Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Sindre Beba, Magnus Melseth Karlsen

Implementation Analysis of Open-Source Static Analysis Tools for Detecting Security Vulnerabilities

Master's thesis in Computer Science
Supervisor: Jingyue Li
June 2019

NTNU
Norwegian University of Science and Technology

Sindre Beba, Magnus Melseth Karlsen

Implementation Analysis of Open-Source Static Analysis Tools for Detecting Security Vulnerabilities

Master's thesis in Computer Science
Supervisor: Jingyue Li
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

A static analysis tool that claims to cover a specific vulnerability has an inherited level of trust to fulfill. Bringing the actual performance and limitations of these tools into the spotlight helps to prove or disprove any question about their applicability. We have analyzed the implementation and identified limitations for three open-source static analysis tools aimed at detecting security vulnerabilities, namely SpotBugs, Find Security Bugs, and ESVD. A common pattern of strengths and limitations emerged as a result of this analysis. The use of taint analysis is an important technique for increased detection rates. A generalizable implementation also seems to benefit the coverage, as code can be reused for new vulnerabilities with only minor modifications. Incomplete collections of vulnerable sources and sinks is a limitation with a significant impact on the detection rates of the static analysis tools; the same applies to weak control- and data-flow analysis. The misclassification of vulnerabilities and an inconsistent or missing confidence ranking also negatively affect the tools. This master thesis serves as an important first step into a scarcely explored part of static analysis tools. The contributions from this master thesis are the detailed performance results produced from over 20,000 vulnerability reports, in addition to describing the implementation of the static analysis tools, identifying limitations, the improvements to address these limitations, and further improvement proposals.

Sammendrag

Brukere må kunne føle seg trygge på at verktøy for statisk kodeanalyse holder sitt løfte om dekningsgrad. Begrensninger, og ikke minst den faktiske ytelsen til disse verktøyene bør gjøres kjent slik at det med sikkerhet kan bestemmes om de er hensiktsmessig for ønsket bruk. Vi har analysert implementasjonen og identifisert begrensninger for tre verktøy for statisk kodeanalyse som er laget for å oppdage sikkerhetshull, og basert på prinsippet om åpen og fritt tilgjengelig kildekode. De tre verktøyene er SpotBugs, Find Security Bugs, og ESVD. Basert på analysen fant vi et sett med felles styrker og begrensninger. Bruken av *taint analysis* er en teknikk som ofte fører til bedre oppdagelsesrate av sikkerhetshull. I tillegg fant vi at en generaliserbar implementasjon ser ut til å påvirke dekningsgraden positivt, ettersom algoritmer enkelt kan bli gjenbrukt for nye sikkerhetshull. Begrensede mengder av kilder og sluker (eng. *source* og *sink*) påvirker oppdagelsesraten svært negativt. Dårlig *control-flow* og *data-flow analysis* er også en hovedårsak til dårlige oppdagelsesrater. Vi fant også at oppdagede sikkerhetshull noen ganger klassifiseres som helt andre typer sikkerhetshull. Denne masteroppgaven utfører en viktig undersøkelse av verktøy for statisk kodeanalyse, noe som ikke tidligere har blitt gjort til en slik grad. Bidragene fra denne masteroppgaven er detaljerte ytelsesresultater basert på mer enn 20 000 eksekveringer på sårbar kode, i tillegg til en beskrivelse av implementasjonen til verktøyene for statisk kodeanalyse, deres begrensninger, forbedringer av disse begrensningene, og forslag til forbedringer for hele felleskapet for statisk kodeanalyse.

Acknowledgment

We would first like to thank our thesis advisor, Associate Professor Jingyue Li of the Department of Computer Science at the Norwegian University of Science and Technology (NTNU). Assoc. Prof. Li was always there to help whenever we ran into a problem or had a question. We are also very grateful for his valuable feedback on our research and report. Without his advice and guidance, we would not have been able to produce the master thesis you are currently reading.

We would also like to thank the program committee, reviewers, and other participants of the Evaluation and Assessment in Software Engineering (EASE) 2019 conference. It was an honor being allowed to present at the conference, and the valuable feedback and questions we received allowed us to further reflect on and improve our research. We would also like to extend a big thank you to NTNU for giving us the opportunity to travel to Denmark to participate in EASE 2019, and to Assoc. Prof. Li for making it possible.

Last but not least, we would like to thank our families and friends for all the support during the process of writing this master thesis. This thesis has taken up a lot of our time and energy, and without your support that would not have been possible.

Thank you,

Sindre Beba and Magnus Melseth Karlsen

Contents

1. Introduction **13**

2. Background **15**
2.1. Importance of Information Security 15
2.2. Critical Security Vulnerabilities 15
2.2.1. Injection 17
2.2.2. Use of Hard-coded Password 17
2.2.3. Path Traversal 18
2.2.4. Cross-Site Scripting 18

3. Related Work **19**
3.1. What is Static Analysis? 19
3.2. Evaluating Static Analysis Tools 20
3.3. Analyzing the Implementation of Static Analysis Tools 22
3.4. Extending and Improving Existing Static Analysis Tools 22

4. Pre-Study **23**
4.1. Coverage and Performance Evaluation 23
4.2. Coverage and Performance Results 25
4.3. Usability Evaluation 26

5. Research Design and Implementation **30**
5.1. Research Motivation 30
5.2. Research Questions 31
5.3. Research Method and Design 31
5.3.1. Research Strategy 31
5.3.2. Data Generation and Analysis 31
5.3.3. Research Paradigm 32
5.4. Research Implementation 32
5.4.1. Selection of the Static Analysis Tools 33
5.4.2. Selection of Vulnerability Categories 34
5.4.3. How Juliet Test Suite is Structured 36
5.4.4. How Parsers are Used to Facilitate the Analysis 38
5.4.5. RQ1 and RQ2: How to Present the Implementation Analysis Results 40
5.4.6. RQ1 and RQ2: Conducting the Implementation Analysis 40
5.4.7. RQ3: Conducting the Limitation Analysis and Producing Proof-of-Concept Improvements 41

6. Research Results **43**
6.1. RQ1: How is Static Analysis Implemented in the SATs from the Pre-Study? 43
6.1.1. SpotBugs . . . 43
6.1.2. Find Security Bugs . . . 46
6.1.3. ESVD . . . 49
6.2. RQ2: How can the Performance of the SATs be Explained by Their Implementation? . . . 51
6.2.1. SpotBugs: Injection Vulnerabilities . . . 52
6.2.2. SpotBugs: Hard-Coded Password . . . 58
6.2.3. SpotBugs: Path Traversal . . . 59
6.2.4. SpotBugs: Cross-Site Scripting . . . 62
6.2.5. Find Security Bugs: Injection Vulnerabilities . . . 67
6.2.6. Find Security Bugs: Hard-Coded Password . . . 73
6.2.7. Find Security Bugs: Path Traversal . . . 77
6.2.8. Find Security Bugs: Cross-Site Scripting . . . 79
6.2.9. ESVD: Injection Vulnerabilities . . . 81
6.2.10. ESVD: Hard-Coded Password . . . 86
6.2.11. ESVD: Path Traversal . . . 88
6.2.12. ESVD: Cross-Site Scripting . . . 90
6.3. RQ3: How can the Limitations of the SATs be Addressed Through Proof-of-Concept Improvements? . . . 91
6.3.1. Addressing the Limitations in SpotBugs . . . 92
6.3.2. Addressing the Limitations in Find Security Bugs . . . 99
6.3.3. Addressing the Limitations in ESVD . . . 106
6.4. Summary . . . 115

7. Discussion **119**
7.1. Use of Taint Analysis . . . 119
7.2. Generalizable Implementation . . . 120
7.3. Incomplete Collections of Sources and Sinks . . . 120
7.4. Control- and Data-flow Analysis Limitations . . . 121
7.5. Prioritized Output . . . 122
7.6. Vulnerability Misclassification . . . 123
7.7. Comparison to Related Work . . . 124
7.8. Limitations of the Juliet Test Suite . . . 125
7.9. Threats to Validity . . . 125

8. Conclusion and Future Work **127**

A. Scientific Paper of our Pre-Study **135**

B. Flow Variants in the Juliet Test Suite **146**

List of Figures

3.1. Example of Control-Flow Graph Notation 20

5.1. Juliet Test Suite: Connect TCP Source Variant 36
5.2. Juliet Test Suite: File Source Variant 36
5.3. Juliet Test Suite: Control-Flow Variant 37
5.4. Juliet Test Suite: Data-Flow Variant 37
5.5. Example of Parser Output 39

6.1. Java Compiler Optimization: Removing Conditional Statement 44
6.2. Java Compiler Optimization: Not Removing Conditional Statement . . . 45
6.3. Find Security Bugs: Resource File 47
6.4. Find Security Bugs: Illustration of Taint Analysis 48
6.5. ESVD: Illustration of Architecture 50
6.6. ESVD: XML-File Containing Sources 51
6.7. Juliet Test Suite: Path Traversal Sinks 62
6.8. Juliet Test Suite: CWE-80 Basic XSS Sink Variants 64
6.9. Bytecode Instructions from CWE-80 Sink without String.replaceAll() . . 65
6.10. Bytecode Instructions from CWE-80 Sink with String.replaceAll() 66
6.11. Find Security Bugs: System Variables Configuration 67
6.12. Find Security Bugs: Password Words 74
6.13. Find Security Bugs: Password Names 77
6.14. ESVD: Missing Sink for LDAP Injection 84
6.15. ESVD: Missing Sinks for HTTP Response Splitting 85
6.16. ESVD: XPath Injection Sink Used in the Juliet Test Suite 86
6.17. SpotBugs: Added Source Cookie.getValue() 95
6.18. SpotBugs: Path Traversal Changes 96
6.19. SpotBugs: Changes to Taint Analysis for String.replaceAll(...) 99
6.20. Find Security Bugs: Tainted System Variables 104
6.21. ESVD: Changes in Test Case for Hard-Coded Password 109
6.22. ESVD: Removing Complexity from XPath Injection Test Case 110
6.23. ESVD: Method Signatures for Connect TCP, Listen TCP, and URL Connection Source Variants 111
6.24. ESVD: Method Signature for File Source Variant 111
6.25. Juliet Test Suite: File Source Variant 112

List of Tables

4.1. Metric Applicability for Natural and Artificial Code 25
4.2. Confirmed and Claimed Coverage of the Static Analysis Tools 26
4.3. Detailed Coverage Data with True and False Positives 27
4.4. Detailed Performance Data with Recall, Precision, and Discrimination Rate 28
4.5. Summary of the Usability Results 29

5.1. Information About the Selected Static Analysis Tools 34
5.2. List of Selected Vulnerabilities 35

6.1. Java Compiler Optimization: Control-flow variants removed 46
6.2. RQ2: Injection Results for SpotBugs 56
6.3. RQ2: Hard-Coded Password Results for SpotBugs 60
6.4. RQ2: Path Traversal Results for SpotBugs 61
6.5. RQ2: XSS Results for SpotBugs 63
6.6. RQ2: Injection Results for Find Security Bugs, CWE-78, CWE-89, CWE-90 69
6.7. RQ2: Injection Results for Find Security Bugs, CWE-113, CWE-643 . . . 72
6.8. RQ2: Hard-Coded Password Results for Find Security Bugs 76
6.9. RQ2: Path Traversal Results for Find Security Bugs 78
6.10. RQ2: XSS Results for Find Security Bugs 79
6.11. RQ2: Injection Results for ESVD 82
6.12. RQ2: Hard-Coded Password Results for ESVD 87
6.13. RQ2: Path Traversal Results for ESVD 89
6.14. RQ2: XSS Results for ESVD 91
6.15. RQ3: Injection Modification Results for SpotBugs 93
6.16. RQ3: Path Traversal Modification Results for SpotBugs 97
6.17. RQ3: XSS Modification Results for SpotBugs 98
6.18. RQ3: Injection Modification Results for Find Security Bugs I 100
6.19. RQ3: Injection Modification Results for Find Security Bugs I 101
6.20. RQ3: Injection Modification Results for Find Security Bugs II 101
6.21. RQ3: Injection Modification Results for Find Security Bugs II 102
6.22. RQ3: Path Traversal Modification Results for Find Security Bugs 105
6.23. RQ3: Injection Modification Results for ESVD I 107
6.24. RQ3: Injection Modification Results for ESVD II 108
6.25. RQ3: Path Traversal Modification Results for ESVD 113
6.26. RQ3: Injection Modification Results for ESVD III 113
6.27. RQ3: XSS Modification Results for ESVD 114
6.28. Summary: Strengths and Limitations I 116

6.29. Summary: Strengths and Limitations II 117
6.30. Summary: Total Results of RQ2 and RQ3 118

B.1. Control-Flow Variants in the Juliet Test Suite 147
B.2. Data-Flow Variants in the Juliet Test Suite 148

Acronyms

API	Application Programming Interface
BCEL	Byte Code Engineering Library
CFG	Control-Flow Graph
CORE	Computing Research and Education Association of Australasia
CRLF	Carriage Return Line Feed
CWE	Common Weakness Enumeration
EASE	Evaluation and Assessment in Software Engineering
ERA	Excellence in Research in Australia
FN	False Negative
FP	False Positive
FPR	False Positive Rate
GDPR	General Data Protection Regulation
IDE	Integrated Development Environment
J2EE	Java 2 Platform, Enterprise Edition
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
OWASP	Open Web Application Security Project
SARD	Software Assurance Reference Dataset
SAT	Static Analysis Tool
SATE	Static Analysis Tool Exposition
SQL	Structured Query Language
TN	True Negative
TP	True Positive
XSS	Cross-Site Scripting

Glossary

Artificial Code Code made to be intentionally vulnerable to serve as a testing platform for static analysis tools.

Basic Block Uninterrupted blocks of code between control-flow statements.

Bug Pattern Code idiom that is likely to be an error.

Bytecode Compiled code (e.g., Java) which is interpreted by the Java Virtual Machine. Located in `.class` files.

Common Weakness Enumeration (CWE) Community-developed list of software weaknesses. The purpose is to serve as a common standard for people working with software security.

Completeness A static analysis tool is complete if all its reports are correct. That is, it has no false positives, but it may have false negatives.

Constant Pool Contains compile-time constants formed from literals, and cannot be changed after compilation.

Context-Sensitivity A context-sensitive static analysis tool takes the actual function parameters as well as global variables into consideration when analyzing the code. It is able to track the data between classes.

Control-Flow The order of execution in a program.

Control-Flow Analysis A static code analysis technique to analyze and thus determine the control-flow of a program. The result is a control-flow graph.

Control-Flow Graph A directed graph consisting of nodes (basic blocks) and edges (control-flow statements) depicting the control-flow of a program.

Control-Flow Statement A code statement that forks the control-flow. E.g., `if`-statements and `while`-loops.

Coverage How many and which security vulnerabilities a static analysis tool detects.

Data-Flow The change in assignments of variables during the execution of a program, i.e., what each variable is assigned to at each time.

Data-Flow Analysis (DFA) Monitors the data-flow and how the variables are used by utilizing a control-flow graph. At each node in the control-flow graph, data-flow analysis collects information about what the values are for incoming and outgoing variables.

Detector A class within a static analysis tool that is designed to detect a certain set of vulnerabilities, e.g., an SQL injection detector.

Discrimination Rate The percentage of test cases a static analysis tool discriminates. An SAT discriminates a vulnerability if it only reports a true positive on it, but no false positives.

Early Detection Detecting security vulnerabilities while the user is writing the code.

False Negative (FN) Incorrect report of no vulnerability. The code contains a vulnerability, but no vulnerability is reported.

False Positive (FP) Incorrect detection of a vulnerability. No vulnerability exists in the code, but a vulnerability is reported.

False Positive Rate (FPR) The percentage of detections that are false positive.

Flow Variant Variant of a test case that includes specific control-flow statements that add complexity to the test case.

Interprocedural Analysis See Context-Sensitivity.

Java Virtual Machine (JVM) Virtual machine that runs Java bytecode.

Method Signature Identifies a method. Consists of the method name and the parameter list.

Natural Code Source code of any application designed to be used.

Performance How well a static analysis tool performs. Consists of the performance metrics: recall, precision, and discrimination rate.

Precision The percentage of detections that are true positives.

Recall The percentage of vulnerabilities that are detected out of the total amount.

Sanitization-Point A place in the code where untrusted data is processed and changed into trusted data.

Sink A place in the code where untrusted data leaves the boundary of the program.

Soundness A static analysis tool is sound if it reports all bugs in the code. That is, it has no false negatives, but it may have false positives.

Source A place in the code where external input enters the program.

Source Variant Variant of a test case that uses a specific source.

Static Analysis Technique used to analyze the source code of an application without executing it.

Static Analysis Tool (SAT) A tool using static analysis to detect software flaws.

Taint Analysis Analysis technique to detect vulnerable input retrieved from a source that is used in a sink. Uses both control-flow and data-flow analysis.

Test Case File in Juliet Test Suite that includes exactly one vulnerability as well as at least one non-flawed construct meant to represent a potential false positive.

True Negative (TN) Correct report of no vulnerability. No vulnerability exists in the code, and no vulnerability is reported.

True Positive (TP) Correct detection of a vulnerability. The code contains a vulnerability, which is correctly detected and reported.

1. Introduction

As society gets more dependent on technology, the importance of securing information systems increases. A solution to reduce the number of security vulnerabilities is static analysis tools (SAT). With these tools included directly into the Integrated Development Environment (IDE) of the user, they have never been easier to use. Unfortunately, not a lot of research has been conducted to compare and evaluate the existing IDE-integrated SATs, leaving the users to take the developers claims of quality for face value.

With the aim to provide more information about open-source IDE-integrated SATs, we conducted a pre-study where we reported on their actual coverage, their performance, and their usability. This was achieved by evaluating them using a credible framework for test cases and commonly used performance metrics. The SATs were evaluated on the most critical security vulnerabilities according to OWASP [2017], a reputable source for web application security. The results showed that the existing solutions had obvious limitations and a worrying discrepancy between what they claimed and what they actually did. Coverage was focused around vulnerabilities connected to injection and access control, while other categories were left untouched. Several of the tools also had a high false positive rate, a leading factor for why users do not adopt static analysis tools [Christakis and Bird, 2016].

In this master thesis we continue our work from last year by further analyzing the results. We select the three SATs utilizing data-flow analysis and take a detailed look at their implementation. We will conduct an implementation analysis as well as a limitation analysis to better understand why the SATs perform as they did in our pre-study. By making modifications to the tools, we test our hypotheses about the implementation as well as demonstrate their limitations. Our contribution comes in the form of detailed performance results, an explanation of the SATs' implementation, identifying limitations, code modifications improving the performance results, and further improvement proposals. It is a contribution to those who seek to adopt such tools as well as developers who want to contribute to the field.

The results of this thesis uncover several strengths and limitations in the implementation of the SATs. Taint analysis is a technique that gives superior results, especially a high precision. An implementation that is generalizable performs well and is accessible to modifications and improvements. Incomplete collections of sources and sinks are a recurring limitation of the SATs; the same applies to inadequate algorithms for control-flow and data-flow analysis. Prioritized output can be a helpful feature when the precision is imperfect, and misclassification of vulnerabilities can cause confusion and detections to be inaccurate or overlooked.

The thesis starts with necessary background knowledge in chapter 2 and a description of related work in chapter 3. A summary of our pre-study is presented in chapter 4 with the complete scientific paper included in appendix A. Then, the research design and research implementation are presented in chapter 5. That includes the research questions. We present the results of our research questions in chapter 6, including the implementation analysis and limitation analysis of the static analysis tools. The results are then further discussed in chapter 7, where we also present some observations based on the results. Finally, we conclude in chapter 8 and present some ideas of what can be done in the future following our research.

2. Background

2.1. Importance of Information Security

Data breaches are common occurrences in the newspapers today. As mentioned in a press release by Facebook [2018a,b], hackers managed to gain access to 30 million Facebook accounts. In a press release by Google [2018a,b], a security vulnerability in Google+ was exposing private data of 52.5 million users. Earlier this year, as reported by Wired [Greenberg, 2019], a collection consisting of 2.2 billion stolen usernames and passwords was published online. Special counsel Robert S. Mueller's investigation into Russian interference in the 2016 U.S. presidential election found that the GRU used simple SQL injection in their attacks [Mueller and U.S. Department of Justice, 2019, p. 50]. More generally, each data breach costs an average of 3.86 million US dollars [Ponemon Institute, 2018], with some predicting that cybercrime will cost the world 6 trillion US dollars annually by 2021 [Cybersecurity Ventures, 2017]. Securing information systems is therefore an essential part when protecting important information and reducing costs.

The introduction of the General Data Protection Regulation (GDPR) in the EU has also put information security at the top of the agenda for many companies. Heavy fines await if the regulations are not followed, as proven for the Norwegian municipality Bergen [IAPP, 2018] and the German company Knuddels [Tellerreport, 2018], which got a fine of respectively 1.6 million NOK (approximately 164,000 euros) and 20,000 euros for not encrypting user passwords.

One of the ways to secure information systems is by removing security vulnerabilities in the software code. Carelessness, complexity, or lack of necessary knowledge can lead to the unintended inclusion of security vulnerabilities by developers. To combat this, some have chosen to turn their attention to static analysis tools. These tools can give real-time feedback while the developer is writing the code, or be used on-demand by manually running the SAT. Baca, Carlsson, and Lundberg [2008] noted that the use of an *a posteriori* static analysis tool reduced the maintenance cost by an average of 17%.

2.2. Critical Security Vulnerabilities

There are many types of security vulnerabilities, some more relevant than others. Open Web Application Security Project (OWASP) is known as a reputable source for web application security, and the OWASP Top 10 list of security vulnerabilities is highly regarded. The list consists of the ten most critical categories of security vulnerabilities. The most recent report is presented below in order of importance [OWASP, 2017].

A1 Injection Untrusted data is sent as a command or query to a vulnerable interpreter. Examples of injection vulnerabilities include SQL injection, OS injection, LDAP injection, etc.

A2 Broken Authentication Incorrect implementation of session or authentication management, allowing attackers access to passwords, session tokens, etc.

A3 Sensitive Data Exposure Weakly protected data can be stolen or modified, resulting in identity fraud or other crimes. This is especially important in financial or healthcare settings.

A4 XML External Entities Poorly configured XML processors might evaluate external entity references when given XML documents. This can lead to remote code execution, internal port scanning, etc.

A5 Broken Access Control Attackers can exploit missing or broken access restrictions. This allows access to unauthorized functionality and data, e.g., sensitive files.

A6 Security Misconfiguration Misconfiguration or use of insecure default configurations can lead to a range of different attacks.

A7 Cross-Site Scripting Untrusted data is included into a web page without validation or sanitization. This allows the possibility of script executions in the victim's browser.

A8 Insecure Deserialization Deserialization flaws can lead to remote code execution, which again might lead to other attacks.

A9 Using Components with Known Vulnerabilities Frameworks and libraries used in an application can lead to data loss or server takeover. These components run with the same privileges as the application, and might undermine application defenses and enable attacks.

A10 Insufficient Logging & Monitoring Insufficient logging and monitoring will increase the time before a breach is detected and reacted to. This can allow attackers further access and more time to attack systems, pivot to other systems, etc.

All of these categories include several different types of vulnerabilities, all with a unique CWE entry. The Common Weakness Enumeration (CWE) is a community-developed list of software weaknesses. The purpose of CWE is to serve as a common standard for people working with software security [MITRE, 2018a]. The list consists of several CWE entries where each entry represents a weakness. Each entry has its own number for easy reference and can have relations to other CWE entries.

2.2.1. Injection

Injection vulnerabilities encapsulate all vulnerabilities that are caused by improper handling of an input from an outside source [MITRE, 2019b]. The input should be sanitized, also referred to as neutralized, before it is further used by the application. In the absence of doing so, the input can potentially alter the control-flow of the application in order to achieve the attacker's goal.

There are many different kinds of injection vulnerabilities. They can share several similarities, but also differ from each other substantially.

For OS command injection, the vulnerability is caused by a constructed OS command receiving unsanitized input [MITRE, 2019c]. The input can alter the behavior of the OS command by using special elements. This is a severe vulnerability as it allows attackers to execute commands directly on the OS. In addition to sanitizing the input, it is a good idea to run the command with the lowest privileges possible.

Similarly, the vulnerabilities SQL injection, LDAP injection, and XPath injection are also caused by a constructed command receiving input which has not been sanitized [MITRE, 2018e,f, 2019d]. However, in these cases an SQL, LDAP, or XPath command is respectively exploited instead of an OS command. This can lead to the attacker either bypassing authorization, reading sensitive data, or altering the underlying database or directory. For SQL injections, it is encouraged to use prepared statements that automatically sanitize the input.

For HTTP response splitting, unvalidated data is inserted into the header of an HTTP request [MITRE, 2018b]. By injecting CRLF into the HTTP header, the attacker can split the HTTP response into two responses with complete control of the second response which will be rendered in the user's browser.

2.2.2. Use of Hard-coded Password

Software that contains a hard-coded password used for authentication is vulnerable to misuse of this password. Not only can the password be read directly from the source code or from decompiling the binary code, but it also increases the possibility of password guessing [MITRE, 2019a; OWASP, 2016]. Misuse of hard-coded passwords can be difficult to detect, and if detected it can be difficult to fix. MITRE [2019a] and OWASP [2016] both claim that if a hard-coded password is used, it is almost certain that a malicious user will be able to access the account in question.

MITRE [2019a] describes two different types of hard-coded password usage: inbound authentication and outbound communication. The former is used when authenticating with the software containing the hard-coded password, and the latter is used when the software containing the hard-coded password is authenticating with another service.

Countermeasures against such attacks can be simple but depend on the usage of the password. If the hard-coded password is to be compared against user input, apply strong one-way hashing to the hard-coded password and compare the hashes instead. If the hard-coded password is to be used for an outbound connection, the password should be

stored outside of the code in a secure and encrypted database which is protected from unauthorized access by both local system users and outsiders.

2.2.3. Path Traversal

A path traversal attack is an attack that aims to access files outside the intended web root folder [OWASP, 2015b]. This can be used to read files that the attacker is not supposed to have access to, such as sensitive password files. Depending on what the attacker is able to access, this can allow further attacks to different services on the machine.

Path traversal can be divided into relative path traversal and absolute path traversal. A relative path traversal attack allows the attacker to traverse the server's file system by using sequences that resolves into the parent directory, e.g. "`../`" [MITRE, 2018c]. Absolute path traversal allows the attacker to traverse the server's file system by specifying the absolute path to a file [MITRE, 2018d].

Countermeasures include proper file system permissions and whitelisting input [OWASP, 2015a]. An example of proper file system permissions is a *chroot jail*, blocking the web server from referencing anything outside the web root folder. While some countermeasures will work for both relative and absolute path traversal, other countermeasures might only affect one or the other. Sanitizing the use of "`../`", which is a sequence of characters that will resolve into the parent directory, will only affect relative path traversal.

2.2.4. Cross-Site Scripting

Cross-Site Scripting (XSS) attacks are a form of injection attacks, where an attacker sends malicious scripts as input to web pages [OWASP, 2018a]. The malicious input is often browser-side scripts, such as JavaScript. Through different processes, this malicious script is sent back to an unsuspecting victim's web browser, where it is executed. Because the script seems to come from a trusted source, the web browser does not know it is malicious, thereby allowing access to cookies and sensitive information.

XSS attacks can roughly be divided into two categories; stored and reflected XSS attacks. A stored XSS attack occurs when the web page permanently stores the malicious script on the server, e.g., in the database of a messaging service. Once another user requests to download the message from the server, the malicious script is executed by the browser. A reflected XSS attack, on the other hand, is a type of attack where the server immediately reflects the malicious script. This can occur when using search engines, where the input from the search is also shown on the search result page.

A successful XSS attack can steal another user's cookies, including the possibility of hijacking the user session. It can also redirect the user to another URL, or modify the content on the current page. Due to the vast amount of attack vectors, there are many different ways to protect yourself from XSS attacks happening in your code [OWASP, 2019]. This includes sanitizing user input, whitelisting or blacklisting allowed input, and protecting cookies against theft.

3. Related Work

3.1. What is Static Analysis?

Boulanger [2011] defines static analysis as a generic term that can be applied to any tool that analyzes an application without executing it. Two common ways to measure static analysis tools are *soundness* and *completeness*. There are different interpretations of these concepts, but Delaitre et al. [2018] define them as:

Soundness - a static analysis tool is sound if it reports all bugs in the code. That is, it has no false negatives, but it may have false positives.

Completeness - a static analysis tool is complete if all its reports are correct. That is, it has no false positives, but it may have false negatives.

According to Emanuelsson and Nilsson [2008], no static analysis tool fulfills both of these criteria. However, when designing a static analysis tool, one has to consider the trade-off between them and pick one to aim for. Traditionally, most static analysis tools have aimed for soundness, but that is no longer the case. Christakis and Bird [2016] discovered through surveys that developers do not tolerate many false positives. In fact, most of the developers that participated only accepted a false positive rate below 15%.

Control-Flow Analysis

Most programming languages execute code in a certain order. This order is chosen by the programmers themselves in the way they structure the code. Languages that have this *order of execution* are called imperative programming languages [Van Roy and Haridi, 2004, p. 406], and the order of execution is referred to as *control-flow* [Aho et al., 2007, p. 399]. The control-flow is determined by *control-flow statements* that fork the order of execution into two or more paths. Two well-known examples of such statements are `if`-statements and `while`-loops.

Control-flow analysis is a static code analysis technique to analyze and thus determine what the control-flow of a program is. The result of control-flow analysis is a visualization of the control-flow called a *control-flow graph* [Allen, 1970]. It is a directed graph consisting of nodes and edges. The nodes represent uninterrupted blocks of code called *basic blocks*, while the edges represent jumps in the control-flow. Figure 3.1 shows an example of how an `if`-statement and a `while`-loop is depicted in a control-flow graph.

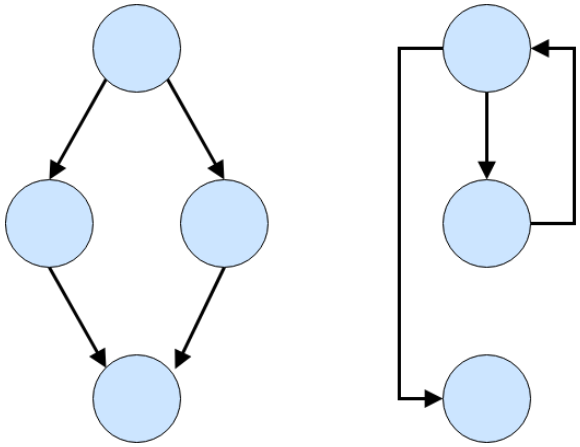

Figure 3.1.: The control-flow graph notation for an *if-then-else statement* (left) and a *while loop* (right).

Data-Flow Analysis

In computer programming, variables are declared and then assigned a value. An example can be the variable `x` being assigned the value `1`. As this is a variable, `x` can be assigned a new value later in the code, e.g., `2`. The change in assignments of variables during the execution of a program is called *data-flow*. In other words, the data-flow consists of what each variable is assigned to at each time. Data-flow analysis (DFA) monitors the data-flow and how the variables are used by utilizing a control-flow graph [Aho et al., 2007, p. 597]. At each node in the control-flow graph, DFA collects information about incoming and outgoing assignments.

Static analysis tools that utilize control- and data-flow analysis, often use it for monitoring how untrusted data moves through the control-flow. Tripp et al. [2009] presented taint analysis as a solution to vulnerabilities tied to information flow. All of these vulnerabilities can be described as tainted data originating from a vulnerable *source* moving through the control-flow to an exploitable *sink* without first being *sanitized*. Taint analysis has since become a common technique used by many static analysis tools.

Emanuelsson and Nilsson [2008] describe static analysis tools that track the data-flow across multiple files as *context-sensitive*. A context-sensitive static analysis tool takes the actual function parameters as well as global variables into consideration when analyzing the code. This is also known as interprocedural analysis and leads to a higher precision and longer analysis time.

3.2. Evaluating Static Analysis Tools

An early evaluation of static analysis tools was performed by Rutar, Almazan, and Foster [2004]. They evaluated five static analysis tools on natural code in the form of a set of Java applications. The SATs were Bandera, ESC/Java 2, FindBugs, JLint, and

PMD. Their results showed that all of the SATs had unique detections, and the authors proposed to combine the tools into a single meta-tool to achieve improved results.

Oyetoyan et al. [2018] evaluated six SATs on artificial code using the Juliet Test Suite. They evaluated FindBugs, Find Security Bugs, SonarQube, JLint, LAPSE+, and an undisclosed commercial tool on all of the test cases present in the Juliet Test Suite. Like Rutar, Almazan, and Foster [2004], they also concluded that a single tool was not enough to cover all vulnerabilities. Neither the open-source nor the commercial tool performed as well as expected. Through interviews, they also found that developers were interested in using SATs, but that there were several obstacles to overcome.

Similarly, Charest, Rodgers, and Wu [2016] evaluated the open-source SATs CodePro AnalytiX, JLint, FindBugs, and VisualCodeGrepper on a selected subset of the Juliet Test Suite. By using metrics such as recall and precision, they found that the tools performed poorly. However, the authors argued it was not too alarming as other studies had shown similar results for commercial tools.

The Static Analysis Tool Exposition (SATE) evaluated many SATs for the languages Java, PHP, and C. The tools were undisclosed, but were evaluated on both production software as well as the Juliet Test Suite. The results were presented in the paper by Delaitre et al. [2018]. In 2018, six different tools participated in the Java track. The results of SATE are anonymized due to the participants' wish for confidentiality. The report showed varying results for the Java tools and that they generally struggled with increased complexity like control- and data-flow. They also showed varying results across the different forms of test cases.

AlBreiki and Mahmoud [2014] also tested SATs on a Software Assurance Reference Dataset (SARD) like the Juliet Test Suite. The static analysis tools, Yasca, CAT.NET, and FindBugs, supported both .NET and Java. The results showed that SATs could to an extent be effective to find security holes, but were not enough to uncover all vulnerabilities in software.

To discover how SATs performed for concurrency bugs and whether open-source or commercial tools were superior, Al Mamun et al. [2010] evaluated four static analysis tools. Coverity Prevent, Jtest, FindBugs, and Jlint were evaluated on programs from a benchmark as well as selected bug patterns. They concluded that it was not possible to clearly distinguish the commercial and open-source tools from each other.

Tripathi and Gupta [2014] evaluated the effectiveness and efficiency of the four SATs FindBugs, CodePro AnalytiX, UCDetector, and PMD. They were evaluated on four small Java projects with added mutant bugs. PMD proved to be the SAT that detected the most bugs, but did not detect those of high severity. CodePro Analytix detected bugs of every severity rank including the highest.

There also exists research that focuses on evaluating individual tools without comparisons such as the one by Vetro, Morisio, and Torchiano [2011]. They evaluated FindBugs in terms of false positives and precision. This was achieved by executing FindBugs on 301 different university projects. Out of 77 FindBugs bug patterns, four were reliably precise while 14 had negligible precision.

3.3. Analyzing the Implementation of Static Analysis Tools

After an extensive search, we were not able to find any previous research into analyzing the implementation of static analysis tools similarly to what we do in this master thesis. It seems no one has tried to take it a step further by explaining the results of their evaluation by analyzing the implementation. The closest we could find was the textbook explanation of the implementation of the static analysis tool CodePeer by Baird et al. [2011].

3.4. Extending and Improving Existing Static Analysis Tools

In this master thesis, we will prove our points of the implementation by modifying the SATs. Therefore, we identify previous work performed in extending and improving existing static analysis tools.

Ware and Fox [2008] conducted an evaluation of eight static analysis tools, one of them being FindBugs. They discovered that a significant number of code flaws were not detected by any of the tools. Consequently, they wrote FindBugs detectors for five of these code flaws which were not detected in their evaluation.

It is mainly FindBugs that has received enhancement from scientific research. Both Al-Ameen, Hasan, and Hamid [2011] and Vestola [2012] added new bug patterns for the static analysis tool. Both did so in an effort to improve the coverage and performance of FindBugs as they believed in its usefulness.

Shen, Zhang, et al. [2008] also extended FindBugs by adding bug patterns. These were targeted towards AspectJ, an aspect-oriented programming extension for Java. The 17 additional bug patterns were evaluated on a set of open-source AspectJ projects, and the authors were able to confirm seven already known bugs in addition to 257 unknown bugs.

Shen, Fang, and Zhao [2011] continued their efforts in FindBugs by proposing a new ranking system for it. The ranking system was based on the results of an initial analysis as well as feedback from users in whether they considered the detection to be true or false. The authors conducted an evaluation of three large applications and found that their ranking system improved both recall and precision for the top 60% of reports. The source code was posted online, but is unfortunately no longer available.

4. Pre-Study

Prior to this master thesis, we conducted a pre-study where we evaluated five different open-source IDE-integrated static analysis tools for Java. Our research consisted of two contributions, a practical evaluation of coverage and performance as well as a theoretical evaluation of usability.

The results of the pre-study have concurrently with the work on this thesis been rewritten into a scientific paper and published on its own to the Evaluation and Assessment in Software Engineering (EASE) 2019 conference [Li, Beba, and Karlsen, 2019]. EASE is ranked as an A-conference by both the Excellence in Research in Australia (ERA) [2010] and Computing Research and Education Association of Australasia (CORE) [2018]. The scientific paper is included in appendix A.

During the work on this master thesis, we discovered we had made an error during the evaluation of ESVD for the categories A5 Broken Access Control and A7 Cross-Site Scripting. We reported in our paper that CWE-23 Relative Path Traversal and CWE-36 Absolute Path Traversal were not detected by ESVD. However, after a new evaluation, both of them gets reported. All of the CWE entries in A7 Cross-Site Scripting also produced higher numbers than first reported. Our theory is that this happened because of ESVD's unstable behavior with repeated crashes and freezes. In addition to this, we discovered that ESVD does in fact cover CWE-259 Hard-coded Password despite not claiming to do so. We also discovered a bug in the behavior of Find Security Bugs for flow variant 81 in the Juliet Test Suite. Flow variant 81 is explained in Table B.2. This bug results in Find Security Bugs sometimes not reporting a true or false positive for this specific flow variant, and slightly changed the results for all injection detectors, the path traversal detectors, and the cross-site scripting detectors. All results are adjusted accordingly, and the original results can be seen in our paper in appendix A.

4.1. Coverage and Performance Evaluation

The static analysis tools we evaluated were ASIDE, LAPSE+, SpotBugs, Find Security Bugs, and ESVD. ASIDE was created in an effort to teach secure programming. The idea behind the SAT is to report the vulnerabilities directly inside the IDE while the user is typing, which was uncommon at the time. LAPSE+, on the other hand, has to be manually executed and analyzes the whole project each time. Neither of them utilizes data-flow analysis, which also leads to poor results.

