usability evaluation of static analysis tools for security. In *Proceedings of the Sixteenth Symposium on Usable Privacy and Security, SOUPS 2020*, 2020.
- [Sny21] Snyk. Deepcode. 2021. Online; accessed January 2021.
- [Son21] SonarSource. Sonarqube. <https://www.sonarqube.org>, 2021. Online; accessed January 2021.
- [Spo] SpotBugs. Find security bugs. <https://find-sec-bugs.github.io/>. Online; accessed January 2021.
- [Spr] Java Spring. Java spring. <https://spring.io/>.
- [SRH96] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1–2):131–170, oct 1996.
- [SSA15] Janet Siegmund, Norbert Siegmund, and Sven Apel. Views on internal and external validity in empirical software engineering. In *2015 IEEE/ACM 37th*

- IEEE International Conference on Software Engineering*, volume 1, pages 9–19. IEEE, 2015.
- [Sto74] M. Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the royal statistical society. Series B (Methodological)*, pages 111–147, 1974.
- [SVC06] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley Sons, Inc., Hoboken, NJ, USA, 2006.
- [SvGJ⁺15] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspán, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15*, pages 598–608. IEEE Press, 2015.
- [SY86] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, 1986.
- [Tea17] Eclipse DeepLearning4j Development Team. DeepLearning4j: Open-source distributed deep learning for the jvm. <http://deeplearning4j.org>, 2017. Accessed 15.04.2020.
- [Tem] Pebble Templates. Pebble. <https://pebbletemplates.io/>.
- [TGPA14] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin. Aletheia: Improving the usability of static security analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14*, pages 762–774, New York, NY, USA, 2014. ACM.
- [TLC⁺16] Tyler W. Thomas, Heather Lipford, Bill Chu, Justin Smith, and Emerson Murphy-Hill. What questions remain? an examination of how developers understand an interactive static analysis tool. In *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*, Denver, CO, June 2016. USENIX Association.
- [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*, page 87–97, New York, NY, USA, 2009. Association for Computing Machinery.
- [TSBB17] J. Thomé, L. K. Shar, D. Bianculli, and L. C. Briand. Joanaudit: A tool for auditing common injection vulnerabilities. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017*, pages 1004–1008, New York, NY, USA, 2017. ACM.
- [TTCL18] Tyler W. Thomas, Madiha Tabassum, Bill Chu, and Heather Lipford. Security during application development: An application security expert perspective. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI '18*, pages 262:1–262:12, New York, NY, USA, 2018. ACM.
- [VdSCC⁺08] A Van der Stock, D Cruz, J Chapman, D Lowery, E Keary, M Morana, D Rook, J Williams, and P Prego. Owasp code review guide v1. 1. *The OWASP Foundation Guidelines*, 2008.

- [Ver20] Veracode. Veracode. <https://www.veracode.com/>, 2020. Online; accessed January 2020.
- [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.
- [WFHP16] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2016.
- [WROR18] 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. *ACM Trans. Priv. Secur.*, 21(3), April 2018.
- [WZW⁺15] Jim Witschey, Olga Zielinska, Allaire Welk, Emerson Murphy-Hill, Chris Mayhorn, and Thomas Zimmermann. Quantifying developers’ adoption of security tools. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, pages 260–271, New York, NY, USA, 2015. ACM.
- [Zan02] Mirko Zanotti. Security typings by abstract interpretation. In Manuel V. Hermenegildo and Germán Puebla, editors, *Static Analysis*, pages 360–375, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
- [ZGSN17] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. Effective interactive resolution of static analysis alarms. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.
- [ZSO⁺17] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. 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, 2017.

Appendix

7.1 Supplementary material for Chapter 3

7.1.1 Survey Questions

The survey from the study described in Section 3.2 consists of 7 parts. We list all questions from the survey that are relevant for this paper.

Part 1: Questions for all roles

- 1 To what extent do you agree with the following statements? [likert scale: "strongly agree", "agree", "disagree", "strongly disagree", "do not know"]
 - 1.1 For the topic *Secure Software Engineering* (SSE) our team invests the right amount of time.
 - 1.2 In our team, we have members who are responsible for *Security*.
 - 1.3 We have clearly defined regulations and policies how to develop secure software.
 - 1.4 We have the right amount of tools to support us in developing secure software.
 - 1.5 Our development process and the existing tools are enough for our needs.
 - 1.6 Security requirements (e.g. secure processing of confidential data) are clearly defined in our team.
- 2 What is your team responsible for regarding your software product? [multiple choice: "Deployment/Server configuration", "Programming", "Testing", "Build processes", "Software product operation", "Requirements", "Architecture and design"]

Part 2: Questions related requirements

- 3 Is your role involved in requirements elicitation? [single choice: "Yes", "No" (if no, then part 2 is skipped)]
- 4 Please answer the following questions [single choice: "Yes", "No", "I don't know"]
 - 4.1 Is security considered during activities related to requirements management?
 - 4.2 Do you have security experts that check the security requirements?
- 5 To what extent do you agree with the following statement related to security requirements? [likert scale: "strongly agree", "agree", "disagree", "strongly disagree", "do not know"]
 - 5.1 Our current processes should be more precise and clear.
 - 5.2 More or better tools would help us to perform our tasks with higher quality.

