



**Luciano Sampaio Martins de Souza**

**Early Vulnerability Detection for  
Supporting Secure Programming**

**DISSERTAÇÃO DE MESTRADO**

**DEPARTAMENTO DE INFORMÁTICA**  
Programa de Pós-Graduação em  
Informática



**Luciano Sampaio Martins de Souza**

**Early Vulnerability Detection for  
Supporting Secure Programming**

**DISSERTAÇÃO DE MESTRADO**

Dissertation presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática, PUC-Rio, as partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Alessandro Fabricio Garcia

Rio de Janeiro  
January 2015



**Luciano Sampaio Martins de Souza**

**Early Vulnerability Detection for  
Supporting Secure Programming**

Dissertation presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática, PUC-Rio, as partial fulfillment of the requirements for the degree of Master in Informatics.

**Prof. Alessandro Fabricio Garcia**

Advisor

Departamento de Informática – PUC-Rio

**Prof. Anderson Oliveira da Silva**

Departamento de Informática – PUC-Rio

**Prof. Marcelo Blois Ribeiro**

GE Global Research

**Prof. Marcos Kalinowski**

UFJF

**Prof. José Eugenio Leal**

Coordinator of the Centro

Técnico Científico da PUC-Rio

Rio de Janeiro, January 15<sup>th</sup>, 2015

All rights reserved

### **Luciano Sampaio Martins de Souza**

The author graduated in Computer Science from the University Tiradentes (UNIT) in 2006. He received a Graduate Degree with emphasis on Web Development from the University Tiradentes (UNIT) in 2011. His main research interest is: Software Development.

#### Bibliographic data

Souza, Luciano Sampaio Martins de

Early vulnerability detection for supporting secure programming / Luciano Sampaio Martins de Souza; advisor: Alessandro Fabricio Garcia. – 2015.  
132 f. :il. (color.) ; 30 cm

Dissertação (mestrado) Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2015.  
Inclui bibliografia

1. Informática – Teses. 2. Vulnerabilidade de segurança. 3. Detecção contínua. 4. Análise de fluxo de dados. I. Garcia, Alessandro Fabricio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004



## Acknowledgments

First, I would like to thank my parents and my brother. There are no words to describe my gratitude for all the love, trust, encouragement and admiration received. If I am reaching another goal in my life is because of you. You are my foundation, inspiration and motivation.

Secondly, I thank Fernanda Monteiro to be on my side at all times in this dissertation. Her dedication, discipline and patience were inspiring. Her love, affection and understanding were the driving force that enabled me to reach this goal.

My deepest gratitude goes to my advisor, Alessandro Garcia, for all the patience, for the enormous contribution to my academic growth and, above all, the friendship built over these two years of my Masters. His professionalism, dedication and energy for work are exemplary.

I thank all the friends and members of the OPUS research group. Your comments and discussions are always enriching.

My appreciation also goes to all my friends, in particular Igor Oliveira, Rafael Oliveira and Eiji Barbosa.

I thank all the teachers of the Informatics Department of PUC-Rio for their contribution in my education. I also thank all the staff of the department for their services.

Finally, I also thank CNPq and PUC-Rio for financial aid, without which this work would not have been possible.

To all of you, my sincere thanks.

## Abstract

Souza, Luciano Sampaio; Garcia, Alessandro Fabricio (Advisor). **Early Vulnerability Detection for Supporting Secure Programming**. Rio de Janeiro, 2015. 132p. MSc. Dissertation – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Secure programming is the practice of writing programs that are resistant to attacks by malicious people or programs. Programmers of secure software have to be continuously aware of security vulnerabilities when writing their program statements. They also ought to continuously perform actions for preventing or removing vulnerabilities from their programs. In order to support these activities, static analysis techniques have been devised to find vulnerabilities in the source code. However, most of these techniques are built to encourage vulnerability detection a posteriori, only when developers have already fully produced (and compiled) one or more modules of a program. Therefore, this approach, also known as *late detection*, does not support secure programming but rather encourages posterior security analysis. The lateness of vulnerability detection is also influenced by the high rate of false positives, yielded by *pattern matching*, the underlying mechanism used by existing static analysis techniques. The goal of this dissertation is twofold. First, we propose to perform continuous detection of security vulnerabilities while the developer is editing each program statement, also known as *early detection*. Early detection can leverage his knowledge on the context of the code being created, contrary to late detection when developers struggle to recall and fix the intricacies of the vulnerable code they produced from hours to weeks ago. Our continuous vulnerability detector is incorporated into the editor of an integrated software development environment. Second, we explore a technique originally created and commonly used for implementing optimizations on compilers, called *data flow analysis*, hereinafter referred as DFA. DFA has the ability to follow the path of an object until its origins or to paths where it had its content changed. DFA might be suitable for finding if an object has a vulnerable path. To this end, we have implemented a proof-of-concept Eclipse plugin for continuous vulnerability detection in Java programs. We also performed two empirical studies based on several industry-strength systems to evaluate if the code security can be improved through DFA and early vulnerability detection. Our studies confirmed that: (i) the use of data flow analysis significantly reduces

the rate of false positives when compared to existing techniques, without being detrimental to the detector performance, and (ii) early detection improves the awareness among developers and encourages programmers to fix security vulnerabilities promptly.

## **Keywords**

Early detection; security vulnerability; data flow analysis; secure programming.

## Resumo

Souza, Luciano Sampaio; Garcia, Alessandro Fabricio. **Detecção de vulnerabilidades de segurança em tempo de programação com o intuito de dar suporte a programação segura**. Rio de Janeiro, 2015. 132p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Programação segura é a prática de se escrever programas que são resistentes a ataques de pessoas ou programas mal-intencionados. Os programadores de software seguro precisam estar continuamente cientes de vulnerabilidades de segurança ao escrever as instruções de código de um programa. Eles precisam estar preparados para executar continuamente ações para prevenir ou removê-las de seus programas. Neste cenário, as técnicas de análise estática foram concebidas para encontrar vulnerabilidades no código-fonte. No entanto, a maioria das técnicas existentes são construídas de uma maneira a incentivar a detecção de vulnerabilidade tardiamente, apenas quando os desenvolvedores já tenham produzido (e compilado) por completo um ou mais módulos de uma aplicação. Portanto, esta abordagem, também conhecida como *detecção tardia*, não promove programação segura, mas apenas análise retrospectiva de segurança. O atraso na detecção de vulnerabilidades também é influenciado pela alta taxa de falsos positivos, gerados pelo *casamento de padrões*, mecanismo comumente usado por técnicas de análise estática. Esta dissertação tem dois objetivos. Em primeiro lugar, nós propomos promover detecção de vulnerabilidades, enquanto o desenvolvedor está editando cada instrução do programa, também conhecida como *detecção antecipada*. A detecção antecipada pode aproveitar o conhecimento do desenvolvedor sobre o contexto do código que está sendo desenvolvido, ao contrário da detecção tardia em que os desenvolvedores enfrentam dificuldades para lembrar detalhes do código vulnerável produzido a horas ou semanas atrás. Nosso detector de vulnerabilidades é incorporado ao editor de um ambiente integrado de desenvolvimento de software. Em segundo lugar, vamos explorar uma técnica criada e comumente utilizada para a implementação de otimizações em compiladores, chamada de *análise de fluxo de dados*, doravante denominada como DFA. DFA tem a capacidade de seguir os caminhos de um objeto, até a sua origem ou para caminhos onde o seu conteúdo tenha sido alterado. DFA pode ser adequado para encontrar se um objeto tem um

ou mais caminhos vulneráveis. Para isso, implementamos um plugin Eclipse, como prova de conceito, para detecção antecipada de vulnerabilidades em programas Java. Depois disso, foram realizados dois estudos empíricos baseados em vários sistemas da indústria para avaliar se a segurança de um código fonte produzido pode ser melhorada através de DFA e detecção contínua de vulnerabilidades. Nossos estudos confirmaram que: (i) análise de fluxo de dados reduz significativamente a taxa de falsos positivos, quando comparada com técnicas existentes, sem prejudicar o desempenho do detector, e (ii) a detecção antecipada melhora a consciência entre os desenvolvedores e os incentiva a corrigir vulnerabilidades de segurança prontamente.

### **Palavras-chave**

Vulnerabilidades de segurança; Detecção contínua; Análise de fluxo de dados.

## Contents

1 Introduction	16
1.1. Problem Description	18
1.2. Limitations of Related Work	21
1.3. Goals and Overview of the Solution	24
1.4. Research Questions	25
1.5. Dissertation Structure	27
2 Background and Related Work	28
2.1. Security Taxonomy	28
2.2. Types of Security Vulnerability	29
2.3. Detection of Security Vulnerabilities	32
2.3.1. Manual Inspection	33
2.3.2. Dynamic Analysis	39
2.3.3. Static Analysis	41
2.4. False Positives and False Negatives	44
2.5. Late Detection and Early Detection	46
2.6. Related Work	49
2.6.1. Lapse+	50
2.6.2. ASIDE	50
2.6.3. CodePro Analytics	51
2.7. Lack of Knowledge on Secure Programming	51
3 Data-Flow-Driven Heuristics for Vulnerability Detection	54
3.1. Data Flow Analysis with Context-Sensitivity	55
3.2. Entry-Point	57
3.3. Exit-Point	58
3.4. Sanitization-Point	59
3.5. Algorithm	60
3.6. Supported Vulnerabilities	62
3.6.1. Command Injection	63
3.6.2. Cookie Poisoning	64
3.6.3. Cross-Site Scripting (XSS)	66