SpotBugs is the spiritual successor to the popular bug detector FindBugs that has not been updated since 2015 [SpotBugs, 2018]. FindBugs is the result of the academic work of Hovemeyer and Pugh [2004] in their article "Finding Bugs is Easy." SpotBugs detects all sorts of bugs, not just security vulnerabilities. However, it is not a style checker and

focuses on bugs that actually cause errors in programs. Over 400 different bug patterns are used by SpotBugs, and they are divided into different categories [SpotBugs, 2017, 2018]. The security category has bug patterns for cross-site scripting, HTTP response splitting, path traversal, insecure passwords, and SQL injection.

Find Security Bugs is a custom plugin for SpotBugs that extends its coverage and improves its performance by adding more bug patterns. It is able to use all of the methods and functionality of SpotBugs, in addition to new functionality it provides itself. At the time of the pre-study, Find Security Bugs added 128 different bug patterns [Find Security Bugs, 2018a]. Among these were bug patterns for injection, authentication, broken access control, and cross-site scripting.

ESVD is an SAT in the form of a plugin for the Eclipse IDE. The motivation behind it is to utilize early detection to warn the user of potential security vulnerabilities while writing the code. In addition, ESVD focuses on vulnerabilities originating from user input and claims to utilize context-sensitive data-flow analysis to track the input from source to sink. The motivation behind this is to reduce the number of false positives. However, as our pre-study revealed, ESVD produced poor results and did not live up to what it claimed. A total of 11 security vulnerabilities are included, and among them are injection vulnerabilities, cookie poisoning, cross-site scripting, log forging, path traversal, and security misconfiguration.

We wanted to only evaluate the SATs for the most critical and relevant security vulnerabilities. To achieve this, we used the OWASP Top 10 list as a reference. The list is explained in more detail in section 2.2.

In order to measure performance we utilized the performance metrics used at the Static Analysis Tool Exposition (SATE) [Delaitre et al., 2018] as these are commonly used metrics that are also used in previous research such as the ones by Charest, Rodgers, and Wu [2016] and Oyetoyan et al. [2018]. The metrics are defined as follows:

- **Recall** is the percentage of vulnerabilities detected.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

- **Precision** is the percentage of detections which are true positives. It is a good indication of noise in the SAT's reports.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

- **Discrimination Rate** is the number of test cases where the SAT reports a true positive without also reporting a false positive. It tells if the tool is able to see the difference between a true and a false positive.

$$\text{Discrimination Rate} = \frac{\text{Number of Discriminations}}{\text{Number of Test Cases}}$$

To use these metrics, we needed a vulnerable code base to evaluate on that provided the necessary data to calculate the metrics. Natural code, i.e., the source code of any application, does not provide this information as one does not know how many vulnerabilities are in the source code. However, artificial code, which is usually a framework made intentionally vulnerable, does provide this information. With this in mind, using artificial code was the only suitable option for us. Which metrics are applicable are shown in Table 4.1.

Table 4.1.: The metric applicability for natural and artificial code according to Delaitre et al. [2018]. N/A means not applicable.

Metric	Natural Code	Artificial Code
Coverage	Limited	Applicable
Recall	N/A	Applicable
Precision	Applicable	Applicable
Discrimination	N/A	Applicable

We selected the Juliet Test Suite v1.3 created by NSA [2012] and NIST [2017, 2018]. It is a collection of intentionally vulnerable, artificial code. Its purpose is to serve as a testing platform for static analysis tools. It consists of over 28,000 test cases which are categorized under 112 different CWE entries [NIST, 2017]. Each test case includes exactly one vulnerability as well as at least one non-flawed construct meant to represent a potential false positive. A single CWE entry can have up to thousands of test cases spanning over simple cases, control-flow cases and data-flow cases, where the latter are more challenging to detect. The test cases in the Juliet Test Suite have both a source variant (functional variant) as well as a flow variant, as explained in the documentation by NSA [2012].

4.2. Coverage and Performance Results

The results of the evaluation are presented in Table 4.3. The numbers for Find Security Bugs does not include detections made by SpotBugs as all detectors for SpotBugs were turned off when evaluating Find Security Bugs' own detectors. Usually their results are combined when using the Find Security Bugs extension for SpotBugs, but we wanted to evaluate them individually.

A summary of the coverage is presented in Table 4.2. It shows how many categories of vulnerabilities the SATs claimed to detect and what they actually covered. A category was considered covered if the SAT got any true positives for it. By reading the table, it was clear that the coverage was generally poor. Find Security Bugs was the only SAT that had a coverage above half, with a 62% confirmed coverage. However, we would still argue this number should have been higher for an SAT. In addition to the low percentages, ESVD and LAPSE+ also had a worrying discrepancy between what they claimed and what they actually detected. We considered this worrying as it meant these SATs

could not be relied upon. By looking at Table 4.3, it became obvious that the coverage was also unevenly distributed. Certain categories, such as injection vulnerabilities, broken access control, and cross-site scripting were vastly more covered than others.

Table 4.2.: Confirmed and claimed coverage of the IDE-integrated static analysis tools. Full coverage corresponds to covering all 29 vulnerability categories.

Tools	Confirmed Coverage		Claimed Coverage	
ASIDE	12	41%	12	41%
ESVD	7	24%	13	45%
LAPSE+	8	28%	11	38%
SpotBugs	8	28%	8	28%
Find Security Bugs	18	62%	19	66%

The calculated performance metrics are presented in Table 4.4. Since ASIDE did not categorize its detections, it was impossible to verify if the detections were relevant to the test cases. This meant some irrelevant detections might have been included, leading to the true and false positive numbers in our results being higher than what ASIDE deserved. LAPSE+ required manual effort instead of having an automatic data-flow analysis implemented. We did not conduct this manual data-flow analysis and it was not included in the results.

Both ESVD and SpotBugs had a low recall, but a high precision. Opposite to this, LAPSE+ had a low precision, but a high recall. This showed how SATs have to deal with the trade-off between recall and precision, and it was clear these had chosen the opposite. This also leads to all three having a poor discrimination rate. ASIDE performed average with good results for a handful of vulnerabilities. The only SAT that performed remarkably well was Find Security Bugs. With the highest coverage, it also had a recall and precision close to 100% for most vulnerabilities with a few exceptions. It even performed well for CWE-89 SQL Injection, where the other SATs struggled to achieve a high recall while maintaining a low precision.

4.3. Usability Evaluation

The usability of the SATs were evaluated on the metrics listed in Table 4.5. All the tools had at least one CWE entry with a much greater number of false positives than true positives. ESVD and SpotBugs had a single high false positive number that skewed the false positive rate (FPR) for these two SATs. Situations like these were the reason why we used two different types of FPRs. The "averaged false positive rate" metric averaged the FPR for each entry with equal weight. The FPR of the tools were way higher than the 15% that half of developers accept according to Christakis and Bird [2016].

A countermeasure for a high false positive rate is prioritized output telling the user which detections are the most critical or confident. ESVD prioritized the detections in criticalness by using numbers, while SpotBugs and Find Security Bugs ranked them by both criticalness and confidence using words and colors. Another countermeasure for a

Table 4.3.: Detailed coverage data, showing the number of true and false positives. A hyphen (-) indicates that the plugin does not claim to cover the CWE entry. Each entry is listed with its unique ID and its total number of test cases.

CWE			IDE-Integrated Static Analysis Tools									
ID	Name		ASIDE		ESVD		LAPSE+		SpotBugs		FindSecBugs	
A1 Injection		**Total**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
78	OS Command Injection	444	185	0	49	0	444	624	-	-	380	60
89	SQL Injection	2220	3[a]	3[a]	1440	2280	2220	3060	2220	3000	1900	300
90	LDAP Injection	444	185	0	0	0	0	0	-	-	380	60
113	HTTP Response Splitting	1332	555	795	0	0	0	0	57	0	990	0
134	Use of Externally-Controlled Format String	666	148	212	-	-	-	-	-	-	462	0
643	XPath Injection	444	185	265	0	0	444	1248	-	-	380	60
A2 Broken Authentication		**Total**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
256	Unprotected Storage of Credentials	37	-	-	-	-	-	-	-	-	-	-
259	Use of Hard-coded Password	111	-	-	20	3	-	-	15	0	48	0
321	Use of Hard-coded Cryptographic Key	37	-	-	-	-	-	-	-	-	16	0
523	Unprotected Transport of Credentials	17	-	-	-	-	-	-	-	-	-	-
549	Missing Password Field Masking	17	-	-	-	-	-	-	-	-	-	-
A3 Sensitive Data Exposure		**Total**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
315	Cleartext Storage of Sensitive Information in a Cookie	37	-	-	-	-	-	-	-	-	0	0
319	Cleartext Transmission of Sensitive Information	370	-	-	-	-	-	-	-	-	259	369
325	Missing Required Cryptographic Step	34	-	-	-	-	-	-	-	-	-	-
327	Use of a Broken or Risky Cryptographic Algorithm	34	-	-	-	-	-	-	-	-	17	0
328	Reversible One-Way Hash	51	-	-	-	-	-	-	-	-	51	0
329	Not Using a Random IV with CBC Mode	17	-	-	-	-	-	-	-	-	17	0
614	Sensitive Cookie in HTTPS Session Without 'Secure' Attribute	17	-	-	-	-	-	-	-	-	16	0
759	Use of a One-Way Hash without a Salt	17	-	-	-	-	-	-	-	-	-	-
760	Use of a One-Way Hash with a Predictable Salt	17	-	-	-	-	-	-	-	-	-	-
A5 Broken Access Control		**Total**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
23	Relative Path Traversal	444	108	0	49	0	444	624	19	0	380	60
36	Absolute Path Traversal	444	108	0	49	0	444	624	16	0	380	60
566	Auth. Bypass Through User-Controlled SQL Primary Key	37	36	0	-	-	37	0	-	-	-	-
A6 Security Misconfiguration		**Total**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
395	NullPointerException Catch to Detect NULL Pointer Deference	17	-	-	0	0	-	-	-	-	-	-
396	Declaration of Catch for Generic Exception	34	-	-	0	0	-	-	-	-	-	-
397	Declaration of Throws for Generic Exception	4	-	-	0	0	-	-	-	-	-	-
A7 Cross-Site Scripting		**Total**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
80	Basic XSS	666	642	900	70	0	666	936	19	0	666	90
81	Improper Neutralization of Script in an Error Message	333	321	450	35	0	0	0	19	0	333	45
83	Improper Neutralization of Script in Attributes in a Web Page	333	108	0	35	0	333	468	19	0	333	45

[a] ASIDE generates an exception when running on these test cases.

high false positive rate is the ability to hide false positives. Both ASIDE and ESVD had the option, but the action is irreversible.

SpotBugs and Find Security Bugs were the only SATs that provided sufficient information about the problem. Find Security Bugs also gave examples of how the vulnerabilities could be fixed. ASIDE and ESVD, on the other hand, provided quick fixes through a third-party application programming interface (API). Unfortunately, the fixes were often too general to be relevant to the vulnerability.

All the tools utilized early detection except for LAPSE+, which had to be manually executed. SpotBugs and Find Security Bugs also provided this as an option. Only SpotBugs and Find Security Bugs could manually analyze a single file.

All the results are shown in Table 4.5, and more details are found in our paper.

Table 4.4.: Detailed performance data, showing recall, precision, and discrimination rate. The names of the CWE entries are slightly shortened, see Table 4.3 for the full names. A hyphen (-) indicates that the plugin does not cover the CWE.

CWE		Tools														
ID	Name	ASIDE			ESVD			LAPSE+			SpotBugs			FindSecBugs		
A1	**Injection**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
78	OS Command Injection	42%	100%	42%	11%	100%	11%	100%	42%	0%	-	-	-	86%	86%	72%
89	SQL Injection	0%	50%	0%	65%	39%	0%	100%	42%	0%	100%	43%	0%	86%	86%	72%
90	LDAP Injection	42%	100%	42%	0%	N/A	N/A	0%	N/A	N/A	-	-	-	86%	86%	72%
113	HTTP Response Splitting	42%	41%	0%	0%	N/A	N/A	0%	N/A	N/A	4%	100%	4%	74%	100%	74%
134	Externally-Controlled Format String	22%	41%	0%	-	-	-	-	-	-	-	-	-	69%	100%	69%
643	XPath Injection	42%	41%	0%	0%	N/A	N/A	100%	26%	0%	-	-	-	86%	86%	72%
A2	**Broken Authentication**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
256	Unprotected Credentials Storage	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
259	Hard-coded Password	-	-	-	18%	87%	16%	-	-	-	14%	100%	14%	43%	100%	43%
321	Hard-coded Cryptographic Key	-	-	-	-	-	-	-	-	-	-	-	-	43%	100%	43%
523	Unprotected Credentials Transport	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
549	Missing Password Field Masking	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
A3	**Sensitive Data Exposure**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
315	Cleartext Sensitive Info in Cookie	-	-	-	-	-	-	-	-	-	-	-	-	0%	N/A	N/A
319	Sensitive Cleartext Transmission	-	-	-	-	-	-	-	-	-	-	-	-	70%	41%	0%
325	Missing Required Crypto. Step	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
327	Broken/Risky Crypto. Alg.	-	-	-	-	-	-	-	-	-	-	-	-	50%	100%	50%
328	Reversible One-Way Hash	-	-	-	-	-	-	-	-	-	-	-	-	100%	100%	100%
329	Not Random IV in CBC Mode	-	-	-	-	-	-	-	-	-	-	-	-	100%	100%	100%
614	Missing 'Secure' in HTTPS Cookie	-	-	-	-	-	-	-	-	-	-	-	-	94%	100%	94%
759	One-Way Hash, no Salt	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
760	One-Way Hash, Predictable Salt	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
A5	**Broken Access Control**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
23	Relative Path Traversal	24%	100%	24%	11%	100%	11%	100%	42%	0%	4%	100%	4%	86%	86%	72%
36	Absolute Path Traversal	24%	100%	24%	11%	100%	11%	100%	42%	0%	4%	100%	4%	86%	86%	72%
566	SQL PK Auth. Bypass	97%	100%	97%	-	-	-	100%	100%	100%	-	-	-	-	-	-
A6	**Security Misconfiguration**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
395	Catching NULL Pointer Deference	-	-	-	0%	N/A	N/A	-	-	-	-	-	-	-	-	-
396	Catch for Generic Exception	-	-	-	0%	N/A	N/A	-	-	-	-	-	-	-	-	-
397	Throws for Generic Exception	-	-	-	0%	N/A	N/A	-	-	-	-	-	-	-	-	-
A7	**Cross-Site Scripting**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
80	Basic XSS	96%	42%	2%	11%	100%	11%	100%	42%	0%	3%	100%	3%	100%	88%	86%
81	Script in Error Message	32%	100%	32%	11%	100%	11%	0%	N/A	N/A	6%	100%	6%	100%	88%	86%
83	Script in Attributes in a Web Page	96%	42%	2%	11%	100%	11%	100%	42%	0%	6%	100%	6%	100%	88%	86%

Table 4.5.: Summary of the usability results.

	Metric	ASIDE	ESVD	LAPSE+	SpotBugs	FindSecBugs
FP rate	Averaged false positive rate	29%	9%	53%	7%	9%
	False positive rate of total result	50%	57%	60%	56%	13%
Tool output	What is the problem	×	✓	✓	✓	✓
	Why is it a problem	N/A	×	×	✓	✓
	How to fix the problem	N/A	×	×	×	✓
	Prioritized output	×	✓	×	✓	✓
	Quick fixes	✓	✓	×	×	×
	(E)arly or (L)ate detection	E	E	L	E/L	E/L
	Can suppress warnings	✓	✓	×	×	×
	Eclipse Environment integration	✓	✓	✓	✓	✓
	Available on Eclipse Marketplace	×	×	×	✓	×
	(I)mmediate or (N)egotiated interruptions	N	N	N	N	N
	Easily extendable	×	×	×	✓	×
	Possible to analyze single file only	×	×	×	✓	✓
	Possible to analyze single method only	×	×	×	×	×

5. Research Design and Implementation

Chapter 5 explains our research approach and implementation. In section 5.1 our motivation for this research project is presented, then our three research questions are listed in section 5.2. Our research method and design is explained in section 5.3. This includes our research strategy, our data generation and analysis approach, and our research paradigm. Finally, our research implementation is described in section 5.4.

5.1. Research Motivation

Many of the code errors today can be detected in real-time while writing the code. If a semicolon is forgotten, you immediately get a red squiggly line below the relevant part of the code. More advanced errors might not be immediately detected, but will not allow the application to compile before being fixed. Even runtime errors are often detected in a timely manner, because they occur during normal usage. Security vulnerabilities, on the other hand, get no red squiggly line, do not cause compilation errors, and can be present in an application for a long time without being detected through normal usage. When researching possible solutions to this problem, the results were less than satisfactory.

Our pre-study showed us that the coverage is unevenly distributed among the vulnerability categories, and that the performance is lacking. The very high false positive rate we saw would definitely scare away a good portion of the potential user base. Considering the devastating effects security vulnerabilities can have on software, we find that quite worrisome.

When looking for literature explaining the behavior of the SATs we evaluated in the pre-study, we found barely any related work. We found a book explaining the implementation of a different tool for a different programming language, but this was not in relation to the performance or results of the tool. Other related work have improved SATs by adding new detectors, but none have improved SATs by enhancing existing detectors. It should also be noted that neither the explanation of the implementation nor the addition of new detectors are related to security vulnerabilities, but rather towards general software bugs. We found this lack of related work disappointing, given the large impact these SATs could have not only on software developers, but also the users of software products. More details about the related work are presented in chapter 3.

Our research aims to understand the implementation of these SATs and how the implementation relates to the results from our pre-study. This will lead to an explanation of the limitations we found in the SATs with accompanying proofs for our claims. These proofs come in the form of code modifications that confirm the limitations and improve the performance. Based on our findings, we will propose recommendations for further development of static analysis tools.

5.2. Research Questions

From the research motivation mentioned in section 5.1, we have formulated three research questions that will be explored in our master thesis. The three research questions are as follows:

RQ1. How is static analysis implemented in the SATs from the pre-study?

RQ2. How can the performance of the SATs be explained by their implementation?

RQ3. How can the limitations of the SATs be addressed through proof-of-concept improvements?

5.3. Research Method and Design

This section will summarize the research method and design of our study. This includes the research strategy, data generation method, and data analysis method. In addition, this section will describe the research paradigm considered to hold true for us as the authors of this master thesis.

5.3.1. Research Strategy

A set of static analysis tools and security vulnerabilities will be included from the pre-study for further analysis. The SATs will be tested and analyzed on the selected vulnerabilities, and limitations and possible modifications will be explored. These steps are further described in section 5.4.

Our research will be answered through experiments, as defined by Oates [2005]. The experiments will focus on the relationships between cause and effect. We will be observing the outcome from us testing our hypotheses about why the SATs behave the way they do. There is before and after measurements, and the factors that can affect the results are controlled.

When explaining the SATs' implementation, we will create hypotheses about their behavior that we try to confirm or reject. Considering we have access to the source code of these SATs, we will alter the code in an attempt to prove or disprove these hypotheses. This is strongly in line with the experiments strategy described by Oates [2005].

An important aspect of good research is reproducibility. To make it easier to reproduce the results of our master thesis, a detailed implementation description and the necessary source code changes will be published. More about this will be described in section 5.4.

5.3.2. Data Generation and Analysis

The data generation method will partially consist of collecting the results from the SATs being executed on vulnerable code, and partially from documents and related research describing the design and implementation of the SATs.

RQ1 will mainly be based on the documents produced by the authors of the SATs themselves. The word "document" is used broadly speaking, including both design models, research papers, and source code. The data generated from these documents will be analyzed qualitatively.

RQ2 is partially based on the same documents as for RQ1, in addition to the results we produce when executing the SATs on vulnerable code. The data generated from the documents will still be analyzed qualitatively, while the data generated from executing the SATs will be analyzed quantitatively. Quantitatively data analysis is done through the use of mathematical approaches to calculate the coverage and performance of the SATs.

RQ3 is only based on the results we produce when executing the SATs on vulnerable code. Data generated from this will be analyzed qualitatively. This includes comparing the before and after measurements, to explore any potential changes in the results caused by the modifications to the source code.

5.3.3. Research Paradigm

In academic research, the experiment strategy is at the heart of the scientific method and positivism [Oates, 2005]. This research assumes the world to be ordered, and in which it can be investigated objectively. Claiming vulnerabilities in code are caused by specific instructions can be repeated or refuted. Claiming a static analysis tool can detect a set of vulnerable instructions can be repeated or refuted. Differences in the before and after measurements can be used as proof of our hypotheses. This approach to research aligns with the positivism paradigm. The research is based on facts, with quantitative and qualitative data which is seen as objective.

5.4. Research Implementation

As part of our research, we will conduct an implementation analysis and a limitation analysis for IDE-integrated SATs targeting security vulnerabilities. The implementation analysis will answer RQ1 and RQ2, and will uncover limitations in the implementation. These limitations are analyzed further in the limitation analysis that answers RQ3.

Section 5.4 explains our approach to implementing our research and gives insight into our thought process and decisions. Section 5.4.1 covers the research and selection of the SATs while section 5.4.2 covers the research and selection of the vulnerability categories. We structure our presentation of the SATs' performance similarly to how the Juliet Test Suite is structured, which is explained in section 5.4.3. Then we present our parsers and how they are utilized in section 5.4.4. Section 5.4.5 and 5.4.6 explain how we are presenting and conducting our implementation analysis, before section 5.4.7 ends this chapter by explaining how we are conducting the limitation analysis and how we are producing the proof-of-concept improvements.

Note that this master thesis is a continuation of our pre-study, and some decisions have already been thoroughly explained in the research paper. Although these decisions will

be mentioned, the thorough explanation will not. The reader is referred to the pre-study in chapter 4 or the paper in appendix A for the full explanation of these decisions.

5.4.1. Selection of the Static Analysis Tools

Criteria

Researching and analyzing static analysis tools is a time-consuming task. Thus, it is important to balance research time with research contribution as we select the SATs to analyze. The pre-study defines the available pool of static analysis tools, namely ASIDE, ESVD, LAPSE+, SpotBugs, and Find Security Bugs. In order to only evaluate the SATs where our research will produce a significant contribution, we decide on the following criteria:

1. The SAT must be easy to find and install.
2. The SAT must work properly with recent releases of the Eclipse IDE.
3. The SAT must use state-of-the-art detection algorithms.

These specifications are the result of a long reflection process into what we want our research contribution to be. The two first criteria are important for the user to be able to find and use the SATs. If too much effort is needed to find or install the SAT, or the user is forced to use an old release of Eclipse, the likelihood of use is drastically reduced. The third criterion is important when it comes to the contribution of our research. It is less interesting to analyze simpler detection algorithms, as they often produce worse results than state-of-the-art detection algorithms. In other words, analyzing the best performing detection algorithms is a greater contribution.

Selection

Only SpotBugs and ESVD are available on the Eclipse Marketplace, but ESVD seems to be misconfigured and impossible to directly download from the marketplace. It is possible to install ESVD by following the link present in the error message given by Eclipse. Find Security Bugs is easy to find and download from the internet, while LAPSE+ is somewhat harder to find. ASIDE does not have any executable available, and the only way to install it is to compile its source code.

Four of the five SATs can run in recent versions of Eclipse. Only LAPSE+ must be run in an older release from 2010, namely Eclipse Helios. Helios is likely to be missing important features present in the nine years since its release, and is slightly harder to find, download, and install than more recent editions of Eclipse.

ESVD, SpotBugs, and Find Security Bugs are the only three tools that use data-flow analysis. ESVD is also of special interest, as it claims to use context-sensitive DFA while producing subpar performance results. ASIDE and LAPSE+ are using older and less capable techniques of detecting security vulnerabilities, such as pattern matching.

It should be mentioned that ESVD frequently crashed during the pre-study. We decided to excuse this behavior, as our interest lies in the advanced algorithms it claims to use, and not the usability of the tool. Although this is an annoyance for us, it is uncertain if this occurs during regular use on natural code.

Based on the criteria and explanation above, the selected static analysis tools are ESVD, SpotBugs, and Find Security Bugs. Even though ESVD does not fulfill our first criterion, we still believe it is worth further analysis and we see it as a better contender than ASIDE or LAPSE+. It fulfills the third criteria which we see as more important and assert some bold claims about context-sensitive DFA. See Table 5.1 for some additional details and a summary about the SATs. Note that due to ESVD requiring changes to its output-producing source code as described in the research paper in appendix A, the SAT was acquired from its source code repository. Also note that we are using a newer version of SpotBugs than in the pre-study. The updated version of SpotBugs has not changed anything related to the vulnerabilities we are analyzing [SpotBugs, 2019a], which we have also verified by producing the same results using both versions of SpotBugs.

Table 5.1.: Information about the selected static analysis tools.

SAT	Downloaded From	Version	Version Date
ESVD	GitHub [Sampaio, 2016]	0.4.2	Jul 2016
SpotBugs	Eclipse Marketplace [SpotBugs, 2019b]	3.1.11	Jan 2019
FindSecBugs	Project Webpage [Find Security Bugs, 2019]	1.8.0	Jun 2018

5.4.2. Selection of Vulnerability Categories

Criteria

The static analysis tools have different implementations for different vulnerabilities. Because of the time-consuming nature of implementation analysis, we deem it necessary to narrow the scope from the pre-study to a smaller set of vulnerabilities. As the work from the pre-study depends on OWASP Top 10 and the Juliet Test Suite, we will continue to utilize them. The criteria and reasons for selecting OWASP Top 10 and the Juliet Test Suite can be found in the pre-study in chapter 4.

To narrow the list of vulnerabilities to analyze, we define the criterion that the vulnerability must allow us to compare the results and implementations between multiple static analysis tools. This criterion is based on our desire to accumulate information that can be used to present a list of shared and individual strengths and limitations for all of the static analysis tools. We believe it is interesting to see how the SATs approach the same task, and if they do so differently. This information will uncover how different solutions perform and can guide developers in the future to make educated decisions when developing SATs.

Table 5.2.: This table shows the selected vulnerabilities in yellow. The original table can be seen in Table 4.4. Only vulnerabilities where at least two of the SATs claim to cover the vulnerability are included.

CWE		Tools								
ID	Name	ESVD			SpotBugs			FindSecBugs		
A1	**Injection**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
78	OS Command Injection	11%	100%	11%	-	-	-	86%	86%	72%
89	SQL Injection	65%	39%	0%	100%	43%	0%	86%	86%	72%
90	LDAP Injection	0%	N/A	N/A	-	-	-	86%	86%	72%
113	HTTP Response Splitting	0%	N/A	N/A	4%	100%	4%	74%	100%	74%
134	Externally-Controlled Format String	-	-	-	-	-	-	69%	100%	69%
643	XPath Injection	0%	N/A	N/A	-	-	-	86%	86%	72%
A2	**Broken Authentication**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
256	Unprotected Credentials Storage	-	-	-	-	-	-	-	-	-
259	Hard-coded Password	18%	87%	16%	14%	100%	14%	43%	100%	43%
321	Hard-coded Cryptographic Key	-	-	-	-	-	-	43%	100%	43%
523	Unprotected Credentials Transport	-	-	-	-	-	-	-	-	-
549	Missing Password Field Masking	-	-	-	-	-	-	-	-	-
A3	**Sensitive Data Exposure**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
315	Cleartext Sensitive Info in Cookie	-	-	-	-	-	-	0%	N/A	N/A
319	Sensitive Cleartext Transmission	-	-	-	-	-	-	70%	41%	0%
325	Missing Required Crypto. Step	-	-	-	-	-	-	-	-	-
327	Broken/Risky Crypto. Alg.	-	-	-	-	-	-	50%	100%	50%
328	Reversible One-Way Hash	-	-	-	-	-	-	100%	100%	100%
329	Not Random IV in CBC Mode	-	-	-	-	-	-	100%	100%	100%
614	Missing 'Secure' in HTTPS Cookie	-	-	-	-	-	-	94%	100%	94%
759	One-Way Hash, no Salt	-	-	-	-	-	-	-	-	-
760	One-Way Hash, Predictable Salt	-	-	-	-	-	-	-	-	-
A5	**Broken Access Control**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
23	Relative Path Traversal	11%	100%	11%	4%	100%	4%	86%	86%	72%
36	Absolute Path Traversal	11%	100%	11%	4%	100%	4%	86%	86%	72%
566	SQL PK Auth. Bypass	-	-	-	-	-	-	-	-	-
A6	**Security Misconfiguration**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
395	Catching NULL Pointer Deference	0%	N/A	N/A	-	-	-	-	-	-
396	Catch for Generic Exception	0%	N/A	N/A	-	-	-	-	-	-
397	Throws for Generic Exception	0%	N/A	N/A	-	-	-	-	-	-
A7	**Cross-Site Scripting**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
80	Basic XSS	11%	100%	11%	3%	100%	3%	100%	88%	86%
81	Script in Error Message	11%	100%	11%	6%	100%	6%	100%	88%	86%
83	Script in Attributes in a Web Page	11%	100%	11%	6%	100%	6%	100%	88%	86%

Selection

When filtering the vulnerabilities from the pre-study for the aforementioned criterion, we get the vulnerabilities highlighted in Table 5.2. All of these vulnerabilities are claimed to be covered by at least two of the SATs. They are described in section 2.2. 7,215 distinct variations of the 11 selected vulnerabilities exist in the Juliet Test Suite. Considering the fact that we are executing three different SATs on these 7,215 test cases, this results in 20,313 different vulnerability reports, excluding the 1,332 vulnerabilities which SpotBugs does not claim to cover.

5.4.3. How Juliet Test Suite is Structured

When evaluating the SATs, we utilize the natural structure of Juliet Test Suite. There are multiple test cases for each vulnerability, and each of these test cases has a unique composition of which source is used and what control- and data-flow complexity is added. These will be referred to as respectively source variants and flow variants.

A source variant is a variant of a test case using a specific source. The Juliet Test Suite consists of similar cases where the only difference being what source is used. This way, it is possible to discover if an SAT is not able to detect a specific source. Figure 5.1 and 5.2 are two examples of the source variants Connect TCP and File. These lines of code retrieve potentially vulnerable input from respectively a TCP connection and a File.

```
/* SOURCE VARIANT: Read data using an outbound tcp connection */
socket = new Socket("host.example.org", 39544);
reader = new InputStreamReader(socket.getInputStream(), "UTF-8");
readerBuffered = new BufferedReader(reader);
data = readerBuffered.readLine();
```

Figure 5.1.: Simplified version of the Connect TCP source variant used in the Juliet Test Suite.

```
/* SOURCE VARIANT: Read data from a file */
file = new File("C:\\data.txt");
stream = new FileInputStream(file);
reader = new InputStreamReader(stream, "UTF-8");
readerBuffered = new BufferedReader(reader);
data = readerBuffered.readLine();
```

Figure 5.2.: Simplified version of the File source variant used in the Juliet Test Suite.

A flow variant of a test case includes specific control-flow statements that add complexity to the test case. Every vulnerability category include the same 37 flow variants. The first flow variant is called the baseline and includes no added complexity. It is the easiest of the test cases. The next group is called control-flow cases. There are 18 of them, and they hide the source or sink within control-flow statements to make them harder to detect. Figure 5.3 is an example of this where the source from Figure 5.1 is wrapped inside an `if`-statement. Figure 5.1 represents a baseline case. The final group is called data-flow cases. Figure 5.4 shows how the vulnerable data is retrieved from the source in one class, and used in a sink in another class. Data-flow cases can send the data between methods or classes, and are the most complex test cases to detect as they require a sophisticated algorithm to track the flow of the vulnerable input. More

information on each specific flow variant is located in appendix B.

Control-Flow Variant 02

```
String data;

/* CONTROL-FLOW VARIANT: If-statement with boolean literal */
if(true) {
    /* SOURCE VARIANT: Read data using an outbound tcp connection */
    socket = new Socket("host.example.org", 39544);
    reader = new InputStreamReader(socket.getInputStream(), "UTF-8");
    readerBuffered = new BufferedReader(reader);
    data = readerBuffered.readLine();
}
```

Figure 5.3.: Simplified version of control-flow variant 02 used in the Juliet Test Suite. It is a more complex test case than the corresponding baseline case in Figure 5.1.

Data-Flow Variant 51 - Class A

```
/* SOURCE VARIANT: Read data using an outbound tcp connection */
socket = new Socket("host.example.org", 39544);
reader = new InputStreamReader(socket.getInputStream(), "UTF-8");
readerBuffered = new BufferedReader(reader);
data = readerBuffered.readLine();

/* DATA-FLOW VARIANT: Send data from one class to another */
(new ClassB()).badSink(data);
```

Data-Flow Variant 51 - Class B

```
public void badSink(String data ) {
    ...
    /* SINK: Relative Path Traversal */
    File file = new File(root + data);
    ...
}
```

Figure 5.4.: Simplified version of data-flow variant 51 used in Juliet Test Suite. Class A retrieves the input from the source and sends it to Class B where it used in a sink. It is a more complex test case than the corresponding baseline case in Figure 5.1 and control-flow case in Figure 5.3.

5.4.4. How Parsers are Used to Facilitate the Analysis

Before carrying out the implementation analysis, the static analysis tools have to be installed and the parsers must be able to understand the output of the SATs. This process is explained in the research paper in appendix A. This section will only discuss what is new for the master thesis. The modifications to ESVD are the same as for the pre-study, while SpotBugs and Find Security Bugs do not require modifications to properly function with the parsers.

In order to automate the process of testing the static analysis tools, we develop parsers that analyze the raw output data from the SATs. Without an automated way of analyzing the results, we would have to manually analyze at least 20,313 vulnerability reports, which is not only error-prone, but also too time-consuming to be feasible for this thesis. Considering any proof-of-concept improvements for RQ3 will result in new vulnerability reports, this could easily total over 100,000 vulnerability reports, showing the significance of the parsers. The parsers share some similarities with the parsers from our pre-study, but are now capable of producing more comprehensive results. The new and improved parsers can separate results based on source variants from the Juliet Test Suite, merge similar categories, and present more information about available sources and sinks. As can be seen in Figure 5.5, the parsers calculate true and false positives, as well as false negatives, recall, precision, and the discrimination rate for each flow variant and for each source variant.

The output from ESVD is difficult to export and automatically parse, as opposed to the detailed output that can be exported from SpotBugs and Find Security Bugs. We made slight modifications to how ESVD presents its output by also including more details about its detections. We are particularly careful to only change the textual output from the SAT, without changing any logic that could alter the detection algorithm. The modified source code is available at `https://github.com/Beba-and-Karlsen/ide-plugins-modified`, and is identical to the modifications used in the pre-study.

The parsers are written in Python 3 with modularity in mind. There are two different SAT-specific output readers, called `esvd.py` and `spotbugs.py`. As Find Security Bugs is a plugin for SpotBugs, both output their results in the same format. These SAT-specific output readers are necessary to interpret the reports produced by the SATs. The two output readers then send their results to `plugincommon.py` that takes care of the remaining result analysis. The parsers can be explained as executing the following steps:

1. **Read vulnerability reports** - The parser runs through the file containing the vulnerability reports, extracting relevant information such as file name, CWE ID, vulnerability category, and in which test case it is detected.

2. **Filter vulnerability reports** - Then, the results are sent to `plugincommon.py` that checks whether each vulnerability is a true positive, false positive, or not relevant. It then adds it to the respective source variant list. Whether it is a true or false positive is based on the method it is detected in. All flawed methods

```
python spotbugs.py .\FindSecBugs_injections\cwe643_orig\
Detailed output written to
 ↪  .\FindSecBugs_injections\cwe643_orig\parser_report.csv
######################
CWE643
--------------------
####### Environment, Property
Type            TP      FP      FN      rec     prec    disc
Baseline        0       0       1       0%      0%      0%
Control-Flow    0       0       18      0%      0%      0%
Data-Flow       5       5       13      28%     50%     0%
Total           5       5       32      14%     50%     0%
####### File, PropertiesFile, URLConnection, connect_tcp, console_readLine,
 ↪  database, getCookies, getParameter, getQueryString, listen_tcp
Type            TP      FP      FN      rec     prec    disc
Baseline        1       0       0       100%    100%    100%
Control-Flow    18      0       0       100%    100%    100%
Data-Flow       18      5       0       100%    78%     72%
Total           37      5       0       100%    88%     86%
####### Total #######
Type            TP      FP      FN      rec     prec    disc
Baseline        10      0       2       83%     100%    83%
Control-Flow    180     0       36      83%     100%    83%
Data-Flow       190     60      26      88%     76%     60%
Total           380     60      64      86%     86%     72%
####### Other info #######
Not relevant: 0
Total numb. of vuln. in Juliet CWE: 444
Total numb. of source var. in Juliet CWE: 12
Total numb. of sinks per source var. in Juliet CWE: 1
```

Figure 5.5.: An example of output from the parsers when executed on the XPath results produced by Find Security Bugs. Calculates and divides the results into categories of source and flow variants, and calculates the total of all source and flow variants.

in Juliet Test Suite are named in an identifiable way. The vulnerability report is deemed relevant if the vulnerability category corresponds to the CWE entry of the test case. This is checked by comparing the vulnerability category to a set of predefined categories provided by the SAT developer.

3. **Calculate test results** - When the filtering is completed, `plugincommon.py` calculates the number of true positives, false positives, false negatives as well as the recall, precision, and discrimination rate for each source and flow variant.

4. **Print and log results** - In the end, the parser prints the results to screen and logs

all results to a log file. The final output of running the parser for Find Security Bugs on the XPath test cases is shown in Figure 5.5.

The source code is available at `https://github.com/Beba-and-Karlsen/ide-plugin-parsers/tree/master/parser-for-master-thesis`.

5.4.5. RQ1 and RQ2: How to Present the Implementation Analysis Results

Analyzing the implementation of large and advanced static analysis tools will generate a vast amount of data. Therefore, it is important to create clear guidelines on how this information should be presented. A decision on how comprehensive the implementation explanation should be is also required. On the one hand, it is possible for us to explain the implementation in terms that are specific to a particular SAT, requiring the reader to have extensive and intimate knowledge of the SAT's implementation. On the other hand, the information can be presented in a way that requires the reader to only have knowledge of general techniques and algorithms related to static analysis. We choose the latter, as we want our research to be generalizable and contribute to the entire SAT community. Thus, we will translate implementation specific expressions and names into terms that can be understood by anyone with general knowledge about static analysis.

In addition to the textual explanation mentioned above, we will also present numerical and quantitative results in the form of result tables. These tables will contain the performance results of the detectors. The results are based on our findings from the pre-study, but with a finer granularity. More detail on how the results are presented is available in the introduction of section 6.2.

5.4.6. RQ1 and RQ2: Conducting the Implementation Analysis

Analyzing the implementation of static analysis tools is a difficult task. In addition to reading and analyzing the source code directly, it can be useful to examine the documentation. Unfortunately the literature and documentation available are limited and incomplete. Due to this, most of the implementation analysis is carried out by studying the source code and executing the SATs on the vulnerable code.

Our study consists of executing the static analysis tools, parsing the output, identifying what works and fails, and ascertaining why. The SATs will analyze individual vulnerability categories in the Juliet Test Suite, in some cases multiple times to ensure there is no problem during execution. The detection results will then be extracted and given to the parser which shows how the SAT performs for each source and flow variant. The parser also outputs a readable file of all detections, including the test case identifier where it was reported and exactly what was reported. The parser helps us see where the SAT performs well or poorly, which serves as a starting point for what to look for in the SAT's implementation. Based on this, we form hypotheses about the implementation that we later test by making modifications to the source code of the SAT. The modified source code is compiled, and the SAT is yet again executed on the same vulnerability as earlier. This process allows us to identify limitations and clearly see the effects of modifications to the detection algorithms.