Part 3: Questions related design and architecture

- 6 Is your role involved in design and architecture? [single choice: "Yes", "No" (if no, then part 3 is skipped)]
- 7 Please answer the following questions [single choice: "Yes", "No", "I don't know"]
- 7.1 Is security considered during activities related to design and architecture?
 - 7.2 Do you have a process to check the security properties of the design with respect to the implemented program?
 - 7.3 Do you have security experts that check the architecture from security perspective?
- 8 To what extent do you agree with the following statement related to secure design and architecture? [likert scale: "strongly agree", "agree", "disagree", "strongly disagree", "do not know"]
- 8.1 Our current processes should be more precise and clear.
 - 8.2 More or better tools would help us to perform our tasks with higher quality.

Part 4: Questions related to implementation and testing

- 9 Is your role involved in implementation and testing? [single choice: "Yes", "No" (if no, then part 4 is skipped)]
- 10 Please answer the following questions [single choice: "Yes", "No", "I don't know"]
- 10.1 Is security considered during implementation?
 - 10.2 Are there templates, in particular standards for implementing secure software?
 - 10.3 Do you use tools to automatically check security checks of the implemented code?
 - 10.4 Do you have a process to check the security properties of the code?
- 11 Who checks the code agains security vulnerabilities? [multiple choice with option for free text: "Same person who wrote the code", "Another person from the team who did not write the code", "Internal security team", "External security team", "Nobody", "Other"]
- 12 Which IDEs are used within your team? [multiple choice with option for free text: "Eclipse", "IntelliJ Idea", "NetBeans", "Visual Studio Code", "Visual Studio", "vi / vim", "Notepad++ (or similar editor)", "Apple Xcode", "Other"]
- 13 Which programming languages are used primarily within your team? [multiple choice with option for free text: "Java", "JavaScript/TypeScript", "C#", "C", "C++", "Kotlin", "Objective-C", "Python", "Ruby", "PHP", "Go", "SQL", "Swift", "Rust", "R", "Other"]
- 14 When is the program checked against security vulnerabilities? [multiple choice with option for free text: "During implementation in the IDE", "Before each commit in the repository", "After each commit from the server pipeline", "Before official release", "During one sprint", "Never", "Other"]
- 15 To what extent do you agree with the following statement related to secure software implementation and testing? [likert scale: "strongly agree", "agree", "disagree", "strongly disagree", "do not know"]
- 15.1 Our current processes should be more precise and clear.

15.2 More or better tools would help us to perform our tasks with higher quality.

Part 5: Questions related to static analysis tools

- 16 Does your team use static analysis tools, such as static analysis tools? [single choice: "Yes", "No" (if no, then part 5 is skipped)]
- 17 Which static analysis tools are used within your team? [open question]
- 18 To what extent do you agree with the following statements? [likert scale: "strongly agree", "agree", "disagree", "strongly disagree", "do not know"]
- 18.1 The tools we use return the results fast enough for our needs.
- 18.2 The number of reported warnings that are false (false positives) is too high.
- 18.3 I can confirm the true warnings (true positives) easily.
- 18.4 The tools we use often report true warnings.
- 18.5 The messages of the warnings help me to fix the issues in the code.
- 18.6 I am willing to write our project-specific custom rules for the static analysis tools.
- 18.7 I am willing to label the false warnings to give feedback to the tools so that the tools can improve in the future.
- 18.8 I have experience in writing custom rules for some of the tools.
- 19 Please sort the following statements based on importance, where 1 has the highest importance. [sorting of statements from 1 to 4]
- The analysis should finish within few seconds.
 - The messages of the reported findings should be understandable and provide hints how to fix the issues.
 - It should be easy for me to understand and adapt the rules of the tools according to my needs.
 - The tool should return only very few false warnings.
- 20 Where should the warnings from the static analysis tools be reported? [sorting of statements from 1 to 3]
- In my IDE (e.g. Eclipse)
 - On an internal website (e.g. Jenkins)
 - In our ticket system (e.g. Jira)

Part 6: Questions related to software operation and maintenance

- 21 Is your role involved in software operation and maintenance? [single choice: "Yes", "No" (if no, then part 6 is skipped)]
- 22 Please answer the following questions [single choice: "Yes", "No", "I don't know"]
- 22.1 Does your team performs a final security review before each release?
- 22.2 Are there automatic security checks for each release?
- 22.3 Are there automatic security checks during operation?