3.6.4. HTTP Response Splitting	67
3.6.5. LDAP Injection	68
3.6.6. Log Forging	70
3.6.7. Path Traversal	71
3.6.8. Reflection Injection	72
3.6.9. Security Misconfiguration	74
3.6.10. SQL Injection	75
3.6.11. XPath Injection	76
3.7. Current Limitations	78
4 Early Vulnerability Detector: Implementation	80
4.1. Architecture	80
4.1.1. Verifier	81
4.1.2. Analyzer	82
4.1.3. Manager	83
4.1.4. Reporter	83
4.2. Call Graph	84
4.2.1. Clean Call Graph	84
4.2.2. Prime Call Graph	85
4.3. Features	85
5 Evaluation	87
5.1. Study 1: Accuracy Benchmarking	88
5.1.1. Testing Environment	89
5.1.2. Open-Source Projects	90
5.1.3. Supported Vulnerabilities	91
5.1.4. Precision, Recall and F-measure	93
5.1.5. Study 1: Results	94
5.2. Study 2: Late vs. Early Detection—A Quasi-Experiment	104
5.2.1. Methodology	105
5.2.2. Study 2: Results	109
5.3. Concluding Remarks	119
6 Conclusion	121

6.1. Contributions	122
6.2. Future work	123
7 References	125
Appendix 1 - Participant Profile Questionnaire	132
Appendix 2 - System Requirements	133



## List of Figures

Figure 1 False positive from a tool (ASIDE [Zhu 2012]) that uses pattern matching.	20
Figure 2 False positive from a tool (Lapse+ [Livshits 2006]) that uses pattern matching.	22
Figure 3 Lapse+ view displaying the detected vulnerabilities.	22
Figure 4 False negative from a tool (ASIDE [Zhu 2012]) that uses pattern matching.	23
Figure 5 Sample of several security vulnerabilities in a small piece of code.	35
Figure 6 Exception message being displayed to the user.	38
Figure 7 ZAP's screenshot.	41
Figure 8 Lapse+ view displaying the detected vulnerabilities.	42
Figure 9 Examples of pattern matching limitations.	43
Figure 10 ASIDE's false negative of SQL injection.	45
Figure 11 Late detection workflow.	46
Figure 12 Early detection workflow.	47
Figure 13 Early vulnerability detection from ASIDE and ESVD (section 4).	48
Figure 14 Code snippet with SQL injection vulnerability.	52
Figure 15 Code snippet without SQL injection vulnerability.	52
Figure 16 Data Flow Analysis representation.	55
Figure 17 Data Flow - Context Insensitive - CodePro Analytics [Google 2001].	57
Figure 18 Data Flow - Context Sensitive - ESVD.	57
Figure 19 Example of Entry-Point.	58
Figure 20 Example of Exit-Point.	59
Figure 21 Sanitization-Point.	60
Figure 22 Infinite loops.	62
Figure 23 Command Injection vulnerability.	63
Figure 24 Command Injection mitigation.	64
Figure 25 Cookie Poisoning - Problem.	65
Figure 26 Cookie Poisoning - Mitigation.	66
Figure 27 Cross-Site Scripting - Problem.	66
Figure 28 Cross-Site Scripting - Mitigation.	67

Figure 29 HTTP Response Splitting - Problem.	67
Figure 30 HTTP Response Splitting - Mitigation.	68
Figure 31 LDAP - Problem.	69
Figure 32 LDAP - Mitigation.	69
Figure 33 Log Forging - Problem.	70
Figure 34 Log Forging - Mitigation.	71
Figure 35 Path Traversal - Problem.	72
Figure 36 Path Traversal - Mitigation.	72
Figure 37 Reflection Injection - Problem.	73
Figure 38 Reflection Injection - Mitigation.	74
Figure 39 Security misconfiguration - Problem.	75
Figure 40 Security misconfiguration - Mitigation.	75
Figure 41 SQL injection - Problem.	76
Figure 42 SQL injection - Mitigation.	76
Figure 43 XPath Injection - Problem.	77
Figure 44 XPath Injection - Mitigation.	78
Figure 45 False positives on containers generated by ESVD.	79
Figure 46 False positives on containers generated by ASIDE, CodePro Analytics and Lapse+.	79
Figure 47 ESVD Plugin Architecture.	81
Figure 48 Implemented Verifiers.	82
Figure 49 Verifiers of the Security Vulnerability Analyzer.	82
Figure 50 Security Vulnerability View.	84
Figure 51 Types of Interactions.	84
Figure 52 Clean Call Graph.	85
Figure 53 Prime Call Graph.	85
Figure 54 Equation of Precision.	94
Figure 55 Equation of Recall.	94
Figure 56 Equation of F-measure.	94
Figure 57. BlueBlog.	96
Figure 58 PersonalBlog.	96
Figure 59 PersonalBlog false positive.	97
Figure 60 WebGoat.	97
Figure 61 False positive on WebGoat.	98

Figure 62 Roller.	98
Figure 63 Roller false positive.	99
Figure 64 Pebble.	99
Figure 65 Pebble false positive.	100
Figure 66 NCO.	100
Figure 67 NCO false positive.	101
Figure 68 Compilation of results from all analyzed projects.	102
Figure 69 Memory Usage.	103
Figure 70 Execution time.	104

## List of Tables

Table 1 Brief description of the top 10 most found security vulnerabilities.	32
Table 2 Example of a simple checklist to perform manual inspection.	34
Table 3 List of Entry-Points.	58
Table 4 List of Exit-Points.	59
Table 5 List of Sanitization-Points.	60
Table 6 Characteristics of the tools used in our evaluation.	88
Table 7 Benchmark applications.	91
Table 8 Supported vulnerabilities.	93
Table 9 Compilation of results from all analyzed projects.	102
Table 10 Description of the tasks of the coding exercise.	106
Table 11 Number of participants and average of years of experience.	110
Table 12 Final number of participants and average of years of experience.	111
Table 13 Distribution of the participants on each group.	112
Table 14 Programming time (hours) per group.	113
Table 15 Experiment time (hours) and tasks performed by each participant.	114
Table 16 Number of vulnerabilities added, removed and left.	115
Table 17 Vulnerabilities added, removed and left during the experiment.	116
Table 18 Security vulnerabilities reported during the experiments.	117
Table 19 Average time (hours) until a vulnerability was added.	118

# 1

## Introduction

Secure programming is the practice of writing software systems that are resistant to attacks by malicious people or programs [Apple 2013]. In order to promote secure programming, developers have to be continuously aware of security vulnerabilities when writing their program statements. They need to be prepared to continuously perform actions for preventing and removing vulnerabilities from their programs. Security vulnerability (or simply vulnerability) is a flaw within a software system that can be exploited to allow an attacker to reduce the system's information assurance [Organization for Internet Safety 2004]. An attacker is a person or application that intends to cause damage to a software system. By exploiting a security vulnerability, an attacker takes advantage of this vulnerability, typically for malicious purposes, such as stealing information or causing damage to a computer system.

In the context of this dissertation, we are particularly concerned with security vulnerabilities introduced by programmers when adding or editing code statements. Unfortunately, existing software development environments, such as Eclipse [“Eclipse” [S.d.]], NetBeans [“NetBeans” [S.d.]] and others, often do not offer the means to make programmers aware they are writing insecure code. Therefore, if a company or a developer wants support for performing secure programming, they have to use additional external tools, such as: IBM Appscan [IBM 2001], Lapse+ [Livshits 2006] and others. However, these solutions frequently do not fit properly on the development workflow, because they either are not integrated into the development environments or do not detect the vulnerabilities exactly when they are added into the source code. Consequently, they only support “a posteriori” security analysis in the source code rather than supporting actual secure programming. Thus, both novice and experienced developers are not encouraged to detect and remove security vulnerabilities in the code they are editing.

According to a well-established report about security statistics of vulnerabilities found in the source code, published in May 2013 [Grossman 2013], 86% of all audited websites contained at least one serious security vulnerability in their source code. By serious it means a vulnerability through which an attacker could take control over all, or some part of the website. Unfortunately, developers commonly become concerned about security vulnerabilities only after someone reports a critical security problem. In other words, developers become aware when the vulnerabilities have already been exploited and, as a result, finance and reputation might have been harmed.

For instance, in 2001 a single vulnerability present in the source code of the Microsoft's Internet Information Server (IIS) cost an estimated \$2 billion dollars of damage [Cowley 2001]. After that, Microsoft's chairman Bill Gates demanded employees to focus on building more secure software to avoid this type of problems in the future. This is simply one case from many, where a company only starts to worry about security when it is already too late. However, this passive behavior is not recommended, companies should take pro-active actions in order to truly make a software system secure or at least resistant to attacks. This can be accomplished by adding security awareness among developers and into the development workflow. To corroborate this information, there is a study from the National Institute of Standards and Technology (NIST) that states that the number of new source code vulnerabilities discovered each year has more than doubled in the last decade and this number is still rising [Telang and Wattal 2005].

The aforementioned scenarios are alarming and reinforce that secure programming should be intertwined in the workflow of software programmers. Explicit support for early vulnerability detection would likely encourage both students and developers to acquire concrete knowledge about the subject in the context of their academic and industrial software projects. Every software programmer should become aware of, at least, the most common security vulnerabilities when writing statements and functions in their code, so they can early observe and treat security vulnerabilities accordingly.

## 1.1. Problem Description

Developers should be aware of emerging security vulnerabilities as they write their program statements. The expectation that every developer would become a security specialist is not feasible in software projects. However, if a company wants to perform secure programming on all of its projects, at least the most common security vulnerabilities should be handled by the programmer who is adding or editing the code, leaving only the more complex ones to actual security specialists. These specialists are highly skilled workers and they cost significantly more than programmers. Therefore, their time on each project should be optimized to the fullest. In order to achieve this, developers should receive tooling support to continuously detect and remove security vulnerabilities in their programming context. The early detection and removal of security threats are expected to decrease the chances of vulnerabilities being exploited by attackers.