Find Security Bugs has an inconsistent behavior regarding one particular test case in the Juliet Test Suite for all of the injection detectors, in addition to the path traversal and cross-site scripting detectors. It sometimes detects a true and a false positive, sometimes only a true or a false positive, and sometimes nothing. We reported this changing detection result to the Find Security Bugs community in February 2019, but it has not been fixed so far. We choose to include any true or false positive that occurs at least once for these test cases. Considering that this affects both true and false positives, and that the detector's implementation is created in such a way that it is supposed to generate both a true and false positive on each of the test cases [Arteau, 2019], we found this to be the fairest approach to deal with this bug.

ESVD crashes if the size of the project is too large, such as the SQL injection test cases. We circumvented this problem by dividing the test code into smaller portions which we tested individually. This does not change the final results, as the individual test results are combined afterward. However, ESVD would still crash on these smaller portions, requiring us to consistently restart the Eclipse IDE. We were careful to only include results when ESVD completed the analysis.

5.4.7. RQ3: Conducting the Limitation Analysis and Producing Proof-of-Concept Improvements

Analyzing the implementation in search of limitations requires some sort of verification of the results. This verification will take the form of proof-of-concept improvements, where modifications to the source code will be used to prove that limitations are present. The goal of RQ3 is not the improvement itself, but rather to describe the effect of the limitations and the feasibility of the described solutions. As such, a proof-of-concept improvement to the source code is a well-suited tool. This approach also brings authenticity to the results of RQ2, as it not only relates to limitations but also to the implementation of the SATs.

The SATs' source code will be downloaded from their respective code archives and will at first be compiled without changes to ensure that the source code correctly assembles into the same executable provided by the developers. This approach is especially important considering improvements by the developers themselves may have been added into the code archives, but not yet included into the distributed executables.

When we have ensured that the source code is capable of presenting a comparison baseline, the code changes can take place. All changes to the source code will be saved and uploaded to our GitHub page for reproducibility. Reproducibility is an essential aspect of trustworthy research, which is why it is important for us to publish all of our data generation points. All of the code modifications are available at `https://github.com/Beba-and-Karlsen/sat-limitation-proofs`.

Restricting the scope of limitation proofs is important when defining how far we will go to verify our analysis. It is not necessary to implement an advanced DFA detector from scratch to prove that another detector does not utilize data-flow analysis. On the other hand, claiming that missing sinks or improper taint analysis leads to reduced detection capabilities requires a proof-of-concept code alteration. Our proof-of-concept

improvements will be limited to existing code within the individual detectors, and will not extend to significant re-writes of the underlying frameworks. It is important that the improvements are limited to proving our limitation claims.

Situations may arise where limitations potentially have two conflicting improvements which cannot be combined. An example of such a situation is cases where one improvement can result in increased true and false positives, while another improvement results in decreased true and false positives. Both of these alterations can be desirable for different user groups. Some users can accept a higher false positive rate in exchange for a higher recall, while other users want to sacrifice some true positives in exchange for better precision. In cases like this, we will present both possibilities.

To present the limitation proofs, we will show both the original results and the results of the altered detectors. The results of the original detectors will be presented as a part of RQ2, while the results of the improved detectors belong to RQ3. The limitations will also be textually discussed for RQ3, as opposed to only showing numerical data of the improvements.

6. Research Results

This chapter will present the results of our research. Section 6.1 presents the results of RQ1, describing how static analysis is implemented in the SATs. While the general implementation is described in section 6.1, the implementation details specific to the individual detectors are presented in section 6.2 which also presents the results of RQ2. Section 6.3 answers RQ3 by addressing the implementation limitations, including proof-of-concept improvements. Any code modifications used as a proof can be found at `https://github.com/Beba-and-Karlsen/sat-limitation-proofs`.

6.1. RQ1: How is Static Analysis Implemented in the SATs from the Pre-Study?

Section 6.1 will give insight into the implementation of the SATs covered in this thesis. Section 6.1.1 will explain the implementation of SpotBugs, while Find Security Bugs is covered in section 6.1.2. Finally, the implementation of ESVD is explained in section 6.1.3.

6.1.1. SpotBugs

SpotBugs utilizes *bug patterns* to discover bugs. A bug pattern is defined as "a code idiom that is likely to be an error" by Hovemeyer and Pugh [2004]. This is done by analyzing the Java bytecode using the Byte Code Engineering Library (BCEL).

Both control-flow and data-flow analysis is used by SpotBugs and implemented as part of an internal framework [Hovemeyer and Pugh, 2004]. This enables the detectors to focus on the detection aspect and eases the work of constructing new detectors.

SpotBugs categorizes each bug occurrence into a rank category and a confidence category. The rank indicates how severe the bug is and can be of concern, troubling, scary, or scariest in order of increased severity. The default setting of SpotBugs is to report on all rank categories, but it is possible to modify the rank required and whether it should be reported as an error or warning. The confidence indicates how confident SpotBugs is in the bug's authenticity and can be either low, medium, or high. SpotBugs does not report vulnerabilities with low confidence by default, but it is possible to change the confidence requirement.

How SpotBugs Utilizes Bytecode and BCEL

As Lindholm et al. [2018, ch. 1.2] explain in the specification, the Java Virtual Machine (JVM) used to execute Java programs does not understand the Java programming

```java
public void bad() throws Throwable {
    String data;
    if (true) {
        data = "7e5tc4s3";
    } else {
        data = null;
    }

    Connection connection = null;
...
```

```java
public void bad() throws Throwable {
    String data = "7e5tc4s3";
    Connection connection = null;
...
```

Figure 6.1.: The Java compiler removes a conditional statement that always takes one specific branch.

language. Instead, it has its own set of instructions called Java bytecode. In order to execute Java programs, a compiler must first generate bytecode by interpreting `.java` files. The result is `.class` files consisting of bytecode corresponding to each compiled `.java` file. Instead of working directly with the Java bytecode, it is possible to use the Byte Code Engineering Library (BCEL) which is part of the Apache Commons project [Apache, 2019]. It is a library that creates abstractions of the bytecode in the form of objects and is intended to ease working with Java bytecode.

As mentioned, SpotBugs analyzes Java bytecode instead of the Java programming language. This is done by using the abstractions provided by BCEL. An advantage of analyzing bytecode is that the compiler will optimize parts of the code. A relevant example is `if`-statements that will always be `true`. As mentioned, `if`-statements are a type of control-flow statements that fork the order of execution. However, if an `if`-statement is always `true`, the compiler will exclude the `if`-statement in the bytecode and only include its content as shown in Figure 6.1. However, if the `if`-statement can either be true or false, e.g., by using a variable, the `if`-statement remains in the bytecode as in Figure 6.2. By looking at the code, it is clear the `if`-statement is always true as the `privateTrue` variable is never changed. However, the compiler does not take this into consideration and only considers the fact that variables are mutable, so it cannot be certain.

In the Juliet Test Suite, a total of 8 control-flow variants are optimized by the Java

```java
private boolean privateTrue = true;

/* uses badsource and badsink */
public void bad() throws Throwable {
    String data;
    if (privateTrue) {
        data = "7e5tc4s3";
    } else {
        data = null;
    }
...
```

```java
private boolean privateTrue = true;

public void bad() throws Throwable {
    String data;
    if (this.privateTrue) {
        data = "7e5tc4s3";
    } else {
        data = null;
    }
...
```

Figure 6.2.: The Java compiler does not remove a conditional statement that always takes one specific branch due to the theoretical possibility it can take another branch.

compiler which removes the control-flow statement. This way, these test cases will be similar to the baseline for SpotBugs, and it will detect the vulnerability as long as it detects the baseline. It does not need to utilize control-flow analysis. The affected flow variants of the Juliet Test Suite are listed in Table 6.1. More information on the flow variants of the Juliet Test Suite is located in appendix B.

Table 6.1.: Control-flow variants for the Juliet Test Suite that are optimized by the Java compiler and are thus at the same level of complexity as the baseline for SpotBugs.

Flow Variant	Condition of `if`-Statement
02	The `boolean` value `true`.
03	The equation `5==5`.
04	A `private static final` constant set to the `boolean` value `true`.
06	An equation between a `private static final` constant set to `5` and the `int` value `5`.
09	A `public static final` constant from another class set to the `boolean` value `true`.
13	An equation between a `public static final` constant from another class set to `5` and the `int` value `5`.
Flow Variant	**Description**
16	Both the bad and good source are encapsulated in `while(true)`-statements that loops once.
17	The sources are not encapsulated, however, the sink is encapsulated in a `for`-statement that loops once.

6.1.2. Find Security Bugs

Find Security Bugs is very similar to SpotBugs when it comes to its implementation. It mostly consists of added vulnerability detectors based on the functionality provided by SpotBugs. In addition to the many new detectors, it also adds techniques for detecting new classes of vulnerabilities. Taint analysis is one of these new techniques, and is applied by many detectors to find connections between provided sources and sinks where tainted data is handled. There are some implementations of similar techniques in SpotBugs, but these are highly specific to the detector using them, e.g., a taint analysis algorithm that only works for one vulnerability. The general implementation of taint analysis in Find Security Bugs is more capable than those found in SpotBugs and allows many different detectors to use this algorithm.

Many of the detectors in Find Security Bugs use resource files to list their vulnerable sources and sinks. See Figure 6.3 for an example of a resource file. Detectors will read and use these lists when scanning the code. This makes it possible to easily add or update the detectors without changing the detector's code itself. A situation where this will become useful is in cases where a new vulnerability is discovered by the security community. In such a case, all that is needed to add the new vulnerable sink is to append a single line to the resource file.

Test-driven development is at the core of Find Security Bugs. Every detector needs a set of vulnerable and non-vulnerable sample code which the detector will need to

```
java/security/KeyStore.load(Ljava/io/InputStream;[C)V#0
javax/crypto/spec/PBEKeySpec.<init>([C)V#0
javax/crypto/spec/PBEKeySpec.<init>([C[BI)V#2
javax/crypto/spec/PBEKeySpec.<init>([C[BII)V#3
java/net/PasswordAuthentication.<init>(Ljava/lang/String;[C)V#0
javax/security/auth/callback/PasswordCallback.setPassword([C)V#0
java/security/KeyStore$PasswordProtection.<init>([C)V#0
javax/security/auth/kerberos/KerberosKey.<init>(Ljavax/security/auth/k
javax/net/ssl/KeyManagerFactory.init(Ljava/security/KeyStore;[C)V#0
```

Figure 6.3.: A resource file from Find Security Bugs used to detect hard-coded passwords. The file is called `password-methods-all.txt` and contains vulnerable sinks. The sink in the first line is referencing the `load(...)` method of the `java.security.KeyStore` class, where the `load(...)` method accepts an `InputStream` as its first argument and a `char array` as its second argument. The "`V`" close to the end defines the sink as a void method, i.e. it does not return anything. The "`#0`" at the end of the line defines that it is the first argument (from the right) that is injectable, in this case the `char array`.

detect and ignore, respectively [Find Security Bugs, 2018b]. When Find Security Bugs is compiled, all detectors are evaluated using their sample code, and compilation will stop if these tests do not pass. The test-driven development provides a higher code quality that is more resistant towards bugs being introduced later in the development process, as changes still need to pass the previously written test cases.

In addition to supporting Java code analysis, Find Security Bugs has support for Apache Groovy, Scala, and Kotlin. These three are separate programming languages that compile into JVM bytecode, making it possible for Find Security Bugs to utilize much of the existing functionality of SpotBugs to detect new bug patterns in these languages. Although Java detectors cannot be directly reused for e.g., Scala, the underlying analysis techniques can be reused as a result of Find Security Bugs analyzing JVM bytecode as opposed to analyzing Java source code. Find Security Bugs also supports J2EE, the enterprise edition of Java.

The compiler optimization described in section 6.1.1 for SpotBugs has the same effect on Find Security Bugs. This results in some control-flow variants in the Juliet Test Suite being optimized in such a way that control-flow analysis is not required to properly analyze the test case. See the example in Figure 6.1 and the full list of optimized control-flow variants in Table 6.1.

Implementation of Injection Detector and Taint Analysis

The underlying logic for the injection detector in Find Security Bugs is a general taint detector. See the data-flow and taint analysis descriptions in section 3.1 for an introduc-

tion to taint analysis, and see Figure 6.4 for a simplified diagram of the taint analysis implementation in Find Security Bugs. Many other detectors are based on this general injection detector, including some of the hard-coded password detectors, the path traversal detector, the cross-site scripting detector, and of course the detectors for all of the different injection vulnerabilities. All of the injection detectors use this general taint detector with only small changes, such as defining which vulnerable sinks are relevant and some logic to determine the confidence of the detection.

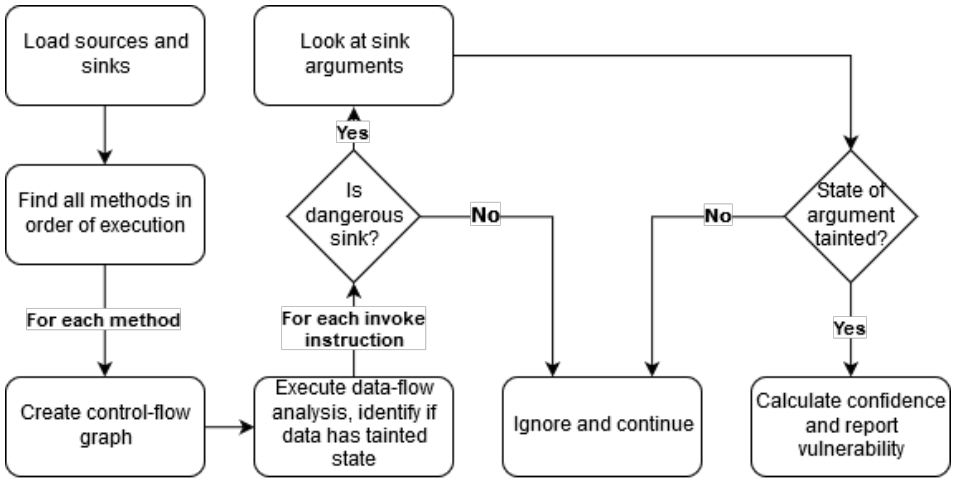

Figure 6.4.: A simplified diagram of the taint analysis implementation in Find Security Bugs.

The taint detector analyzes all of the methods in each class in the order of which they are executed. For each method, a control-flow graph is created, and a data-flow analysis is executed where each invoke instruction in the control-flow graph is analyzed. If the invoke instruction matches a known dangerous sink from the configuration files, it will look at the parameters and potentially report it as a vulnerability.

Possible tainted data is given properties using *tags*. There are many different reasons a tag can be assigned to a possible taint, but in the case of SQL injection taint detection, a tag is assigned either for saying the data has been properly sanitized or that the data contains apostrophes which have been properly encoded to not interfere with an SQL query's apostrophes. As an example, take the `encodeForSQL(Codec, String)` method from the OWASP Enterprise Security API [OWASP, 2018b]. This method will sanitize the data and make it safe from SQL injections, which results in the sanitized data being tagged as safe from SQL injections. Another similar method is the `escapeSql(String)` method from Apache Commons Lang [Apache, 2018].

In addition to tags, possible tainted data has a *state*. The state can be either tainted, unknown, safe, null, or invalid. A hard-coded integer will automatically be given a safe state, while the configuration file can define that data from a list of specific methods is tainted. An example of a method output configured to automatically be given a tainted state is the `Cookie.getName()` method from the `javax.servlet.http` package, as a

cookie name can be changed by a malicious user.

When all invoke instructions have been compared to the list of vulnerable sinks, and the tags and states have been updated, the detector will choose what to report and which confidence ranking will be assigned to the vulnerability. It is very common for detectors based on the injection and taint analysis code to override the confidence logic. If not overridden, the confidence will be given as follows:

- If the state is set to tainted, set the confidence to high;
- If the state is set to unknown or invalid, set the confidence to normal;
- If the state is set to safe, ignore it.

6.1.3. ESVD

The implementation of ESVD is a proof of concept and not a finished product [Sampaio and Garcia, 2019]. This could be the reason for why it performed poorly in our pre-study, however, we will still take a thorough look at its implementation. Unfortunately, it is difficult for us to know whether limitations are due to poor implementation or because it is unfinished. Be that as it may, our goal is not to judge its implementation, but rather to discover good implementation ideas and limitations in order to contribute with ideas to improve the state of the art.

The architecture of ESVD consists of four main components: the manager, the reporter, the analyzer, and verifiers. The vulnerability detection occurs in the verifiers, and there is one for each type of security vulnerability. They are similar in function to the detectors of SpotBugs and Find Security Bugs. However, ESVD analyzes Java code instead of bytecode. All verifiers use the same detection algorithm based on tracking vulnerable input from a source to a sink. However, both the verifiers for SQL injection and security misconfiguration have additional detection methods. It is the job of the analyzer to keep track of the verifiers. It represents a category of verifiers, e.g., all verifiers detecting security vulnerabilities. The reporter takes the results from the analyzer and reports it to the end-user within the IDE. ESVD allows for multiple different analyzers and reporters, but the current implementation only has one of each. The manager keeps track of the analyzer and the reporter as well as the settings chosen by the end-user. An illustration of the architecture is shown in Figure 6.5.

ESVD utilizes context-sensitive data-flow analysis (DFA). Because of the resource demanding nature of context-sensitive DFA, coupled with the fact that ESVD utilizes early detection and constantly runs in the background, ESVD does not scan every file except for the first execution. By remembering some of the information from each scanned source code file, it only re-scans files that have been modified. This greatly reduces the number of files that are scanned continuously in the background, thereby reducing the resource demand of the tool.

During analysis, some source code can cause an infinite loop, resulting in the tool crashing. To combat these cases, the ESVD algorithm contains a recursion control. When recursion is detected, ESVD will not scan the method invocation repeatedly.

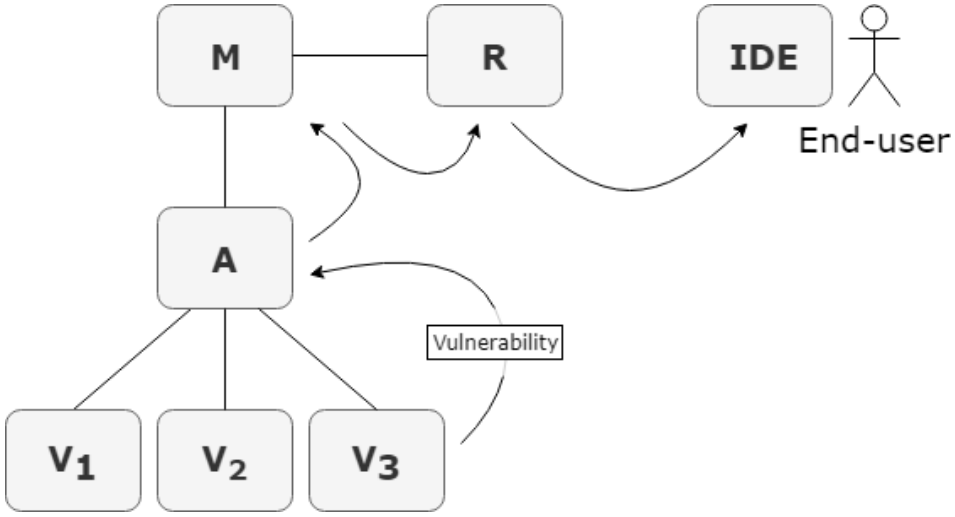

Figure 6.5.: The relation between the verifiers (V), analyzer (A), manager (M), and reporter (R) of ESVD as well as how a vulnerability is sent from a verifier to the IDE of the end-user.

The way ESVD is implemented causes it to depend heavily on the Eclipse API. It uses abstractions of code constructs provided by the Eclipse API to scan through the files. This means ESVD only works as an Eclipse plugin.

As mentioned, ESVD has a single algorithm which detects all vulnerabilities that it covers. This algorithm identifies sources and sinks and looks for vulnerable paths between them. ESVD analyzes the code method-by-method. All of the code in the method is analyzed, however, ESVD only cares about method invocations. It finds these by step-wise narrowing down the code abstractions provided by the Eclipse API. For example, when it analyzes an `if`-statement, it will narrow it down to analyze the `if`-expression and the content of the `if`-statement.

When a method invocation is found, it is inspected. ESVD checks if it is either a source, a sink, or a sanitization method. If the method invocation is a sink, the following steps are executed:

1. The parameters of the method invocation are retrieved.
2. Each parameter is compared with a list of what sort of values they are allowed to contain.
3. If a parameter has restrictions, e.g., requiring sanitization or not allowing string concatenations, it is further inspected to check if it originates from a vulnerable source. If the user has ignored the vulnerability by adding an annotation in the code, the inspection does not occur.
   3.1 If the parameter originates from a vulnerable source, ESVD marks the data-flow deriving from the source as vulnerable.

3.2 If the parameter passes through a sanitization-point, the inspection stops and ESVD deems it as not a vulnerability.

4. If the data-flow leading up to the sink is marked as vulnerable, i.e. it originates from a vulnerable source, the vulnerability is reported.

In order to detect sources, sinks, and sanitization-points, ESVD uses XML-files containing a resource list for each one. Figure 6.6 shows the first two entries in the source list. Sources and sanitization-points are universal for all kinds of vulnerabilities, while sinks are unique for each type of vulnerability. When a method invocation is found as explained above, the method signature is compared with these resource lists. These lists include 75 sources, 141 sinks, and 52 sanitization-points [Sampaio and Garcia, 2016].

entry_point.xml

```
<!-- javax.servlet.ServletRequest -->
<entrypoint id="01">
  <qualifiedname>javax.servlet.ServletRequest</qualifiedname>
  <methodname>getAttribute</methodname>
  <parameters type="java.lang.String" />
</entrypoint>
<entrypoint id="02">
  <qualifiedname>javax.servlet.ServletRequest</qualifiedname>
  <methodname>getAttributeNames</methodname>
</entrypoint>
```

Figure 6.6.: The first two entries of the XML-file containing sources in ESVD. Each of the sources are specified by package name, class name, method name, and what parameters it has. The first source method has one parameter which is a string, while the second source method has none. Note that ESVD refers to a source as an *entry-point*.

A variable receiving input from a vulnerable source is considered tainted. By using what is called tainted propagation, every other element in contact with this tainted element is also marked as tainted. Elements flowing through containers, such as arrays or lists, are automatically marked as tainted if one of the other elements in the container is tainted. The current implementation of ESVD is not able to correctly make a distinction between separate elements in a container, which leads to an increase in false positives.

6.2. RQ2: How can the Performance of the SATs be Explained by Their Implementation?

In this section, we take a closer look at how the test results in the pre-study can be explained by the detector implementations used by the SATs. Each section covers the

implementation of a single SAT for a type or group of vulnerabilities. Sections 6.2.1 to 6.2.4 cover the SpotBugs detectors, sections 6.2.5 to 6.2.8 cover the detectors for Find Security Bugs, while 6.2.9 to 6.2.12 cover ESVD's detectors. The detectors are presented in the order of injection, hard-coded password, path traversal, and then cross-site scripting for each SAT.

RQ2 focuses on the security aspect of the implementations and does not put emphasis on the programming approach. Thus, the description avoids the use of terms as well as names specific to the different SATs. This is due to the fact that this is an analysis of the static analysis tools' ability to detect vulnerabilities, not an analysis into how they are structured.

When presenting the results, the tables are structured as follows:

- Each table presents the results for a single SAT specified in its table description.
- Each table presents the results for one or more vulnerability categories separated into different columns. Which vulnerability category a column represents is specified by its CWE number and name in the top two rows of the table.
- Each vulnerability category has results for recall, precision, and discrimination rate. These are separated into three different columns.
- The table is divided by different source variants, indicated by a blue header row. Multiple source variants are grouped together if they have identical results.
- For each group of source variants there are four rows of results. The first three are separated into different flow variants. These are the single *baseline* case, the 18 *control-flow* cases, and the 18 *data-flow* cases. The fourth line is a subtotal of all the flow variants for this group of source variants.
- Finally, the last blue header row contains the totaled results of all source variants. These are also separated into flow variants as described above.

This structuring of the results is based on how the Juliet Test Suite divides its test cases into groups. We find it natural to group our own results in a similar way as it emphasizes limitations in specific source or flow variants. Detailed information on each flow variant is available in appendix B. The only exceptions are the tables for the hard-coded password detectors. These do not have multiple source variants, but are instead divided into different sinks.

6.2.1. SpotBugs: Injection Vulnerabilities

SpotBugs has detectors for two of the injection vulnerabilities, namely SQL injection and HTTP response splitting. As opposed to Find Security Bugs and ESVD, the injection detectors in SpotBugs are not based on a common underlying framework for injection detection. Due to the different implementations of the detectors, they will be described individually.

Although the SQL injection detector has a 100% recall, it has a precision of 43% and a 0% discrimination rate. Contrarily, the HTTP response splitting detector has a 100% precision but a very low recall of 4%. While the SQL injection detector's poor precision and discrimination rate are due to poor confidence ranking, the HTTP response splitting detector's low recall is due to missing sources.

SQL Injection

Detector Name	`FindSqlInjection`
Files	`FindSqlInjection.java`
Vuln. Type	1) `SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE` 2) `SQL_PREPARED_STATEMENT_GENERATED_FROM_NONCONSTANT_STRING`
Limitations	– Reports on all use of string concatenation in conjunction with a sink

The SpotBugs detector for SQL injection uses a combination of control-flow and data-flow analysis to find use of non-constant strings in SQL statements. Such an approach should, in theory, detect almost all possible forms of SQL injection, as a constant from the *constant pool* cannot contain user input. The constant pool contains compile-time constants formed from literals, and cannot be changed after compilation. The implementation in SpotBugs is having significant problems determining if data is a constant or not, resulting in a lot of false positives. The results can be seen in Table 6.2.

First of all, the detector will identify any method that contains an SQL sink. This information is used in later analysis steps. Then, for each class, the detector will look at each method individually. The method analysis will first execute a data-flow analysis of the values within the method. This is used to identify any method parameters where the value is unchanged before being used in an SQL sink. Afterward, the method analysis will generate a control-flow graph (CFG) and process it twice. The two CFG processing passes can be summarized as follows:

1. **First CFG pass.** The detector does not take immediate action on this CFG pass, but rather remembers what it sees so it can be used in the next CFG pass.
   1.1 Identify changes to data, e.g., a new literal being pushed to *the stack* or a method invocation returning a value to the stack.
      - The JVM stack contains, among other things, local variables. It is analogous to the runtime stack in conventional languages such as C.
   1.2 When a string literal is pushed to the stack, check if it contains a comma or a quotation mark at the start or the end of the string.

1.3 Look for strings being appended to other strings, check if any of these strings are considered unsafe, look for method invocations where the method returns a string, etc.

2. **Second CFG pass**. Look at how the data in the first pass - string appends, use of quotation marks, etc. - are being used in relation to SQL injection sinks. When a sink is found, look at the arguments sent to the sink.

   2.1 If the argument is a constant string, take no action.

   2.2 If the argument comes directly from the method's parameters, without being changed during the method, take no action. This is implemented to reduce false positives.

   - If string concatenation is used on the parameter, the SQL injection detector in SpotBugs will always believe that the parameter is changed. This is due to SpotBugs only seeing the string concatenation and not the unchanged method parameter.

   2.3 If the argument is neither a constant string nor an unchanged parameter, evaluate where the argument was created. If it is unsafe, a vulnerability is reported and the confidence is calculated.

SpotBugs calculates the confidence of a vulnerability by considering the content of the data used in the SQL injection sink. The SQL injection detector will rank the confidence of a vulnerability as low, normal, or high. Vulnerabilities with a low confidence ranking are not shown to the user by default in SpotBugs. When a possible SQL injection has been identified, the confidence of the detected vulnerability is calculated like this:

High confidence If an unsafe string concatenation has been used, the sink's data is tainted, and both an opening and a closing quotation mark is present in the SQL query.

Normal confidence If an unsafe string concatenation has been used, the sink's data is **not** tainted, and both an opening and a closing quotation mark is present in the SQL query.

Normal confidence If an unsafe string concatenation has been used, the sink's data is tainted, and a comma is present in the SQL query.

Low confidence If an unsafe string concatenation has been used, the sink's data is **not** tainted, and a comma is present in the SQL query. By default not shown to the user.

Low confidence If string concatenation has not been used. By default not shown to the user.

The problem with this approach is that the entire detection algorithm is dependent on string concatenations. As an example, take a look at the following code:

```
String data = "foo";
Boolean result = sqlStatement.execute("insert into users (status)
 ↪  values ('updated') where name='"+data+"'");
```

This code will concatenate three strings: `"insert into ..."`, `"foo"`, and `"'"`. The detector sees a string concatenation and it sees open and closing quotation marks. The detector will mark the string concatenation as unsafe, because the instruction before adding "foo" is not considered safe. However, this code does not have any unsafe string concatenations. The bytecode instruction before adding `"foo"` to the string is simply loading the string literal onto the stack, which is not considered safe. Instead of looking at the instruction before the concatenation, SpotBugs should be looking at the instruction which created the value. The methods for finding the creation location of values are actually implemented in the detector, it is just not used in this case. It is however used later in the code, counteracting the flawed value creation locator mentioned above. Due to all of the above, this will result in a reported vulnerability of normal confidence, even though there is no chance of a vulnerability being present.

On the other hand, the code below will allow a malicious user to execute arbitrary commands at the SQL database.

```
Socket socket = new Socket("example.org", 8081);

/* read input from socket */
InputStreamReader readerInputStream = new
 ↪  InputStreamReader(socket.getInputStream(), "UTF-8");
BufferedReader readerBuffered = new
 ↪  BufferedReader(readerInputStream);

String data = readerBuffered.readLine();
Boolean result = sqlStatement.execute(data);
```

This code example is highly dangerous, but not reported to the user. This is mainly due to the code not doing any string concatenations. There are no string literals to check, so it will find no commas or quotation marks. It correctly detects the `data` variable as a taint source, but since no string concatenations are carried out, the detection of tainted data will not affect the vulnerability confidence. Although the code above is unlikely to ever be seen in production software, it serves as an example for the lack of care SpotBugs has given to taint analysis. Even though the code above is very clearly feeding tainted data to an SQL sink, this is given a low confidence and therefore not reported to the user by the default settings in SpotBugs.

The high recall shown in Table 4.3 can be considered misleading when compared to the actual performance of the SQL injection detector. All the test cases in the Juliet Test Suite contain string concatenations, thereby resulting in this detector finding all of the vulnerabilities. Similar amounts of true positives could be achieved by reporting every

Table 6.2.: Test results of SpotBugs for the injection vulnerabilities.

	CWE-89 SQL Injection			**CWE-113 HTTP Resp. Sp.**		
Connect TCP, Console ReadLine, Cookies Servlet, Database, Environment, File, Listen TCP, Properties File, Property, Query String Servlet, URL Connection						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	50%	0%	0%	0%	0%
Control-Flow	100%	37%	0%	0%	0%	0%
Data-Flow	100%	50%	0%	0%	0%	0%
Subtotal	100%	43%	0%	0%	0%	0%
Parameter Servlet						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	50%	0%	100%	100%	100%
Control-Flow	100%	37%	0%	89%	100%	89%
Data-Flow	100%	50%	0%	11%	100%	11%
Subtotal	100%	43%	0%	51%	100%	51%
Total of all Source Variants						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	50%	0%	8%	100%	8%
Control-Flow	100%	37%	0%	7%	100%	7%
Data-Flow	100%	50%	0%	1%	100%	1%
Total	100%	43%	0%	4%	100%	4%

line executing SQL commands, even the safe ones, but that does not make it a good detector. This is apparent when looking at the amounts of false positives in Table 4.3: 3000.

HTTP Response Splitting

Detector Name	`CrossSiteScripting`
Files	`CrossSiteScripting.java`
Vuln. Type	1) `HRS_REQUEST_PARAMETER_TO_COOKIE` 2) `HRS_REQUEST_PARAMETER_TO_HTTP_HEADER`
Strengths	+ Perfect precision of 100%
Limitations	− Detects only 1 of 12 source variants − Poor taint analysis

The SpotBugs detector for HTTP response splitting is looking for the most obvious cases of the vulnerability. It shares the same detector as the one SpotBugs uses for cross-site scripting and path traversal vulnerabilities. The detector uses taint analysis combined with methods for detecting if tainted data reaches a small number of hard-coded sinks, as opposed to resource files containing the sinks like Find Security Bugs and ESVD. The sources that the detector is able to track are few and cover little of the test cases in the Juliet Test Suite. Contrarily to the low recall of 4%, the detector achieves a 100% precision; the HTTP response splitting detector generates no false positives. The results can be seen in Table 6.2.

The HTTP response splitting detector works by analyzing each bytecode instruction on the instruction stack. It compares each instruction with a hard-coded list of vulnerable sinks. The detector then asks the underlying taint analysis if the value reaching the sink is tainted. If the data is tainted, a vulnerability is reported.

When determining the confidence of the reported vulnerability, SpotBugs looks at the source of the tainted data. If the source is `HttpServletRequest.getParameter()`, the vulnerability is given a high confidence. All other reported vulnerabilities are given a normal confidence.

In addition to utilizing the underlying taint analysis present in SpotBugs, the detector also has to do some taint analysis itself. This is required for cases where the taint analysis provided by the SpotBugs framework performs poorly and is unable to connect the data to a source and sink.

The underlying taint analysis algorithm is implemented into a class for managing the instruction stack - a class that is shared by many detectors in SpotBugs. This class does not only contain shared code, but also contains many different techniques that are specific to an individual detector. The taint analysis part of this class is only used by this HTTP response splitting detector, and the sources that are tracked are highly specific to HTTP response splitting, although both the taint analysis and the sources are somewhat usable to the cross-site scripting and path traversal vulnerabilities that are also reported by this detector. Not only is this confusing, but it breaks with many of the design principles found in object-oriented programming. It also makes it difficult to add or modify code belonging to this detector. This might be the reason so few sources are detected.

The lack of sources defined in the taint analysis is the main contributor to the bad recall. Only three sources are defined, namely:

- `HttpServletRequest.getParameter();`
- `HttpServletRequest.getQueryString();`
- `HttpServletRequest.getHeader();`

Out of these three, the first two are sources that exist in the Juliet Test Suite. Only the first source, `getParameter()`, gets any results. Although `getQueryString()` exists in the Juliet Test Suite, the tainted value is immediately broken down into substrings, which the taint analysis fails to track. This lack of sources is the reason SpotBugs is unable to detect anything other than the *Get Parameter Servlet* source variant as seen in Table 6.2.

The second largest contributor to the low recall of 4% is also due to the taint analysis, more specifically the low ability to track data through complex data-flow. As mentioned above, this is the reason SpotBugs is unable to find anything using the `getQueryString()` source in the Juliet Test Suite. The taint analysis is able to track data through simple data operations such as the use of `StringBuilder`, but not the more complex cases. This is the reason SpotBugs is unable to perform better for the control-flow and data-flow variants in the Juliet Test Suite, seen in Table 6.2.

6.2.2. SpotBugs: Hard-Coded Password

Detector Name	`DumbMethodInvocations`
Files	`DumbMethodInvocations.java`
Vuln. Type	`DMI_CONSTANT_DB_PASSWORD`
Strengths	+ Perfect precision of 100%
Limitations	− Detects only 1 of 3 sinks − No control-flow analysis

SpotBugs' detections are limited when it comes to hard-coded passwords. It only detects hard-coded database passwords for the Java method `java.sql.DriverManager.getConnection()`. The implementation is part of the detector called `DumbMethodInvocations` which detects multiple bug patterns.

First, it finds all methods that either directly or indirectly invokes the `getConnection()`-sink. The code is analyzed method-by-method, and a control-flow graph is created for each method. The basic blocks in the control-flow graph are then examined for invoke instructions in the bytecode. Since the detector is looking for invocations of the `getConnection()`-sink, only invoke instructions are relevant to analyze further. If the instruction is indeed an invoke instruction, the detector then checks if it is one of the methods that invokes `getConnection()`. If so, the detector checks if

the password argument of `getConnection()` is hard-coded, i.e. a literal or a constant instantiated as a literal. If it is indeed hard-coded, the detector reports the vulnerability with normal confidence.

The test results for CWE-259 Use of Hard-coded Password in SpotBugs are shown in Table 6.3. Note that the table is divided into different sinks instead of source variants. This is because the source of hard-coded passwords is string literals instead of specific methods. From the results, it is clear that SpotBugs' implementation only detects usage of hard-coded password in conjunction with `java.sql.DriverManager`. Even though it reports 44% true positives for the control-flow variants, the detector does not utilize control-flow analysis. The eight true positives are not due to the detector, but rather the Java compiler optimizing the code and removing the control-flow statement as explained in section 6.1.1. However, the six true positives for data-flow are indeed the detector's merit. By finding the methods that directly or indirectly invokes `getConnection()`, SpotBugs is able to detect the vulnerability on the first method call even though the data travels through several methods and classes. However, it only detects data-flow test cases when the data is transferred as an argument, not when a method returns the value. Nor does it detect the vulnerability if the data is hidden in a data structure such as an `array` or a `Map`.

6.2.3. SpotBugs: Path Traversal

Detector Name	`CrossSiteScripting`
Files	`CrossSiteScripting.java`
Vuln. Type	1) `PT_ABSOLUTE_PATH_TRAVERSAL` 2) `PT_RELATIVE_PATH_TRAVERSAL`
Strengths	+ Perfect precision of 100%
Limitations	− Detects only 1 of 12 source variants − Poor taint analysis − Depends on poor taint analysis to differentiate between relative and absolute path traversal

The SpotBugs detector for path traversal is looking for the most obvious cases of relative and absolute path traversal. The detector contains logic to detect HTTP response splitting, path traversal, and cross-site scripting. The detection logic for the three aforementioned vulnerabilities share the same taint analysis and sources, but differs in which sinks are defined and how the confidence ranking is calculated. The taint analysis used by this shared detector is described in section 6.2.1. Only the logic that is specific to path traversal detection in SpotBugs will be described in this section. The sources that the detector is able to track are few and cover little of the test cases in the Juliet Test Suite. Although the taint analysis has problems following data from source to sink, the

Table 6.3.: Test results of SpotBugs for CWE-259 Use of Hard-coded Password.

	CWE-259 Hard-Coded Pwd		
Driver Manager			
Flow Variant	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%
Control-Flow	44%	100%	44%
Data-Flow	33%	100%	33%
Subtotal	41%	100%	41%
Kerberos Key, Password Authentication			
Flow Variant	**Rec.**	**Prec.**	**Disc.**
Baseline	0%	0%	0%
Control-Flow	0%	0%	0%
Data-Flow	0%	0%	0%
Subtotal	0%	0%	0%
Total of all Sinks			
Flow Variant	**Rec.**	**Prec.**	**Disc.**
Baseline	33%	100%	33%
Control-Flow	15%	100%	15%
Data-Flow	11%	100%	11%
Total	14%	100%	14%

main reason for the bad recall of 4% is due to missing sources. Contrarily to the low recall of 4%, the detector achieves a 100% precision; the path traversal detector generates no false positives. The results can be seen in Table 6.4.