23 To what extent do you agree with the following statements related to secure software operation and maintenance? [likert scale: "strongly agree", "agree", "disagree", "strongly disagree", "do not know"]

23.1 Our current processes should be more precise and clear.

23.2 More or better tools would help us to perform our tasks with higher quality.

Part 7: Meta-data questions

24 How many employees has your company? [single choice: "1-3", "4-10", "11-50", "51-250", "251-1000", "> 1000"]

25 In which domain operate your company? [open question]

26 How many employees work in software development? [single choice: "1-50", "51-250", "> 250"]

27 What is your position? [multiple choice with option for free text: "Management", "Project lead", "Product owner", "Software development (requirements, implementation, testing)", "Security analyst", "Information security officer", "Other"]

28 How many years of experience in software development do you have? [single choice: "< 2 years", "2-5 years", "6-10 years", "> 10 years"]

29 How many members has your team? [single choice: "1-5", "6-15", "16-30", "> 30"]

30 What type of applications do you develop? [multiple choice with option for free text: "Mobile", "Desktop", "Web", "Embedded", "Server", "Other"]

7.1.2 Interview Questions

The semi-structured interviews from the study described in Section 3.2 were conducted with product owners (PO) and executives (E). The questions are grouped into topics. Each question is marked with PO and/or ME, indicating that it was asked to interviewees with the specific role.

Introduction

1 Please describe your role. What are the main tasks you perform on a daily basis? [PO/E]

2 Do you actively participate in the software development? If yes, in which of the following areas: Requirements, Design, Implementation, Testing, and Operations? [PO/E]

3 IT-Security - what priority has this topic for you personally, privately and professionally? [PO/E]

Product

4 Do you define security requirements for your products? If yes, which? Do you prioritize security requirements in the beginning of the design of a new product? Do you require a final penetration test? Do you require security certifications and standards? [PO]

5 Do you define user stories from the perspective of an attacker (evil-user-stories)? Do you define "evil" roles in the team? [PO]

6 Which security requirements come from the users of your products? [PO]

Development process

- 7 Is Security an important topic during sprint planning and retrospective? If yes, what is usually discussed? [PO]

Organization and personel

- 8 In your opinion, are the software developers aware of security? If no, do you take any actions to increase the awareness? [PO/E]
- 9 Do you think that the software developers need trainings for new competences in SSE? Do the software developers have the same opinion? [PO/E]
- 10 Do you wish more expertize in SSE for yourself? [PO/E]
- 11 Which skills with respect to SSE do you expect from the software developers? [PO/E]
- 12 Are there resources for external or internal trainings for the software developers in SSE topics? Do software developers have time for self-learning in SSE topics? [PO/E]
- 13 Which actions do you take to ensure that the software developers have the right competences in SSE? [E]

Tools

- 14 What is your opinion in using free tools during software development? Do the software developers use free tools? [PO/E]
- 15 Do you think that the software developers in your team need more tools to improve the security of the software? [PO/E]
- 16 Is there a reserved budget for tools that are focused on security? [E]
- 17 What is your opinion when software developers using SAST tools invest time in providing feedback to the tools, for example labeling false warnings in the tool? [E]

Optional

- 18 Are there clear responsibilities when security incidence happen? If yes, are you involved in those events? [PO/E]
- 19 Are you involved in defining the processes for software development? If yes, is security addressed in the processes? [E]
- 20 Do you or your team attend events with focus on security? [E]
- 21 Are you aware of the offers in security trainings in Germany? If yes, are you happy with these offers? Do you know if your team shares your opinion? [E]

Supervised theses

The following is a list of bachelor and master theses at Paderborn University (co-)supervised by the author of this thesis.

- *Explorative research on taint analysis for Kotlin*, master thesis by Ranjith Krishnamurthy, 2022
- *Soot-based configuration generator for analysis writers*, master thesis by Shreyas Kottur Shivananda, 2021
- *Extending fluentTQL: Specifying taint-flows through a domain-specific language*, master thesis by Enri Ozuni, 2021
- *Evaluation of Call Graph Construction for Python*, master thesis by Sriteja Kummita, 2020
- *Transformation of Taint and Typestate Specifications*, master thesis by Alexander Lorsch, 2020
- *Detection of methods of interest for security based on software documentation*, master thesis by Oshando Johnson, 2020
- *Authentication and authorization checker for Java web systems*, bachelor thesis by Tobias Petrasch, 2018
- *Evaluation of machine learning algorithms for automatic detection of security-relevant methods*, bachelor thesis by Parviz Nasiry, 2018