Support for early detection should be provided in the context of program edition. Otherwise, developers might be unconscious about the security vulnerabilities emerging in their code. Vulnerabilities should be fixed before the program goes to testing or production. Ideally, they should be detected and fixed before the programmer's code is committed into the project's repository. If done afterwards, developers might spend hours, days or weeks to find out and fix vulnerabilities in their code [Baca et al. 2008]. In fact, there is recently a trend to investigate solutions that support early detection of some implementation problems, such as modularity problems [Albuquerque et al. 2014] and exception handling flaws [Barbosa et al. 2012]. However, there is limited knowledge on how to specifically support early detection of security vulnerabilities in programs. Zhu et al. [Zhu 2012] were the only authors to recently (2013) create a prototype solution, called ASIDE (Application Security plugin for Integrated Development Environment), that performs early detection of security vulnerabilities. However, when ASIDE was compared to other static analyzers (see section 5.1.5), it resulted in a much higher amount of false positives. False positive is the incorrect indication of the presence of a vulnerability ["An Overview Of Vulnerability Scanners" 2008]. As described by Nadeem et al. [Nadeem et al. 2012], several false positives discourage the use of existing static analysis solutions (including

Zhu's prototype) for security vulnerability detection. Unfortunately, according to Nadeem et al. [Nadeem et al. 2012], existing solutions result in a high rate of false positives, because most of them are strictly based on a low-accurate technique called *pattern matching*.

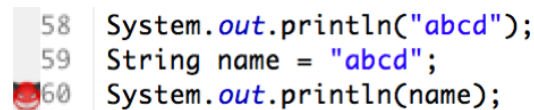
*Pattern matching* is a technique for checking if a pattern matches a given sequence of tokens (letters, numbers, punctuation, and certain symbols) [Kohli and Joshi 2013]. In order to be positive, this match has to be exact, otherwise the outcome is negative. In the context of security vulnerabilities, this technique compares the source code that is being analyzed against a code template that usually represents a security vulnerability. These code templates are stored on what is known as *knowledge base*. One problem with this technique is the fact that it does not consider program's context (of variables and methods) when searching for vulnerabilities. For instance, some important contextual information about variables are: where they were created, which values were assigned to them, and the like. As far as methods are concerned, the vulnerability detection should be sensitive to which methods invoke the current one, what are possible contents of the provided parameters, and so forth. All these contextual issues are simply ignored by solutions that apply pattern-matching technique. ASIDE [Zhu 2012], the only existing solution (to the best of our knowledge) that supports early vulnerability detection, is also a representative of tools that apply *pattern matching* to identify security vulnerabilities in the source code.

*Pattern matching* solutions usually perform four main actions for analyzing and finding security vulnerabilities in the source code. First, they read line by line in each program file. Second, when they find a method, they verify if that method matches with one of the code templates from their *knowledge base*. Third, if the method matches, they verify what is the element that is being passed as a parameter to the method. Finally, if the parameter is one of the elements, they consider unsecure, they report it as vulnerable, otherwise, they consider it as secure. In order to demonstrate this behavior, Figure 1 has a code snippet that was analyzed by ASIDE. ASIDE does not recognize line 58 as vulnerable, because the method *System.out.println* with a *string literal* is not in its *knowledge base*. The reason for that is the idea that a *string literal* is inserted into the code by the developer. Therefore, s/he does not want to insert vulnerabilities into her/his own



source code. Thus, a *string literal* cannot have a malicious content in it and is considered secure.

On the other hand, the technique flags (icon on the left - red devil) line 60 as vulnerable, because the method `System.out.println` with a *variable* is registered in its *knowledge base*. The reason is that variables can receive their content from users and this content can contain malicious code, which can be used to attack the application. Therefore, such variables should not be trusted. Although variables can, in fact, contain malicious content, if we carefully analyze the code snippet depicted in Figure 1, it is possible to observe that the only possible content from variable *name* is a *string literal* “abcd”. Therefore, following the rule that a *string literal* cannot have a malicious content in it, this line should also be considered secure.



```

58 System.out.println("abcd");
59 String name = "abcd";
60 System.out.println(name);

```

Figure 1 False positive from a tool (ASIDE [Zhu 2012]) that uses pattern matching.

The strict use of *pattern matching* is not able to correctly state that the line is secure in the example above, because it does not have the ability to follow the data flow of the variable being passed into the method `System.out.println()`. The consequence of the limitations of this nature is that existing security analysis tools yield from 20% to 30% of false positives [Baca 2009]. These percentages may not seem very high at first, but the corresponding absolute values often represent hundreds of false warnings (code elements incorrectly flagged as vulnerable) even for small or medium-sized programs. Therefore, the underlying technique used to find security vulnerabilities should produce results that are more accurate.

In particular, the use of early detection without employing a high accuracy technique would not be sufficient for achieving secure programming. Developers would be discouraged to write secure programs if they often become frustrated by continuously treating a high amount of false positives when editing their code. Unfortunately, we downloaded and tested all the existing operational solutions and they presented these problems. To this end, this dissertation proposes the use of a different technique, named *context-sensitive data flow analysis* [Hammer et al. 2006]. This technique is able to consider program's context (of variables and methods) when searching for vulnerabilities in the context of early detection. We

will investigate if the use of this technique affects the final program security positively or negatively. To the best of our knowledge, we have not seen any other study addressing this problem in the literature.

## 1.2. Limitations of Related Work

There are dozens of available solutions intended to perform detection of security vulnerabilities in the source code [OWASP 2003a]. Some noteworthy examples are SSVChecker [Dehlinger et al. 2006], FindBug [Pugh and Loskutov 2006], ASIDE [Zhu 2012], Lapse+ [Livshits 2006], CodePro Analytics [Google 2001], Fortify HP [HP 2002] and AppScam IBM [IBM 2001]. They can be categorized in several different ways, such as which vulnerabilities they give support and how they perform their detection. However, none of them properly address the problems mentioned in the previous section.

First, the vast majority of the existing solutions are not integrated in the developer's workflow. In other words, they are executed only a posteriori, when developers have fully produced (and compiled) a method, a class or the code relative to entire days or weeks of programming. This disconnection of vulnerability detection with the developer's workflow, compounded with additional reasons [Xie et al. 2011], such as tight deadlines, will increase the probability of developers forgetting or simply deciding not to run the external detection tools. In fact, developers may assume it is not their responsibility to perform secure programming, as the software development environment does not naturally encourage her/him to do it. This behavior of performing vulnerability detection only after the code has been completed is known as *late detection* of security vulnerabilities.

Late detection does not enable to fully realize the notion of secure programming (section 1), because it reports any possible vulnerability only afterwards, when the code is already complete. In other words, these solutions only support security analysis in the source code rather than actual secure programming. A consequence of applying late detection approaches is that all those hours and lines of code written by developers may have been worthless. Even worse, developers usually spend a considerable amount of time actually

removing vulnerabilities of their code [Baca et al. 2008]. Existing late detection approaches report a long list of vulnerabilities. Then, a programmer has to examine each one in a location of the source code where they not always remember about the context required to fully understand and remove the vulnerability.

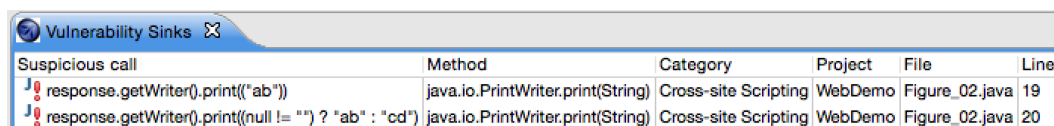
The second problem of existing solutions is the exploitation of a vulnerability detection technique with low accuracy. Lapse+ [Livshits 2006] is an example of a tool that also applies *pattern matching* to identify security vulnerabilities in the source code. As it can be seen in Figure 2 and Figure 3, Lapse+ correctly recognizes line 18 as secure. As already mentioned, a *string literal* is inserted into the code by the developer and s/he does not want to insert vulnerabilities into her/his own source code. Thus, a *string literal* cannot have a malicious content in it. On the other hand, it incorrectly flags lines 19 and 20 as vulnerable to cross-site scripting (XSS) [OWASP 2013a]. Cross-site scripting (XSS) is a vulnerability that occurs whenever an application takes untrusted data and sends it to a web browser without proper validation. After that, this untrusted data could be used to cause a severe damage on the website's visitors. Lapse+ produces these false warnings as it does not analyze the data flow of the *parenthesis* (line 19) and the *ternary operator* (line 20) being passed into the method `response.getWriter().print()` to discover that these contents are also secure.

```

18 response.getWriter().print("ab");
19 response.getWriter().print(("ab"));
20 response.getWriter().print((null != "") ? "ab" : "cd");

```

Figure 2 False positive from a tool (Lapse+ [Livshits 2006]) that uses pattern matching.

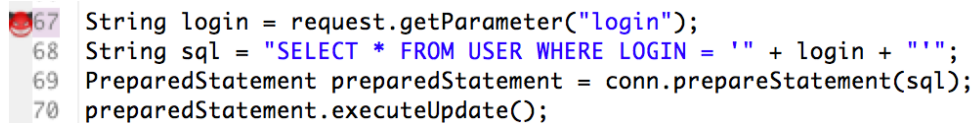


Suspicious call	Method	Category	Project	File	Line
response.getWriter().print(("ab"))	java.io.PrintWriter.print(String)	Cross-site Scripting	WebDemo	Figure_02.java	19
response.getWriter().print((null != "") ? "ab" : "cd")	java.io.PrintWriter.print(String)	Cross-site Scripting	WebDemo	Figure_02.java	20

Figure 3 Lapse+ view displaying the detected vulnerabilities.