The detector looks for sinks that accept a file or directory path. This is achieved by comparing each bytecode instruction on the instruction stack to a list of vulnerable sinks. If a possible vulnerable sink is located, the detector asks the underlying taint analysis if the value reaching the sink is tainted. If the data is tainted, a vulnerability is reported.

There are in total 21 vulnerable sinks present in the path traversal detector. They are defined in the underlying SpotBugs framework, and are also used by other detectors. Two examples of the included sinks are as follows:

- `java.io.File(String);`
- `java.io.FileReader(String);`

Table 6.4.: Test results of SpotBugs for the path traversal vulnerabilities.

	CWE-23 Relative Path Trv.			CWE-36 Absolute Path Trv.		
Parameter Servlet						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%
Control-Flow	89%	100%	89%	50%	100%	50%
Data-Flow	11%	100%	11%	33%	100%	33%
Subtotal	51%	100%	51%	43%	100%	43%
Connect TCP, Console ReadLine, Cookies Servlet, Database, Environment, File, Listen TCP, Properties File, Property, Query String Servlet, URL Connection						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	0%	0%	0%	0%	0%	0%
Control-Flow	0%	0%	0%	0%	0%	0%
Data-Flow	0%	0%	0%	0%	0%	0%
Subtotal	0%	0%	0%	0%	0%	0%
Total of all Source Variants						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	8%	100%	8%	8%	100%	8%
Control-Flow	7%	100%	7%	4%	100%	4%
Data-Flow	1%	100%	1%	3%	100%	3%
Total	4%	100%	4%	4%	100%	4%

Determining if the vulnerability is an absolute or relative path traversal is based on the source of the data. If the source is `HttpServletRequest.getParameter(String)`, the vulnerability is determined to be an absolute path traversal. Any other source is determined to be a relative path traversal. This peculiar way to determine the vulnerability type is at first glance difficult to understand, as any source can lead to both types of path traversal. The reason lies in the inability of the taint analysis to track the data. In the case of relative path traversal, a common way to combine the tainted user input "`data`" and the pre-determined root path "`root`" is to concatenate the strings, e.g., `new File(root + data)`. Figure 6.7 contains one relative and one absolute path traversal sink, where the code example on the top shows a relative path traversal with the string concatenation. The taint analysis is unable to follow the data through string concatenations, therefore losing track of the data source. If the taint analysis is able to verify

Relative Path Traversal

```
root = "C:\\uploads\\";
/* Sink: Relative Path Traversal */
File file = new File(root + data);
```

Absolute Path Traversal

```
/* Sink: Absolute Path Traversal */
File file = new File(data);
```

Figure 6.7.: Simplified versions of the sinks used in the Juliet Test Suite for relative and absolute path traversal. The only difference is the string concatenation for relative path traversal.

that the data entering the sink has `getParameter(String)` as a source, it knows that no string concatenation has occurred, as it has not lost track of the data due to string concatenation. Determining if the vulnerability is a relative or absolute path traversal based on if the taint analysis loses track of the data is a limitation, as it relies on the taint analysis to have problems tracking data.

6.2.4. SpotBugs: Cross-Site Scripting

Detector Name	`CrossSiteScripting`
Files	`CrossSiteScripting.java`
Vuln. Type	1) `XSS_REQUEST_PARAMETER_TO_JSP_WRITER` 2) `XSS_REQUEST_PARAMETER_TO_SERVLET_WRITER` 3) `XSS_REQUEST_PARAMETER_TO_SEND_ERROR`
Strengths	+ Perfect precision of 100%
Limitations	– Detects only 1 of 12 source variants – Poor taint analysis

The cross-site scripting detector in SpotBugs is able to detect between 3% and 6% of the XSS vulnerabilities in the Juliet Test Suite. The sources and taint analysis used by the XSS detector is the same as the one SpotBugs uses for HTTP response splitting and path traversal vulnerabilities. The difference lies in the sinks and confidence raking. The main reasons for the bad results are, as with HTTP response splitting and path traversal, the bad taint analysis and the few sources that are detected. The taint analysis and the defined sources are described in section 6.2.1, while the sinks and confidence rating will be discussed below. Using taint analysis and a few hard-coded sources and sinks, the detector is only able to detect 57 out of 1332 possible XSS vulnerabilities in the Juliet Test Suite. The detailed results can be seen in Table 6.5.

Table 6.5.: Test results of SpotBugs for the cross-site scripting vulnerabilities.

	CWE-80 Basic XSS			CWE-81 XSS Error Msg.			CWE-83 XSS Attrib.		
Parameter Servlet									
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	50%	100%	100%	100%	100%	100%	100%	100%	100%
Control-Flow	44%	100%	89%	89%	100%	89%	89%	100%	89%
Data-Flow	6%	100%	11%	11%	100%	11%	11%	100%	11%
Subtotal	26%	100%	51%	51%	100%	51%	51%	100%	51%
Connect TCP, Database, File, Cookies Servlet, Query String Servlet, Listen TCP, Properties File, URL Connection									
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	0%	0%	0%	0%	0%	0%	0%	0%	0%
Control-Flow	0%	0%	0%	0%	0%	0%	0%	0%	0%
Data-Flow	0%	0%	0%	0%	0%	0%	0%	0%	0%
Subtotal	0%	0%	0%	0%	0%	0%	0%	0%	0%
Total of all Source Variants									
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	6%	100%	6%	11%	100%	11%	11%	100%	11%
Control-Flow	5%	100%	5%	10%	100%	10%	10%	100%	10%
Data-Flow	1%	100%	1%	1%	100%	1%	1%	100%	1%
Total	3%	100%	3%	6%	100%	6%	6%	100%	6%

There are five sinks that the XSS detector tracks. These five sinks cover all of the sink variants used by the Juliet Test Suite. The following sinks are present in the detector, where the last three allow variations of the sinks, visualized by writing "`*`" to specify that any characters can be placed there:

- `javax.servlet.http.HttpServletResponse.sendError(...)`
- `javax.servlet.jsp.JspWriter.write(...)`
- `javax.servlet.jsp.JspWriter.print*(...)`
- `java.io.*Writer.print*(...)`
- `java.io.*Writer.write*(...)`

CWE-80 Basic XSS has two different ways of sending data to the sink, as can be seen in Figure 6.8. The upper code box in Figure 6.8 shows a string concatenation of a

literal and the `data` variable being sent to the `PrintWriter.println(...)` sink. The code box on the bottom shows almost the same, with the exception of the `data` variable being modified by a `String.replaceAll(...)` method. The taint analysis in the XSS detector is able to handle the data in the upper code box, but not the lower one.

CWE80_XSS__Servlet_connect_tcp_01.java

```
response.getWriter().println("<br>bad(): data = " + data);
```

CWE80_XSS__CWE182_Servlet_connect_tcp_01.java

```
response.getWriter().println("<br>bad(): data = " +
↪  data.replaceAll("(<script>)", ""));
```

Figure 6.8.: Shows two different variants of sending data to the sink for CWE-80 Basic XSS.

The string concatenation in both of the cases shown in Figure 6.8 will result in a `StringBuilder` being created behind the scenes, see Figure 6.9 and 6.10. The XSS detector for SpotBugs is able to follow data through a normal string concatenation, but not through the use of `String.replaceAll(...)`. Figure 6.10 shows the bytecode instructions generated from the `replaceAll` variant. When comparing Figure 6.9 with Figure 6.10, note that the bytecode instructions are almost the same except for the instructions with offset 356, 358, and 360 in Figure 6.10 which are not present in Figure 6.9. When `StringBuilder.append(...)` is executed on offset 363 in Figure 6.10, the XSS detector will look at the item on the top of the stack to determine if it is tainted or not. Since the taint analysis is unable to track data through `replaceAll`, the top of the stack will be unknown, resulting in the `StringBuilder` not being marked as tainted. Considering the CWE-80 Basic XSS category in the Juliet Test Suite contains 333 vulnerable test cases that pass the data through `replaceAll`, this results in a lower recall for the CWE-80 category.

CWE80_XSS__Servlet_connect_tcp_01.java

```
response.getWriter().println("<br>bad(): data = " + data);
```

javap -p -c -constants CWE80_XSS__Servlet_connect_tcp_01.class

```
// String "<br>bad(): data = "
350: ldc               #88
// StringBuilder."<init>":(LString;)V
352: invokespecial     #90
// The "data" variable
355: aload_3
// StringBuilder.append:(LString;)LStringBuilder;
356: invokevirtual     #93
// StringBuilder.toString:()LString;
359: invokevirtual     #97
// PrintWriter.println:(LString;)V
362: invokevirtual     #100
```

Figure 6.9.: The bytecode instructions generated from one of the sinks in the Juliet Test Suite for CWE-80 Basic XSS. See Figure 6.10 for a similar sink using a `String.replaceAll()`. Some bytecode instructions have been omitted for brevity. Some method signatures have been shortened for brevity.

CWE80_XSS__CWE182_Servlet_connect_tcp_01.java

```
response.getWriter().println("<br>bad(): data = " +
↪  data.replaceAll("(<script>)", ""));
```

javap -p -c -constants CWE80_XSS__CWE182_Servlet_connect_tcp_01.class

```
// String "<br>bad(): data = "
350: ldc             #88
// StringBuilder."<init>":(LString;)V
352: invokespecial   #90
// The "data" variable
355: aload_3
// String "(<script>)"
356: ldc             #93
// String ""
358: ldc             #19
// String.replaceAll:(LString;LString;)LString;
360: invokevirtual   #95
// StringBuilder.append:(LString;)LStringBuilder;
363: invokevirtual   #101
// StringBuilder.toString:()LString;
366: invokevirtual   #105
// PrintWriter.println:(LString;)V
369: invokevirtual   #108
```

Figure 6.10.: The bytecode instructions generated from one of the sinks in the Juliet Test Suite for CWE-80 Basic XSS. See Figure 6.9 for a similar sink not using a `String.replaceAll()`. Some bytecode instructions have been omitted for brevity. Some method signatures have been shortened for brevity.

6.2.5. Find Security Bugs: Injection Vulnerabilities

Find Security Bugs has detectors for all of the injection vulnerabilities tested for in the pre-study. All of the injection detectors are based on the injection and taint analysis implementation described in section 6.1.2. The injection detectors use a combination of control-flow and data-flow analysis to find tainted sources passed through to vulnerable sinks without passing through a sanitization point. It does so quite well, with good results for all of the injection vulnerabilities. Find Security Bugs has a high recall of between 74% and 86%, and a high precision rate of 86% to a 100% for all of the injection vulnerabilities in the Juliet Test Suite. Different implementations of the confidence ranking result in slightly different recall and precision between the different injection categories, due to some preferring zero false positives while others accept some false positives in exchange for less false negatives. All false positives and false negatives are exclusively located in the data-flow cases of the Juliet Test Suite.

In general, all the injection detectors suffer from Find Security Bugs' decision to exclude a set of vulnerable sources from its taint analysis. As can be seen in Figure 6.11, the authors of Find Security Bugs have on purpose defined `System.getenv(String)` and `System.getProperty(String)` as safe sources for data to enter the application. These two sources are used to retrieve tainted data in the Juliet Test Suite's source variants Environment and Property, respectively. For the SQL injection category in the Juliet Test Suite, this results in 14% false negatives. Similar numbers of false negatives are seen in the other injection detectors because of the two missing vulnerable sources. Both environment variables and system properties can be tampered with, either by a user with access to the system, another application running on the same system, or by a different vulnerability in the same application. Assuming environment variables and system variables are safe is not based in reality. Although they might be unusual attack vectors, they are not safe from tainted data.

```
- usually safe for web applications
java/lang/System.clearProperty(Ljava/lang/String;)Ljava/lang/String;:SAFE
java/lang/System.getenv()Ljava/util/Map;:SAFE
java/lang/System.getenv(Ljava/lang/String;)Ljava/lang/String;:SAFE
java/lang/System.getProperty(Ljava/lang/String;)Ljava/lang/String;:SAFE
java/lang/System.getProperty(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;:SAFE
```

Figure 6.11.: A configuration file in Find Security Bugs which defines certain data sources as safe, resulting in reduced numbers of true positives.

The false positives and false negatives that are produced by the injection detectors in Find Security Bugs are exclusively located in the complex data-flow test cases in the Juliet Test Suite. There are no false positives or false negatives in the baseline or control-flow cases, see Table 6.6 and 6.7. There are five data-flow cases where the taint analysis in Find Security Bugs is unable to track the data. Two of the five problematic data-flow cases are passing the value between methods using a class field. Another two of the five data-flow cases process and wrap the data into other data structures, such as a Java `Container` or by serializing and deserializing the data. The taint analysis

framework in Find Security Bugs loses track of all four of the test cases mentioned so far, and the tainted data is given an "unknown" state, as discussed in section 6.1.2. The fifth and final of the problematic data-flow cases is using class-based inheritance, where both a safe and vulnerable test case is based on the same class, making it hard to follow which is good and which is bad. The taint analysis framework is unable to track the data in the safe part of the class-based inheritance data-flow test case, but not the vulnerable part of the same test case. Some of the individual injection detectors below will report a vulnerability where data with an unknown state reaches a vulnerable sink, resulting in these five difficult data-flow cases producing five true and five false positives. An example of the effect of reporting the five true and five false positives can be seen in Table 6.6 for the SQL injection detector. Even though the SQL injection detector is unable to track the Environment and Property source variants, the SQL injection detector still achieves a 28% recall. Since the five true and false positives are the result of the taint analysis being unable to track the data back to its source, it does not matter that the Environment and Property source variants are considered safe by Find Security Bugs, as the taint analysis is unable to actually see what the data source is. The HTTP response splitting detector will not report data with an unknown state reaching a vulnerable sink, resulting in these five difficult data-flow cases producing zero false positives and one true positive, as the taint analysis is able to track the vulnerable part of the class-based inheritance test case discussed above.

OS Command Injection

Detector Name	`CommandInjectionDetector`
Files	`CommandInjectionDetector.java`
Vuln. Type	1) `COMMAND_INJECTION` 2) `SCALA_COMMAND_INJECTION`
Strengths	+ Perfect recall of 100%
Limitations	− Detects only 10 of 12 source variants

The OS command injection detector has a 100% recall for 10 of the 12 source variants present in the Juliet Test Suite. For the Environment and Property source variants it does not consider the sources as tainted as previously explained, resulting in a total recall of 86% instead of 100%. The 88% precision is due to false positives in the complex data-flow test cases in the Juliet Test Suite, with no false positives on the baseline cases or on the control-flow cases.

As with all of the injection detectors in Find Security Bugs, very little of the underlying injection detection algorithm is detector specific. The detector specific code only consist of defining relevant sinks and changing the confidence ranking of detections. Specifically for OS command injection, the confidence ranking is defined as follows:

High confidence If the state is set as tainted.

Table 6.6.: Test results of Find Security Bugs for the injection vulnerabilities CWE-78, CWE-89, and CWE-90.

	CWE-78 OSC Injection			CWE-89 SQL Injection			CWE-90 LDAP Injection		
Connect TCP, Console ReadLine, Cookies Servlet, Database, File, Listen TCP, Parameter Servlet, Properties File, Query String Servlet, URL Connection									
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%	100%	100%	100%
Control-Flow	100%	100%	100%	100%	100%	100%	100%	100%	100%
Data-Flow	100%	78%	72%	100%	78%	72%	100%	78%	72%
Subtotal	100%	88%	86%	100%	88%	86%	100%	88%	86%
Environment, Property									
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	0%	0%	0%	0%	0%	0%	0%	0%	0%
Control-Flow	0%	0%	0%	0%	0%	0%	0%	0%	0%
Data-Flow	28%	50%	0%	28%	50%	0%	28%	50%	0%
Subtotal	14%	50%	0%	14%	50%	0%	14%	50%	0%
Total of all Source Variants									
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	83%	100%	83%	83%	100%	83%	83%	100%	83%
Control-Flow	83%	100%	83%	83%	100%	83%	83%	100%	83%
Data-Flow	88%	76%	60%	88%	76%	60%	88%	76%	60%
Total	86%	86%	72%	86%	86%	72%	86%	86%	72%

Normal confidence If the taint is not safe, i.e. the state being unknown or invalid, and is not tagged as having sanitized the data.

As can be seen in Table 6.6, for the Environment and Property source variants, it scores a 28% recall and 50% precision for the data-flow variants, even though it does not recognize the sources as tainted. As mentioned above, this is due to the detector reporting on data with an unknown state. In all other test cases, it is able to follow the data from source to sink.

SQL Injection

Detector Name	`SqlInjectionDetector`
Files	`SqlInjectionDetector.java`
Vuln. Type	1) `SQL_INJECTION_HIBERNATE` 2) `SQL_INJECTION_JDO` 3) `SQL_INJECTION_JPA` 4) `SQL_INJECTION_JDBC` 5) `SQL_INJECTION_SPRING_JDBC` 6) `SCALA_SQL_INJECTION_SLICK` 7) `SCALA_SQL_INJECTION_ANORM` 8) `SQL_INJECTION_TURBINE`
Strengths	+ Perfect recall of 100%
Limitations	− Detects only 10 of 12 source variants

The Find Security Bugs detector for SQL injection is very similar to the OS command injection detector, with the only difference being how the vulnerability confidence is calculated and which sinks are considered as potentially vulnerable. Rather than the sinks being targeted at OS command injections, the sinks are in the SQL injection detector targeted at SQL methods. The results can be seen in Table 6.6. The SQL injection detector has defined the following confidence ranking:

High confidence If the state is set as tainted.

Normal confidence If the state is not safe, i.e. the state being unknown or invalid, and the string has not encoded its apostrophes.

Low confidence If the taint is not safe, and the string has encoded its apostrophes.

LDAP Injection

Detector Name	`LdapInjectionDetector`
Files	`LdapInjectionDetector.java`
Vuln. Type	`LDAP_INJECTION`
Strengths	+ Perfect recall of 100%
Limitations	− Detects only 10 of 12 source variants

The LDAP injection detector in Find Security Bugs has the exact same results as the OS command injection detector. In fact, the implementation is the exact same as for OS command injection, except for the vulnerable sinks being targeted at LDAP injection

rather than OS command injection. The results for the LDAP injection detector can be seen in Table 6.6, and since the implementation is the same as for the OS command injection detector, the explanation of the LDAP injection detector's implementation will not be repeated.

HTTP Response Splitting

Detector Name	`HttpResponseSplittingDetector`
Files	`HttpResponseSplittingDetector.java`
Vuln. Type	`HTTP_RESPONSE_SPLITTING`
Strengths	+ Perfect precision of 100%
Limitations	− Detects only 10 of 12 source variants

Although the HTTP response splitting detector is based on the same underlying code as the other injection detectors, it behaves differently. This leads to a total recall of 74% instead of the 86% recall that can be seen for the other detectors, but also a higher total precision of 100% compared to 86% for the other detectors. SQL injection, LDAP injection, and OS command injection have five data-flow cases for each source variant where it reports a true and a false positive even though it is unable to see the data source. The HTTP response splitting detector has a different confidence ranking for potential vulnerabilities, resulting in it only reporting cases where it is certain that the data is tainted. This confidence ranking results in four less true positives, and five less false positives for each source variant. Technically, the detector does in fact report on test cases where it is unsure if the data is tainted, but it gives them a low confidence resulting in the vulnerability not being reported when the default configuration of SpotBugs is used.

The HTTP response splitting detector has defined the following confidence ranking:

Normal confidence If the state is set as tainted.

Low confidence If the state is not safe, i.e. the state being unknown or invalid.

The full results can be seen in Table 6.7. All the reported vulnerabilities are given a normal confidence, meaning that the detector is sure that the data is tainted. The low confidence vulnerabilities are ignored in a normally configured SpotBugs and Find Security Bugs plugin, and are not included in Table 6.7.

Table 6.7.: Test results of Find Security Bugs for the injection vulnerabilities CWE-113 and CWE-643.

	CWE-113 HTTP Resp. Sp.			CWE-643 XPath Injection		
Connect TCP, Console ReadLine, Cookies Servlet, Database, File, Listen TCP, Parameter Servlet, Properties File, Query String Servlet, URL Connection						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%
Control-Flow	100%	100%	100%	100%	100%	100%
Data-Flow	78%	100%	78%	100%	78%	72%
Subtotal	89%	100%	89%	100%	88%	86%
Environment, Property						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	0%	0%	0%	0%	0%	0%
Control-Flow	0%	0%	0%	0%	0%	0%
Data-Flow	0%	0%	0%	28%	50%	0%
Subtotal	0%	0%	0%	14%	50%	0%
Total of all Source Variants						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	83%	100%	83%	83%	100%	83%
Control-Flow	83%	100%	83%	83%	100%	83%
Data-Flow	65%	100%	65%	88%	76%	60%
Total	74%	100%	74%	86%	86%	72%

XPath Injection

Detector Name	`XPathInjectionDetector`
Files	`XPathInjectionDetector.java`
Vuln. Type	`XPATH_INJECTION`
Strengths	+ Perfect recall of 100%
Limitations	− Detects only 10 of 12 source variants

The XPath injection detector in Find Security Bugs has the exact same results and implementation as the OS command injection and LDAP injection detectors. The only

difference is that the vulnerable sinks are targeted towards XPath injection. The results for the XPath injection detector can be seen in Table 6.7, and since the implementation is the same as for the OS command injection detector, the explanation of the XPath injection detector's implementation will not be repeated.

6.2.6. Find Security Bugs: Hard-Coded Password

Find Security Bugs has implemented multiple detectors for finding hard-coded passwords. By extending the underlying detectors from SpotBugs, these cover approximately half of the relevant test cases in the Juliet Test Suite. On its own, meaning without the help of the detectors from SpotBugs, it finds 43% of the vulnerabilities.

There are five different detectors for finding hard-coded passwords. Some are complex while others are much simpler. The five detectors are as follows:

- HardcodedPasswordEqualsDetector
- HardcodePasswordInMapDetector
- IntuitiveHardcodePasswordDetector
- GoogleApiKeyDetector
- ConstantPasswordDetector

Each detector will be discussed individually. Even though the only vulnerabilities detected in the Juliet Test Suite are due to one of the detectors, it is important to remember that all of the detectors serve a valuable purpose. The Juliet Test Suite does not test for all possible variants of hard-coded passwords, and in its current form only tests for cases that the `ConstantPasswordDetector` is able to find. We have decided to describe the other detectors briefly without presenting strengths and limitations, as they do not produce any measurable results when tested on the Juliet Test Suite and we cannot verify the effects of implementation strengths and limitations.

Detecting Hard-Coded Passwords in `equals()`

Detector Name	`HardcodedPasswordEqualsDetector`
Files	1) `HardcodedPasswordEqualsDetector.java`
	2) `AbstractHardcodedPasswordEqualsDetector.java`
Vuln. Type	`HARD_CODE_PASSWORD`

This detector's purpose is to find code instances where a variable is compared to a string literal, where the variable contains user input and the string contains the hard-coded password. It will detect these two cases:

- `variable.equals("string literal");`

- `"string literal".equals(variable);`

The detector finds code where the `equals(...)` method is used, and tags these as possible injection points. It also looks for variable names which have a meaning similar to password. The full list of words resembling password can be seen in Figure 6.12, where also foreign words are included. Each variable with such a name is marked as a password variable. If a password variable is used in conjunction with a possible injection point, the injection point is reported and given a normal confidence.

```
//Passwords in various language
//http://www.indifferentlanguages.com/words/password
PASSWORD_WORDS.add("password");
PASSWORD_WORDS.add("motdepasse");
PASSWORD_WORDS.add("heslo");
PASSWORD_WORDS.add("adgangskode");
PASSWORD_WORDS.add("wachtwoord");
PASSWORD_WORDS.add("salasana");
PASSWORD_WORDS.add("passwort");
PASSWORD_WORDS.add("passord");
PASSWORD_WORDS.add("senha");
PASSWORD_WORDS.add("geslo");
PASSWORD_WORDS.add("clave");
PASSWORD_WORDS.add("losenord");
PASSWORD_WORDS.add("clave");
PASSWORD_WORDS.add("parola");
//Others
PASSWORD_WORDS.add("secretkey");
PASSWORD_WORDS.add("pwd");
```

Figure 6.12.: The different words which are suspected to be used for storing passwords. The list is shared by several detectors in Find Security Bugs.

Detecting Hard-Coded Passwords when Assigning `Map` Values

Detector Name	`HardcodePasswordInMapDetector`
Files	1) `HardcodePasswordInMapDetector.java`
	2) `AbstractHardcodePasswordInMapDetector.java`
Vuln. Type	`HARD_CODE_PASSWORD`

The detector's purpose is to find code instances where a hard-coded password is added to a `Map`. It supports various types of `Maps`, such as `HashMap`, `Hashtable`, and `Properties`. The following three cases will be detected:

- `map.put(key, value);`
- `map.putIfAbsent(key, value);`

- `properties.setProperty(key, value);`

Any calls to the three sinks above will be marked as a potential injection point. Similarly to the `equals(...)` detector above, the key must have a name from Figure 6.12 with a meaning similar to password. In this case, the key can also have the name `java.naming.security.credentials`, as this is used in e.g. LDAP authentication. If `value` is a string from the constant pool and `key` is a password word, it is reported as an injection point and is given normal confidence.

Detecting Hard-Coded Passwords in Unknown API Calls

Detector Name	`IntuitiveHardcodePasswordDetector`
Files	`IntuitiveHardcodePasswordDetector.java`
Vuln. Type	`HARD_CODE_PASSWORD`

The goal of this detector is to find hard-coded passwords in calls to any class or API. It will mark all of the following as vulnerabilities, where `MyCustomClient` and `HomeDoor` are arbitrary classes or API calls:

- `MyCustomClient.setMyPwd("secret123");`
- `HomeDoor.password("pass123");`

For every invocation instruction in the code, the detector checks if the method argument is a string from the constant pool. Then, it checks if the method name equals one of the words in Figure 6.12. It will also trigger if the method name starts with "`set`", and contains one of the words. If the aforementioned requirements are fulfilled, the detector will report an injection point with normal confidence.

Detecting Hard-Coded Passwords in Google Maps API Sample Code

Detector Name	`GoogleApiKeyDetector`
Files	`GoogleApiKeyDetector.java`
Vuln. Type	`HARD_CODE_PASSWORD`

The `GoogleApiKeyDetector` is a very specific detector. It detects hard-coded private keys in sample code provided by Google for a Java client library for the Google Maps API. The sample code is very hard to find at the time of writing, as Google has updated their API. The changes to the Google Maps API renders this code useless, and we have decided to not describe this detector any further.

Table 6.8.: Test results of Find Security Bugs for CWE-259 Use of Hard-coded Password.

	CWE-259 Hard-Coded Pwd		
Total of all Sinks (Driver Manager, Kerberos Key, Password Authentication)			
Flow Variant	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%
Control-Flow	78%	100%	78%
Data-Flow	6%	100%	6%
Total	43%	100%	43%

Detecting Hard-Coded Passwords with Data-Flow Analysis

Detector Name	`ConstantPasswordDetector`
Files	`ConstantPasswordDetector.java`
Vuln. Type	`HARD_CODE_PASSWORD`
Strengths	+ Perfect precision of 100%
Limitations	− DFA only works within a method

This detector is more advanced than the other hard-coded password detectors in Find Security Bugs. It can detect hard-coded passwords through data-flow analysis contained within a method, in addition to hard-coded passwords in class fields. It is unable to carry out context-sensitive data-flow analysis, resulting in many false negatives for the data-flow cases in the Juliet Test Suite. By using a resource file containing 35 vulnerable sinks, it is able to detect multiple vulnerabilities as listed in Table 6.8. The following vulnerable sinks are just some of what this detector will mark as a vulnerability:

- `KeyStore.load(...);`
- `KeyManagerFactory.init(...);`
- `DriverManager.getConnection(...);`

For each class, the detector examines static class fields being initialized. This is done before processing the rest of the class, so that when analyzing the different methods, it can be determined if they use a constant password assigned in a class field. The detector then analyzes the instructions from each method, looking for string literals, array storage, conversions into a constant array, and invocations.

Tracking data is difficult, especially through conditional statements where the code branches. To be able to track data through branches, the detector tags variables as hard-coded, effectively marking them as dangerous. Hard-coded variables contain string literals from the constant pool and cannot be changed by user input. The tracking through conditional statements works as follows:

- If all reachable paths in a conditional statement assign a hard-coded value to the same variable, the variable is tagged as hard-coded.
- If at least one of the reachable paths assign a value that is not hard-coded, the variable is not tracked. The reasoning behind not marking it as hard-coded is to reduce false positives. On the other hand, it can potentially increase false negatives.

Two cases will result in a vulnerability being reported by this detector. The first case is when a class field has a suspicious name, defined by the regular expression in Figure 6.13. If such a suspicious name is detected in a field name, it will report it with normal confidence. This behavior can detect hard-coded passwords in fields, even though the detector is unable to execute context-sensitive analysis of the usage of these fields. Reporting suspicious field names without checking if the value ever reaches a sink might lead to false positives, but the Juliet Test Suite does not contain test cases for this behavior. The second case is when a vulnerable sink is reached by a variable tagged as hard-coded. This will result in the vulnerability being reported with high confidence.

```
private static final String PASSWORD_NAMES =
".*(pass|pwd|psw|secret|key|cipher|crypt|des|aes|mac|private|sign|cert).*";
```

Figure 6.13.: The different strings which are suspected to be used for storing passwords in the data-flow detector in Find Security Bugs.

6.2.7. Find Security Bugs: Path Traversal

Detector Name	`PathTraversalDetector`
Files	`PathTraversalDetector.java`
Vuln. Type	1) `PATH_TRAVERSAL_IN` 2) `PATH_TRAVERSAL_OUT` 3) `SCALA_PATH_TRAVERSAL_IN`
Strengths	+ Perfect recall of 100%
Limitations	– Detects only 10 of 12 source variants – Cannot differentiate between relative and absolute path traversal

Table 6.9.: Test results of Find Security Bugs for the path traversal vulnerabilities.

	CWE-23 Relative Path Trv.			CWE-36 Absolute Path Trv.		
Connect TCP, Console ReadLine, Cookies Servlet, Database, File, Listen TCP, Parameter Servlet, Properties File, Query String Servlet, URL Connection						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%
Control-Flow	100%	100%	100%	100%	100%	100%
Data-Flow	100%	78%	72%	100%	78%	72%
Subtotal	100%	88%	86%	100%	88%	86%
Environment, Property						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	0%	0%	0%	0%	0%	0%
Control-Flow	0%	0%	0%	0%	0%	0%
Data-Flow	28%	50%	0%	28%	50%	0%
Subtotal	14%	50%	0%	14%	50%	0%
Total of all Source Variants						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	83%	100%	83%	83%	100%	83%
Control-Flow	83%	100%	83%	83%	100%	83%
Data-Flow	88%	76%	60%	88%	76%	60%
Total	86%	86%	72%	86%	86%	72%

The path traversal detector in Find Security Bugs has a total recall and precision of 86%, and a discrimination rate of 72%, see Table 6.9. The implementation is based on the same taint analysis found in the injection detectors in section 6.2.5, and will therefore not be repeated here. With the exception of different sinks and a slightly modified confidence ranking algorithm, the implementation is identical to those found in the injection detectors of Find Security Bugs. The detector does not detect if the vulnerability is a relative or absolute path traversal vulnerability, but rather reports all vulnerabilities as simply path traversal vulnerabilities.

Find Security Bugs does not consider the Environment and Property source variants used in the Juliet Test Suite as vulnerable. This is discussed in section 6.2.5, but also affects the path traversal detector. The ten other source variants are detected, and all vulnerabilities are found in addition to a few false positives. The reason for these false

Table 6.10.: Test results of Find Security Bugs for the cross-site scripting vulnerabilities.

	CWE-80 Basic XSS			CWE-81 XSS Error Msg.			CWE-83 XSS Attrib.		
Total of all Source Variants (Connect TCP, Database, File, Cookies Servlet, Query String Servlet, Listen TCP, Parameter Servlet, Properties File, URL Connection)									
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%	100%	100%	100%
Control-Flow	100%	100%	100%	100%	100%	100%	100%	100%	100%
Data-Flow	100%	78%	72%	100%	78%	72%	100%	78%	72%
Total	100%	88%	86%	100%	88%	86%	100%	88%	86%

positives are the same as for the injection detectors: when the taint analysis loses track of the data source, but the data enters a vulnerable sink, a vulnerability is reported with a medium confidence. When the taint analysis is able to track the data source, and the data enters a vulnerable sink without sanitization, a vulnerability is reported with a high confidence.

The path traversal detector makes no effort to differentiate between relative and absolute path traversal. Generally, both relative and absolute path traversal use the same sinks, making it harder to separate these two vulnerabilities. The taint analysis in Find Security Bugs does not provide any techniques for separating between the vulnerabilities either.

6.2.8. Find Security Bugs: Cross-Site Scripting

Find Security Bugs contains three different cross-site scripting detectors. The cross-site scripting detectors for Java servlet and JavaServer Pages are based on the same taint analysis framework as the injection detectors described in section 6.2.5. Although the implementation specific to these two path traversal detectors will be explained, the details of the underlying taint analysis algorithm will not be repeated. The third XSS detector is not aimed at detecting cross-site scripting vulnerabilities, but rather weak or improper prevention against XSS. Out of the three XSS detectors, only the detector aimed at Java servlet XSS vulnerabilities is relevant for the vulnerabilities present in the Juliet Test Suite. The two other detectors cannot be evaluated on the Juliet Test Suite, and for that reason, we will not present their strengths and limitations as we cannot verify their effect.

Cross-Site Scripting in Java Servlet

Detector Name	`XssServletDetector`
Files	`XssServletDetector.java`
Vuln. Type	`XSS_SERVLET`
Strengths	+ Perfect recall of 100% + Detects all source variants
Limitations	− Uncertain detections are still reported

The XSS detector for Java servlet is based on the same taint analysis framework as the injection detectors in Find Security Bugs. As can be seen in Table 6.10, a few false positives are produced for the data-flow cases, resulting in a 88% precision. This is due to the taint analysis losing track of the data source. When a variable with an unknown value reaches a vulnerable sink, the XSS detector chooses to report a vulnerability.

A problem with the injection detectors in section 6.2.5 was two missing source variants, namely Environment and Property. These two source variants are not present in the Juliet Test Suite XSS test cases, and for that reason do not affect the results of the XSS detector in Find Security Bugs.

The following confidence rating is given to XSS vulnerabilities by this detector:

High confidence If the state is set as tainted.

Normal confidence If the state is not safe, i.e. the state being unknown or invalid, and the string has not encoded its apostrophes, quotation marks, or less than signs.

Low confidence If the taint is not safe, and the string has encoded its apostrophes or quotation marks, in addition to encoding less than signs.

To not interfere with the XSS detector for JavaServer Pages, the XSS detector for Java servlet will ignore sinks which are to be detected by the JavaServer Pages detector, and vice versa.

Cross-Site Scripting in JavaServer Pages

Detector Name	`XssJspDetector`
Files	`XssJspDetector.java`
Vuln. Type	`XSS_JSP_PRINT`

The cross-site scripting detector for finding vulnerabilities in JavaServer Pages is almost identical to the XSS detector for Java servlet. The only difference is the sinks. None of these sinks are present in the Juliet Test Suite, and therefore produce no results.

Use of Weak Cross-Site Scripting Prevention

Detector Name	`XSSRequestWrapperDetector`
Files	`XSSRequestWrapperDetector.java`
Vuln. Type	`XSS_REQUEST_WRAPPER`

The last cross-site scripting detector in Find Security Bugs is not looking for XSS vulnerabilities, but rather improper attempts at sanitizing XSS vulnerabilities. According to the author of this detector, an XSS filter published by Ricardo Zuasti around 2012 is easy to bypass [Find Security Bugs, 2012] and therefore unsafe to use.

By looking at some properties that are isolated to the specific XSS filter published by Zuasti, the detector is able to identify when the weak XSS filter is in use. The detector specifically looks for classes that extend the `HttpServletRequestWrapper` class, and contain a method named `stripXSS(...)`. No data-flow or taint analysis is needed, as this detector simply executes something similar to pattern matching in an attempt to locate the use of this XSS filter.

6.2.9. ESVD: Injection Vulnerabilities

Out of the six injection vulnerabilities covered in our pre-study, ESVD claims to cover all of them except for use of externally-controlled format string. However, as the pre-study results reveal in Table 4.3, ESVD does in fact only cover OS command injection and SQL injection. All of LDAP injection, HTTP response splitting, and XPath injection get zero true and false positives. This is despite the fact that all of the detectors in ESVD use the same underlying algorithm as explained in section 6.1.3.

OS Command Injection

Detector Name	`VerifierCommandInjection`
Files	`VerifierCommandInjection.java`
Vuln. Type	`Command Injection`
Strengths	+ Perfect precision of 100%
Limitations	− Detects only 7 of 12 source variants − Does not track data through `if`- and `switch`-statements − No context-sensitive data-flow analysis

The results in Table 6.11 show that the OS command injection detector only detects 7 out of the 12 source variants used by the Juliet Test Suite. For the seven detected source variants, only 19% of the vulnerabilities are reported. It is especially the control-flow and data-flow cases the detector is struggling to detect with a recall of respectively

11% and 22%. However, the detector maintains a perfect precision of 100%, meaning it detects no false positives.

Table 6.11.: Test results of ESVD for the injection vulnerabilities.

	CWE-78 OSC Injection			**CWE-89 SQL Injection**		
Cookies Servlet, Database, Environment, Parameter Servlet, Properties File, Property, Query String Servlet						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	50%	0%
Control-Flow	11%	100%	11%	94%	35%	0%
Data-Flow	22%	100%	22%	33%	50%	0%
Subtotal	19%	100%	19%	65%	39%	0%
Connect TCP, Console ReadLine, File, Listen TCP, URL Connection						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	0%	0%	0%	100%	50%	0%
Control-Flow	0%	0%	0%	94%	35%	0%
Data-Flow	0%	0%	0%	33%	50%	0%
Subtotal	0%	0%	0%	65%	39%	0%
Total of all Source Variants						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	58%	100%	58%	100%	35%	0%
Control-Flow	6%	100%	6%	94%	65%	0%
Data-Flow	13%	100%	13%	33%	50%	0%
Total	11%	100%	11%	65%	39%	0%

The five source variants not detected by the OS command injection detector are Connect TCP, Console ReadLine, File, Listen TCP, and URL Connection. When inspecting the implementation of the detector, we discover that none of the sources are included in ESVD's resource lists. As mentioned in section 6.1.3, ESVD has a shared list of sources that every detector utilizes. If a source is not included in the list, the source will not be detected by any of the detectors. Consequently, these source variants will not be detected by any of the other detectors in ESVD either.

The control-flow variants detected by the OS command injection detector use a `while`-loop and a `for`-loop. It turns out that the underlying algorithm of ESVD is able to track the data through these two control-flow statements, but struggles with `if`-statements and `switch`-statements where it detects nothing. Since most of the control-flow variants in the Juliet Test Suite include `if`-statements, the detector only detects 11% of the

control-flow variants.

The OS command injection detector detects data-flow variants where data is sent between methods within the same class. However, when the data moves between classes, i.e. context-sensitive data-flow analysis, the detector is not able to detect anything. This results in only a 22% recall for the data-flow variants. This is especially surprising as utilizing context-sensitive data-flow analysis was the biggest selling point of ESVD [Sampaio and Garcia, 2016].

When inspecting the implementation to determine why the detector performs poorly for the more complex flow variants, we discovered that the cause lies in the underlying algorithm of ESVD. In other words, it is not a limitation of only the OS command injection detector, but it affects all of the detectors in ESVD. Therefore, we should expect to see similar results produced by the detectors for the other vulnerabilities covered in this thesis.

SQL Injection

Detector Name	`VerifierSQLInjection`
Files	`VerifierSQLInjection.java`
Vuln. Type	`Sql Injection`
Limitations	– Reports on all use of string concatenation in conjunction with a sink – Uses pattern matching, instead of data-flow analysis

The second injection vulnerability covered by ESVD is SQL injection. The results of the SQL injection detector in Table 6.11 seem at first glance far better than the results for the OS command injection detector. All twelve source variants are covered, and it has a 65% recall. However, there is also a significant amount of false positives which leads to a precision of merely 39%.

As mentioned in section 6.1.3, the SQL injection detector is one of two detectors in ESVD that has a unique detection algorithm. The SQL injection detector still uses the underlying algorithm of ESVD, but has a different approach to when a vulnerability is reported. Instead of reporting a vulnerability when tainted data from a source reaches a sink, the SQL injection detector reports on all cases where a concatenated string is used in conjunction with a sink. The only criterion is that the concatenated string stems from string variables and not literals, i.e., a string originating from user input. The detector does not look at the content of the string variable and whether it originates from a vulnerable source. Thus, the detector does not actually utilize data-flow analysis and is more similar to basic pattern matching. This results in many true and false positives being reported. The discrimination rate of 0% is an indication that the detector is not able to differ true positives from false positives and does not use data-flow analysis.

An interesting observation is that all of the false positives are detected in test cases consisting of a safe source and an exploitable sink. This is because all detectors in ESVD only report a vulnerability when it finds an exploitable sink. A vulnerable source on its own triggers nothing.

LDAP Injection

Detector Name	`VerifierLDAPInjection`
Files	`VerifierLDAPInjection.java`
Vuln. Type	`LDAP Injection`
Limitations	– Detects nothing because of missing sink

The LDAP injection detector is not able to detect any vulnerabilities in the Juliet Test Suite. Examining the implementation reveals that the sink used in the Juliet Test Suite is not included in ESVD's resource lists. The method signature of the missing sink showed in Figure 6.14 is the method `InitialDirContext.search(String, String, SearchControls)`. Without including this method signature in the resource list of LDAP injection sinks, the detector will not be able to detect any of the test cases for LDAP injection in the Juliet Test Suite.

CWE90_LDAP_Injection__connect_tcp_01.java

```
directoryContext = new InitialDirContext(environmentHashTable);
/* POTENTIAL FLAW: data concatenated into LDAP search, which could
   result in LDAP Injection */
String search = "(cn=" + data + ")";
answer = directoryContext.search("", search, null);
```

Figure 6.14.: Simplified version of the LDAP injection sink used in the Juliet Test Suite.

HTTP Response Splitting

Detector Name	`VerifierHTTPResponseSplitting`
Files	`VerifierHTTPResponseSplitting.java`
Vuln. Type	`HTTP Response Splitting`
Limitations	– Detects nothing because of missing sink

The HTTP response splitting detector in ESVD detects no vulnerabilities. Similarly to the detector for LDAP injection, this is because the sinks used in the Juliet Test Suite

are not included in the resource list. The test cases for HTTP response splitting in the Juliet Test Suite uses three different sinks called `addCookie`, `addHeader`, and `setHeader`. There are three times as many test cases for HTTP response splitting as opposed to, e.g., LDAP injection, as each source variant is repeated three times, once for each sink. The three sinks used in the Juliet Test Suite are shown in Figure 6.15.

CWE113_HTTP_Response_Splitting__connect_tcp_addCookieServlet_01.java

```
Cookie cookieSink = new Cookie("lang", data);
/* POTENTIAL FLAW: Input not verified before inclusion in the cookie */
response.addCookie(cookieSink);
```

CWE113_HTTP_Response_Splitting__connect_tcp_addHeaderServlet_01.java

```
/* POTENTIAL FLAW: Input from file not verified */
response.addHeader("Location", "/author.jsp?lang=" + data);
```

CWE113_HTTP_Response_Splitting__connect_tcp_setHeaderServlet_01.java

```
/* POTENTIAL FLAW: Input not verified before inclusion in header */
response.setHeader("Location", "/author.jsp?lang=" + data);
```

Figure 6.15.: Simplified version of the HTTP Response Splitting sinks used in the Juliet Test Suite.

XPath Injection

Detector Name	`VerifierXPathInjection`
Files	`VerifierXPathInjection.java`
Vuln. Type	`XPath Injection`
Limitations	– Detects nothing despite including the sink

The detector for XPath injection is not able to detect any vulnerabilities. However, contrary to the LDAP injection and HTTP response splitting detectors, the sink is indeed included in the resource list for XPath injection. Nonetheless, the detector is still not able to detect the vulnerability. This is odd as all detectors use the same underlying algorithm which should lead to identical results as long as the sinks are included.

We hypothesize that the XPath injection detector fails to track the data through the natural complexity of the test cases in the Juliet Test Suite. As shown in Figure 6.16, even the baseline test case for XPath injection sends the data through additional control-flow statements and string operations before it reaches the sink `xPath.evaluate(...)`. Since ESVD struggles with `if`-statements as mentioned earlier, this might explain why the XPath injection detector detects nothing.

CWE643_Xpath_Injection__connect_tcp_01.java

```
...
if (data != null) {
    String [] tokens = data.split("||");
    if (tokens.length < 2) {
        return;
    }
    String username = tokens[0];
    String password = tokens[1];
    XPath xPath = XPathFactory.newInstance().newXPath();
    InputSource inputXml = new InputSource(xmlFile);
    /* POTENTIAL FLAW: user input is used without validate */
    String query = "//users/user[name/text()='" + username +
                   "' and pass/text()='" + password + "']" +
                   "/secret/text()";
    String secret = (String)xPath.evaluate(query, inputXml,
     ↪  XPathConstants.STRING);
}
```

Figure 6.16.: The baseline test case for XPath injection in the Juliet Test Suite. The sink `xPath.evaluate(...)` is encapsulated within an `if`-statement, while the data is processed by several string operations. This additional complexity might be the cause for why the XPath injection detector is not able to detect any vulnerabilities.

6.2.10. ESVD: Hard-Coded Password

Detector Name	`VerifierSecurityMisconfiguration`
Files	`VerifierSecurityMisconfiguration.java`
Vuln. Type	`Security Misconfiguration`
Strengths	+ Detects 89% of the control-flow variants + Good precision of 87%
Limitations	– Mislabeled as Security Misconfiguration – Detects only 1 of 3 sinks – Inconsistent behavior in the CFA and DFA algorithms when using literals instead of sources

Despite not stating it anywhere in its documentation, ESVD is able to detect hard-coded passwords. When analyzing the source code to identify why it is reported, we discover that instead of being labeled as hard-coded password, it is labeled as security misconfiguration. This also caused confusion during our pre-study as described in chapter 4. We reported that ESVD did not cover the vulnerability as can be seen in the research

paper in appendix A.

Just as the SQL injection detector, the hard-coded password detector of ESVD has a unique detection algorithm. It is still partly based on the underlying algorithm of ESVD, but instead of tracking data from source to sink, the detector reports when literals are used as a hard-coded password. This means the detector does not use the resource list for sources, but it does use a unique resource list for sinks. Since a password variable can be assigned a literal value and later be used in a sink, the detector still needs to utilize data-flow analysis.

Table 6.12.: Test results of ESVD for CWE-259 Use of Hard-coded Password.

	CWE-259 Hard-Coded Pwd		
Driver Manager			
Flow Variant	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%
Control-Flow	89%	89%	83%
Data-Flow	17%	75%	11%
Subtotal	54%	87%	49%
Kerberos Key, Password Authentication			
Flow Variant	**Rec.**	**Prec.**	**Disc.**
Baseline	0%	0%	0%
Control-Flow	0%	0%	0%
Data-Flow	0%	0%	0%
Subtotal	0%	0%	0%
Total of all Sinks			
Flow Variant	**Rec.**	**Prec.**	**Disc.**
Baseline	33%	100%	33%
Control-Flow	30%	89%	28%
Data-Flow	6%	75%	4%
Total	18%	87%	16%

As can be seen in Table 6.12, the only sink included is Driver Manager. The other two sinks, namely Kerberos Key and Password Authentication, are not included in the resource lists. Contrary to the OS command injection detector, the detector for hard-coded passwords reports more true and false positives. It detects additional control-flow variants and misses one data-flow variant which the OS command injection detector successfully detect. By examining the source code of the hard-coded password detector, we found that nothing in the detector warrants this change in behavior. The detector has

its own way to deal with literals instead of sources, but does not change the underlying control- and data-flow algorithms. The cause of the difference in the results is due to an inconsistent behaviour by the control- and data-flow algorithms when detecting literals instead of sources.

Additionally, the detector also detects hard-coded usernames as well as when usernames and passwords are set to `null`. Since this is not part of CWE-259 Use of Hard-coded Password, we have decided to not include these in our results. We do not count them as false positives, as they are not wrong detections, but rather irrelevant detections.

6.2.11. ESVD: Path Traversal

Detector Name	`VerifierPathTraversal`
Files	`VerifierPathTraversal.java`
Vuln. Type	`Path Traversal`
Strengths	+ Perfect precision of 100%
Limitations	– Detects only 7 of 12 source variants – Does not track data through `if`- and `switch`-statements – No context-sensitive data-flow analysis – Does not differentiate between relative and absolute path traversal

In the pre-study, we reported that ESVD did not detect a single path traversal vulnerability. During the work on this master thesis, we discovered that it does in fact detect something. As mentioned in chapter 4, we believe this error was caused by ESVD's unstable behaviour with constant freezes and crashes.

Contrarily to the SQL injection detector and the hard-coded password detector, the path traversal detector does not have a unique detection algorithm. Instead, it uses the same underlying algorithm in ESVD as the OS command injection detector. Consequently, the results in Table 6.13 are identical to the results of OS command injection in Table 6.11 with the same source and flow variants detected. The explanation of the implementation for the OS command injection detector in section 6.2.9 is valid for the path traversal detector as well.

The difference between the test cases for relative and absolute path traversal in the Juliet Test Suite is whether the input retrieved from the source is a relative path or an absolute path. However, the path traversal detector in ESVD does not separate between relative and absolute path traversal, and they are both labeled as merely "Path Traversal". An interesting detail with the sinks used by the Juliet Test Suite for path traversal is that they are not ordinary methods, but rather object instantiations as shown in Figure 6.7. However, ESVD only uses methods for sinks and not classes. ESVD solves

Table 6.13.: Test results of ESVD for the path traversal vulnerabilities.

	CWE-23 Relative Path Trv.			**CWE-36 Absolute Path Trv.**		
Cookies Servlet, Database, Environment, Parameter Servlet, Properties File, Property, Query String Servlet						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%
Control-Flow	11%	100%	11%	11%	100%	11%
Data-Flow	22%	100%	22%	22%	100%	22%
Subtotal	19%	100%	19%	19%	100%	19%
Connect TCP, Console ReadLine, File, Listen TCP, URL Connection						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	0%	0%	0%	0%	0%	0%
Control-Flow	0%	0%	0%	0%	0%	0%
Data-Flow	0%	0%	0%	0%	0%	0%
Subtotal	0%	0%	0%	0%	0%	0%
Total of all Source Variants						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	58%	100%	58%	58%	100%	58%
Control-Flow	6%	100%	6%	6%	100%	6%
Data-Flow	13%	100%	13%	13%	100%	13%
Total	11%	100%	11%	11%	100%	11%

this by including the method signature of the constructor for the class. This shows ESVD is quite versatile in what it considers a sink which is a strength of the implementation.

6.2.12. ESVD: Cross-Site Scripting

Detector Name	`VerifierCrossSiteScripting`
Files	`VerifierCrossSiteScripting.java`
Vuln. Type	`Cross-Site Scripting (XSS)`
Strengths	+ Perfect precision of 100% + Detects both sinks of CWE-80 Basic XSS
Limitations	− Detects only 5 of 9 source variants − Does not track data through `if`- and `switch`-statements − No context-sensitive data-flow analysis

The cross-site scripting detector of ESVD uses the same implementation as the OS command detector and the path traversal detector. This means the cross-site scripting detector detects the same source and flow variants as the other two. The results in Table 6.14 are identical to the results for OS command injection in Table 6.11 and path traversal in Table 6.13. The explanation given for the OS command injection detector in section 6.2.9 is valid for the cross-site scripting detector as well.

Table 6.14.: Test results of ESVD for the cross-site scripting vulnerabilities.

	CWE-80 Basic XSS			CWE-81 XSS Error Msg.			CWE-83 XSS Attrib.		
Cookies Servlet, Database, Parameter Servlet, Properties File, Query String Servlet									
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%	100%	100%	100%
Control-Flow	11%	100%	11%	11%	100%	11%	11%	100%	11%
Data-Flow	22%	100%	22%	22%	100%	22%	22%	100%	22%
Subtotal	19%	100%	19%	19%	100%	19%	19%	100%	19%
Connect TCP, File, Listen TCP, URL Connection									
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	0%	0%	0%	0%	0%	0%	0%	0%	0%
Control-Flow	0%	0%	0%	0%	0%	0%	0%	0%	0%
Data-Flow	0%	0%	0%	0%	0%	0%	0%	0%	0%
Subtotal	0%	0%	0%	0%	0%	0%	0%	0%	0%
Total of all Source Variants									
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	56%	100%	56%	56%	100%	56%	56%	100%	56%
Control-Flow	6%	100%	6%	6%	100%	6%	6%	100%	6%
Data-Flow	12%	100%	12%	12%	100%	12%	12%	100%	12%
Total	11%	100%	11%	11%	100%	11%	11%	100%	11%

6.3. RQ3: How can the Limitations of the SATs be Addressed Through Proof-of-Concept Improvements?

Section 6.3 presents the proof-of-concept improvements for addressing the limitations of SpotBugs, Find Security Bugs, and ESVD. By modifying the source code of the SATs we will confirm or deny hypotheses about what causes the limitations uncovered in section 6.2.

In section 6.3.1 we address the limitations in SpotBugs, presenting proof-of-concept improvements and measuring the difference these improvements produce. We do the same in section 6.3.2 and section 6.3.3 for Find Security Bugs and ESVD respectively.

Proof-of-concept improvements will be given by testing modifications to the detectors' source code. The code changes are published for transparency and replicability on our GitHub repository, `https://github.com/Beba-and-Karlsen/sat-limitation-proofs`.

Similarly to how the results are presented in section 6.2 for RQ2, we will use the Juliet Test Suite to measure changes in the detectors' capabilities. These results can

then easily be compared to the results from section 6.2. At the bottom of each result table, we will add the results from the same detector in RQ2. This additional row will be useful when comparing the performance of the modified source code.

6.3.1. Addressing the Limitations in SpotBugs

We have been able to address the limitations for the SQL injection, HTTP response splitting, path traversal, and cross-site scripting detectors in SpotBugs. The improvements will be described below. The hard-coded password detector in SpotBugs is specifically aimed at a narrow part of the possible hard-coded password vulnerabilities that exist, and cannot easily be extended to detect other types of hard-coded password.

Removing the Reliance on String Concatenations in the SQL Injection Detector

The main limitation of the SQL injection detector is the calculation of vulnerability confidence, which often results in a detection having a confidence too low to be reported by SpotBugs. The requirement of string concatenations being present before any vulnerability can be reported is the most obvious limiting factor. It is unclear why such a requirement was implemented. A string concatenation can combine multiple safe constant strings into a non-constant string, and it is counter-intuitive that such a series of instructions are the driving factor of what is reported as a vulnerability. A tainted source can be successfully detected, but is completely ignored if the code does not also have a string concatenation somewhere.

Additionally, the algorithm for identifying if an unchanged parameter is used by an SQL injection sink is flawed. This check often fails, resulting in increased false positives, but also increased true positives. The problem is due to SpotBugs seeing the SQL query input as originating from a string concatenation, and not the possibly tainted method parameter that is part of the string concatenation.

To prove better results are achievable, we suggest a different post-processing of the data collected during analysis. However, without a larger re-write that includes context-sensitive data-flow analysis, the detector will continue to struggle with the data-flow variants. The improved post-processing calculates if a taint has been seen, if string appends and possibly unsafe string appends has been seen, and if open and close quotation marks both have been seen. Then, this confidence rating is used:

High confidence If tainted data, unsafe append, and open and closing quotation marks have been seen.

Normal confidence If tainted data has been seen, in addition to either an unsafe append or open and closing quotation marks.

Low confidence If an unsafe append and open and closing quotation marks have been seen, but no tainted data.

This will result in 360 fewer true positives, but 2220 fewer false positives, an increase in precision from 43% to 70% and an increase in the discrimination rate from 0% to 49%.

Table 6.15.: Test results of SpotBugs for the injection vulnerabilities. The CWE-89 detector has an altered prioritization algorithm, while the CWE-113 detector has six additional sources added. The original results can be seen in Table 6.2.

	CWE-89 SQL Injection			**CWE-113 HTTP RS**		
Connect TCP, Console ReadLine, Cookies Servlet, Database, Environment, File, Listen TCP, Properties File, Property, Parameter Servlet, URL Connection						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%
Control-Flow	100%	90%	89%	89%	100%	89%
Data-Flow	67%	52%	6%	11%	100%	11%
Subtotal	84%	70%	49%	51%	100%	51%
Query String Servlet						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	0%	0%	0%
Control-Flow	100%	90%	89%	0%	0%	0%
Data-Flow	67%	52%	6%	0%	0%	0%
Subtotal	84%	70%	49%	0%	0%	0%
Total of all Source Variants						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	92%	100%	92%
Control-Flow	100%	90%	89%	81%	100%	81%
Data-Flow	67%	52%	6%	10%	100%	10%
Total	84%	70%	49%	47%	100%	47%
RQ2 Total (Table 6.2)	100%	43%	0%	4%	100%	4%

It is possible for a normal user to change the lowest confidence that will be reported in the SpotBugs UI. This is by default set to normal confidence, but if changed to low confidence the user will get the same number of true and false positives as the original detector. We believe this to be the best of both worlds; a default configuration will sacrifice some true positives to seriously reduce the number of false positives, while a user with very high security requirements still can find all the vulnerabilities while having to filter through many false positives. The results of the improved detector can be seen in Table 6.15, showing only the vulnerabilities with medium and high confidence.

Adding Sources for HTTP Response Splitting, Path Traversal, and Cross-Site Scripting

The main limitations for the HTTP response splitting, path traversal, and cross-site scripting detector, except for the weak taint analysis, is the lack of sources that are detected. We have analyzed possible additional sources that can be added to the three detectors detector. The following additional sources are able to positively change the results on the Juliet Test Suite:

- `javax.servlet.http.Cookie.getValue()`
- `java.lang.System.getenv(...)`
- `java.util.Properties.getProperty(...)`
- `java.lang.System.getProperty(...)`
- `java.sql.ResultSet.getString(...)`
- `java.io.BufferedReader.readLine(...)`

The added sources result in improved results for all three vulnerability categories, but to some varying degrees. For the HTTP response splitting vulnerability, the true positives increased from 57 to 627 TP, increasing the recall and discrimination rate from 4% to 47%. The results can be seen in Table 6.15. For the path traversal vulnerability, only the relative path traversal category saw improvements. The logic for differentiating between relative and absolute path traversal has a limitation that will be discussed later, but when that limitation is addressed the same results will be seen for absolute path traversal too. For now, only the relative path traversal vulnerability saw an increase in true positives from 19 to 209, increasing the recall and discrimination rate from 4% to 47%. The improvements to the relative path traversal category can be seen in Table 6.16, but note that the results for absolute path traversal shown in Table 6.16 are after the improvements to how relative and absolute path traversal are differentiated from each other which is described later. Improvements are also seen for the cross-site scripting vulnerability, where we see a sharp increase in the recall and discrimination rate. For CWE-80 Basic XSS the recall and discrimination rate increase from 3% to 23%, while CWE-81 and CWE-83 increased from 6% to 46%. This increase clearly shows that the missing sources in the taint analysis is a large contributor to the poor results. The slightly lower improvements seen for CWE-80 Basic XSS compared to the other two XSS categories are due to the subpar taint analysis capabilities of SpotBugs, and will be addressed later in this section. Table 6.17 shows the improved results for the cross-site scripting vulnerabilities, but note that the results for CWE-80 Basic XSS include improvements to the taint analysis that will be presented later.

For the HTTP response splitting, path traversal, and cross-site scripting vulnerabilities, no false positives are added. Half of the improved detection capabilities are the result of the first five sources in the list above, while the other half of the increase is

the result of the last source. The last source, `BufferedReader`, although mostly used to read possibly tainted data, can be used in such a way that it contains untainted data. This choice is further discussed in section 7.3.

OpCodeStack.java

```
if (seen == Const.INVOKEVIRTUAL && "getValue".equals(methodName)
        && "javax/servlet/http/Cookie".equals(clsName)) {
    pop();
    Item result = new Item("Ljava/lang/String;");
    result.setServletParameterTainted();
    result.source = XFactory.createReferencedXMethod(dbc);
    result.setPC(dbc.getPC());
    push(result);
    return;
}
```

Figure 6.17.: One of the five sources added to SpotBugs. Finds usages of `Cookie.getValue()`, and marks the returned value as tainted. This includes popping the item off the instruction stack, altering it, then pushing it back onto the instruction stack.

Manual, and to some extent difficult, changes to the source code had to be made when adding the six additional sources listed above. Items had to be popped off the instruction stack, altered, then pushed back onto the instruction stack. Binary JVM bytecode had to be decompiled and analyzed to understand which instructions to detect. See Figure 6.17 for one of the five sources that was added. The algorithm to detect vulnerable sources is only used by the combined HTTP response splitting, path traversal, and cross-site scripting detector, as opposed to being shared by many detectors. Making it difficult to add additional sources might result in less maintenance and improvement. Not sharing the algorithm for detecting vulnerable sources lead to improvements for one detector not benefiting other detectors. An improved storage repository for sources, combined with a generalizable taint analysis framework, would be beneficial to all the detectors in SpotBugs.

The limited ability to perform data-flow analysis is the reason for why the combined HTTP response splitting, path traversal, and cross-site scripting detector does not go above a 47% recall. As can be seen in Table 6.15, the detector performs poorly on control-flow and data-flow variants for HTTP response splitting. The same results are seen for path traversal and cross-site scripting as well. When the detector is unable to follow the data from source to sink, which is harder for the data-flow variants compared to the control-flow variants, it is unable to identify the data as tainted. When the data cannot be identified as tainted, no vulnerability is reported. The limited data-flow capabilities also result in one of the source variants, Query String Servlet, having a zero percent recall, see Table 6.15, 6.16, and 6.17. Not all vulnerable sources can be directly used as

a string value - some values must be converted from other data types. One example of data being converted and otherwise processed is the use of string tokenization in Query String Servlet. In Java, a string tokenization is used to separate a string into sub-strings based on a delimiter. The detector does not understand that the data from the string tokenization originates from the tainted string, but rather see the data originating from the tokenizer itself. This is problematic, as the string tokenizer source can be something innocent and safe. This is one of the reasons for the recall not being higher.

Separating Between Relative and Absolute Path Traversal

Although the relative path traversal detector saw large improvements when adding vulnerable sources as described above, the absolute path traversal detector did not produce improved results. This is due to how the path traversal detector differentiates between relative and absolute path traversal. The taint analysis is unable to properly track tainted data for the relative path traversal vulnerability due to string concatenations, as discussed in section 6.2.3. Not only is the poor taint analysis a limitation, but the way in which the detector decides if it has lost track of the tainted data is also a limitation. The detector compares the data source to a hard-coded source. If the data source does not match the hard-coded source, it concludes that the data could not be tracked and reports a relative path traversal. By altering the detector in such a way that it verifies if the data source is actually lost, the absolute path traversal also improves drastically. See the code modifications in Figure 6.18. This results in an increase in recall and discrimination rate from 4% to 40%. See Table 6.16 for the full results for both relative and absolute path traversal.

CrossSiteScripting.java

```
String bugPattern = (path != null && path.getReturnValueOf() != null ) ?
 ↪  "PT_ABSOLUTE_PATH_TRAVERSAL" : "PT_RELATIVE_PATH_TRAVERSAL";
```

Figure 6.18.: The changes to the path traversal detector in SpotBugs to determine if the taint analysis has lost track of data.

Improving the Tracking Capabilities of the Taint Analysis

The cross-site scripting detector in SpotBugs has a recall and discrimination rate of between 3% and 6%, but a precision of 100%. This is mainly due to two factors: few detected sources and a taint analysis that is struggling to follow data. Adding vulnerable sources has been carried out above, while this section will make improvements to the taint analysis.

The taint analysis capabilities of the XSS detector in SpotBugs severely limits its ability to track data from source to sink. While this holds true for all three of the XSS CWEs, it especially affects the CWE-80 Basic XSS test cases in the Juliet Test Suite. As described in section 6.2.4, half of the CWE-80 test cases executes a `String.replaceAll()`

Table 6.16.: Test results of SpotBugs for the path traversal vulnerabilities with additional tainted sources added and changes to taint analysis. The original results can be seen in Table 6.4.

	CWE-23 Relative Path Trv.			CWE-36 Absolute Path Trv.		
Connect TCP, Console ReadLine, Cookies Servlet, Database, Environment, File, Listen TCP, Properties File, Property, Parameter Servlet, URL Connection						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%
Control-Flow	89%	100%	89%	50%	100%	50%
Data-Flow	11%	100%	11%	33%	100%	33%
Subtotal	51%	100%	51%	43%	100%	43%
Query String Servlet						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	0%	0%	0%	0%	0%	0%
Control-Flow	0%	0%	0%	0%	0%	0%
Data-Flow	0%	0%	0%	0%	0%	0%
Subtotal	0%	0%	0%	0%	0%	0%
Total of all Source Variants						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	92%	100%	92%	92%	100%	92%
Control-Flow	81%	100%	81%	46%	100%	46%
Data-Flow	10%	100%	10%	31%	100%	31%
Total	47%	100%	47%	40%	100%	40%
RQ2 Total (Table 6.4)	4%	100%	4%	4%	100%	4%

on the data before it is sent to the sink. The data tracking capabilities in the XSS detector is unable to understand that the output of the `replaceAll` method is tainted, resulting in half of the test cases not being detected. The taint analysis can be modified to track data through e.g. a `replaceAll` method, which we have shown by altering the taint analysis algorithm for the XSS detector. An excerpt of the changes carried out on the taint analysis algorithm can be seen in Figure 6.19. This change results in 152 new true positives.

In total, the added sources and the improved taint analysis algorithm increased the true positives from 19 to 304 for the CWE-80 test cases and from 19 to 152 for each

Table 6.17.: Test results of SpotBugs for the cross-site scripting vulnerabilities with additional tainted sources added and changes to tracking capabilities of taint analysis. The original results can be seen in Table 6.5.

	CWE-80 Basic XSS			CWE-81 XSS Error Msg.			CWE-83 XSS Attrib.		
Connect TCP, Database, File, Cookies Servlet, Parameter Servlet, Listen TCP, Properties File, URL Connection									
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%	100%	100%	100%
Control-Flow	89%	100%	89%	89%	100%	89%	89%	100%	89%
Data-Flow	11%	100%	11%	11%	100%	11%	11%	100%	11%
Subtotal	51%	100%	51%	51%	100%	51%	51%	100%	51%
Query String Servlet									
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	0%	0%	0%	0%	0%	0%	0%	0%	0%
Control-Flow	0%	0%	0%	0%	0%	0%	0%	0%	0%
Data-Flow	0%	0%	0%	0%	0%	0%	0%	0%	0%
Subtotal	0%	0%	0%	0%	0%	0%	0%	0%	0%
Total of all Source Variants									
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	89%	100%	89%	89%	100%	89%	89%	100%	89%
Control-Flow	79%	100%	79%	79%	100%	79%	79%	100%	79%
Data-Flow	10%	100%	10%	10%	100%	10%	10%	100%	10%
Total	46%	100%	46%	46%	100%	46%	46%	100%	46%
RQ2 (Tab. 6.5)	3%	100%	3%	6%	100%	6%	6%	100%	6%

of the CWE-81 and CWE-83 test cases. No false positives were added, resulting in the precision remaining at 100%. The detailed results after the improvement can be seen in Table 6.17.

OpCodeStack.java

```
if (
"java/lang/String".equals(clsName) && "replaceAll".equals(methodName) &&
"(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;".equals(signature)
&& getStackDepth() >= 3) {
    // first.replaceAll(second, third)
    Item third = getStackItem(0);
    Item second = getStackItem(1);
    Item first = getStackItem(2);

    if (first.isServletParameterTainted() ||
    second.isServletParameterTainted() ||
    third.isServletParameterTainted()) {
        servletRequestParameterTainted = true;
    }
}
```

Figure 6.19.: Excerpt from added tracking capabilities in SpotBugs. Finds usages of `String.replaceAll(...)`, checks if any of the possible values are tainted, and marks the resulting stack item accordingly.

6.3.2. Addressing the Limitations in Find Security Bugs

We have been able to address the limitations for all of the five injection detectors, in addition to the path traversal and cross-site scripting detectors in Find Security Bugs. The limitations that are addressed below are the inconsistent vulnerability confidence ranking and missing sources.

As discussed in section 6.2.6, there are five hard-coded password detectors in Find Security Bugs. The collection of detectors for hard-coded passwords in Find Security Bugs mostly consists of simple logic to accomplish a simple goal, with the exception of the more advanced data-flow analysis detector. The simple detectors are detecting what is expected of them, and their only limitations would be the narrow amount of vulnerable code patterns they detect. The data-flow analysis detector is trying to accomplish more, and while also detecting more than the others, still lacks logic for detecting many control- and data-flow cases. The hard-coded password detector that utilizes data-flow analysis is missing the ability to track data through multiple methods or classes, and adding that functionality would require a complete re-write of the detector.

While the path traversal detector is able to reliably identify path traversal vulnerabilities, the detector is not able to separate the identified vulnerabilities into relative or absolute path traversal. The ability to separate them can be useful when trying to identify countermeasures that can be implemented in the code, although some countermeasures such as whitelisting input is the same for both types of path traversal. The sinks for both types of path traversal are the same, and trying to separate the two path traversal types would not allow the detector to keep using the taint analysis framework

that is provided by Find Security Bugs, as this framework does not contain any helpful data to classify the path traversal type. For these reasons, the ability to identify if the vulnerability is a relative or absolute path traversal would require a complete re-write of the detector.

Table 6.18.: Test results of Find Security Bugs for the injection vulnerabilities with additional tainted sources added. The original results can be seen in Table 6.6. This table aims for perfect recall, while Table 6.20 aims for perfect precision.

	CWE-78 OSC Injection			**CWE-89 SQL Injection**			**CWE-90 LDAP Injection**		
Total of all Source Variants (Connect TCP, Console ReadLine, Cookies Servlet, Database, Environment, File, Listen TCP, Parameter Servlet, Property, Properties File, Query String Servlet, URL Connection)									
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%	100%	100%	100%
Control-Flow	100%	100%	100%	100%	100%	100%	100%	100%	100%
Data-Flow	100%	78%	72%	100%	78%	72%	100%	78%	72%
Total	100%	88%	86%	100%	88%	86%	100%	88%	86%
RQ2 (Tab. 6.6)	86%	86%	72%	86%	86%	72%	86%	86%	72%

Alternative Vulnerability Confidence Ranking

The injection, path traversal and cross-site scripting detectors in Find Security Bugs take two different approaches when it comes to prioritizing soundness or completeness. There is no definitive answer to which is the correct approach. Some users want less false positives and are willing to sacrifice some true positives for that possibility [Christakis and Bird, 2016]. On the other hand there might be users dealing with very sensitive data that are willing to look through a lot of false positives in exchange for the analysis to detect all of the vulnerabilities. This is an example of the trade-off between soundness and completeness described in section 3.1. We have made code changes demonstrating both of these approaches. The HTTP response splitting detector already has zero false positives but with less true positives, so for this detector we have shown that it can be changed to detect all of the vulnerabilities if it allows some false positives. For the other four injection detectors, in addition to the path traversal and cross-site scripting detectors, we have done the opposite. These six detectors already find all of the vulnerabilities but also include some false positives. With our code changes they can eliminate all of the false positives, but they have to sacrifice some of the true positives. The code modifications are changing the confidence ranking algorithm to favor either

Table 6.19.: Test results of Find Security Bugs for the injection vulnerabilities with additional tainted sources added and confidence ranking changes to CWE-113. The original results can be seen in Table 6.7. This table aims for perfect recall, while Table 6.21 aims for perfect precision.

	CWE-113 HTTP RS			**CWE-643 XPath Injection**		
Total of all Source Variants (Connect TCP, Console ReadLine, Cookies Servlet, Database, Environment, File, Listen TCP, Parameter Servlet, Property, Properties File, Query String Servlet, URL Connection)						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%
Control-Flow	100%	100%	100%	100%	100%	100%
Data-Flow	100%	78%	72%	100%	78%	72%
Total	100%	88%	86%	100%	88%	86%
RQ2 Total (Table 6.7)	74%	100%	74%	86%	86%	72%

Table 6.20.: Test results of Find Security Bugs for the injection vulnerabilities with additional tainted sources added and confidence ranking changes. The original results can be seen in Table 6.6. This table aims for perfect precision, while Table 6.18 aims for perfect recall.

	CWE-78 OSC Injection			**CWE-89 SQL Injection**			**CWE-90 LDAP Injection**		
Total of all Source Variants (Connect TCP, Console ReadLine, Cookies Servlet, Database, Environment, File, Listen TCP, Parameter Servlet, Property, Properties File, Query String Servlet, URL Connection)									
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%	100%	100%	100%
Control-Flow	100%	100%	100%	100%	100%	100%	100%	100%	100%
Data-Flow	78%	100%	72%	78%	100%	72%	78%	100%	72%
Total	89%	100%	86%	89%	100%	86%	89%	100%	86%
RQ2 (Tab. 6.6)	86%	86%	72%	86%	86%	72%	86%	86%	72%

Table 6.21.: Test results of Find Security Bugs for the injection vulnerabilities with additional tainted sources added and confidence ranking changes to CWE-643. The original results can be seen in Table 6.7. This table aims for perfect precision, while Table 6.19 aims for perfect recall.

	CWE-113 HTTP RS			**CWE-643 XPath Injection**		
Total of all Source Variants (Connect TCP, Console ReadLine, Cookies Servlet, Database, Environment, File, Listen TCP, Parameter Servlet, Property, Properties File, Query String Servlet, URL Connection)						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%
Control-Flow	100%	100%	100%	100%	100%	100%
Data-Flow	78%	100%	72%	78%	100%	72%
Total	89%	100%	86%	89%	100%	86%
RQ2 Total (Table 6.7)	74%	100%	74%	86%	86%	72%

recall or precision by specifying if data with an unknown state reaching a vulnerable sink should be reported or not. By reporting data with an unknown state, recall is favored, and vice versa. When recall is favored, the results for the injection detectors are as shown in Table 6.18 and 6.19. If precision is favored, the results for the injection detectors are as shown in Table 6.20 and 6.21. Note that Table 6.18, 6.19, 6.20, and 6.21 also include the two additional sources that are added in the next section. The results for the path traversal and XSS detectors are, after similar code modifications, the exact same as the injection detectors, and will not be repeated in their own tables. For both the path traversal and XSS detectors, modifications to the confidence ranking increases the precision from 88% to 100%, while slightly reducing the recall from 100% to 89%. It is up to the developers of Find Security Bugs to choose which approach to take, which is why we have demonstrated that both are possible. In our opinion, the default behavior of Find Security Bugs should be to have zero false positives, as this is favored by most users [Christakis and Bird, 2016; Johnson et al., 2013]. If the user wants to see all of the uncertain true and false positives, they can lower the threshold for showing possible vulnerabilities. This option is also included in our code changes, as the uncertain detections are reported with a low confidence, which by default is not shown to the user unless explicitly chosen in the settings.

The modifications of the confidence ranking are slightly different between all the detectors. The main difference is that the modifications that favor recall reports a vulnerability when data with an unknown state reaches a vulnerable sink, while the modifications favoring precision only reports a vulnerability if the state of the data is set as tainted.

The confidence ranking for the modifications to favor recall can be summarized as follows:

High confidence If the state is set as tainted.

Normal confidence If the state is not safe, i.e. the state being unknown or invalid.

The confidence ranking for the modifications to favor precision can be summarized as follows:

Normal confidence If the state is set as tainted.

Low confidence If the state is not safe, i.e. the state being unknown or invalid. Low confidence vulnerabilities are not shown by default, but can be enabled in the settings.

The different confidence that the injection, path traversal, and cross-site scripting detectors report is a problem in other ways too. While the HTTP response splitting detector reports its most certain detections as medium and uncertain detections as low confidence, the other detectors report their most certain detections as high and uncertain detections as medium confidence. There is no developer guidelines that say which is the correct approach, although an advanced user guide states that the latter is how the injection detectors should behave [Find Security Bugs, 2017]. While missing developer guidelines is a limitation, the fact that the detectors use a different confidence ranking is also a limitation. If the user adjusts the plugin settings to hide false positives for the SQL injection detector - that is hiding both low and medium confidence detections - it will also hide every single detection from the HTTP response splitting detector, as even the certain detections are only given a medium confidence. A consistent behavior should be expected.

Adding Missing Sources

The two missing sources are `System.getenv()` and `System.getProperty()`. As discussed in section 6.2.5, these are considered safe by choice by the developers. Although there is no documentation of the following workaround, it is possible to enable these two missing sources as possible tainted data. By adding a special system variable into your system of choice, both sources will be considered unsafe. See Figure 6.20 for a visualization of how to enable this functionality. This greatly improves the number of true positives detected by the injection and path traversal detectors, without affecting the precision. The results for the improved injection detectors can be seen in Table 6.18 and 6.19, while the results for the improved path traversal detector can be seen in Table 6.22. It is unclear why this is an opt-in feature rather than an opt-out one.

Adding the two missing sources has no effect on the results of the XSS detector in Find Security Bugs, as the cross-site scripting test cases in the Juliet Test Suite do not use the Environment and Property source variants. If the Environment and Property source variants were used by the XSS test cases, it would result in equal improvements

of the results for the cross-site scripting detector, as the XSS detector is also missing these two sources.