In another example (Figure 4), the statement on line 69 is using a *preparedStatement* object, which is the recommended object to handle database queries. However, this fact alone is not sufficient to prevent SQL (Structured Query Language) injection [OWASP 2013b] vulnerability, because the query (*sql* variable) is concatenating the SQL command (line 68) with the *login* variable,

which might contain unsafe data sent by the user. SQL injection occurs when untrusted data is sent to an interpreter as part of a command or query. This data can trick the interpreter into executing unintended commands or accessing data without proper authorization. The *knowledge base* of ASIDE [Zhu 2012] states that if the code is using a *preparedStatement* object, it can be considered secure. However, as it will be described in more detail in section 3.6.10, this is not sufficient and the vulnerability persists.



```

67 String login = request.getParameter("login");
68 String sql = "SELECT * FROM USER WHERE LOGIN = '" + login + "'";
69 PreparedStatement preparedStatement = conn.prepareStatement(sql);
70 preparedStatement.executeUpdate();

```

Figure 4 False negative from a tool (ASIDE [Zhu 2012]) that uses pattern matching.

Therefore, existing automated approaches support inaccurate security analysis and do not encourage secure programming. These characteristics might also lead to a number of other side effects. For instance, the execution time of the security detection tool might be high if the developer runs it after they concluded the implementation of its program. We noticed on Lapse+ [Livshits 2006] and CodePro Analytics [Google 2001] that, depending on the size of the project, the report with all the security vulnerabilities may take minutes to be generated. Developers do not want to wait a long period to receive feedback about potential security problems in their code. This problem happens because every time the tool runs, it scans all the files of the selected projects, even if only one file has been modified since the last time the tool ran, thus, making developers more prone to execute it only occasionally. This is a design choice explicitly made by developers of these tools as they are typically used for posterior security analysis of the entire software project. If the code edits by the programmer are checked early - i.e. as soon as they introduce new code statements, the list is not going to be exhaustive. Incremental checking could be applied and developers will be encouraged to fix and avoid it while coding. However, the high rate of false negatives, imposed by *pattern-matching* solutions, would not make the realization of incremental checking viable. The programmers would have to handle too many false warnings for every change made in the program.

In summary, the key limitation of existing solutions is twofold: late detection and the use of *pattern matching* leading to a high rate of false positives. Late detection seems to be the most common approach. However, it does not fully

enable secure programming. We found only ASIDE [Zhu 2012] partially support early detection (see section 2.6.2). However, it also relies on pattern matching to identify vulnerability candidates. From all the solutions we have analyzed, most of them use pattern matching as their underlying detection technique. The only exception is CodePro Analytics [Google 2001], which relies on a limited form of data flow analysis (section 3.1). CodePro Analytics also does not support early vulnerability detection. Finally, for all existing solutions, the rate of false positives is high, reaching up to 50% in certain cases [Baca 2009].

### 1.3. Goals and Overview of the Solution

There are several existing solutions [OWASP 2003a] to find security vulnerabilities in the source code (section 1.2). However, only a subset of them was operational. In other words, developers are actually able to download and work with only a small set of existing solutions. For all the cases, including the operational ones, they did not fit properly in the development's workflow and presented a high rate of false positives. These may be two of the key reasons that explain why these tools are not being widely used by programmers, even though the importance of security is recently growing [Xie et al. 2011][Blyth 2004][Willis et al. 2006].

Based on these motivating factors, we propose the combination of two ideas on this dissertation. First, we propose to support a change from the default behavior of late detection to early detection. We believe this is the best way to provide actual support for secure programming. Second, we propose new heuristics to find security vulnerabilities using a technique named *context-sensitive data flow analysis* [Hammer et al. 2006] instead of using *pattern matching* or *context-insensitive data flow analysis* [Hammer et al. 2006], the most frequently used techniques. We expect the use of data flow analysis will decrease the rate of false positives yielded by existing solutions. Consequently, the improvement on the accuracy detection will likely to encourage developers to detect and remove vulnerabilities in their source code. Finally, we design and execute two empirical evaluations. They are aimed to improve our understanding about the use of early vulnerability detection based on data flow analysis. The first

evaluation intends to verify if developers receiving continuous detection support (i.e. early detection) could produce more secure code than developers receiving support afterwards, i.e. only at the end (late detection) of their programming session. The second evaluation was specifically targeted at measuring the accuracy of our prototype (using *data flow analysis*) compared to other existing solutions.

After our detection heuristics were created, we designed and implemented a prototype. This prototype enabled us to verify if and to what extent our detection heuristics could decrease the rate of false positives when compared to other techniques. Our heuristics focused on vulnerabilities that occur on web applications that stem from program input and output not being properly validated. Although these heuristics are generic and can be implemented to the context of other programming languages, the first (and current) version of our prototype only provides support for the Java<sup>1</sup> programming language. This choice was driven by the fact that Java is one of the most popular programming languages in the world [Zeichick 2012]. The prototype is a plugin for the Eclipse<sup>2</sup> IDE (integrated development environment), which is the most popular IDE used for the Java programming language[Geer 2005]. The plugin, called *ESVD - Early Security Vulnerability Detector*, can be downloaded from the Eclipse Marketplace [Sampaio and Garcia 2014].

#### **1.4. Research Questions**

The main goal of our research was to find out if and to what extent, early vulnerability detection, based on the use of *data flow analysis*, could help developers on improving the security of their programs. To achieve this goal, as previously mentioned, we developed a prototype that performs constant background verifications in the source code being edited by a programmer. The vulnerability detection algorithm takes into consideration the *data flow* of the program statements under analysis. The technical objectives of our investigation can be better characterized in two research questions, which are individually

---

<sup>1</sup><https://www.oracle.com/java/>

<sup>2</sup><https://www.eclipse.org>

described and discussed below. We address our research questions through two empirical studies, as stated in section 1.3.

Our first research question aims at confirming (or refuting) whether the use of data flow analysis improves the accuracy of vulnerability detection in the source code:

RQ1: Can data flow analysis decrease the rate of false positives when compared to pattern matching?

For a technique to be considered accurate, it should not result in a high number of false positives. To achieve this goal, the technique should also take into consideration possible contents of variables, values returned by methods, reflection, recursion and others. However, it is not clear if and to what extent the use of data flow analysis outperforms the use of pattern matching. For instance, the use of the former can eventually generate other unknown forms of false positives, i.e. not currently generated by the use of pattern matching.

Our second research question is concerned in checking whether an early detection approach can help developers produce more secure software systems:

RQ2: Can the early detection approach help developers produce more secure code when compared to late detection?

Early vulnerability detection might encourage proper involvement from developers in secure programming. Developers will receive real-time vulnerability notifications about the source code they are currently working on. Then, developers are likely to have proper knowledge to understand those vulnerabilities and perform repairing actions. When vulnerability detection is performed afterwards, the developer might not have the same knowledge to understand the vulnerable code. As a consequence, they might leave more vulnerabilities unhandled. Therefore, we want to investigate: (i) if making the programmer aware early about security problems can help them to produce more secure source code, or (ii) if promoting this constant involvement can frustrate developers into a point where they are discouraged to think about secure programming.

## **1.5. Dissertation Structure**

The remainder of this dissertation is structured as follows. Section 2 presents the theoretical background required to understand the main concepts of this dissertation. This section also describes the main existing studies about the subject discussed on this dissertation. Section 3 describes the heuristics created to find security vulnerabilities in the source code. This section also describes the components, which compose our algorithm. It also discusses which vulnerabilities are supported by our heuristics, how these vulnerabilities occur in the source code and what is necessary to remove them. Section 4 presents the software architecture of our implemented solution. Section 5 discusses the empirical evaluations to explicitly address our research questions (section 1.4). Finally, section 6 concludes this dissertation, by showing the main contributions made and discussing some possibilities and future research directions.



## 2 Background and Related Work

This section presents the theoretical background required to understand the main concepts of this dissertation. Section 2.1 describes the main keywords used in the field of software security. Section 2.2 discusses the concept of security vulnerability and presents the most common vulnerability types found on web applications. Section 2.3 presents the existing techniques and respective tooling support for detecting occurrences of security vulnerabilities. As all the existing static analysis techniques are of heuristic nature, they lead to detection mistakes. These detection mistakes are classified as either false positives or false negatives. These concepts are discussed in section 2.4, and enable us to assess and understand the degree of accuracy of existing techniques. Section 2.5 presents the main advantages and disadvantages of both early and late detection for vulnerability detection. Section 2.6 presents the main existing studies about the subject discussed on this dissertation. Finally, Section 2.7 describes empirical evidence about the developer's lack of knowledge on how to perform secure programming.

### 2.1. Security Taxonomy

Software security, as any other field of software engineering, has its own unique terminology. The understanding of this terminology is required in order to understand the topics presented in this dissertation. We used the commonly referenced taxonomy from [Tsipenyuk et al. 2005] and their definitions are presented below.

- *Input* - Data provided by the user of a software system or by another application.
- *Malicious input* - Input that is intended to cause harm to the application or to other users.

- *Untrusted/Unvalidated input* - Input that has not been compared to a range of expected values or has not removed malicious data from its content to ensure it is safe to use.
- *Trusted/Validated/Sanitized input* - Input that has been compared to a range of expected values and has removed (if any) malicious data from its content to ensure it is safe to use.
- *Encode/Escape input* - The process of converting some input, e.g. a sequence of characters (letters, numbers, punctuation, and certain symbols) into a specialized digital format, such as HTML tags.
- *Decode/Unescape input* - The process of converting an encoded input back into its original sequence of characters.
- *Security vulnerability* - Security vulnerabilities (or simply vulnerabilities) is a flaw within a software system that can be exploited to allow an attacker to reduce the system's information assurance [Organization for Internet Safety 2004]. In other words, an attacker can exploit a program vulnerability to cause damage to the application or its users.
- *Process/Analyze/Scan a file* - In the context of this dissertation, it means searching (reading and processing) all lines of a program file for security vulnerabilities.

## 2.2. Types of Security Vulnerability

In the context of this dissertation, a program is considered secure when it is able to resist attacks by malicious people or programs. These attacks can have different goals, such as stealing information/money, shutdown the application, or pretending a user is someone else. Most of these attacks are only possible because the source code of a software system contains security vulnerabilities.