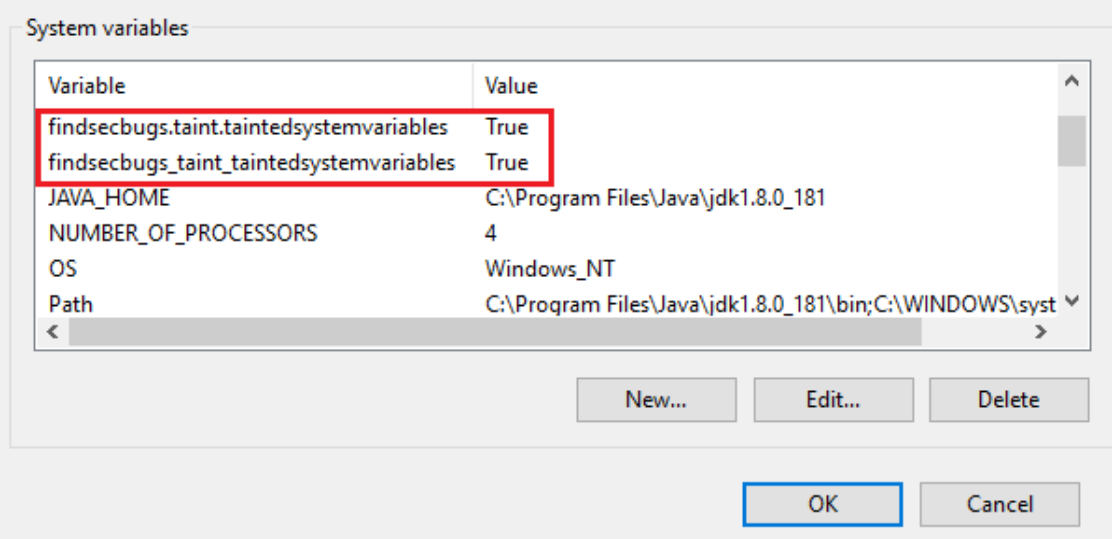

Figure 6.20.: The Microsoft Windows setting to define system variables, which in this case is used to define the data sources in Figure 6.11 as tainted. To be easier to use in environments where system variables cannot use periods, it is possible to use underscores instead.

Table 6.22.: Test results of Find Security Bugs for the path traversal vulnerabilities with additional tainted sources added. The original results can be seen in Table 6.9.

	CWE-23 Relative Path Trv.			**CWE-36 Absolute Path Trv.**		
Total of all Source Variants (Connect TCP, Console ReadLine, Cookies Servlet, Database, Environment, File, Listen TCP, Parameter Servlet, Properties File Property, Query String Servlet, URL Connection)						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%
Control-Flow	100%	100%	100%	100%	100%	100%
Data-Flow	100%	78%	72%	100%	78%	72%
Total	100%	88%	86%	100%	88%	86%
RQ2 Total (Table 6.9)	86%	86%	72%	86%	86%	72%

6.3.3. Addressing the Limitations in ESVD

In section 6.2.9, we created hypotheses about what caused the poor results for ESVD. Our hypotheses are that the algorithm for detecting SQL injection through string concatenations is faulty, the XPath detector struggling with the natural complexity of the test cases, as well as the resource lists of ESVD missing sources and sinks. In this section, we will attempt to prove these claims by modifying the source code of ESVD and reevaluating the tool. We are able to address all of the limitations except for the struggling XPath detector and the missing sinks for the hard-coded password detector.

Removing String Concatenation Detection from the SQL Injection Detector

As mentioned in section 6.2.9, ESVD produces many true and false positives for SQL injection. This is because the SQL injection detector reports on all cases of string concatenation used in conjunction with SQL injection sinks.

When removing this unique detection method for SQL injection, the SQL injection detector uses the underlying algorithm of ESVD instead. The results in Table 6.23 are more similar to the results of the OS command injection detector, which is logical as they now use the same implementation. However, there is an exception in control-flow variant 21 which is detected for SQL injection, but not for OS command injection. This is very curious as the two detectors use the same algorithm with the only difference being which sinks are used. This points to inconsistencies in the control-flow analysis of ESVD. It is likely that this is the result of ESVD being a proof of concept instead of a finished product.

Adding Sinks for LDAP Injection and HTTP Response Splitting

ESVD did not detect any vulnerabilities for LDAP injection or HTTP response splitting. As explained in section 6.2.9, this is because ESVD lacks the sinks used by the test cases in the Juliet Test Suite.

By adding the missing sinks, the LDAP injection and HTTP response splitting detectors are able to detect some occurrences of the vulnerabilities as shown in Table 6.24. The results for LDAP injection are identical to OS command injection, while the results for HTTP response splitting are identical to SQL injection when detection of string concatenation is removed.

When executing the HTTP response splitting detector with standard settings in ESVD, the detector will only report on one of the three sinks used in the Juliet Test Suite. It reports on `addCookie`, but not on `addHeader` or `setHeader`. This is strange as all sinks have now been added. By manually inspecting the test cases, we discover that ESVD reports cross-site scripting on `addHeader` and `setHeader`. By altering the settings and turning off the cross-site scripting detector, ESVD reports HTTP response splitting on these instead. This exposes a critical limitation in ESVD's implementation as it does not report multiple vulnerabilities on a single line of code. If a false positive is first reported, in this case cross-site scripting, then a true positive may not be reported as in this case with HTTP response splitting.

Table 6.23.: Test results of ESVD for CWE-89 SQL Injection when removing the detection algorithm for use of string concatenation in conjunction with a sink. The original results can be seen in Table 6.11.

	CWE-89 SQL Injection		
Cookies Servlet, Database, Environment, Parameter Servlet, Properties File, Property, Query String Servlet			
Flow Variant	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%
Control-Flow	17%	100%	17%
Data-Flow	22%	100%	22%
Subtotal	22%	100%	22%
Connect TCP, Console ReadLine, File, Listen TCP, URL Connection			
Flow Variant	**Rec.**	**Prec.**	**Disc.**
Baseline	0%	0%	0%
Control-Flow	0%	0%	0%
Data-Flow	0%	0%	0%
Subtotal	0%	0%	0%
Total of all Source Variants			
Flow Variant	**Rec.**	**Prec.**	**Disc.**
Baseline	58%	100%	58%
Control-Flow	10%	100%	10%
Data-Flow	13%	100%	13%
Total	13%	100%	13%
RQ2 Total (Table 6.11)	65%	39%	0%

Table 6.24.: Test results of ESVD for CWE-90 LDAP Injection and CWE-113 HTTP Response Splitting when adding the missing sinks. The original results were 0% recall, precision, and discrimination rate for both of CWE-90 and CWE-113.

	CWE-90 LDAP Injection			**CWE-113 HTTP Resp. Sp.**		
Cookies Servlet, Database, Environment, Parameter Servlet, Properties File, Property, Query String Servlet						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%
Control-Flow	11%	100%	11%	17%	100%	17%
Data-Flow	22%	100%	22%	22%	100%	22%
Subtotal	19%	100%	19%	22%	100%	22%
Connect TCP, Console ReadLine, File, Listen TCP, URL Connection						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	0%	0%	0%	0%	0%	0%
Control-Flow	0%	0%	0%	0%	0%	0%
Data-Flow	0%	0%	0%	0%	0%	0%
Subtotal	0%	0%	0%	0%	0%	0%
Total of all Source Variants						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	58%	100%	58%	58%	100%	58%
Control-Flow	6%	100%	6%	10%	100%	10%
Data-Flow	13%	100%	13%	13%	100%	13%
Total	11%	100%	11%	13%	100%	13%
RQ2 Total	0%	0%	0%	0%	0%	0%

Adding Sinks for Hard-Coded Password

By adding the two missing sinks for hard-coded passwords as discussed in section 6.2.10, namely Kerberos Key and Password Authentication, one would believe these test cases should be detected. However, the hard-coded password detector is still not able to detect the vulnerabilities, even after adding the sinks to the resource list. Both of the sinks use a `char` array as a parameter, and the `String` input must be converted into a `char` array before injected into the sinks. Unfortunately, the hard-coded password detector is not able to track the input through the conversion from `String` to `char` which leads to no vulnerabilities detected.

We changed one of the test cases for both sinks to use a hard-coded `char` array instead, as shown in Figure 6.21, and ESVD was able to detect it. This experiment confirms that it is the `String` to `char` conversion that the detector is not able to track and prevents ESVD from detecting the vulnerabilities.

```
String data = "7e5tc4s3";

/* POTENTIAL FLAW: data used as password in KerberosKey() */
KerberosKey key = new KerberosKey(principal, data.toCharArray(), null);
```

```
char[] array = { '7', 'e', '5', 't', 'c', '4', 's', '3' };

/* POTENTIAL FLAW: data used as password in KerberosKey() */
KerberosKey key = new KerberosKey(principal, array, null);
```

Figure 6.21.: Simplified version of the test case for the Kerberos Key sink. Changing the hard-coded variable from a `String` to a `char` array makes ESVD detect the vulnerability.

Removing Complexity of Baseline Case for XPath Injection

As mentioned in section 6.3.3, we hypothesize that the reason for why the XPath injection detector is not able to detect any of the vulnerabilities is the natural complexity of the test cases in the Juliet Test Suite. To test this hypothesis, we modify one of the baseline test cases and remove unnecessary statements as shown in Figure 6.22. We remove the `if`-statement encapsulating the sink as well as the string operations altering the `data` variable containing the vulnerable input. However, even with these changes the XPath injection detector does not work, indicating that the implementation of ESVD is the cause for this and not the Juliet Test Suite.

CWE643_Xpath_Injection__connect_tcp_01.java

```
...
if (data != null) {
    String [] tokens = data.split("||");
    if (tokens.length < 2) {
        return;
    }
    String username = tokens[0];
    String password = tokens[1];
    XPath xPath = XPathFactory.newInstance().newXPath();
    InputSource inputXml = new InputSource(xmlFile);
    /* POTENTIAL FLAW: user input is used without validate */
    String query = "//users/user[name/text()='" + username +
                   "' and pass/text()='" + password + "']" +
                   "/secret/text()";
    String secret = (String)xPath.evaluate(query, inputXml,
     ↪  XPathConstants.STRING);
}
```

CWE643_Xpath_Injection__connect_tcp_01.java (modified)

```
...
XPath xPath = XPathFactory.newInstance().newXPath();
InputSource inputXml = new InputSource(xmlFile);
/* POTENTIAL FLAW: user input is used without validate */
String secret = (String)xPath.evaluate(data, inputXml,
 ↪  XPathConstants.STRING);
```

Figure 6.22.: Removing the complexity from the baseline test case for XPath injection.

Adding Sources for all Detectors

As mentioned in sections 6.2.9, 6.2.11, and 6.2.12, there are five source variants none of the detectors in ESVD detect any vulnerabilities for. These are Connect TCP, Console ReadLine, File, Listen TCP, and URL Connection. However, ESVD is easy to extend as it uses a general algorithm which retrieves sources and sinks from resource lists stored in XML-files.

The resource lists consist of method signatures. Thus, a source must be in the form of a method in order for ESVD to recognize it. All of Connect TCP, Listen TCP, and URL Connection use a method called `getInputStream()`, however, from two different classes. By adding these two method signatures to the resource list as shown in Figure 6.23, all of the detectors in ESVD should be able to detect these source variants as well.

The test cases for the Console ReadLine and File source variants do not use a method

entry-point.xml

```
<!-- java.net.URLConnection -->
<entrypoint id="82">
  <qualifiedname>java.net.URLConnection</qualifiedname>
  <methodname>getInputStream</methodname>
</entrypoint>
<!-- java.net.Socket -->
<entrypoint id="83">
  <qualifiedname>java.net.Socket</qualifiedname>
  <methodname>getInputStream</methodname>
</entrypoint>
```

Figure 6.23.: The method signatures added to the XML-file including the resource list for sources in ESVD. The method signatures correspond to the source variants: Connect TCP, Listen TCP, and URL Connection.

entry-point.xml

```
<!-- java.io.File -->
<entrypoint id="84">
  <qualifiedname>java.io.File</qualifiedname>
  <methodname>File</methodname>
  <parameters type="java.lang.String" />
</entrypoint>
```

Figure 6.24.: The method signature added to the XML-file including the resource list for sources in ESVD. The method signature corresponds to the File source variant.

as a source, but rather object instantiations. As mentioned in section 6.2.11, ESVD is capable of using an instantiation as a sink, and it turns out it works just as well with a source. The File source variant in the Juliet Test Suite uses instantiations of `java.io.File`. By adding the constructor for this class to the lists of sources as shown in Figure 6.24, ESVD detects the File source variant as well.

In the test cases for the File source variant, the `java.io.File` object is always subsequently used in a `FileInputStream` object as shown in Figure 6.25. This causes an issue for the path traversal detector which has `FileInputStream` listed as a sink, and consequently produces a lot of false positives when `Java.io.File` is added as a source. The sink included in ESVD for path traversal has the method signature `java.io.FileInputStream(Object)`. However, the parameter should be a `String` since the vulnerability is caused by receiving a tainted path variable originating from user input. With `Object` as the parameter, ESVD will report on all cases of `FileInputStream`

```
String data = "";
file = new File("C:\\data.txt");
stream = new FileInputStream(file);
reader = new InputStreamReader(stream, "UTF-8");
readerBuffered = new BufferedReader(reader);
```

Figure 6.25.: Simplified version of the File source variant used in the Juliet Test Suite. The `File` always passes through the `FileInputStream` which causes a false positive for the path traversal detector.

no matter the datatype of the argument since every class in Java inherits the `Object` class. Changing the method signature to `java.io.FileInputStream(String)` should solve this problem. However, when reevaluating ESVD after these changes, it reports absolutely nothing for path traversal. This repeated erratic behavior of ESVD as discussed throughout section 6.2 and 6.3 indicates that ESVD is further from a finished product than it seemed like initially.

In the case of the source variant Console ReadLine, the source is the instantiation of `InputStreamReader(System.in, "UTF-8")` where `System.in` specifies to read from the console. By adding its constructor to the resource list of sources, ESVD detects this as well. An `InputStreamReader` converts byte streams to character streams, and is technically not a vulnerable source in itself. It is by passing `System.in` as an argument that it becomes insecure. Unfortunately, ESVD can not use a class field as a source, so it cannot mark `System.in` as a source, and the only option is to use `InputStreamReader`. We will discuss this further in section 7.3.

The results after adding these sources in addition to the previously discussed proof-of-concept improvements are presented in Table 6.26, 6.25, and 6.27. For the path traversal detector, we have used `java.io.InputStreamReader` as the source for the File source variant instead of `java.io.File` because of the issues presented above. In total, this shows that ESVD is capable of producing significantly better results if all sources and sinks are included. However, the recall of ESVD is only between 18% and 22% for the different vulnerabilities. This can not compare to Find Security Bugs which has a recall between 89% and 100% for all vulnerabilities except for hard-coded password which has 43%.

Table 6.25.: Test results of ESVD for the path traversal vulnerabilities when adding the missing sources. The original results can be seen in Table 6.13.

	CWE-23 Relative Path Trv.			**CWE-36 Absolute Path Trv.**		
Total of all Source Variants (Connect TCP, Console ReadLine, Cookies Servlet, Database, Environment, File, Listen TCP, Parameter Servlet, Properties File, Property, Query String Servlet, URL Connection)						
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%
Control-Flow	11%	100%	11%	11%	100%	11%
Data-Flow	22%	100%	22%	22%	100%	22%
Total	19%	100%	19%	19%	100%	19%
RQ2 Total (Table 6.13)	11%	100%	11%	11%	100%	11%

Table 6.26.: Test results of ESVD for the injection vulnerabilities when adding the missing sources as well as keeping changes previously discussed in section 6.3.3. The original results can be seen in Table 6.11, and the results of previous changes can be seen in Table 6.23 and Table 6.24.

	CWE-78 OSC Injection			**CWE-89 SQL Injection**			**CWE-90 LDAP Injection**			**CWE-113 HTTP Resp. Sp.**		
Total of all Source Variants (Connect TCP, Console ReadLine, Cookies Servlet, Database, Environment, File, Listen TCP, Parameter Servlet, Properties File, Property, Query String Servlet, URL Connection)												
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Control-Flow	11%	100%	11%	17%	100%	17%	11%	100%	11%	17%	100%	17%
Data-Flow	22%	100%	22%	22%	100%	22%	22%	100%	22%	22%	100%	22%
Total	19%	100%	19%	22%	100%	22%	19%	100%	19%	22%	100%	22%
RQ2 (Tab. 6.11)	11%	100%	11%	65%	39%	0%	0%	0%	0%	0%	0%	0%

Table 6.27.: Test results of ESVD for the cross-site scripting vulnerabilities when adding the missing sources. The original results can be seen in Table 6.14

	CWE-80 Basic XSS			**CWE-81 XSS Error Msg.**			**CWE-83 XSS Attrib.**		
Total of all Source Variants (Connect TCP, Console ReadLine, Cookies Servlet, Database, Environment, File, Listen TCP, Parameter Servlet, Properties File, Property, Query String Servlet, URL Connection)									
Flow Variant	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
Baseline	100%	100%	100%	100%	100%	100%	100%	100%	100%
Control-Flow	11%	100%	11%	11%	100%	11%	11%	100%	11%
Data-Flow	22%	100%	22%	22%	100%	22%	22%	100%	22%
Total	19%	100%	19%	19%	100%	19%	19%	100%	19%
RQ2 (Tab. 6.14)	11%	100%	11%	11%	100%	11%	11%	100%	11%

6.4. Summary

In this section, we will present summarized results from sections 6.2 and 6.3. When analyzing the implementations of the SATs, we discovered both strengths and limitations. To give a concise summary of our findings, we present them in Table 6.28 and 6.29, where strengths are indicated by + and limitations by −.

This summary table is divided into general implementation as well as detector-specific strengths and limitations. The general implementation corresponds to all detectors used by the SAT, while the detector-specific ones are divided into the same vulnerability categories used for the results in sections 6.2 and 6.3. Be aware that some categories share the same algorithm, and consequently share the same limitations and strengths as well. This is the case for Find Security Bugs, where all the injection vulnerabilities are combined in the table.

Table 6.30 presents all of the changes in the performance results from section 6.2 and 6.3. This table demonstrates how the proof-of-concept improvements affected the SATs' performance. Improved results are written in green, while reduced performance is written in red.

Table 6.28.: General strengths and limitations of the detectors as well as specifics for injection vulnerabilities.

	SpotBugs	Find Security Bugs	ESVD
General Implementation	– No guidelines for confidence.	+ Shared list for sources between vulnerabilities. – No guidelines for confidence.	+ Uses a general algorithm for all detectors. + Taint analysis gives fewer false positives. + Shared list for sources between vulnerabilities. – Detects only 7 of 12 source variants. – Does not track data through `if`- and `switch`-statements. – No context-sensitive DFA. – Cannot report multiple vulnerabilities on the same line of code.
Injection Vulnerabilities			
CWE-78 OS Cmd Injection	*Not covered by any detectors.*	+ Taint analysis gives fewer false positives. – Detects only 10 of 12 source variants. – Loses track of data in advanced data-flow variants. – Inconsistent confidence ranking where the HTTP response splitting detector targets precision, while the other detectors target recall.	*No detector-specific strengths or limitations.*
CWE-89 SQL Injection	– Reports on all use of string concatenation in conjunction with a sink.		– Reports on all use of string concatenation in conjunction with a sink. – Uses pattern matching instead of DFA
CWE-90 LDAP Injection	*Not covered by any detectors.*		– Detects nothing because of missing sink.
CWE-113 HTTP Resp. Split.	– Detects only 1 of 12 source variants. – Poor taint analysis.		– Detects nothing because of missing sink.
CWE-643 XPath Injection	*Not covered by any detectors.*		– Detects nothing despite including the sink.

Table 6.29.: Strengths and limitations of the detectors for hard-coded passwords, path traversal and cross-site scripting.

	SpotBugs	Find Security Bugs	ESVD
Hard-Coded Password			
CWE-259 Hard-coded Pwd	– Only detects database passwords. – No control-flow analysis.	+ Shared list of common names for password variables. – DFA only works within a method.	– Mislabeled as Security Misconfiguration. – Only detects database passwords. – Inconsistent behavior in the CFA and DFA algorithms when using literals instead of sources.
Path Traversal			
CWE-23 Rel. Path Trav.	– Detects only 1 of 12 source variants. – Poor taint analysis. – Depends on poor taint anaylsis to differentiate between relative and absolute path traversal	+ Taint analysis gives fewer false positives. – Detects only 10 of 12 source variants. – Loses track of data in advanced data-flow variants. – Cannot differentiate between relative and absolute path traversal.	– Cannot differentiate between relative and absolute path traversal.
CWE-36 Abs. Path Trav.			
Cross-Site Scripting			
CWE-80 Basic XSS **CWE-81 XSS Error Msg.** **CWE-83 XSS Attributes**	– Detects only 1 of 9 source variants. – Poor taint analysis. – Does not detect `replaceAll` sink.	+ Taint analysis gives fewer false positives. + Detects all source variants. + Detects both the regular and the replaceAll sink. – Loses track of data in advanced data-flow variants.	+ Detects both the regular and the replaceAll sink.

Table 6.30.: The total results for ESVD, SpotBugs, and Find Security Bugs for all vulnerabilities. The results of both RQ2 and RQ3 are included to demonstrate the performance improvement of the modifications made in RQ3. Green numbers indicate increased performance, while red numbers indicate decreased performance. For Find Security Bugs, we have included the results that prioritize precision over recall.

		ESVD			SpotBugs			FindSecBugs		
Injection										
		Rec.	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
CWE-78 OS Cmd Inj.	RQ2	11%	100%	11%	0%	0%	0%	86%	86%	72%
	RQ3	19%	100%	19%	0%	0%	0%	89%	100%	86%
CWE-89 SQL Inj.	RQ2	65%	39%	0%	100%	43%	0%	86%	86%	72%
	RQ3	22%	100%	22%	84%	70%	49%	89%	100%	86%
CWE-90 LDAP Inj.	RQ2	0%	0%	0%	0%	0%	0%	86%	86%	72%
	RQ3	19%	100%	19%	0%	0%	0%	89%	100%	86%
CWE-113 HTTP R.S.	RQ2	0%	0%	0%	4%	100%	4%	74%	100%	74%
	RQ3	19%	100%	19%	47%	100%	47%	89%	100%	86%
CWE-643 XPath Inj.	RQ2	0%	0%	0%	0%	0%	0%	86%	86%	72%
	RQ3	0%	0%	0%	0%	0%	0%	89%	100%	86%
Broken Authentication										
		Rec.	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
CWE-259 Hard-Coded	RQ2	18%	87%	16%	14%	100%	14%	43%	100%	43%
	RQ3	18%	87%	16%	14%	100%	14%	43%	100%	43%
Broken Access Control - Path Traversal										
		Rec.	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
CWE-23 Rel. Path T.	RQ2	11%	100%	11%	4%	100%	4%	86%	86%	72%
	RQ3	19%	100%	19%	47%	100%	47%	100%	88%	86%
CWE-36 Abs. Path T.	RQ2	11%	100%	11%	4%	100%	4%	86%	86%	72%
	RQ3	19%	100%	19%	40%	100%	40%	100%	88%	86%
Cross-Site Scripting										
		Rec.	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**	**Rec.**	**Prec.**	**Disc.**
CWE-80 Basic XSS	RQ2	11%	100%	11%	3%	100%	3%	100%	88%	86%
	RQ3	19%	100%	19%	46%	100%	46%	89%	100%	86%
CWE-81 XSS Error	RQ2	11%	100%	11%	6%	100%	6%	100%	88%	86%
	RQ3	19%	100%	19%	46%	100%	46%	89%	100%	86%
CWE-83 XSS Attrib.	RQ2	11%	100%	11%	6%	100%	6%	100%	88%	86%
	RQ3	19%	100%	19%	46%	100%	46%	89%	100%	86%

7. Discussion

The discussion chapter will describe the contribution of the findings from chapter 6 and explain new insights that emerged as a result of our research. In section 7.1, we discuss the benefits of taint analysis, while section 7.2 explores the benefits of a generalizable implementation. One of the most recurring limitations we found was missing sources or sinks, which is discussed in section 7.3. The state and limitations of the control- and data-flow analysis present in the SATs are explained in section 7.4. Section 7.5 talks about the benefits of a prioritized output, and the different ways we have seen this being implemented in the three SATs. One of the problems we discovered is misclassification of vulnerabilities, which we discuss further in section 7.6. We compare our research with related work in section 7.7, and the limitations of the Juliet Test Suite are discussed in section 7.8. The last section of this chapter presents possible threats to validity, both internal and external, in section 7.9.

7.1. Use of Taint Analysis

A common technique used by static analysis tools is taint analysis, tracking user input from a vulnerable source to an exploitable sink. Find Security Bugs uses it for many of its detectors, while ESVD is fundamentally built around it. SpotBugs also has some detectors utilizing taint analysis, but not to the extent as the other two.

The big advantage of taint analysis is that it produces fewer false positives. By utilizing data-flow analysis to keep track of the movement of inputs, taint analysis will not report on cases where input is not used by an exploitable sink. This way, false positives are avoided. A low false positive rate is important for developers as described by Christakis and Bird [2016], and taint analysis is a good technique to achieve this.

In section 6.3.2 we proved that Find Security Bugs' injection detectors are able to have a perfect recall or precision depending on which confidence ranking is used. This is achieved while still performing generally well with a high discrimination rate. By looking at the results in Table 6.18, 6.19, 6.20, and 6.21, the strengths of taint analysis are clear. Find Security Bugs is able to achieve 100% recall while still having a precision of 88%. Similarly, it can also achieve a perfect precision with a 89% recall. Identical results are also achieved by the detectors for path traversal and cross-site scripting.

ESVD uses taint analysis for all of its detectors. In section 6.3.3 we made changes to the detectors' implementation in order to detect all vulnerabilities and all source variants. However, as the results in Table 6.25, 6.26, and 6.27 show, ESVD is not able to replicate the results of Find Security Bugs despite both utilizing taint analysis. This is due to a lacking implementation for control- and data-flow analysis.

SpotBugs uses taint analysis in its detectors for SQL injection, HTTP response splitting, path traversal, and XSS. Similarly to ESVD, this implementation cannot compare to that of Find Security Bugs either. The SQL injection detector does not rely solely on taint analysis, but rather on detecting string concatenation. Taint analysis for the HTTP response splitting, path traversal, and XSS detectors does not properly track the data. All of this leads to poor results for SpotBugs as is evident in Table 6.15, 6.16, and 6.17. The implementation for taint analysis in SpotBugs is also detector-specific, which both Find Security Bugs and ESVD have showed is not needed. Taint analysis is a generalizable technique that can be utilized by detectors for many kinds of vulnerabilities. This is a strength that will be further discussed in section 7.2.

7.2. Generalizable Implementation

Section 6.3 showed us how a generalizable implementation can make it easier to change an SAT. As mentioned in section 6.1.1, SpotBugs provides a framework for static analysis on bytecode. This framework is used by all of its detectors, and makes it easy to develop new detectors that expand SpotBugs' capabilities. By using an implementation for static analysis that is generalizable, SpotBugs encourages people to contribute to the project through accessibility. There are several examples of contributions such as the ones by Shen, Zhang, et al. [2008], Shen, Fang, and Zhao [2011], Ware and Fox [2008], Al-Ameen, Hasan, and Hamid [2011], and Vestola [2012].

ESVD also has a generalizable implementation, but has taken it one step further. All of the detectors use the same detection algorithm. The algorithm uses taint analysis and the only difference between detectors is which sinks are used. This means that every detector reaps the benefits of improvements to the algorithm. As shown in our proof in section 6.3.3, it was easy to extend ESVD to detect new sources. Every source was only added once, but used by all of the detectors. This is because all of the detectors share the resource lists for sources as well as sanitization methods.

The taint analysis algorithm of Find Security Bugs is also shared by several detectors and is similar in design to that of ESVD. However, Find Security Bugs also have detector-specific implementations. It has a different take on shared implementation than ESVD. Instead of only having a single algorithm that is shared, Find Security Bugs has different algorithms for taint analysis, control-flow analysis, and data-flow analysis that a detector can opt in to use. Find Security Bugs also has other shared resources such as the password names in Figure 6.12.

7.3. Incomplete Collections of Sources and Sinks

Taint analysis, as discussed in section 7.1, uses lists of sources and sinks in its detection algorithm. Therefore, taint analysis is highly dependent on these resource lists to detect vulnerabilities. If these lists are missing a source or a sink, a vulnerability will consequently be missed as well.

One of the recurring modifications we carried out when answering RQ3 in section 6.3 was adding missing sources and sinks. While all of the SATs had extensive lists, all of them missed some sources and sinks as well. As we believe taint analysis is a powerful tool to detect security vulnerabilities, we propose a mutual effort between developers to construct complete lists of sources, sinks, and sanitization methods. We believe such an effort would be beneficial for the SAT community as a whole as well as the software security community.

When composing such a list, one will have to make decisions on what should be considered a safe or vulnerable source. As mentioned in section 6.2.5, the developers of Find Security Bugs consider the sources `System.getenv()` and `System.getProperty()` to be safe, while we show in 6.3.2 that they should be considered vulnerable as they are possible to exploit. Similarly, we encountered the dilemma of whether methods and classes such as `BufferedReader.readLine()` and `InputStreamReader` should be considered vulnerable sources. On the one hand, they are generally used to handle input. On the other hand, there is nothing inherently vulnerable about them. `BufferedReader.readLine()` is not technically a source as it is not a method that retrieves data from outside the application, but rather a method that input usually passes through. When adding them as sources, they contribute to a higher recall since new source variants are detected, and as long as the SATs use a strong taint analysis algorithm, they will avoid extra false positives as they will only be reported when the data also reaches a sink.

The resource lists for ESVD are composed of method signatures. However, ESVD can use an object instantiation as a source by using the signature of the class constructor. This became apparent when evaluating the source variants Console ReadLine and File as these use an object instantiation as a source. Using a class constructor as a source produce the same results as a method with no more false positives or fewer true positives. The taint analysis framework in Find Security Bugs is similarly able to have class constructors as sources.

7.4. Control- and Data-flow Analysis Limitations

A powerful detection algorithm will increase recall and precision. Few real-life vulnerabilities have a source and sink right next to each other. Related sources and sinks in natural code can be separated by anything from a few lines of code to being spread over different files. Control- and data-flow analysis are vital techniques for detecting and tracking what flows from source to sink. The problem with CFA and DFA is the complexity of implementing them. Many of the false negatives we see in SpotBugs, Find Security Bugs, and ESVD, are due to a badly implemented CFA or DFA.

Missing detections for the control- and data-flow variants in the Juliet Test Suite are the result of badly implemented analysis techniques. In some cases, the analysis algorithms are not even capable of processing the simplest control-flow variants, which is the case for the hard-coded password detector in SpotBugs. ESVD also struggles with control-flow, in particular `if`-statements and `switch`-statements. When considering how vital and common `if`-statements are to the Java programming language, this is

problematic.

The CFA and DFA algorithms implemented in the taint analysis in Find Security Bugs are working well, but the hard-coded password detector cannot use it. While the data sources in injection vulnerabilities stem from the return values of methods, the data source in hard-coded password vulnerabilities stem from string literals directly coded into the source code. The taint analysis framework in Find Security Bugs is built to only allow methods as sources and sinks, meaning that the hard-coded password detector cannot use the underlying taint analysis framework that is available. This results in the hard-coded password detector having to do most of this analysis using differently implemented CFA and DFA techniques. ESVD has solved this problem by modifying the hard-coded password detector to detect literals instead of sources while still utilizing the rest of the underlying taint analysis algorithm. This shows that even SATs with great analysis capabilities, such as Find Security Bugs, can struggle with data-flow when these algorithms cannot be utilized by all detectors.

Another limitation with the detection algorithms is the ability to track data across multiple Java classes. Context-sensitive data-flow analysis is not something we find in SpotBugs and ESVD. The Find Security Bugs detector for identifying hard-coded passwords perform some actions to compensate for this lack of interprocedural analysis, such as looking at the use of hard-coded class fields, but there is no analysis related to tracking changes to this class field before being used in a vulnerable sink. ESVD is able to track data through multiple methods within the same class, but does not track data across multiple classes, i.e., context-sensitive data-flow analysis. On the other hand, the injection detectors in Find Security Bugs are able to perform context-sensitive DFA to some extent. The most difficult data-flow case in the Juliet Test Suite uses code inheritance spread throughout multiple classes. The other difficult data-flow cases in the Juliet Test Suite are similarly complicated. Not even the injection detectors in Find Security Bugs are able to completely follow the interprocedural data-flow between these classes.

7.5. Prioritized Output

Prioritizing detections is an important tool to control the trade-off between soundness and completeness. According to Emanuelsson and Nilsson [2008], no static analysis tool are both sound and complete. Christakis and Bird [2016] discovered through surveys that most developers want a false positive rate below 15%. In many cases, it is hard to reduce the false positive rate while maintaining a high recall, which is why prioritization can make an important difference. Uncertain detections can be given a lower priority or confidence, while certain detections are presented with a higher confidence. This approach allows the user to decide between recall and precision by adjusting the confidence threshold.

From our analysis of SpotBugs, Find Security Bugs, and ESVD, only SpotBugs and Find Security Bugs use a confidence ranking as described above. ESVD has a hard-coded priority ranking based on which vulnerability the SAT regards as the most severe. A

cross-site scripting vulnerability will have the highest priority no matter how confident the detection is. In fact, no attempt is made at determining the confidence based on the available information during the static analysis. On the other had, SpotBugs and Find Security Bugs calculates a confidence based on how certain the SAT is regarding the detection. There might be many suspicious circumstances that indicates a vulnerability even though the origin of the suspicious data is unknown. SpotBugs and Find Security Bugs will in most cases report these detection with a lower priority. We have seen that a change to the confidence calculation can result in thousands of false positives disappearing, as for SpotBugs in section 6.3.1.

The problem with the confidence ranking in SpotBugs and Find Security Bugs is the inconsistencies when deciding which ranking to give. There are no developer guidelines for how certain a detection must be to gain a high confidence. The same goes for the other confidence rankings. This is especially apparent in the injection detectors in Find Security Bugs. These injection detectors are almost completely similar in implementation, except for sinks and confidence ranking. If the user adjusts the minimum visible confidence ranking so that only high confidence rankings are shown, no detections will be reported for HTTP response splitting. The HTTP response splitting detector reports confident detections with a medium confidence ranking, while the other four injection detectors in Find Security Bugs report confident detections with a high confidence ranking. Similarly to this, the other four detectors report uncertain detections with a medium confidence, while the HTTP response splitting detector reports them with a low confidence. This inconsistent behavior is everywhere in SpotBugs and Find Security Bugs, and strongly reduces the users' ability to adjust for recall or precision.

Shen, Fang, and Zhao [2011] implemented a new ranking system for FindBugs. This ranking system is initially based on the likelihood of the bug or vulnerability, but later changes based on user input. We believe that such an adaptive ranking system would be effective, especially when the detectors in SpotBugs and Find Security Bugs produce so diverse confidence rankings. An extension to this idea might involve the user deciding which detections are true and false positives, resulting in the confidence ranking threshold automatically adjusting to hide false positives. Although developer guidelines for deciding confidence will have a larger impact on those who do not want to train the adaptive confidence ranking, there is always the possibility of joining developer guidelines with an adaptive confidence calculation.

7.6. Vulnerability Misclassification

Misclassification of vulnerabilities can lead to confusion and problems when identifying possible countermeasures against a specific vulnerability. All three of the static analysis tools we have looked at incorrectly classifies at least one vulnerability category each. A serious security risk might be overlooked because it is believed to be a trivial vulnerability. In a worst case scenario, this might lead to the vulnerability not being fixed.

ESVD is able to detect 18% of the hard-coded passwords in the Juliet Test Suite. Early detection of hard-coded passwords can save a lot of time and resources later down

the road, as both MITRE [2019a] and OWASP [2016] claim that it is almost certain a malicious user will be able to access an account where the password is hard-coded into the source code. It is therefore unfortunate that the hard-coded passwords detected by ESVD is classified as a security misconfiguration. Hard-coding passwords into source code is not a misconfiguration, but rather an authentication problem. It is also confusing to security experts familiar with the OWASP Top 10 list, as one of the ten categories is named Security Misconfiguration, but shares few similarities with the intended meaning of ESVD's use of the phrase.

All of the three SATs have problems with the classification of path traversal vulnerabilities. Path traversal is further divided into relative and absolute path traversal, and it is differentiating between these that all of the detectors struggle with. SpotBugs has a tendency to classify all path traversals as relative path traversal, due to a peculiar algorithmic choice. We showed that it is possible to improve this part of the SpotBugs algorithm in section 6.3.1. Find Security Bugs and ESVD have no ability to separate the different path traversal vulnerabilities into their correct subcategories, as the taint analysis framework is unable to separate relative and absolute path traversal when both of the path traversal categories utilize the same sources and sinks. As discussed in section 2.2.3, the two different categories of path traversal vulnerabilities might have somewhat different countermeasures, resulting in it being difficult to properly sanitize the user data. In general, it could also be possible to apply a countermeasure that only works for one of the path traversal vulnerabilities, but still tricks the static analysis tool into thinking the data has been sanitized for the other category of path traversal vulnerabilities. For that reason, misclassifications of vulnerabilities can be dangerous.

7.7. Comparison to Related Work

In chapter 3, we presented the related work of our master thesis. Many have evaluated static analysis tools before, such as Charest, Rodgers, and Wu [2016] who evaluated four static analysis tools on test cases from the Juliet Test Suite. Oyetoyan et al. [2018] also evaluated several static analysis tools on the Juliet Test Suite, including SATs such as FindBugs, Find Security Bugs, and LAPSE+. Both Charest, Rodgers, and Wu [2016] and Oyetoyan et al. [2018] have similarities to our research, but also differences. We have focused on vulnerabilities that the OWASP Top 10 list considers the most critical, while they have respectively only selected a few vulnerabilities or evaluated on the whole test suite. What differentiates our research the most is our implementation analysis. We found no research that has gone beyond just presenting the numbers by analyzing the source code like we have.

We have also made modifications to the source code of the SATs in order to prove our hypotheses with proof-of-concept improvements. These changes made to the implementation of the SATs lead to improved performance results. Al-Ameen, Hasan, and Hamid [2011] as well as Ware and Fox [2008] both added new detectors to FindBugs to enhance the SAT's coverage. FindBugs is the only SAT we could find that have received improvements in related work. Shen, Zhang, et al. [2008] and Vestola [2012]

also added new detectors to FindBugs as part of their research. However, none have modified existing detectors to improve the performance of multiple SATs like we did in section 6.3.

7.8. Limitations of the Juliet Test Suite

The limitations of the Juliet Test Suite might have affected the results of this master thesis. There are three main limitations of the Juliet Test Suite: missing vulnerability categories, missing sources and sinks within a vulnerability category, and missing variations of data being passed to a sink. All three limitations will be discussed below.

Although the Juliet Test Suite has been a valuable resource when attempting to understand why an SAT performs poorly, this master thesis has mostly focused on the implementation and limitations of the SATs. The Juliet Test Suite is not necessary to understand the implementation and limitations, but rather a tool to help us understand where to start looking and a tool that is helpful for explaining to the reader how and where the SATs fail and prosper. That being said, it is difficult for us to guarantee that every nuance of the SATs implementation have been understood without also testing on other source code, be it natural code or another test framework. It is also difficult for us to guarantee that every limitation has been identified. As we have seen for ESVD, it has a tendency to crash and stop working on the Juliet Test Suite, which is a problem we have not seen for SpotBugs or Find Security Bugs. This does not mean that SpotBugs and Find Security Bugs are stable though, only that the two are stable when working on the Juliet Test Suite. Similarly, Find Security Bugs performs very well on the test cases in the Juliet Test Suite, but there is no guarantee that it will perform well on any other vulnerable source code.

The Juliet Test Suite does not have test cases for all vulnerabilities, and does not have test cases for every variation within the vulnerability categories. As seen in the pre-study, some of the vulnerability categories from OWASP Top 10 are not covered by the Juliet Test Suite, making it impossible for us to test any potential detector on these vulnerabilities. It is also possible that the sources and sinks in the Juliet Test Suite unfairly favor one of the SATs. Although the Juliet Test Suite contains many sources and sinks, there is always room for more. As seen in the SQL injection test cases in the Juliet Test Suite, all the test cases include a string concatenation when the data is sent to the sink. Since the SQL injection detector in SpotBugs always reports a vulnerability when it sees a string concatenation reaching a vulnerable sink, this structure of the SQL injection test cases favor SpotBugs. Test cases with other variations of data being passed to sink than the ones used in the Juliet Test Suite could have quickly demonstrated this SpotBugs limitation, and possibly many more.

7.9. Threats to Validity

Threats to internal validity include selection bias and experimenter bias. The selection of static analysis tools is based on the available tools from our pre-study, in addition to

the capabilities and techniques found in these SATs. It is possible to rely on erroneous information in a selection process, which is why we described our selection criteria in detail in section 5.4, and based our knowledge of the tools on information provided by the SAT authors themselves. In addition to the SATs, we also selected a list of vulnerabilities to analyze. Ranking vulnerabilities ourselves based on impact and importance is error prone, which is why we decided to rely on a trustworthy and well-established list of vulnerabilities, namely the OWASP Top 10.

There is also a possibility of experimenter bias. In our case, the experimenter bias would relate to non-consciously changing our approach to testing and analyzing the SATs. ESVD is very unstable, and it is vital to verify that none of the results are missed due to a crash. Find Security Bugs produce inconsistent results for one of the flow variants in the Juliet Test Suite. To reduce the experimenter bias threat, it is important to create a list of criteria and steps describing the approach to be taken. Such a description can be seen in section 5.4, where our approach to testing and analyzing the SATs are described in detail.

Threats to external validity include the generalizability and replicability of the results. The best performing static analysis tools from our pre-study are selected, where our pre-study contains the five most popular free and open-source SATs related to detecting security vulnerabilities. Only three SATs are included in this master thesis, which at first might question the generalizability of our results. By looking at the limitations presented in section 6.3 and 6.4, it is apparant that the SATs share a lot of similar limitations. There are a limited number of SATs that fit the criteria set for this master thesis, making the three SATs representative. However, it is not possible to generalize the results to commercial static analysis tools as these are in a completely different category regarding development and research funding.

Replicating qualitatively data analysis can in some cases be difficult. We have to the best of our abilities described and planned the approach, see section 5.4. The results are also described in detail, with quantitative proofs provided on our GitHub page. The necessary source code for parsers and SAT modifications are available for free to encourage replicability.

Using the Juliet Test Suite is, to an extent, both an internal and external threat. Regarding the internal validity, it might be difficult to fully explore the causality when implementation defects are compared to the Juliet Test Suite results. Regarding the external validity, an SAT might be designed to detect the Juliet Test Suite and nothing else, decreasing the generalizability of the research. Although we have not seen any attempts at customizing the SATs to only detect the test cases in the Juliet Test Suite, there is still a possibility that other test suites might lead to different results. Where necessary, we have temporarily modified test cases in the Juliet Test Suite to explore if specific code structures in the test cases are the reason for missing results.

8. Conclusion and Future Work

We have analyzed the implementation and identified limitations for three open-source static analysis tools aimed at detecting security vulnerabilities, namely SpotBugs, Find Security Bugs, and ESVD. A common pattern of strengths and limitations emerged as a result of this analysis. A generalizable implementation and the use of taint analysis are important techniques for increased detection rates. Incomplete collections of sources and sinks is a limitation with a large impact on the detection rates of the static analysis tools; the same applies to weak control- and data-flow analysis. The misclassification of vulnerabilities and an inconsistent or missing confidence ranking also negatively affect the SATs.

A static analysis tool that claims to cover a specific vulnerability has an inherited level of trust to fulfill. Bringing the actual performance and limitations of these tools into the spotlight helps to prove or disprove any question about their applicability. This master thesis, in conjunction with the paper published to EASE 2019, serves to bridge the gap between how well the SATs claim to perform, and how well they actually perform. To produce the performance results presented in this master thesis, at least 20,000 vulnerability reports were analyzed, based on 7,215 distinct test cases spanning 11 vulnerabilities.

We have shown that some of the techniques used by the analyzed detectors seem to be highly beneficial to the coverage and correctness of detections. The detectors that use taint analysis produce fewer false positives than those who do not. This reduction of false positives mainly results from not reporting cases where vulnerable input does not reach an exploitable sink. We also found the detectors based on a generalizable and shared implementation performed much better than the detectors that tried to create their own algorithms of data tracking and vulnerability detection. When a generalizable and shared detection algorithm is used, we found it easier to extend and correct the implementation. Improving or otherwise modifying an individual detector will not lead to improvements for other detectors, but that is not the case for improvements made to a shared algorithm.

We have also identified common limitations that are damaging to the detectors' results. Missing sources and sinks account for a large portion of the false negatives produced by the SATs. We propose a collective effort to compose lists of potentially vulnerable sources and sinks, as we have often experienced that sources or sinks missing from one SAT is present in another. A collective list would benefit all participating static analysis tools. We also identified that many of the false positives and negatives are the result of limited control- and data-flow analysis capabilities. It is rarely the case in natural code that a vulnerable source and sink come right after each other. It is much more common that vulnerable sources and sinks are separated by methods or classes, requiring

a well-performing context-sensitive data-flow analysis. We believe the detectors that perform the best on more complex test cases from the Juliet Test Suite are likely to also perform well on natural code. In addition to badly performing DFA and missing sources and sinks, we saw inconsistent confidence ranking and vulnerability misclassifications. The inconsistent confidence ranking make it difficult to balance the trade-off between soundness and completeness, while the misclassification of vulnerabilities can lead to the vulnerability being ignored or create confusion about which countermeasure to apply.

There exists no similar research where the implementation and limitations of open-source static analysis tools for detecting security vulnerabilities have been analyzed. This master thesis serves as an important first step into analyzing how these static analysis tools are able to perform the way they do. Although it is possible to feel a sense of frustration when looking at the identified limitations, we are left with a bright outlook on the future of these SATs. It is our opinion that the development effort invested, and the performance of these SATs are impressive when considering the development is fueled by volunteers working without pay. Although much work is left before any of them will be completely sound and correct, all three tools have made great strides in the field of static analysis.

It is possible to extend our implementation and limitation analysis to other static analysis tools, and to cover other security vulnerabilities or software bugs. Especially the relationship between open-source and commercial SATs is interesting, although it might be challenging to receive permission to publish detailed implementation descriptions of commercial tools.

One of the criteria in our research implementation limits the selected vulnerabilities to those where we could compare the SATs' implementations and limitations. If this criterion is removed, there are additional detectors from our pre-study which can be analyzed. The detectors do not necessarily have to be geared at security vulnerabilities from the OWASP Top 10 either, but can look at different security vulnerabilities or software bugs.

Another interesting variation of our research is to change or improve the test suite. Although the Juliet Test Suite contains many different source and flow variants, it is not exhaustive. As we have seen for the hard-coded password detectors in Find Security Bugs, many of the detectors are aimed at vulnerabilities not present in the Juliet Test Suite. It is possible to use another test suite, to combine multiple of them, or to extend the Juliet Test Suite with new test cases. It is even possible to test on natural code. All of these possibilities can help identifying different strengths and limitations in the SATs.

This master thesis has analyzed the implementation of open-source static analysis tools for detecting security vulnerabilities, something no one has previously carried out. There are many potential paths for future work, and this master thesis presents important building blocks that future work can build upon. The new information contributed by our research comes in the form of thorough explanations of the SATs' implementation in section 6.1, granular and detailed performance results in section 6.2, identifying limitations and applying code modifications to improve the performance results in section 6.3,

and the improvement proposals presented in chapter 7. The proposed improvements are not only relevant to SpotBugs, Find Security Bugs, and ESVD, but also to other existing and future SATs. It is necessary to understand the inner workings of the static analysis tools that help achieve secure software, and we believe our master thesis has completed this important first step of the process.

Bibliography

Aho, Alfred V. et al. (2007). *Compilers: Principles, Techniques, & Tools*. 2nd ed. Boston: Pearson/Addison Wesley.

Al Mamun, Abdullah et al. (2010). "Comparing Four Static Analysis Tools for Java Concurrency Bugs". In: *Third Swedish Workshop on Multi-Core Computing (MCC-10)*.

AlBreiki, Hamda Hasan and Qusay H. Mahmoud (2014). "Evaluation of static analysis tools for software security". In: *2014 10th International Conference on Innovations in Information Technology, IIT 2014*. Al Ain, United Arab Emirates: IEEE, pp. 93–98. ISBN: 9781479972128. DOI: 10.1109/INNOVATIONS.2014.6987569.

Allen, Frances E. (1970). "Control Flow Analysis". In: *Proceedings of a Symposium on Compiler Optimization*, pp. 1–19. DOI: 10.1145/800028.808479.

Al-Ameen, Mahdi Nasrullah, Monjurul Hasan, and Asheq Hamid (2011). "Making Findbugs More Powerful". In: *ICSESS 2011 - Proceedings: 2011 IEEE 2nd International Conference on Software Engineering and Service Science*. IEEE, pp. 705–708. ISBN: 978-1-4244-9698-3. DOI: 10.1109/ICSESS.2011.5982427. URL: https://doi.org/10.1109/ICSESS.2011.5982427.

Apache (2018). *Commons Lang*. URL: https://commons.apache.org/proper/commons-lang/ (visited on Mar. 25, 2019).

— (2019). *Apache Commons BCEL*. URL: https://commons.apache.org/proper/commons-bcel/ (visited on Feb. 11, 2019).

Arteau, Philippe (2019). *Random chance of detection for some files in Juliet 1.3 CWE89 SQL Injection*. URL: https://github.com/find-sec-bugs/find-sec-bugs/issues/456#issuecomment-476248348 (visited on May 29, 2019).

Baca, Dejan, Bengt Carlsson, and Lars Lundberg (2008). "Evaluating the Cost Reduction of Static Code Analysis for Software Security". In: *Proceedings of the third ACM SIGPLAN workshop on programming languages and analysis for security*. PLAS '08. New York, NY, USA: ACM, pp. 79–88. ISBN: 9781595939364. DOI: 10.1145/1375696.1375707. URL: http://doi.acm.org/10.1145/1375696.1375707.

Baird, Steve et al. (2011). "CodePeer – Beyond Bug-finding with Static Analysis". In: *Static Analysis of Software : The Abstract Interpretation*. 1st. Hoboken, NJ, USA: Wiley-ISTE. Chap. 5, pp. 177–206. ISBN: 978-1848213203.

Boulanger, Jean-Louis (2011). *Static Analysis of Software : The Abstract Interpretation*. 1st. Hoboken, NJ, USA: Wiley-ISTE, p. 331. ISBN: 978-1848213203.

Charest, Thomas, Nick Rodgers, and Yan Wu (2016). "Comparison of Static Analysis Tools for Java Using the Juliet Test Suite". In: *Proceedings of the 11th International Conference on Cyber Warfare and Security, ICCWS 2016*, pp. 431–438. ISBN: 9781910810828.

Christakis, Maria and Christian Bird (2016). "What developers want and need from program analysis: an empirical study". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*. New York, New York, USA: ACM Press, pp. 332–343. ISBN: 9781450338455. DOI: 10.1145/2970276.2970347. URL: http://dl.acm.org/citation.cfm?doid=2970276.2970347.

Computing Research and Education Association of Australasia (CORE) (2018). *CORE Conference Portal*. URL: http://portal.core.edu.au/conf-ranks/ (visited on May 14, 2019).

Cybersecurity Ventures (2017). *2017 Cybercrime Report*. Tech. rep.

Delaitre, Aurelien et al. (2018). *SATE V Report: Ten Years of Static Analysis Tool Expositions*. Tech. rep. National Institute of Standards and Technology. DOI: 10.6028/NIST.SP.500-326.

Emanuelsson, Pär and Ulf Nilsson (2008). "A Comparative Study of Industrial Static Analysis Tools". In: *Electronic Notes in Theoretical Computer Science* 217.C, pp. 5–21. ISSN: 15710661. DOI: 10.1016/j.entcs.2008.06.039.

Excellence in Research in Australia (ERA) (2010). *Conference Rankings*. URL: http://www.conferenceranks.com/data/era2010_conference_list.pdf (visited on May 14, 2019).

Facebook (2018a). *An Update on the Security Issue*. URL: https://newsroom.fb.com/news/2018/10/update-on-security-issue/ (visited on Feb. 15, 2019).

— (2018b). *Security Update: Exploited "View As"*. URL: https://newsroom.fb.com/news/2018/09/security-update/ (visited on Feb. 14, 2019).

Find Security Bugs (2012). *XSSRequestWrapper is a weak XSS protection*. URL: https://find-sec-bugs.github.io/bugs.htm%7B%5C#%7DXSS%7B%5C_%7DREQUEST%7B%5C_%7DWRAPPER (visited on June 13, 2019).

— (2017). *Injection detection*. URL: https://github.com/find-sec-bugs/find-sec-bugs/wiki/Injection-detection (visited on Apr. 29, 2019).

— (2018a). *Find Security Bugs - The SpotBugs plugin for security audits of Java web applications*. URL: https://find-sec-bugs.github.io/ (visited on Oct. 18, 2018).

— (2018b). *Writing a detector: Everything start with a test*. URL: https://github.com/find-sec-bugs/find-sec-bugs/wiki/Writing-a-detector#everything-start-with-a-test (visited on Feb. 17, 2019).

— (2019). *Find Security Bugs - The SpotBugs plugin for security audits of Java web applications*. URL: https://find-sec-bugs.github.io/ (visited on May 9, 2019).

Google (2018a). *Expediting changes to Google+*. URL: https://www.blog.google/technology/safety-security/expediting-changes-google-plus/ (visited on Feb. 14, 2019).

— (2018b). *Project Strobe: Protecting your data, improving our third-party APIs, and sunsetting consumer Google+*. URL: https://www.blog.google/technology/safety-security/project-strobe/ (visited on Feb. 14, 2019).

Greenberg, Andy (2019). *Hackers Are Passing Around a Megaleak of 2.2 Billion Records*. URL: https://www.wired.com/story/collection-leak-usernames-passwords-billions/ (visited on Feb. 15, 2019).

Hovemeyer, David and William Pugh (2004). "Finding Bugs is Easy". In: *SIGPLAN Not.* 39.12, pp. 92–106. ISSN: 0362-1340. DOI: 10.1145/1052883.1052895. URL: http://doi.acm.org/10.1145/1052883.1052895.

IAPP (2018). *Norwegian DPA Warns Municipality of Potential GDPR Violation*. URL: https://iapp.org/news/a/norwegian-dpa-warns-municipality-of-potential-gdpr-violation/ (visited on Feb. 15, 2019).

Johnson, Brittany et al. (2013). "Why don't software developers use static analysis tools to find bugs?" In: *Proceedings of the 2013 International Conference on Software Engineering*. San Francisco, CA, USA: IEEE Press, pp. 672–681. ISBN: 9781467330763. URL: https://dl.acm.org/citation.cfm?id=2486877.

Li, Jingyue, Sindre Beba, and Magnus Melseth Karlsen (2019). "Evaluation of Open-Source IDE Plugins for Detecting Security Vulnerabilities". In: *Proceedings of the Evaluation and Assessment on Software Engineering*. EASE '19. Copenhagen, Denmark: ACM, pp. 200–209. ISBN: 978-1-4503-7145-2. DOI: 10.1145/3319008.3319011. URL: http://doi.acm.org/10.1145/3319008.3319011.

Lindholm, Tim et al. (2018). *The Java Virtual Machine Specification - Java SE 11 Edition*. URL: https://docs.oracle.com/javase/specs/jvms/se11/html/index.html (visited on Feb. 11, 2019).

MITRE (2018a). *About CWE*. URL: https://cwe.mitre.org/about/index.html (visited on Feb. 14, 2019).

— (2018b). *CWE-113: Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting')*. URL: https://cwe.mitre.org/data/definitions/113.html (visited on May 30, 2019).

— (2018c). *CWE-23: Relative Path Traversal*. URL: https://cwe.mitre.org/data/definitions/23.html (visited on May 31, 2019).

— (2018d). *CWE-36: Absolute Path Traversal*. URL: https://cwe.mitre.org/data/definitions/36.html (visited on May 31, 2019).

— (2018e). *CWE-643: Improper Neutralization of Data within XPath Expressions ('XPath Injection')*. URL: https://cwe.mitre.org/data/definitions/643.html (visited on May 30, 2019).

— (2018f). *CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')*. URL: https://cwe.mitre.org/data/definitions/90.html (visited on May 30, 2019).

— (2019a). *CWE-259: Use of Hard-coded Password*. URL: https://cwe.mitre.org/data/definitions/259.html (visited on Feb. 14, 2019).

— (2019b). *CWE-74: Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')*. URL: https://cwe.mitre.org/data/definitions/74.html (visited on May 28, 2019).

— (2019c). *CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')*. URL: https://cwe.mitre.org/data/definitions/78.html (visited on May 28, 2019).

— (2019d). *CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')*. URL: https://cwe.mitre.org/data/definitions/89.html (visited on May 28, 2019).

Mueller, Robert S. and U.S. Department of Justice (2019). *Report On The Investigation Into Russian Interference In The 2016 Presidential Election*. Tech. rep. March. URL: https://www.justice.gov/storage/report.pdf.

NIST (2017). *Test Suites*. URL: https://samate.nist.gov/SRD/testsuite.php (visited on Oct. 31, 2018).

— (2018). *Juliet 1.3 Test Suite: Changes From 1.2*. URL: https://samate.nist.gov/SRD/resources/Juliet_1.3_Changes_From_1.2.pdf.

NSA (2012). *Juliet Test Suite v1.2 for Java User Guide*. URL: https://samate.nist.gov/SARD/resources/Juliet_Test_Suite_v1.2_for_Java_-_User_Guide.pdf.

Oates, B J (2005). *Researching Information Systems and Computing*. SAGE Publications. ISBN: 9781446203620.

OWASP (2015a). *File System*. URL: https://www.owasp.org/index.php/File_System (visited on May 31, 2019).

— (2015b). *Path Traversal*. URL: https://www.owasp.org/index.php/Path_Traversal (visited on May 31, 2019).

— (2016). *Use of hard-coded password*. URL: https://www.owasp.org/index.php/Use_of_hard-coded_password (visited on Feb. 14, 2019).

— (2017). *OWASP Top 10 - 2017 The Ten Most Critical Web Application Security Risks*. Tech. rep.

— (2018a). *Cross-site Scripting (XSS)*. URL: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS) (visited on Oct. 12, 2018).

— (2018b). *OWASP Enterprise Security API*. URL: https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API (visited on Mar. 25, 2019).

— (2019). *Cross-Site Scripting Prevention Cheat Sheet*. URL: https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.md (visited on May 31, 2019).

Oyetoyan, Tosin Daniel et al. (2018). "Myths and Facts About Static Application Security Testing Tools: An Action Research at Telenor Digital". In: *Agile Processes in Software Engineering and Extreme Programming*. Springer International Publishing, pp. 86–103. DOI: 10.1007/978-3-319-91602-6_6.

Ponemon Institute (2018). *2018 Cost of a Data Breach Study: Global Overview*. Tech. rep.

Rutar, Nick, Christian B. Almazan, and Jeffrey S. Foster (2004). "A Comparison of Bug Finding Tools for Java". In: *15th International Symposium on Software Reliability Engineering*, pp. 245–256. ISSN: 1071-9458. DOI: 10.1109/ISSRE.2004.1.

Sampaio, Luciano (2016). *TCM_Plugin*. URL: https://github.com/lsampaioweb/TCM_Plugin (visited on Nov. 30, 2018).

Sampaio, Luciano and Alessandro Garcia (2016). "Exploring context-sensitive data flow analysis for early vulnerability detection". In: *Journal of Systems and Software* 113, pp. 337–361. ISSN: 01641212. DOI: 10.1016/j.jss.2015.12.021. URL: 10.1016/j.jss.2015.12.021.

— (2019). *Early Security Vulnerability Detector - ESVD*. URL: https://marketplace.eclipse.org/content/early-security-vulnerability-detector-esvd/ (visited on Feb. 25, 2019).

Shen, Haihao, Jianhong Fang, and Jianjun Zhao (2011). "EFindBugs: Effective Error Ranking for FindBugs". In: *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pp. 299–308. DOI: 10.1109/ICST.2011.51.

Shen, Haihao, Sai Zhang, et al. (2008). "XFindBugs: EXtended FindBugs for AspectJ". In: *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE '08. New York, NY, USA: ACM, pp. 70–76. ISBN: 978-1-60558-382-2. DOI: 10.1145/1512475.1512490. URL: http://doi.acm.org/10.1145/1512475.1512490.

SpotBugs (2017). *Bug descriptions*. URL: https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html (visited on Oct. 19, 2018).

— (2018). *SpotBugs - Find bugs in Java Programs*. URL: https://spotbugs.github.io/ (visited on Oct. 18, 2018).

— (2019a). *Changelog*. URL: https://github.com/spotbugs/spotbugs/blob/3.1.11/CHANGELOG.md (visited on June 15, 2019).

— (2019b). *SpotBugs Eclipse plugin*. URL: https://marketplace.eclipse.org/content/spotbugs-eclipse-plugin (visited on Feb. 5, 2019).

Tellerreport (2018). *Knuddels: Chat platform must pay after hacker attack fine*. URL: http://www.tellerreport.com/tech/--knuddels--chat-platform-must-pay-after-hacker-attack-fine-.S1xs7l4Am.html (visited on Feb. 15, 2019).

Tripathi, Akash Kumar and Atul Gupta (2014). "A Controlled Experiment to Evaluate the Effectiveness and the Efficiency of Four Static Program Analysis Tools for Java Programs". In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE '14. New York, NY, USA: ACM, 23:1–23:4. ISBN: 978-1-4503-2476-2. DOI: 10.1145/2601248.2601288. URL: http://doi.acm.org/10.1145/2601248.2601288.

Tripp, Omer et al. (2009). "TAJ: effective taint analysis of web applications". In: *Pldi* 44.6, p. 87. ISSN: 0362-1340. DOI: 10.1145/1542476.1542486. URL: http://dl.acm.org/citation.cfm?id=1542476.1542486.

Van Roy, Peter and Seif Haridi (2004). *Concepts, Techniques, and Models of Computer Programming*. Cambridge: MIT Press. ISBN: 0-262-22069-5.

Vestola, Mikko (2012). "Evaluating and enhancing FindBugs to detect bugs from mature software: Case study in Valuatum". MA thesis. Aalto University. URL: http://urn.fi/URN:NBN:fi:aalto-201210043220.

Vetro, Antonio, Maurizio Morisio, and Marco Torchiano (2011). "An empirical validation of FindBugs issues related to defects". In: *15th Annual Conference on Evaluation & Assessment in Software Engineering (EASE 2011)*. Durham, UK: IET, pp. 144–153. ISBN: 978-1-84919-509-6. DOI: 10.1049/ic.2011.0018.

Ware, Michael S. and Christopher J. Fox (2008). "Securing Java Code: Heuristics and an Evaluation of Static Analysis Tools". In: *Proceedings of the 2008 Workshop on Static Analysis*. SAW '08. New York, NY, USA: ACM, pp. 12–21. ISBN: 978-1-59593-924-1. DOI: 10.1145/1394504.1394506. URL: http://doi.acm.org/10.1145/1394504.1394506.

Appendix A.

Scientific Paper of our Pre-Study

This appendix contains the published scientific paper of our pre-study. In its original form, our pre-study was our specialization project as part of the TDT4501 course at NTNU. It was concurrently with the work on this master thesis published on its own to the Evaluation and Assessment in Software Engineering (EASE) 2019 conference. EASE is ranked as an A-conference by both the Excellence in Research in Australia (ERA) [2010] and Computing Research and Education Association of Australasia (CORE) [2018].

During the work on this master thesis, we discovered we had made an error during the evaluation of ESVD for the categories A5 Broken Access Control and A7 Cross-Site Scripting. We reported in our paper that CWE-23 Relative Path Traversal and CWE-36 Absolute Path Traversal were not detected by ESVD. However, after a new evaluation, both of them gets reported. All of the CWE entries in A7 Cross-Site Scripting also produced higher numbers than first reported. Our theory is that this happened because of ESVD's unstable behavior with repeated crashes and freezes. In addition to this, we discovered that ESVD does in fact cover CWE-259 Hard-coded Password despite not claiming to do so. We also found that some of the results for Find Security Bugs were slightly incorrect due to a bug in the SAT itself. All results in our master thesis are adjusted accordingly, and the original results are kept in this scientific paper.

Evaluation of Open-Source IDE Plugins for Detecting Security Vulnerabilities

Jingyue Li*
Norwegian University of Science and Technology
Trondheim, Norway
jingyue.li@ntnu.no

Sindre Beba
Norwegian University of Science and Technology
Trondheim, Norway
sindrbeb@alumni.ntnu.no

Magnus Melseth Karlsen
Norwegian University of Science and Technology
Trondheim, Norway
magnumk@alumni.ntnu.no

ABSTRACT

Securing information systems has become a high priority as our reliance on them increases. Global multi-billion dollar companies have their critical information regularly exposed, costing them money and impairing their users' privacy. To defend against security breaches, IDE-integrated plugins to detect and remove security vulnerabilities in the first place are being used more frequently. More information about these plugins is needed in order to improve the state of the art within the field. Five open-source IDE plugins which can identify and report vulnerabilities are evaluated. We evaluate and compare how many categories of vulnerabilities the plugins can detect, how well the plugins detect the vulnerabilities, and how user-friendly the output of the plugin is to the developers. Our results show that certain vulnerabilities such as injection and broken access control are vastly covered by most plugins, while others have been completely ignored. A discrepancy between the claimed and actually confirmed coverage of the plugins is discovered, underlining the importance of this research. High false positive rate and obvious limitations in usability show that more work is needed before these plugins can be widely used and relied upon in a corporate setting.

CCS CONCEPTS

• **Security → Software security**;

KEYWORDS

Software security, static analysis tools, vulnerability detection, integrated development environment, plugin

ACM Reference Format:
Jingyue Li, Sindre Beba, and Magnus Melseth Karlsen. 2019. Evaluation of Open-Source IDE Plugins for Detecting Security Vulnerabilities. In *Evaluation and Assessment in Software Engineering (EASE '19), April 15–17, 2019, Copenhagen, Denmark.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3319008.3319011

*All authors contributed equally to this research.

EASE '19, April 15–17, 2019, Copenhagen, Denmark

ACM ISBN 978-1-4503-7145-2/19/04…$15.00
https://doi.org/10.1145/3319008.3319011

1 INTRODUCTION

As society gets more dependent on technology, the importance of securing information systems increases. A solution to reduce the number of security vulnerabilities is static analysis tools. With these tools included directly into the Integrated Development Environment (IDE) of the developer, they have never been easier to use. Unfortunately, not a lot of research has been made to compare and evaluate the existing IDE plugins, leaving the users to blindly accept the developer's claims of quality.

In this study, we aim to provide information about the state-of-the-art open-source IDE plugins. We report on their actual coverage of vulnerabilities, their performance, and their usability. This is achieved by evaluating them using a credible framework, i.e., Juliet Test Suite [12], for test cases and commonly used performance metrics, e.g., recall, precision, and discrimination rate. We select and evaluate five open-source plugins, namely ASIDE [14, 27, 28], ESVD [23, 24], LAPSE+ [2, 15], SpotBugs [8, 25], and Find Security Bugs (FindSecBugs) [6]. The plugins are evaluated on the most critical security vulnerabilities according to OWASP Top 10 [16], a reputable source for web application security. The evaluation looks for limitations of the existing plugins and is conducted with reproducibility in mind.

Our contribution comes in the form of two evaluations, i.e., a practical evaluation focused on coverage and performance as well as a theoretical evaluation of usability. Our results show that the evaluated plug-ins have clear limitations and a worrying discrepancy between what they claim and what they do. Coverage is focused around vulnerabilities connected to injection and access control, while other categories are left untouched. Several of the plugins also have a high false positive rate, which Christakis and Bird [4] identified as a leading cause of disuse. Many of the features, such as quick fixes, are not implemented adequately in the plugins. Insights we get from this study can help guide the open-source communities to improve their plugins further. For users of the plugins, our results can help them to understand better the strengths and weaknesses of the plugins they intend to use.

The rest of the paper is organized as follows. We introduce related studies in Section 2, and explain the design of the study in Section 3. Section 4 presents our evaluation results. We compare our results with related work in Section 5, and conclude our paper and discuss possible future work in Section 6.

2 RELATED WORK

Johnson et al. [9] looked at why static analysis tools, despite their benefits, are not widely used by developers. Through interviews with 20 software developers, they look at how current tools can

be improved. Results show that reasons for developers using static analysis tools are mainly because they automatically find bugs, they are already a part of the development environment, or for raising awareness of potential problems in a team setting. A reason for not wanting to use static analysis tools is poorly presented output. Another reason is large quantities of false positives, sometimes outweighing the true positives in volume.

Christakis and Bird [4] looked at what makes a static program analyzer attractive to developers through a broad survey of 375 Microsoft employees. They found that bad warning messages and slow speeds are reported pain points. It is interesting to note that while too many false positives is a largely reported pain point, too many false negatives is not. 90% of the participants are willing to accept a 5% false positive rate, while 47% of developers accept up to a 15% false positive rate. When forced to pick more bugs or fewer false positives, they typically choose fewer false positives. Developers also want the possibility of analyzing only part of the code, such as a file. Another feature that is often used and often requested is the possibility of suppressing warnings, preferably through code annotations.

Baset and Denning [1] collected information about available IDE plugins and compared them based on information provided by developers or manufacturers. They compared factors such as IDEs and languages supported, availability, scope of feedback, vulnerabilities covered and plugin uptake. The goal of their work was to synthesize the available information for future work within the field. In total, they gathered information about 17 different IDE plugins. Among them were both free and commercial tools as well as closed- and open-source. The vulnerability coverage comparison focused on nine input-validation related vulnerabilities. Of the nine vulnerabilities in question, only ESVD, FindBugs and LAPSE+ claimed to check for six or more. In addition, only ASIDE and Codepro AnalytiX provided *quick fixes* when reporting on vulnerabilities. Baset and Denning did not evaluate the plugins themselves, and all the information they collected came from other sources.

Oyetoyan et al. [19] looked into the capabilities of the freely available, open-source static analysis tools FindBugs, FindSecBugs, SonarQube, JLint, LAPSE+, and an undisclosed commercial tool. They tested the tools on all of the test cases in the Juliet Test Suite. Their results showed that FindSecBugs and LAPSE+ had the most true positives, and that FindSecBugs also had the highest recall and precision. FindSecBugs also had a good discrimination rate, while LAPSE+ was poor. The commercial tool ranked third of all the tools and had a poor precision. In addition to evaluating the efficiency, Oyetoyan et al. also interviewed six developers on their experience with using static analysis tools. They discovered developers had a generally positive attitude towards them, but that there were several barriers that stood in the way such as the need for multiple tools as well as poor performance.

There are multiple other related works. Charest et al. [3] compared the accuracy and precision of CodePro AnalytiX, JLint, FindBugs, and VisualCodeGrepper. The results were generally low, but Charest et al. argued that it was not too alarming as other studies had shown similar recall and precision for other tools, both open-source and commercial. Sadowski et al. [22] present a static analysis platform developed at Google and a philosophy on how such a platform should be created. Google's philosophy on program analysis is to have no false positives, allow the users to contribute with their own detections, reducing confusing tool output by accepting user feedback, and that the program should be analyzed while the user is changing or compiling the code.

3 RESEARCH DESIGN AND IMPLEMENTATION

Existing comparisons of vulnerability detection plugins lack vigorous testing on Java code for many of the free and open-source plugins. Some plugins claim to be developing new detection methods that would be superior to previous tools, but do not confirm this through testing afterwards. Not knowing what vulnerabilities each plugin covers, and its accuracy and performance when scanning for security vulnerabilities, can result in reduced usage or misuse of such tools. The focus on comparing existing free and open-source static analysis tools in regards to usability is non-existent. Considering the usability aspect of such tools plays a significant role in how many developers may choose to use these tools. The lacking amount of comparisons is problematic.

Our research aims to evaluate the vulnerability coverage, performance, and usability of current free and open-source IDE plugins utilizing static analysis on Java applications. For a large amount of free and open-source IDE plugins, such an evaluation does not exist today. We believe that an evaluation like this could bring useful information to developers without deep financial pockets, where buying expensive vulnerability scanners is too costly.

We formulate the following four criteria for selecting the plugins:

(1) The IDE plugins must detect security vulnerabilities, not just code bugs.
(2) The IDE plugins must be freely available and open-source.
(3) The IDE plugins must report detected vulnerabilities inside the IDE without the need for external software.
(4) The IDE plugins must detect vulnerabilities in Java code.

We want to evaluate plugins that are able to mitigate security vulnerabilities in code and can be used by both individuals, development teams, and large corporations. In order to lessen the work it is to adopt a static analysis tool, we want to contribute with information on how to improve an IDE plugin that requires minimal effort to install, learn, and use on a daily basis. This is the reasoning for the first three specifications listed above. To limit the study scope, we focus only on plugins analyzing Java code, as Java is the most used programming language according to TIOBE [26].

3.1 Research Questions

Based on the criteria presented, we formulate three research questions. For current open-source IDE plugins used to identify security vulnerabilities in Java using static code analysis:

RQ1 What is the coverage?
RQ2 How good is the performance?
RQ3 How good is the usability?

3.2 Plugins to be Tested and Test Cases

Based on information in [1], we decide upon the plugins in Table 1. All the plugins are available for the Eclipse IDE, and Eclipse is used for the evaluation. SpotBugs is not covered in [1], but its spiritual predecessor FindBugs is. We decide to focus on SpotBugs instead of FindBugs since SpotBugs is still actively worked on, while FindBugs has not been updated in years. FindSecBugs is not an IDE plugin in the same sense as the others. Instead, it is a plugin for SpotBugs (and FindBugs). That means it requires SpotBugs to run and its purpose is to expand the capabilities of SpotBugs further.

Plugins that are the result of academic work are especially interesting to us as they have more documentation and give more insights into their implementation. Both ASIDE and LAPSE+ are included due to this fact even though they are no longer being supported. Of the five plugins we have selected, only FindBugs (SpotBugs) frequently occurs in previous studies, such as [3], which compare static analysis tools. In other words, this evaluation will also provide new useful information about their coverage, performance, and usability.

Table 1: Information about the selected plugins.

IDE Plugin	Downloaded From	Version	Date
ASIDE	GitHub [28]	1.0.0	Feb 2013
ESVD	GitHub [23]	0.4.2	Jul 2016
LAPSE+	GitHub [2]	2.8.1	Jun 2013
SpotBugs	Eclipse Marketplace [25]	3.1.11	Jan 2019
FindSecBugs	Project Webpage [6]	1.8.0	Jun 2018

ASIDE is created with early detection in mind and with the goal to educate the user in secure programming. However, the static analysis tool does not offer any control- or data-flow analysis. Neither does LAPSE+, but it gives the user the opportunity to track variables manually through its provenance tracker. ESVD is inspired by ASIDE's focus on early detection and improves upon it by including data-flow analysis. It also claims to use inter-procedural analysis. SpotBugs and FindSecBugs analyze the Java bytecode instead of the raw source code like the others. Both of them utilize control-flow, data-flow, and inter-procedural analysis.

In order to test the IDE plugins in Table 1, we need test cases containing security vulnerabilities. Many comparisons of static analysis tools, such as the one by Rutar et al. [21], test the tools on the source code of actual software which can be referred to as *natural code*. As explained by NSA [13], this has both advantages and drawbacks. Two major issues are:

- Identifying false negatives. In natural code, it is often problematic to know how many vulnerabilities there are in the code. Without knowing this, it is impossible to calculate the number of false negatives.
- It is also problematic to know which types of security vulnerabilities are present. Thus, natural code cannot confidently prove which vulnerabilities an IDE plugin covers.

Delaitre et al. [5] also compared different types of test cases and which performance metrics they were applicable to test for. Table 2 shows the applicability of natural and artificial code as according to [5].

Table 2: Metric applicability for natural and artificial code according to Delaitre et al. [5].

Metric	Natural Code	Artificial Code
Coverage	Limited	Applicable
Recall	Not applicable	Applicable
Precision	Applicable	Applicable
Discrimination	Not applicable	Applicable

Given the purpose of our evaluation, it is apparent that using artificial code is beneficial to get the most accurate results. To choose the artificial code to compare the plugins, we consider four possible frameworks with vulnerabilities deliberately inserted: WebGoat [18], SecuriBench Micro [10], OWASP Benchmark [17], and Juliet Test Suite [12, 13]. We conclude that WebGoat is not suitable to test static analysis tools and that both SecuriBench Micro and OWASP Benchmark did not have test cases for enough vulnerabilities. Juliet Test Suite, on the other hand, fulfills both of these criteria.

Juliet Test Suite is a collection of intentionally vulnerable artificial code. Its purpose is to serve as a testing platform for static analysis tools. It consists of over 28,000 test cases which are categorized into 112 different CWE entries [12]. Each test case includes exactly one vulnerability as well as at least one non-flawed construct meant to represent a potential false positive. A single CWE entry can have thousands of test cases spanning over simple cases, control-flow cases and data-flow cases, where the later cases are more difficult to detect. All of this is intuitively incorporated in the naming of the test cases and its containing methods, which are further explained in its documentation [13]. It is still maintained and also used in previous research such as [3] and [19].

3.3 Measurement Metrics

We collect both quantitative and qualitative data to answer our research questions. For RQ1, we need to decide what *test case coverage* we want. Test case coverage is here defined as which security vulnerabilities we will test for. We decide to cover the vulnerabilities in OWASP Top 10 [16] because it is a reputable source for the most common and important web application vulnerabilities.