Some examples of security vulnerabilities are *cross-site scripting* (XSS) [OWASP 2013a] and *unauthorized access* [OWASP 2013c]. They are well-known types of security vulnerability for the damage they can cause. An explanation about XSS is presented in section 1.2. Unauthorized access occurs when an attacker is able to access a resource, such as web page or a specific file without the proper credentials. Credentials are a unique identification (e.g. login,

user id, or email) and password used to identify a person that wants to access an application.

A web software system can be exposed to several other types of security vulnerability. Therefore, there are some initiatives to support the development of secure web systems. The most well known of these initiatives is the Open Web Application Security Project (OWASP). OWASP is an open community dedicated to enabling organizations to develop, purchase, and maintain applications that can be trusted [OWASP 2003b]. In early 2013 the OWASP initiative released the fifth edition of one of the most referenced and respected reports with regard to security vulnerabilities in web applications, known as OWASP Top 10 [OWASP 2013d]. The report contains the top 10 most common type of vulnerabilities found in web applications. Their frequency was computed based on the analysis of more than 500,000 vulnerabilities identified in thousands of applications [OWASP 2013d]. These data are shared by eight datasets from seven companies, including Aspect Security, HP, Minded Security, Softtek, Trustwave - SpiderLabs, Veracode and WhiteHat Security Inc. These companies are specialized in application software security.

The report also ranks the types of vulnerabilities from the most critical to the least critical, in terms of estimates of exploitability, detectability and impact. *Exploitability* means how easy an attacker can exploit the vulnerability. *Detectability* means how easy a developer can detect the vulnerability. *Impact* means how severe is the impact in case the vulnerability is exploited by an attacker. Although the report focuses on vulnerabilities of web software systems, some of them can occur in desktop applications as well.

Table 1 presents a brief characterization of the top 10 security vulnerabilities, following the same order as in the original report. A more complete description of each type of vulnerability can be found in [OWASP 2013d]. Several of these vulnerabilities depart from the same main root problem; they exist only because developers rely on user-provided input and do not properly sanitize them. These vulnerabilities that stem from untrusted inputs are recognized as being the most common and capable of causing severe damage [OWASP 2013d]. Consequently, as it will be presented on section 3.6, these are the vulnerabilities that characterize the focus of this dissertation. Other types of vulnerabilities, such as broken authentication and missing function level access

control, are more complex to be detected by automated solutions, because each program can implement different types of authentication and access control, e.g. using sessions, cookies, URL rewriting, and so forth.

### **1. (SQL/Command) Injection**

Injection flaws, such as SQL (Structured Query Language), OS (operating system), and LDAP (lightweight directory access protocol) injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization [OWASP 2013b]. Authorization is the process of giving someone permission to do or have something [Tsipenyuk et al. 2005].

### **2. Broken Authentication and Session Management**

Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities [OWASP 2013c]. Authentication is the process by which a system verifies the identity of a user who wishes to access it [Tsipenyuk et al. 2005].

### **3. Cross-Site Scripting (XSS)**

XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. XSS allows attackers to execute scripts in the victim's browser, which can steal user sessions, deface web sites, or redirect the user to malicious sites [OWASP 2013a].

### **4. Insecure Direct Object References**

A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other form of protection, attackers can manipulate these references to access unauthorized data [OWASP 2013d]. Access control is a mechanism by which a system grants or revokes the right to access some data or perform some action [Tsipenyuk et al. 2005].

### **5. Security Misconfiguration**

Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, in the context of the security of the source code, sensitive information (e.g. passwords, personal information etc.) should never be stored in plain text or hard-coded into the source code. They should be stored encrypted [OWASP 2013e].

<p><b>6. Sensitive Data Exposure</b></p> <p>Many web applications do not properly protect sensitive data, such as credit cards, tax IDs, and authentication credentials. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser [OWASP 2013d].</p>
<p><b>7. Missing Function Level Access Control</b></p> <p>Most web applications verify function level access rights before making that functionality visible in the UI. However, applications need to perform the same access control checks on the server when each function is accessed. If requests are not verified, attackers will be able to forge requests in order to access functionality without proper authorization [OWASP 2013d].</p>
<p><b>8. Cross-Site Request Forgery (CSRF)</b></p> <p>A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim [OWASP 2013d].</p>
<p><b>9. Using Known Vulnerable Components</b></p> <p>Components, such as libraries, frameworks, and other software modules, usually run with full privileges. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications using components with known vulnerabilities may undermine application defenses and enable a range of possible attacks and impacts [OWASP 2013d].</p>
<p><b>10. Unvalidated Redirects and Forwards</b></p> <p>Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to malware sites, or use forwards to access unauthorized pages [OWASP 2013f]. A malware site is a site intended to cause damage of any of its visitors [Tsipenyuk et al. 2005].</p>

Table 1 Brief description of the top 10 most found security vulnerabilities.

### 2.3.

#### Detection of Security Vulnerabilities

Security vulnerabilities and their consequences to software projects are not new topics [Telang and Wattal 2005][Blyth 2004][Howard et al. 2009]. However, in recent years, they are significantly attracting the growing interest of academic and commercial communities [Nadeem et al. 2012][Xie et al. 2011][Willis et al. 2006]. One of the key reasons for this change is the increasing number of

vulnerabilities being discovered and exploited on software programs from small to big companies, such as: Oracle, Microsoft, Apple and others [Telang and Wattal 2005]. There are three main different approaches to find security vulnerabilities in a software program [Chess and McGraw 2004]: manual inspection, dynamic analysis and static analysis.

These three approaches have been the main subject of several studies [Meier et al. 2005][Kupsch and Miller 2009][Artho and Biere 2005]. These studies have helped us understand that there is no automated solution (based on either dynamic or static analysis) able to fully replace manual inspection. On the other hand, automated solutions are cheaper to be executed and can cover more ground in less time than manual inspection. The next sections discuss the advantages and disadvantages of each of these approaches. We also discuss their applicability for early detection as well as their accuracy in the detection of security vulnerabilities.

### **2.3.1. Manual Inspection**

Manual inspection consists of developers themselves or security specialists carefully reviewing the source code [Kupsch and Miller 2009] in order to find security vulnerabilities. They usually rely on a checklist (as presented in Table 2) to perform this inspection [Meier et al. 2005]. The checklists are available either in specialized books or on-line publications. For instance, Microsoft has created and released its full list of items that its developers should check when performing manual inspection [Meier et al. 2005]. For the sake of illustration and brevity, Table 2 shows two examples of these items in Microsoft's checklist. The first column is the name of the item that is going to be verified. The second column is the corresponding action that developers should take in order to validate if the source code is secure. The full list containing all the items is available from Microsoft's website [Meier et al. 2005].

Check item	What to look for in code
Input/data validation	Look for client-side validation that is not backed by server-side validation, poor validation techniques, and reliance on file names or other insecure mechanisms to make security decisions.
Hard-coded secrets	Look for hard-coded secrets in code by looking for variable names such as "key", "password", "pwd", "secret", "hash", and "salt".

Table 2 Example of a simple checklist to perform manual inspection.

One of the major advantages of manual inspection is the fact that the vulnerability analysis is performed by human beings. Therefore, if the person who is performing the inspection finds something that is not in her/his checklist, but it looks as a suspicious code, she/he is able to investigate the code and discover if it is indeed a vulnerability or something else. For instance, the second item presented on Table 2, informs the code reviewer that she/he needs to look for variable names as “password” or “secret”. However, if she/he finds a variable with the name “MyCredentialsInfo”, the reviewer is still able to understand that although the name is syntactically different from the examples in the checklist, it falls into the same category. This flexible analytical behavior cannot be fully automated.

On the other hand, manual inspection is hard to be successfully performed in programs of reasonable size. Each program under analysis can have hundreds or millions of lines of code and each program unit can contain multiple security vulnerabilities. Therefore, this kind of inspection is an error prone and daunting task [Kupsch and Miller 2009]. Thus, developers can fail to identify one or more of these security vulnerabilities, leaving the software vulnerable to attacks. In fact, the high number of security vulnerabilities is not feasible to be identified and tackled even by experienced programmers of secure software [Kupsch and Miller 2009][Meier et al. 2005].

Figure 5 illustrates how a few code elements – i.e. a method and its accessed attributes – are the source of several instances of security vulnerabilities. The code snippet in Figure 5 contains six vulnerabilities. Each of them is briefly explained below. We also discuss why it might not be trivial to detect each of them with manual inspection.

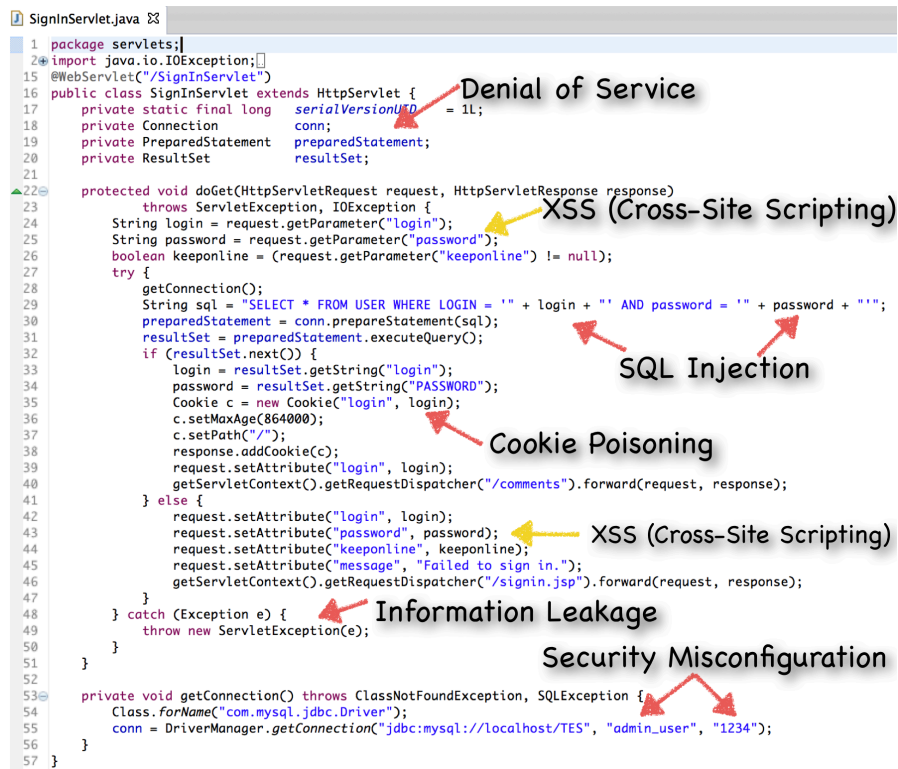


Figure 5 Sample of several security vulnerabilities in a small piece of code.

The first vulnerability in the code of Figure 5 represents a denial of service (DoS). DoS is the act of performing an attack, which prevents the system from providing services to legitimate users [Kessler and Levine 2009]. This vulnerability is a typical type of vulnerability that is hard to be detected even by experienced programmers. The difficulty results from the fact the person, who is reviewing the source code, has to mentally track all variables that were initialized and were not released. Additionally, she/he needs to verify if variables used within loop statements (e.g. *for* and *while* blocks) are receiving their content from user's input. The problem is that users (attackers) could provide values that could make the loop execute forever. For instance, consider the three variables of type *Connection*, *PreparedStatement* and *ResultSet* declared on lines 18-20 of Figure 5. These variable declarations individually require a high amount of resources (memory) from the server. Therefore, it is important that each one of these resources must be released so they can be returned to the server and reused by other users of the application. A non-well-intentioned user could send thousands or even millions of requests to this page, eventually making the server run out of resources. Then, this server would stop serving the legitimate users, whereby causing a denial of service. In the example of Figure 5, the code was small and



simple. However, each class in real life projects can have hundreds of lines of code, making it complicated or impeditive to track where these variables were created and where they were released (if they were) through manual inspection.

The second case of vulnerability in Figure 5 is the occurrence of cross-site scripting (XSS). The lines 24-25 are storing some inputs submitted by the users and sending them on lines 42-43 back to the browser without performing any type of sanitization on them [OWASP 2013a]. An attacker could submit malicious content, such as e.g. `<script>alert(1);</script>`. When this content is executed, it could steal information from other users. According to the OWASP Top 10 report [OWASP 2013a], much more damage can be caused by XSS occurrences. In principle, the line 26 seems to have the same problem as lines 24-25, what could induce a developer to think that this statement might also cause a XSS vulnerability. However, because the only two possible values of this variable are *FALSE* or *TRUE*, even if a malicious code was submitted, it could not cause any harm. Thus, this line can be considered secure. For the sake of brevity, the code snippet above was created containing XSS vulnerability in the same class and same method. However, XSS is also a complicated vulnerability to be found by manual inspection in real programs. There might be a significant “distance” in the program between: (i) the places (e.g. classes) where the variables receive user-provided content, and (ii) the places where this content is sent back to the browser. Additionally, several intermediate classes could be intertwined in this path, thus making the tracking even more difficult.

The third existing vulnerability is a SQL injection. Line 29 is in charge of creating a dynamically composed SQL statement from concatenating values submitted by the user with a pre-defined SQL query. This is an opportunity for a not well-intentioned user to submit malicious data, such as `' or '1' = '1'` and gain control of all the data in the database. Again, lines 30-31 can make the programmer think she/he is safe because the code is using a *PreparedStatement object*. However, as it will be explained on section 3.6.10, if not properly used, the simple fact of having it does not make the source code safe from SQL Injection. What makes SQL injection hard to be detected by manual inspection is the fact that if the code reviewer, who is performing the inspection, might not be fully familiarized with all types of SQL Injection (e.g. error-based, boolean and blind)

[Halfond et al. 2008]. She/he could mistakenly believe the program is secure because the code is not concatenating any strings.

The forth vulnerability is a cookie poisoning. Lines 35-38 create a cookie that is sent to the browser. However, there are at least two problems in this case. The first one is the fact that sensitive information is being added as plain text, which is not recommended. The reason is that the user (or anyone), who is sniffing this connection, is able to read this content with no problem. Sniffing the connection means to capture any data that is being transmitted over a network. Therefore, some sort of encryption should be used. The second problem is the fact that the developer did not set the *HttpOnly* [OWASP 2013g] attribute. If the browser supports it (in fact, most of modern browsers do), it will not allow the user to change the value of this cookie. In the current example, it is added the type of the user into the cookie. However, if the user changes this value from *User* to *Admin*, then the application will accept the next requests of this user as *admin* requests. A third and possible problem is the fact that the developer did not set the *Secure* [OWASP 2013h] attribute. Again, if the browser supports it (as most of modern browsers do), it will only send cookies via secure connection (HTTPS), thus not allowing attackers to sniff the connection and obtain the content inside the cookie. However, as not all web applications use HTTPS, it may count as a problem. Nevertheless, it is important developers know and consider this possibility. Developers usually store all sorts of information in cookies. Because cookies are easy to be manipulated, they can exchange information from the server to the browser and vice-versa. The difficulty of finding a vulnerability in this process lies on the fact that the use of cookies is usually intertwined among several other program statements. These other statements are more important to the functionality being implemented, which ends up hiding the importance of cookies (and their harmfulness). Thus, developers or reviewers may not feel the need to not pay attention on them.

The fifth vulnerability is an information leakage. In case an exception is thrown due to some error detected during the program execution, the *try-catch* block will catch it. However, the line 49 re-throws the error up in the call chain, thus eventually being displayed to the end user. The problem with this undisciplined exception propagation is the nature and amount of information being displayed to the user. This problem can be seen in Figure 6. The printed

error message contains the web server's name, the server version, the database's name, the full stack trace and more information that end users should not know [OWASP 2013i]. These information items are always the first ones an attacker tries to obtain. The knowledge of the server's name, the server version and which programming language the website is using allows the attacker to search for known vulnerabilities and exploit them. This vulnerability is hard to be detected by manual inspection in programs without a well-structured exception handling policy, which is a very common scenario [Barbosa et al. 2012]. In these programs, it becomes very hard for the code reviewer to inspect all classes and execution paths, in order to know what exceptions are being thrown, which ones are being properly handled and, finally, which ones are not. In medium and large-sized projects, this can be very complicated and time consuming.

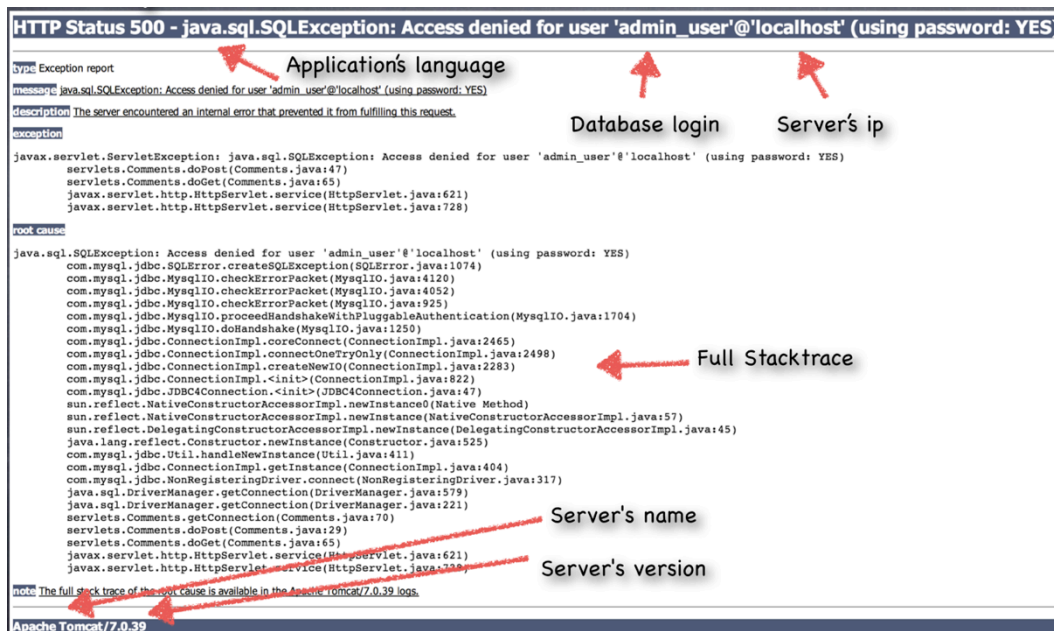


Figure 6 Exception message being displayed to the user.

The sixth and final vulnerability is a security misconfiguration. The line 55 has the database's login and password hard coded, which is not recommended. The problem with this attitude is related to the fact there are projects, such as *Java Decompiler* ["Java Decompiler" [S.d.]], which are able to decompile java byte code, extract the content of classes and reveal this login and password to unauthorized people. This type of information should be encrypted or at least stored in another file [OWASP 2013e]. Sometimes developers are so focused on implementing the requested functionalities that they do not pay attention on basic

things, such as the use of a hard-code password. Maybe that password was only used for testing, but eventually was forgotten in the code and nobody else has ever noticed the need to store the password in another proper location.