For RQ2, we use the performance metrics defined by Delaitre et al. [5]. The metrics are: 1) Recall, which is the percentage of vulnerabilities detected out of the total amount; 2) Precision, which is the percentage of alleged vulnerabilities the plugin reports that are indeed true vulnerabilities; 3) Discrimination Rate, which is the percentage of test cases the plugin discriminates. A plugin discriminates a vulnerability if it only reports a true positive on it, but no false positives.

$$\text{Discrimination Rate} = \frac{\text{Number of Discriminations}}{\text{Number of Test Cases}}$$

For RQ3, we decide to perform a qualitative evaluation. The usability metrics are taken from [4], [9], and [22]. In detail, we compare:

- **Tool Output**. Johnson et al. [9] and Sadowski et al. [22] found that poorly and confusing presented analysis output

was a largely reported problem when trying to figure out why software developers do not use static analysis tools. Developers want to know what the problem is, why it is a problem, and what should be done differently [9]. The tool output will be analyzed qualitatively. The satisfaction requirements are listed below, and must be present for a large portion of the vulnerabilities:

What is the problem? The output clearly states where the problem is in the form of line number and file name. The output also clearly states the vulnerability category of the problem.

Why is it a problem? The output clearly states why the reported vulnerability is a problem through text or an example. This must include the consequences of the vulnerability being in the code.

How to fix the problem? The output clearly states how to fix the problem through text or an example. Quick fixes also satisfy the requirement if all presented quick fixes are valid solutions to the problem, but not in the case where the tool does not try to make a distinction between relevant and irrelevant quick fixes.

- **False Positive Rate**. Christakis and Bird [4] found that over half the participants in their study do not accept a false positive rate above 15%. To present the false positive rate as clearly as possible, we will use two different methods to generate the percentage. The first method is calculating the false positive rate for each CWE, and then averaging the rate into what we call *averaged false positive rate*. The other method is adding all the true and false positives for each CWE into a total true positives and total false positives, and then calculating the false positive rate of the total, which we call *false positive rate of total result*.
- **Prioritized Output**. A high false positive rate might be counteracted by a prioritized output [7]. This metric will look at if the reported vulnerabilities are sorted by priority.
- **Quick Fixes**. Johnson et al. [9] and Sadowski et al. [22] discuss the need for quick fixes which can automatically fix the vulnerability. For each plugin, we investigate if quick fixes are presented to the user in a way that allows it to be automatically applied.
- **Early or Late Detection**. When looking at what the developers preferred, Johnson et al. [9] found that there was little agreement. Some developers preferred the tool to run in the background and immediately notify them of any bugs. Others preferred to integrate it with the compiler or sometimes later in the workflow. We look at whether the plugins utilize either early detection, late detection, or both.
- **Warning Suppression**. We check whether the plugin provides features to suppress the warnings. Both Johnson et al. [9] and Christakis and Bird [4] found that developers wanted the possibility of suppressing specific warnings, preferably through code annotation.
- **Environment Integration**. Developers are more likely to use the static analysis tool if it is already part of the development environment [9]. Sadowski et al. [22] found that the use of static analysis tools dropped when the developer was forced to run the analyzer as a stand-alone binary. This metric will be two binary results of whether the tool can integrate with the Eclipse IDE, and whether the plugin is available through the Eclipse Marketplace.
- **Immediate or Negotiated Interruptions**. Robertson et al. [20] describe the different styles of alerting mechanisms to investigate the impact of different interruption styles on the user. *Negotiated-style interruptions* are interruptions that inform the user of a pending alert without forcing them to acknowledge it at once. *Immediate-style interruptions* are alerts that immediately require the attention of users. Only negotiated-style interruptions were shown to have advantages [20].
- **Extendability**. The extendability metric will look at if the tools can be legally modified or extended. We will consider a tool to be legally modifiable if the source code is freely available, and under a software license where the code can freely and without cost be modified and redistributed. We will consider a tool easily extendable if it has a free API that allows any developer to modify and extend the original tool.
- **Granularity of Analysis**. When Christakis and Bird [4] asked to what level of granularity the developers would like to direct a program analyzer, file level (35%) or method level (46%) were chosen by the majority of participants. We will check if any of the tools support either file or method level granularity.

3.4 Research Implementation

Before conducting the evaluation, we need to assure ourselves that each IDE plugin is available and function properly. This proved not to be the case for ASIDE as the web page dedicated to the ASIDE project consists of mostly dead links to the plugin. This means we have to download the source code and build the plugin ourselves. We also have to change a couple of lines of code to correctly import external libraries as these are sets with an absolute path to the developer's computer. In addition, all plugins run in the newest version of Eclipse at the time, Eclipse Photon, except for LAPSE+ which we only manage to run in the older Eclipse Helios.

In order to automate the process of testing the IDE plugins, we create parsers that take the raw output data of the IDE plugins and transform it into the finished results of true positives, false positives, false negatives, and our performance metrics. This requires us to change the code that outputs the results of ASIDE, ESVD, and LAPSE+ as the information they provide is not adequate. We are particularly careful to only change the textual output from the plugins, without changing any logic that could alter the detection algorithms. The parsers are written in Python and with modularity in mind. The modified source code of the plugins and the source code of the parsers are available at https://github.com/Beba-and-Karlsen/. There are four different versions of the parsers, namely *aside.py*, *esvd.py*, *lapseplus.py* and *spotbugs.py*. As FindSecBugs is a plugin for

SpotBugs, they use the same version. All of these versions include the plugin-specific code for interpreting the bug reports. Then, they send the results to *plugincommon.py* that does the rest. In other words, they are similar in function, but with some implementation differences. They can all be explained as executing the following steps:

(1) **Read bug report** - The parser runs through the file containing the bug reports, extracting relevant information such as file name, CWE ID, vulnerability category, and in which test case it is detected.
(2) **Filter bug report** - The results are sent to *plugincommon.py* that checks whether each vulnerability is a true positive, false positive, or irrelevant. It then adds it to the respective list. Whether it is a true or false positive is based on the method it is detected in. All flawed methods in the Juliet Test Suite are named in a particular way. It is deemed relevant if the vulnerability category corresponds to the CWE entry of the test case. The output of the parsers is compared with the vulnerability categories provided by the plugin developers or documents.
(3) **Calculate test results** - With the filtering done, the parser then calculates the number of true positives, false positives, false negatives, recall, precision, and discrimination rate.
(4) **Print and log results** - In the end, the parser prints the results to screen and logs all results to a log file.

Juliet Test Suite v1.3 covers over 28,000 test cases spanning 112 different CWE entries. To run the plugins directly on the whole project would be very time-consuming and memory demanding. Because of this, we manually pick out the CWE entries we intend to test. Doing this manually is made easier by the fact that each test case in the Juliet Test Suite is categorized and sorted under its corresponding CWE entry. Selecting these CWE entries are based on the external references in OWASP Top 10 [16] and the CWE-1000 Research Concepts [11]. With the help of these resources, we are able to select the relevant CWE entries for each vulnerability category in OWASP Top 10.

However, not all CWE entries have a test case in the Juliet Test Suite. The 14 relevant CWE entries not included are listed in Table 3. In Table 5, all selected CWE entries that are included in the Juliet Test Suite are listed. In total, the plugins are tested on 8,675 test cases. As shown in Table 3, the Juliet Test Suite has no test cases for the CWE entries from categories A4 XML External Entities (XXE), A8 Insecure Deserialization, or A10 Insufficient Logging & Monitoring. In addition, no CWE entry at all matches A9 Using Components with Known Vulnerabilities. This is because this is a category that is more abstract, and is not a concrete weakness. This cannot be tested for by static analysis tools.

Table 3: The relevant CWE entries missing from the Juliet Test Suite.

OWASP Category	CWE Entries not in Juliet Test Suite	
	ID	Name
A1	564	Hibernate Injection
	917	Expression Language Injection
A2	384	Session Fixation
A3	220	Exposure of sens. info through data queries
	326	Weak Encryption
	359	Exposure of Private Information
A4	611	Improper Restriction of XXE
A5	284	Improper Access Control (Authorization)
	285	Improper Authorization
A6	2	Environmental Security Flaws
	16	Configuration
A8	502	Deserialization of Untrusted Data
A10	223	Omission of Security-relevant Information
	778	Insufficient Logging

4 RESEARCH RESULTS

4.1 RQ1: What is the coverage?

For RQ1, the coverage is defined by which and how many vulnerabilities the IDE plugin can detect. To answer what the coverage is of current plugins, we can look at both what they claim to detect and what our evaluation confirms they detect. We are looking at coverage for vulnerabilities in OWASP Top 10 included in the Juliet Test Suite. The results of our evaluation are shown in Table 4 and 5. The data in Table 5 show the number of true positives and false positives for each plugin divided into groups by CWEs. The hyphens (-) indicate that the CWE is not claimed to be covered by the plugin. The CWE is listed with its unique ID, name and its total amount of test cases. Based on the results of Table 5, we made a summary of the confirmed and claimed coverage for each plugin, which is shown in Table 4. The percentages correspond to the number of covered CWE entries covered by the plugins divided by the total number of all the vulnerabilities included in the Juliet Test Suite, which has 29 vulnerability categories.

Note that the results from FindSecBugs do not include the detections from SpotBugs, as we would like the coverage and performance of FindSecBugs to speak for itself. This allows us to evaluate the techniques of FindSecBugs independent from those of SpotBugs. In a typical use case, SpotBugs would by default be concurrently running when using the FindSecBugs extension.

Table 4: Confirmed and claimed coverage of the IDE plugins. Full coverage corresponds to covering all 29 vulnerability categories.

Tools	Confirmed Coverage		Claimed Coverage	
ASIDE	12	41%	12	41%
ESVD	5	17%	13	45%
LAPSE+	8	28%	11	38%
SpotBugs	8	28%	8	28%
FindSecBugs	18	62%	19	66%

The results in Table 4 and 5 show that the claimed coverage of the plugins is not great, with FindSecBugs being the only one with a claimed coverage higher than 50%. The confirmed coverage of our evaluation is even worse. Especially ESVD and LAPSE+ have several vulnerabilities they claim to cover, where our evaluation has shown that they in fact do not. It is worth noting that this could be due to the fact that the plugin does not detect the particular

Table 5: Detailed coverage data, showing the number of true and false positives. A hyphen (-) indicates that the plugin does not cover the CWE. The CWE is listed with its unique ID and its total number of test cases.

CWE			IDE-Integrated Static Analysis Tools									
ID	Name		ASIDE		ESVD		LAPSE+		SpotBugs		FindSecBugs	
A1 Injection		**Total**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
78	OS Command Injection	444	185	0	49	0	444	624	-	-	378	50
89	SQL Injection	2220	3[a]	3[a]	1440	2280	2220	3060	2220	3000	1900	300
90	LDAP Injection	444	185	0	0	0	0	0	-	-	379	50
113	HTTP Response Splitting	1332	555	795	0	0	0	0	57	0	989	0
134	Use of Externally-Controlled Format String	666	148	212	-	-	-	-	-	-	462	0
643	Xpath Injection	444	185	265	0	0	444	1248	-	-	379	49
A2 Broken Authentication		**Total**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
256	Unprotected Storage of Credentials	37	-	-	-	-	-	-	-	-	-	-
259	Use of Hard-coded Password	111	-	-	-	-	-	-	15	0	48	0
321	Use of Hard-coded Cryptographic Key	37	-	-	-	-	-	-	-	-	16	0
523	Unprotected Transport of Credentials	17	-	-	-	-	-	-	-	-	-	-
549	Missing Password Field Masking	17	-	-	-	-	-	-	-	-	-	-
A3 Sensitive Data Exposure		**Total**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
315	Cleartext Storage of Sensitive Information in a Cookie	37	-	-	-	-	-	-	-	-	0	0
319	Cleartext Transmission of Sensitive Information	370	-	-	-	-	-	-	-	-	259	369
325	Missing Required Cryptographic Step	34	-	-	-	-	-	-	-	-	-	-
327	Use of a Broken or Risky Cryptographic Algorithm	34	-	-	-	-	-	-	-	-	17	0
328	Reversible One-Way Hash	51	-	-	-	-	-	-	-	-	51	0
329	Not Using a Random IV with CBC Mode	17	-	-	-	-	-	-	-	-	17	0
614	Sensitive Cookie in HTTPS Session Without 'Secure' Attribute	17	-	-	-	-	-	-	-	-	16	0
759	Use of a One-Way Hash without a Salt	17	-	-	-	-	-	-	-	-	-	-
760	Use of a One-Way Hash with a Predictable Salt	17	-	-	-	-	-	-	-	-	-	-
A5 Broken Access Control		**Total**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
23	Relative Path Traversal	444	108	0	0	0	444	624	19	0	378	52
36	Absolute Path Traversal	444	108	0	0	0	444	624	16	0	378	49
566	Auth. Bypass Through User-Controlled SQL Primary Key	37	36	0	-	-	37	0	-	-	-	-
A6 Security Misconfiguration		**Total**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
395	NullPointerException Catch to Detect NULL Pointer Deference	17	-	-	0	0	-	-	-	-	-	-
396	Declaration of Catch for Generic Exception	34	-	-	0	0	-	-	-	-	-	-
397	Declaration of Throws for Generic Exception	4	-	-	0	0	-	-	-	-	-	-
A7 Cross-Site Scripting		**Total**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
80	Basic XSS	666	642	900	28	0	666	936	19	0	666	76
81	Improper Neutralization of Script in an Error Message	333	321	450	14	0	0	0	19	0	333	38
83	Improper Neutralization of Script in Attributes in a Web Page	333	108	0	14	0	333	468	19	0	333	38

[a] ASIDE generates an exception when running on these test cases.

case that Juliet Test Suite has implemented while detecting others. However, it is still a flaw in the plugin implementation. We find this discrepancy rather alarming. It gives a false sense of security to the user and discredits the integrity of the plugins. It is also worth noting that the distribution of coverage over the different categories is uneven. Injection vulnerabilities, broken access control, and cross-site scripting are heavily covered, while the others are less represented in the plugins' coverage.

4.2 RQ2: How good is the performance?

The calculations of recall, precision, and discrimination rates are based on the results of true positives and false positives shown in Table 5. The calculated performance metrics are presented in Table 6. There are two important notes about the evaluation that may have affected the results:

(1) Because of imprecise detection classification, ASIDE's true positives might be higher than what it actually deserves. It uses only two vulnerability categories which are very general, *input validation vulnerability* and *output encoding vulnerability*. This means we cannot implement proper relevant category checking for ASIDE. In other words, we cannot be confident whether reported vulnerabilities by ASIDE are relevant or not.

(2) LAPSE+ is supposed to be used with a substantial amount of manual effort to complete backward propagation on each result, which cannot be automated. It reports both sources and sinks and requires the user to check whether the data was sanitized between these points. Our results are based on only the automatic results without the manual effort after. This might be the cause of the high amount of false positives LAPSE+ reports.

Table 6: Detailed performance metrics data, showing recall, precision, and discrimination rate. CWE names are slightly shortened, see Table 5 for the full names. A hyphen (-) indicates that the plugin does not cover the CWE.

CWE		Tools														
ID	Name	ASIDE			ESVD			LAPSE+			SpotBugs			FindSecBugs		
A1 Injection		**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
78	OS Command Injection	42%	100%	42%	11%	100%	11%	100%	42%	0%	-	-	-	85%	88%	74%
89	SQL Injection	0%	50%	0%	65%	39%	0%	100%	42%	0%	100%	43%	0%	86%	86%	72%
90	LDAP Injection	42%	100%	42%	0%	N/A	N/A	0%	N/A	N/A	-	-	-	85%	88%	74%
113	HTTP Response Splitting	42%	41%	0%	0%	N/A	N/A	0%	N/A	N/A	4%	100%	4%	74%	100%	74%
134	Externally-Controlled Format String	22%	41%	0%	-	-	-	-	-	-	-	-	-	69%	100%	69%
643	Xpath Injection	42%	41%	0%	0%	N/A	N/A	100%	26%	0%	-	-	-	85%	89%	74%
A2 Broken Authentication		**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
256	Unprotected Credentials Storage	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
259	Hard-coded Password	-	-	-	-	-	-	-	-	-	14%	100%	14%	43%	100%	43%
321	Hard-coded Cryptographic Key	-	-	-	-	-	-	-	-	-	-	-	-	43%	100%	43%
523	Unprotected Credentials Transport	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
549	Missing Password Field Masking	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
A3 Sensitive Data Exposure		**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
315	Cleartext Sensitive Info in Cookie	-	-	-	-	-	-	-	-	-	-	-	-	0%	N/A	N/A
319	Sensitive Cleartext Transmission	-	-	-	-	-	-	-	-	-	-	-	-	70%	41%	0%
325	Missing Required Crypto. Step	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
327	Broken/Risky Crypto. Alg.	-	-	-	-	-	-	-	-	-	-	-	-	50%	100%	50%
328	Reversible One-Way Hash	-	-	-	-	-	-	-	-	-	-	-	-	100%	100%	100%
329	Not Random IV in CBC Mode	-	-	-	-	-	-	-	-	-	-	-	-	100%	100%	100%
614	Missing 'Secure' in HTTPS Cookie	-	-	-	-	-	-	-	-	-	-	-	-	94%	100%	94%
759	One-Way Hash, no Salt	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
760	One-Way Hash, Predictable Salt	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
A5 Broken Access Control		**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
23	Relative Path Traversal	24%	100%	24%	0%	N/A	N/A	100%	42%	0%	4%	100%	4%	85%	88%	74%
36	Absolute Path Traversal	24%	100%	24%	0%	N/A	N/A	100%	42%	0%	4%	100%	4%	85%	89%	74%
566	SQL PK Auth. Bypass	97%	100%	97%	-	-	-	100%	100%	100%	-	-	-	-	-	-
A6 Security Misconfiguration		**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
395	Catching NULL Pointer Deference	-	-	-	0%	N/A	N/A	-	-	-	-	-	-	-	-	-
396	Catch for Generic Exception	-	-	-	0%	N/A	N/A	-	-	-	-	-	-	-	-	-
397	Throws for Generic Exception	-	-	-	0%	N/A	N/A	-	-	-	-	-	-	-	-	-
A7 Cross-Site Scripting		**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
80	Basic XSS	96%	42%	2%	4%	100%	4%	100%	42%	0%	3%	100%	3%	100%	90%	89%
81	Script in Error Message	32%	100%	32%	4%	100%	4%	0%	N/A	N/A	6%	100%	6%	100%	90%	89%
83	Script in Attributes in a Web Page	96%	42%	2%	4%	100%	4%	100%	42%	0%	6%	100%	6%	100%	90%	89%

The calculation of **recall** shows that both ESVD and SpotBugs have poor results when it comes to recall with their respective highest score except for CWE-89 SQL Injection being 65% and 100%. The low scores mean ESVD and SpotBugs cannot reliably find all vulnerabilities in a piece of code, which is a severe limitation. ASIDE has generally low scores as well, but with some exceptions where it is close to full score. LAPSE+ reports all vulnerabilities for a number of CWE entries, but with some exceptions where it reports none. This binary behavior of LAPSE+ may point to it generally performing very well, but it might not cover all that it claims. FindSecBugs, which had the highest coverage, also show very good results, with most CWE entries over 50% and some with full score.

Contrary to their poor results for recall, ESVD and SpotBugs have a high **precision**. This might be due to their implementation as static analysis tools which always have to deal with a trade-off between recall and precision. Another example of this trade-off is LAPSE+ which has a great recall, but has very low precision. FindSecBugs has a precision over 85% for all but one.

Discrimination rate combines recall and precision and will always be a number between zero and the respective recall. Due to this, ESVD, LAPSE+, and SpotBugs all have a low discrimination rate. ESVD and SpotBugs' low discrimination rates are due to their low recall, and LAPSE+ because of its poor precision. ASIDE is not convincing either and only does well on CWE-566, i.e., Authorization Bypass Through User-Controlled SQL Primary Key. FindSecBugs, which has both a good recall and precision, also has a high discrimination rate and proves to be the best solution of these five IDE plugins.

The performance results are varying with clear signs of trade-offs between recall and precision. ESVD and SpotBugs prioritize precision, while LAPSE+ prioritizes recall. This leads to a low discrimination rate for all of the mentioned plugins. Only FindSecBugs has good results for all three performance metrics.

4.3 RQ3: How good is the usability?

All the plugins have one or more CWEs where the number of false positives far outnumbers the number of true positives. As can be seen in Table 7, some tools produce a surprising amount of false

Table 7: Summary of the usability results.

		ASIDE	ESVD	LAPSE+	SpotBugs	FindSecBugs
FP rate	Averaged false positive rate	29%	12%	53%	7%	9%
	False positive rate of total result	50%	60%	60%	56%	13%
Detailed information	What is the problem	×	✓	✓	✓	✓
	Why is it a problem	N/A	×	×	✓	✓
	How to fix the problem	N/A	×	×	×	✓
	Prioritized output	×	✓	×	✓	✓
	Quick fixes	✓	✓	×	×	×
	(E)arly or (L)ate detection	E	E	L	E/L	E/L
	Can suppress warnings	✓	✓	×	×	×
	Eclipse Environment integration	✓	✓	✓	✓	✓
	Available on Eclipse Marketplace	×	×	×	✓	×
	(I)mmediate or (N)egotiated interruptions	N	N	N	N	N
	Easily extendable	×	×	×	✓	×
	Possible to analyze single file only	×	×	×	✓	✓
	Possible to analyze single method only	×	×	×	×	×

positives. Looking closer at the false positive rates of individual CWE categories, all tools have some cases where the rate is close to 60%. The highest false positive rate by all is the one produced by LAPSE+ for the CWE-643 vulnerability class, which is 74%. These numbers are not within the acceptable range of false positives. These high rates can lead to developers not wanting to use such tools in their work. It is important to notice that some tools' false positive rate is generally low except for a few CWEs. SpotBugs has a false positive rate of zero percent with the exception of CWE-89, where it has a false positive rate of 58%. This also shows the importance of the two different measurements of a false positive rate presented in this paper, where one of them tries to balance out this effect by also averaging the false positive rate of each CWE.

The quality of the detection output varies. ASIDE gives no information, and when trying to get more information, it opens a web page where the domain no longer exists. ESVD and LAPSE+ provides a description of what the problem is, but never explains why it is a problem nor how to fix it. SpotBugs and FindSecBugs clearly tell the user why the detected vulnerability is a problem, and FindSecBugs also provides examples of how such vulnerabilities can be fixed.

ESVD, SpotBugs, and FindSecBugs all allow their output to be sorted by priority. ESVD ranks each detected vulnerability based on a number. SpotBugs and FindSecBugs use words, symbols, and colors to show the vulnerability priority. They use words like "scary" with a red bug icon for high priority, and "troubling" with a yellow bug icon next to it for medium priority.

The only two tools that provide quick fixes are ASIDE and ESVD. Both of these give the option of multiple quick fixes, where some of them are not relevant to the current vulnerability at all. Examples of these are "HTML Encoder", "JavaScript Encoder", and "CSS Encoder". Both ASIDE and ESVD seem to produce the exact same quick fixes. These are very simple quick fixes which surround the code with methods from the OWASP Enterprise Security API (ESAPI).

ASIDE and ESVD utilizes early detection by continuously monitoring the workspace for changes. The static code analysis is executed incrementally on small parts of the code while it is being written to ensure quick feedback to the developers. LAPSE+ is using late detection by definition, as it does not automatically scan code. The user has to execute the vulnerability search manually. LAPSE+ will scan each file as if it was the first time, without remembering previous results. SpotBugs, and therefore also the SpotBugs plugin FindSecBugs, utilizes early detection by allowing the user to scan files when they are saved automatically. This can also be turned off so that scans have to be manually executed. That means that SpotBugs supports both early and late detection. SpotBugs can also be integrated into other tools than Eclipse, so that such scans can happen very late if the developer wants that, e.g., right before committing the code to the code repository.

ASIDE and ESVD are the only two tools that can suppress warnings in our evaluation. The result is that the vulnerability warning disappears, with no way of getting it back. This might be a negative experience for the developer if the vulnerability is suppressed by accident.

All of the static analysis tools integrates into the Eclipse IDE. However, the effort to integrate the tools into Eclipse varies. ASIDE is neither in the Eclipse Marketplace nor has any executable ready to be installed in Eclipse. It had to be compiled from source code. ESVD is available on the Eclipse Marketplace, but an error prevents us from downloading it. It seems that the Marketplace is attempting to download the executable from one of the ESVD authors' webpage without success. Therefore the plugin needs to be manually installed into Eclipse. LAPSE+ is not available through the Eclipse Marketplace either, but has an executable that can be downloaded and manually installed. SpotBugs is easily installed through Eclipse Marketplace, i.e., a few clicks are all that is needed. Adding the SpotBugs plugin FindSecBugs requires the user to download the executable from the developer's website. It is then added through the settings of the SpotBugs plugin.

Every tool uses negotiated-style interruptions. None of the tools gives the option of immediate-style interruptions.

All the tools are open-source. This makes it possible for anyone to look at the code. All of the software licenses allow for modification and redistribution of the code. Although all the tools can legally be changed and redistributed, it is difficult to change for most of them. SpotBugs is the only tool that allows other plugins to directly extend the functionality of itself. This can be done through the public SpotBugs API.

SpotBugs and FindSecBugs are the only two tools that allow developers to analyze a single file at a time. ASIDE, ESVD, and LAPSE+ only allow the user to scan the entire project. A finer granularity than file-level analysis is not supported by any of the plugins. Table 7 shows a summary of the usability evaluation results.

5 DISCUSSION

5.1 Comparison with Related Work

The discrepancy we found between claimed and confirmed coverage proves why it is important to test the capabilities of the plugins ourselves and not rely solely on the information provided by the developer. A possible horror situation could be a company which uses an IDE plugin to detect vulnerabilities in their developed code. Being confident in the capabilities of the plugin, they believe they ship software without any severe security vulnerability. However, the plugin does not detect all of the vulnerabilities it claims and the software is shipped with vulnerabilities that cause a security breach and cost the company a lot of money.

Baset and Denning [1] compare the plugins at a purely informational level and use the claimed coverage. Our research contributes with useful and new information about the actual coverage of the plugins that can assist the users in knowing which plugin to use and what to expect from it. While Charest et al. [3] look at coverage for the plugins they compared, they only look for coverage on four different vulnerabilities. In contrast, we look at 29 different CWE entries. Oyetoyan et al. [19] look at all of the 112 CWE entries in the Juliet Test suite, but they did not report the results per CWE entry. Instead, they aggregated the results into categories making it impossible to know which plugin covers which vulnerability. In addition, we tested plugins that have not been covered a lot in previous research. This makes the results of ASIDE, ESVD, LAPSE+, and SpotBugs especially interesting as it is a new contribution to the field. Oyetoyan et al. [19] perform their comparison on all of the 112 CWE entries in the Juliet Test Suite. While it gives their research a wider approach and provides more information, our narrower approach also has an advantage. By narrowing our evaluation down to only vulnerabilities found in OWASP Top 10, we make sure that all of the results are relevant. We only test on vulnerabilities that are considered more important. By testing on all test cases, Oyetoyan et al. [19] may open up for uncommon vulnerabilities to skew their final results. A plugin that does well for uncommon vulnerabilities, but cannot detect common ones, may not be a useful plugin even though it might be overall performing well.

By utilizing the performance metrics used in [5], we generate results that can be compared with other research as we consider these performance metrics the closest to an industry standard. The test cases of Juliet Test Suite are created by the National Security Agency (NSA) and the National Institute of Standards and Technology (NIST) which are trustworthy sources and we believe this enhances the credibility of our results. However, Juliet Test Suite consists of only artificial code. This is not necessarily a negative thing, but it does limit our research. The results indicate how the plugins perform on generated code which is a good indication of what it detects objectively, but it does not say anything about their performance in a real-life setting. Detecting vulnerabilities in natural code is a different matter and the distribution of occurrences by vulnerabilities are very different.

No existing papers are evaluating the usability of multiple static analysis plugins related to detecting security vulnerabilities in the way we did. Some of the plugins have themselves conducted usability evaluations of their tools. Xie et al. [27] conducted a usability evaluation of ASIDE with nine students, where each student used three hours to write code using the plugin. The ASIDE usability evaluation was aimed at evaluating functionality surrounding the plugin itself, without comparisons to other similar tools. Sampaio and Garcia [24] have also evaluated some usability aspects of their own plugin. They conducted an experiment looking at the difference early versus late detection does to a developer's motivation to address reported vulnerabilities. The experiment does not compare ESVD to any other similar tools. Christakis and Bird [4] look at what developers want and need from static analysis tools. Johnson et al. [9] look at why the number of developers using static analysis tools is so low. Sadowski et al. [22] present the guiding philosophy of Google regarding how static analysis tools should behave usability wise. Studies [4], [9], and [22] have all been used as a basis for our usability evaluation, but no direct comparison can be made, because these papers do not compare the usability of different static analysis tools. Given the lack of similar work in existing literature, our evaluation brings forth new information which developers will find useful when deciding which static analysis tool to choose for their own project. In addition, our evaluation gives insights into what existing static analysis tools for detecting security vulnerabilities offer, which in turn can help new tools develop features that existing tools do not currently have.

5.2 Threats to Validity

The main threat to internal validity is selection bias. The selection bias comes from which plugins, vulnerabilities, and metrics that were chosen. To reduce selection bias threat, we have done extensive research into each one in order to make the most sensible decision. We have looked at highly-cited literature and based our decisions on credible sources. Another threat to internal validity is experimenter bias related to the qualitative data analysis of some of the usability metrics. To reduce experimenter bias, the satisfaction requirements are taken from relevant literature regarding the usability of similar tools.

The main threat to external validity is generalizability. The most popular free and open-source static analysis tools were chosen for this evaluation. The limited number of tools that fit our selection criteria make these five tools representative. However, it is not possible to generalize the results to commercial static analysis tools as these are in a completely different category regarding development and research funding.

6 CONCLUSION AND FUTURE WORK

One possible approach to reduce software vulnerabilities is through IDE plugins which alert the developer whenever vulnerable code is written. This allows the vulnerability to be removed at once. There are several open-source vulnerability detection plugins available today. This study presents a coverage, performance, and usability evaluation of five plugins on vulnerability test cases from artificial code. The results of the study show that there are still many categories of vulnerabilities that are not covered by any of the plugins we evaluated. The coverage information published in the plugins' documentation may be misleading. Most plugins have a high false positive rate and are not user-friendly for developers.

To improve the plugins, we can focus on improving all those three aspects evaluated in this study. We believe that improving the coverage of the vulnerabilities, improving performance, and making the plugins more user-friendly will all contribute to more and better use of the plugins and will therefore reduce the number of vulnerabilities in the software code.

REFERENCES

[1] Aniqua Z. Baset and Tamara Denning. 2017. IDE Plugins for Detecting Input-Validation Vulnerabilities. In *2017 IEEE Security and Privacy Workshops (SPW)*. 143–146. https://doi.org/10.1109/SPW.2017.37

[2] Bernhard J. Berger. 2013. lapse-plus. (2013). https://github.com/bergerbd/lapse-plus/

[3] Thomas Charest, Nick Rodgers, and Yan Wu. 2016. Comparison of Static Analysis Tools for Java Using the Juliet Test Suite. In *Proceedings of the 11th International Conference on Cyber Warfare and Security, ICCWS 2016*. 431–438.

[4] Maria Christakis and Christian Bird. 2016. 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*. ACM Press, New York, New York, USA, 332–343. https://doi.org/10.1145/2970276.2970347

[5] Aurelien Delaitre, Bertrand Stivalet, Paul E. Black, Vadim Okun, Athos Ribeiro, and Terry S. Cohen. 2018. *SATE V Report: Ten Years of Static Analysis Tool Expositions*. Technical Report. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.500-326

[6] Find Security Bugs. 2018. Find Security Bugs - The SpotBugs plugin for security audits of Java web applications. (2018). https://find-sec-bugs.github.io

[7] Sarah Heckman and Laurie Williams. 2008. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '08*. ACM Press, New York, New York, USA, 41. https://doi.org/10.1145/1414004.1414013

[8] David Hovemeyer and William Pugh. 2004. Finding Bugs is Easy. *SIGPLAN Not.* 39, 12 (2004), 92–106. https://doi.org/10.1145/1052883.1052895

[9] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, San Francisco, CA, USA, 672–681. https://dl.acm.org/citation.cfm?id=2486877

[10] Benjamin Livshits. 2006. Stanford SecuriBench Micro. (2006). https://suif.stanford.edu/

[11] MITRE. 2018. CWE VIEW: Research Concepts. (2018). https://cwe.mitre.org/data/definitions/1000.html

[12] NIST. 2017. Test Suites. (2017). https://samate.nist.gov/SRD/testsuite.php

[13] NSA. 2012. Juliet Test Suite v1.2 for Java User Guide. (2012).

[14] OWASP. 2016. OWASP ASIDE Project. (2016). https://www.owasp.org/index.php/OWASP_ASIDE_Project

[15] OWASP. 2017. OWASP LAPSE Project. (2017). https://www.owasp.org/index.php/OWASP_LAPSE_Project

[16] OWASP. 2017. OWASP Top 10 - 2017 The Ten Most Critical Web Application Security Risks. (2017).

[17] OWASP. 2018. OWASP Benchmark Project. (2018). https://www.owasp.org/index.php/Benchmark

[18] OWASP. 2018. OWASP WebGoat Project. (2018). https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

[19] Tosin D. Oyetoyan, Bisera Milosheska, Mari Grini, and Daniela S. Cruzes. 2018. Myths and Facts About Static Application Security Testing Tools: An Action Research at Telenor Digital. In *Agile Processes in Software Engineering and Extreme Programming*. Springer International Publishing, 86–103. https://doi.org/10.1007/978-3-319-91602-6_6

[20] T. J. Robertson, Shrinu Prabhakararao, Margaret Burnett, Curtis Cook, Joseph R. Ruthruff, Laura Beckwith, and Amit Phalgune. 2004. Impact of interruption style on end-user debugging. In *Proceedings of the 2004 conference on Human factors in computing systems - CHI '04*, Vol. 6. ACM Press, New York, New York, USA, 287–294. https://doi.org/10.1145/985692.985729

[21] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. 2004. A Comparison of Bug Finding Tools for Java. *15th International Symposium on Software Reliability Engineering* (2004), 245–256. https://doi.org/10.1109/ISSRE.2004.1

[22] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 598–608. https://doi.org/10.1109/ICSE.2015.76

[23] Luciano Sampaio. 2016. TCM_Plugin. (2016). https://github.com/lsampaioweb/TCM_Plugin

[24] Luciano Sampaio and Alessandro Garcia. 2016. Exploring context-sensitive data flow analysis for early vulnerability detection. *Journal of Systems and Software* 113 (2016), 337–361. https://doi.org/10.1016/j.jss.2015.12.021

[25] SpotBugs Team. 2018. SpotBugs Eclipse plugin. (2018). https://marketplace.eclipse.org/content/spotbugs-eclipse-plugin

[26] TIOBE. 2018. TIOBE Index for November 2018. (2018). https://www.tiobe.com/tiobe-index/

[27] Jing Xie, Bill Chu, Heather R. Lipford, and John T. Melton. 2011. ASIDE: IDE Support for Web Application Security. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC '11)*. ACM, New York, NY, USA, 267–276. https://doi.org/10.1145/2076732.2076770

[28] Jun Zhu. 2013. ASIDE-Education. (2013). https://github.com/JunZhuSecurity/ASIDE-Education

Appendix B.

Flow Variants in the Juliet Test Suite

The flow variant indicates whether a test case uses control-flow, data-flow, or neither. Each flow variant has an associated number [NSA, 2012]. Variant 01 is called the *baseline* and uses neither data-flow nor control-flow. It is the simplest form of the vulnerability. Variants 02 through 22 uses control-flow in different variations. An `if`-statement encapsulates the bad source for variants 02 through 14 and the condition for the `if`-statements are listed in Table B.1. The good source is usually encapsulated in an `if`-statement with the condition opposite of the bad source unless otherwise stated. The rest of the control-flow variants are listed in Table B.1 while the data-flow variants are listed in Table B.2. All of the descriptions are based on the documentation provided by NSA [2012].

Table B.1.: Control-Flow Variants in the Juliet Test Suite

Flow Variant	Condition
02	The `boolean` value `true`.
03	The equation `5==5`.
04	A `private static final` constant set to the `boolean` value `true`.
05	A `private` variable set to the `boolean` value `true`.
06	An equation between a `private static final` constant set to `5` and the `int` value `5`.
07	An equation between a `private` variable set to `5` and the `int` value `5`.
08	A `private` method that returns the `boolean` value `true`.
09	A `public static final` constant from another class set to the `boolean` value `true`.
10	A `public static` variable from another class set to the `boolean` value `true`.
11	A `public static` method from another class that returns the `boolean` value `true`.
12	A `public static` method from another class that returns one of the `boolean` values `true` or `false`. In this case the if-statement encapsulating the good source has the same condition.
13	An equation between a `public static final` constant from another class set to `5` and the `int` value `5`.
14	An equation between a `public static` variable from another class set to `5` and the `int` value `5`.
15	The bad source is encapsulated in a `switch`-statements where the control variable is the `int 5` and the case is also the `int 5`.
16	Both the bad and good source are encapsulated in `while(true)`-statements that loops once.
17	The sources are not encapsulated, however, the sink is encapsulated in a `for`-statement that loops once.
21	The bad source is located in a different method within the same class. The control-flow of this method is controlled by an `if`-statement where the condition is a `private` variable which is set in the first method to the `boolean` value `true`.
22	The bad source is located in a different method in a different class. The control-flow of this method is controlled by an `if`-statement where the condition is a `public static` variable which is set in the first method to the `boolean` value `true`.

Table B.2.: Data-Flow Variants in the Juliet Test Suite

Flow Variant	Description
31	Data is copied within the same method.
41	Data is passed as an argument from one method to another in the same class.
42	Data is returned from one method to another in the same class.
45	Data is passed as a private class member variable from one method to another in the same class.
51	Data is passed as an argument from one method to another in different classes in the same package.
52	Data is passed as an argument from one method to another to another in three different classes in the same package.
53	Data is passed as an argument from one method through two others to a fourth; all four methods are in different classes in the same package.
54	Data is passed as an argument from one method through three others to a fifth; all five methods are in different classes in the same package.
61	Data is returned from one method to another in different classes in the same package.
66	Data is passed in an array from one method to another in different classes in the same package.
67	Data is passed in a class from one method to another in different classes in the same package.
68	Data is passed as a member variable in a class from one method to another in different classes in the same package.
71	Data is passed as an Object reference argument from one method to another in different classes in the same package.
72	Data is passed in a Vector from one method to another in different classes in the same package.
73	Data is passed in a LinkedList from one method to another in different classes in the same package.
74	Data is passed in a HashMap from one method to another in different classes in the same package.
75	Data is passed in a serialized object from one method to another in different classes in the same package.
81	Data is passed in an argument to an abstract method called via a reference.

S. Beba, M. M. Karlsen

Implementation Analysis of Open-Source Static Analysis Tools for Detecting Security Vulnerabilities

NTNU
Norwegian University of
Science and Technology