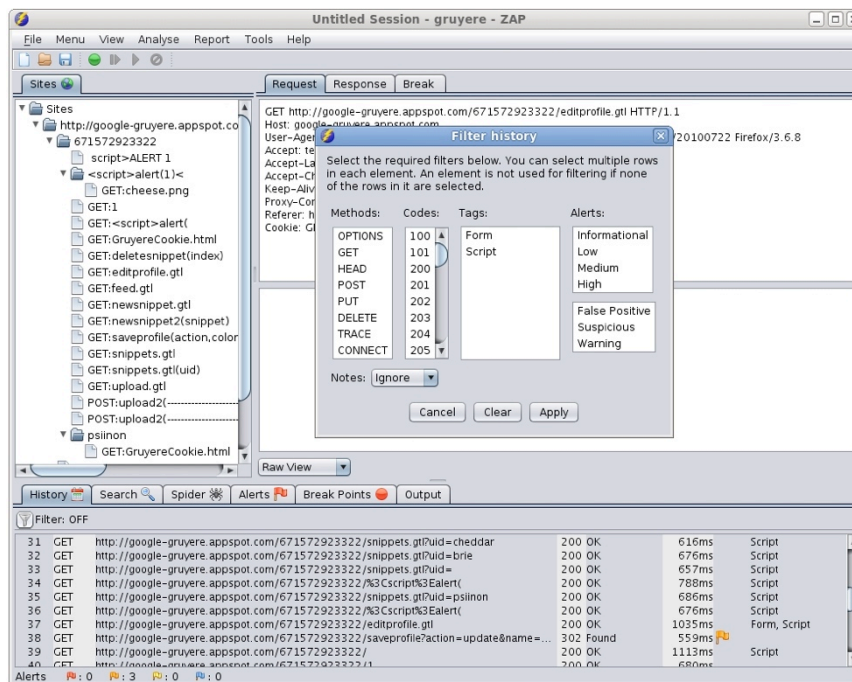
All the aforementioned cases of security vulnerability show how secure programming is difficult to be achieved by relying only on manual inspection. If a simple code snippet with only 57 lines of code (Figure 5) contains six security vulnerabilities, someone can imagine the huge number of vulnerabilities emerging in a project with hundreds or millions of lines of code. Additionally, with developers constantly creating and committing new code statements in a project, an inspection performed a few hours ago could immediately become obsolete. Therefore, manual inspection on medium or large-sized programs can certainly become an error prone and daunting task. Automated support could ameliorate the aforementioned limitations in order to promote secure programming.

### **2.3.2. Dynamic Analysis**

Another approach to find security vulnerabilities is based on dynamic analysis or simply DA. Dynamic analysis is a technique that executes the application and tries to find problems by submitting malicious inputs and checking the returned results [Artho and Biere 2005]. In other words, it analyzes the program behavior during runtime. An advantage of this technique over static analysis (section 2.3.3) is the fact that DA is able to identify vulnerabilities independent on how the source code was implemented. For instance, developers can implement an authentication mechanism on several different ways, e.g. using cookies, sessions and so forth. However, because DA submits the malicious content to the running webpage, it does not need to know how the page was implemented. Additionally, once dynamic analysis finds problems based on the returned results, the rate of false positives is close to zero [Artho and Biere 2005]. Some examples of existing solutions that perform dynamic analysis on web application are ZAP [Bennetts 2012] and Burp [“Burp Suite” [S.d.]]. The features of such dynamic analysis tools for vulnerability detection are very similar. ZAP is used here to illustrate the features of dynamic analysis tools. We selected this tool as it was created and released by the OWASP initiative [Bennetts 2012] (section

2.2). Figure 7 presents a screenshot from the ZAP tool. In order to run the tool, it is necessary to: (i) provide the URL of the website that will be tested, and (ii) select which types of vulnerabilities should be verified. Once the *run* button is pressed, the tool starts submitting thousands of requests containing malicious inputs. A log with the returned results is produced. The lower part of Figure 7 shows the returned results from each of these requests. Based on these results, it is possible to detect the presence of security vulnerabilities in the target web application. For instance, it is possible to observe that, if the tool submitted invalid credentials for the login page and the return was “*welcome user admin*”, this page contains a security vulnerability.

Based on this illustration, we can note a few other advantages of using dynamic analysis solutions, such as ZAP. For instance, applications that contain hundreds of URLs can be analyzed for vulnerabilities in a matter of seconds. In case vulnerabilities are discovered, developers can fix them and execute the process repeatedly. On the other hand, a disadvantage is the fact that DA requires fully-implemented programs in order to test the web pages. Therefore, as already mentioned (section 1.1), if a vulnerability is only detected afterwards, the effort and cost to fix it might be much higher. Additionally, DA usually results in a high rate of false negatives, because the DA tool can only analyze the web pages which it has access to. Finally, DA does not inform the exact location of the vulnerability in the source code. It only informs which vulnerabilities were exploited, leaving developers responsible to find each of the program statements contributing to the security vulnerability.

Figure 7 ZAP's screenshot<sup>3</sup>.

### 2.3.3. Static Analysis

Finally, the last approach for security vulnerability is static analysis or simply SA. SA analyzes the application by examining the source code without executing it [Artho and Biere 2005]. It scans the source code to either compute source code measures or search for known code structure patterns. When the measures are computed or patterns are identified, a report is presented. In the context of this dissertation, this report provides vulnerability warnings and let developers know, among other things, the exact location of each vulnerability candidate.

One of the biggest advantages of this technique is the fact that it does not require fully implemented programs. In other words, it can be executed since the initial programming stages of a software system, even on unfinished program modules. Consequently, SA is the technique selected by most of the existing tools that perform verification on source code to detect security vulnerabilities. Static

<sup>3</sup><https://www.owasp.org/index.php/File:ZAP-ScreenShotHistoryFilter.png>

analysis was also the technique chosen to be used on our proposed solution (section 4).

We can mention several examples of SA tools for vulnerability detection in source code, such as ASIDE [Zhu 2012], Laspe+ [Livshits 2006], CodePro Analytics [Google 2001], Fortify [HP 2002], SSVChecker [Dehlinger et al. 2006] and dozens more. Figure 8 is a screenshot from one of these tools, called Lapse+ [Livshits 2006]. The figure presents the security vulnerabilities found in the source code of the analyzed project. Developers are presented with detailed information about each vulnerability, including the code element (e.g. variable, method or statement) that caused the vulnerability, the type of vulnerability (section 2.2), the project name, the file name and line of code where the vulnerability was found. All these information entries help developers verify if indeed this is a problem that needs to be resolved or it is a false warning yielded by the detection tool.

Suspicious call	Method	Category	Project	File	Line
! ((java.sql.Statement)selectStatement).executeQuery(sel)	java.sql.Statement.execute	SQL Injection	testVulnerabilities	Test4.java	98
! connection.prepareStatement(sel)	java.sql.Connection.prepare	SQL Injection	testVulnerabilities	Test4.java	78
! connection.prepareCall(var1.toString())	java.sql.Connection.prepare	SQL Injection	testVulnerabilities	Test4.java	86
! connection.prepareCall(var1.substring(2))	java.sql.Connection.prepare	SQL Injection	testVulnerabilities	Test4.java	88
! response.sendRedirect(data.substring(data.indexOf(" ") + 1))	javax.servlet.http.HttpServletResponse.sendRedirect	HTTP Response Splitting	testVulnerabilities	Servlet.java	108
! sos.println("Exception Message -> " + e.getMessage())	javax.servlet.ServletOutputStream.println	Cross-site Scripting	testVulnerabilities	PrintTestServlet4.java	144
! response.getWriter().print(data)	java.io.PrintWriter.print	Cross-site Scripting	testVulnerabilities	Servlet.java	143
! out.print(dataString)	java.io.PrintWriter.print	Cross-site Scripting	testVulnerabilities	Test4.java	171
! rt.exec("doStuff.exe" + " " + myUid)	java.lang.Runtime.exec	Command Injection	testVulnerabilities	Test4.java	26
! response1.sendRedirect("/direccionpablo/error.jsp?error=" + (Stri	javax.servlet.http.HttpServletResponse.sendRedirect	HTTP Response Splitting	libretaDirecciones	AddContact.java	110
! response1.sendRedirect("/direccionpablo/error.jsp?error=" + (Stri	javax.servlet.http.HttpServletResponse.sendRedirect	HTTP Response Splitting	libretaDirecciones	AddContact.java	113
! response.sendRedirect("/addressbook/error.jsp?error=" + URLEnc	javax.servlet.http.HttpServletResponse.sendRedirect	HTTP Response Splitting	libretaDirecciones	DBConnection.java	68

Figure 8Lapse+ view displaying the detected vulnerabilities.

The main disadvantage of static analysis is the fact that most of existing implementations rely on simple techniques, such as *pattern matching* [Nadeem et al. 2012]. The code fragment below depicts a real example used to illustrate some of the limitation of the pattern matching technique.

```

16 @Override
17 protected void doGet(HttpServletRequest request,
18                      HttpServletResponse response)
19     throws ServletException, IOException {
20     PrintWriter printWriter = response.getWriter();
21
22     printWriter.print("a");
23     printWriter.print(("b"));
24
25     String d = "d";
26     printWriter.print((null != "") ? "c" : d);
27
28     printWriter.print(getContent(request));
29     printWriter.print(Boolean.parseBoolean(request.getParameter("bad")));
30 }
31
32 private String getContent(HttpServletRequest request) {
33     int i = 5;
34     if (i > 10) {
35         return request.getParameter("bad");
36     } else if (i == 5) {
37         String bad = request.getParameter("bad");
38
39         return bad;
40     }
41
42     return "ok";
43 }

```

Figure 9 Examples of pattern matching limitations.

One of the main problems of *pattern matching* is the fact that it only compares the code that is being analyzed, against a code template (examples of code templates are presented in sections 3.2, 3.3 and 3.4) that usually represents a security vulnerability. However, this is not enough to confirm that it is or not a vulnerable code. Other aspects, such as the application's context (variables and methods) are not taken into consideration when searching for vulnerabilities. In Figure 9, the method *print* from class *PrintWriter* appears five times (lines 22, 23, 26, 28 and 29). For now, it is enough to know that this method should have its parameters investigated for untrusted values.

Applications based on *pattern matching*, such as ASIDE and Lapse+ consider line 22 secure, because the parameter is an element of type *string literal* from the AST (Abstract Syntax Tree) [Kuhn and Olivier 2006] created and hardcoded into the code by the developer. Therefore, considered trusted data. However, line 23 as depicted by the left icon (*red devil*), is reported as vulnerable to cross-site scripting (XSS) [OWASP 2013a] because the parameter is something that is not in their *knowledge base*. The element is of type *parenthesized expression*. If this line is manually analyzed, it is possible to notice it does not contain any vulnerable code. It is printing the same thing as the previous line simply in a different way.

Line 26 is also marked as vulnerable by both tools. This time the parameter is of type *conditional expression*. Nevertheless, the only two possible outputs

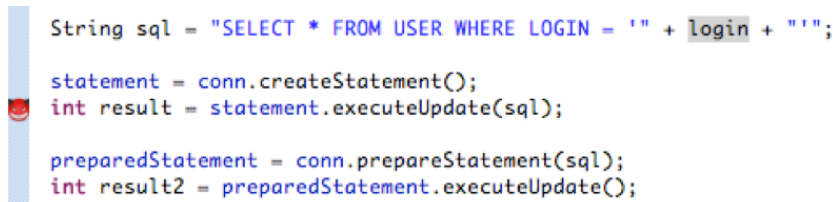


from this condition are a *string literal* “c” or another *string literal* “d”. Once more, not really a vulnerability. In line 28, the parameter is of type *method invocation*. It has three possible outputs; lines 35 and 39 return values considered untrusted and 42 return a trusted value. Based on this, line 28 should be flagged as vulnerable, because it has at least one possible untrusted output. Both tools correctly flag it. They do it not because they were able to identify the untrusted output, but because they flag everything that they do not understand. Finally, line 29 is also flagged as vulnerable. However, the only possible output is a *Boolean* (true or false) value even if the user provides malicious content. The final score in this simple code fragment is two (lines 22 and 28) out of five (lines 22, 23, 26, 28 and 29), thus making *pattern matching* not entirely accurate and trustworthy. Consequently, they discourage many programmers of using them (section 1.2). The concepts of false positive and false negative are presented in the next section.

## 2.4. False Positives and False Negatives

No matter which of the aforementioned techniques for vulnerability detection is used, there will always be the possibility of a false positive or false negative. False positive is the incorrect indication of the presence of a vulnerability. In other words, a warning that does not actually represent a security vulnerability in the program being analyzed. *False negative* is the failure to recognize an existing vulnerability in a program. There is a security vulnerability in the source code, but it is not revealed by the detection technique employed. The reason for the possibility of having false positives and false negatives is threefold. First, manual inspection is able to find vulnerabilities not specified in an inspection checklist. However, because it is a time consuming task it may lead to false negatives, in case one or more modules are not analyzed. Second, dynamic analysis is able to identified vulnerabilities without knowing how it was implemented. However, it only analyzes functionalities which it has access to, this also may lead to false negatives. Finally, static analysis is able to analyze the source code of unfinished modules. However, if the applied technique has a low accuracy, this may lead to false positives.

Figure 10 presents an example of what was stated above. The code snippet was analyzed by a static analysis tool (ASIDE [Zhu 2012]), which correctly flags line 3. Because the developer concatenated the content of variable *sql* and passed that content to the *statement* object. The statement object is known for not making any type of validation on the receiving input, which is certainly not recommended. However, ASIDE assumes that because the code in line 4 is using a *preparedStatement* object, the code can be considered secure. This is a mistake, as it will be described in section 3.6.10, because this fact alone is not sufficient and the code snippet still has a SQL injection vulnerability in it. In other words, this is a false negative, which can be very harmful. Because it might give the false impression that the application is secure when in reality is not.



```
String sql = "SELECT * FROM USER WHERE LOGIN = '" + login + "'";
statement = conn.createStatement();
int result = statement.executeUpdate(sql);
preparedStatement = conn.prepareStatement(sql);
int result2 = preparedStatement.executeUpdate();
```

Figure 10 ASIDE's false negative of SQL injection.

Therefore, companies and developers should not rely solely on one of these approaches. They should think of them as extra layers of security. For instance, static and dynamic analysis solutions could be used to maximize the detection of vulnerabilities. In addition, security specialists should perform manual inspection in order to detect remaining false negatives. In any case, secure programming should be inserted into the developer's workflow so that developers become aware of vulnerabilities as soon as possible, i.e. when writing their code statements. By doing this, developers can handle the most common vulnerabilities as well as discarding false positives. Then, the most complex vulnerabilities and false negatives could be left for the specialists. However, it is important to come up with a high-accuracy solution for vulnerability detection, i.e. with the lowest possible rate of false positives and false negatives. This goal is particularly important if vulnerability detection is integrated into the developer's workflow. Then, the cost of manual inspection by specialists would be kept at a minimum, thereby further reducing the risk of vulnerabilities prevailing in the source code.

## 2.5. Late Detection and Early Detection

The vast majority of the existing solutions are built in a way they are applied a posteriori, only when developers have already fully produced (and compiled) a method, a class or an entire day or week of coding. This existing approach is known as *late detection* of security vulnerabilities. The late detection workflow is depicted in Figure 11. The workflow is usually as follows. First, developers spend hours, days and even weeks implementing their code. Second, if there is any time left until the deadline, they run the security detection tool, which have to scan all the files in all the selected projects. Finally, as a result, a large report of vulnerabilities is presented to the programmer. The developer's next step is to start fixing the code. Although late detection may impose certain drawbacks (section 1.2), it is something that still brings its benefits. Studies [Baca et al. 2008] state that, on average, 17% of cost savings can still be achieved simply by the fact that a late detection tool is used to find security vulnerabilities.



Figure 11 Late detection workflow.

The later and longer it takes to fix a security vulnerability in a program, the more it costs [Guarnieri et al. 2011]. It also increases the chances of that specific vulnerability being exploited by an attacker. Thus, it is important that developers receive tooling support in order to find and remove vulnerabilities as early as possible from their source code. Developers should be aware of security vulnerabilities when they are adding or editing their code statement. This approach is known as *early detection*. As soon as the developer writes a code that is considered vulnerable, warnings of security vulnerability should be displayed. The early detection workflow is depicted in Figure 12. This time, the verification

happens in the source code as new statements are added. As a result, a report containing a few vulnerabilities is presented. Thus, developers have the chance to consider and remove them as they appear, i.e. while their minds are focused in the current context.

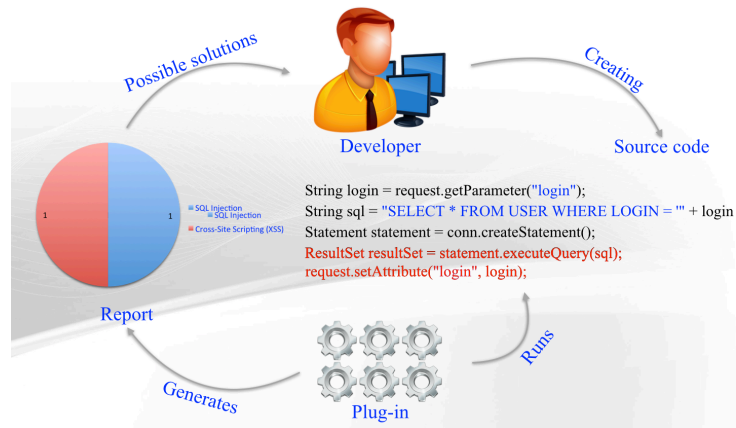


Figure 12 Early detection workflow.

By using the approach of early detection, developers will no longer have to waste time redoing work that could have been done potentially quicker and once. Early detection of vulnerabilities has the goal of constantly supporting developers on the task of secure programming. Developers are constantly reminded about security in the implementation of their program statements. Warnings are displayed to let developers know there is something wrong that requires their attention. However, when these warnings are presented to developers, they should investigate the problem and then fix it or simply ignore it (in case it is a false positive). Thus, if the detection is not accurate enough and generates a high amount of false positives, they may eventually lose interest and even stop using this support. An example is shown in Figure 13.

<pre> 17 @Override 18 protected void doGet(HttpServletRequest request, 19 HttpServletResponse response) 20 throws ServletException, IOException { 21 request.getParameter(""); 22 } 23 } Step 1 </pre>	<pre> 17 @Override 18 protected void doGet(HttpServletRequest request, 19 HttpServletResponse response) 20 throws ServletException, IOException { 21 request.getParameter(""); 22 } 23 } Step 1 </pre>
<pre> 17 @Override 18 protected void doGet(HttpServletRequest request, 19 HttpServletResponse response) 20 throws ServletException, IOException { 21 String bad = request.getParameter("bad"); 22 } 23 } Step 2 </pre>	<pre> 17 @Override 18 protected void doGet(HttpServletRequest request, 19 HttpServletResponse response) 20 throws ServletException, IOException { 21 String bad = request.getParameter("bad"); 22 } 23 } Step 2 </pre>
<pre> 17 @Override 18 protected void doGet(HttpServletRequest request, 19 HttpServletResponse response) 20 throws ServletException, IOException { 21 String bad = request.getParameter("bad"); 22 response.getWriter().print(bad); 23 } 24 } Step 3 </pre>	<pre> 17 @Override 18 protected void doGet(HttpServletRequest request, 19 HttpServletResponse response) 20 throws ServletException, IOException { 21 String bad = request.getParameter("bad"); 22 response.getWriter().print(bad); 23 } 24 } Step 3 </pre>
<pre> 17 @Override 18 protected void doGet(HttpServletRequest request, 19 HttpServletResponse response) 20 throws ServletException, IOException { 21 String bad = request.getParameter("bad"); 22 String safe = ESAPI.encoder().encodeForHTML(bad); 23 response.getWriter().print(safe); 24 response.getWriter().print(ESAPI.encoder().encodeForHTML(bad)); 25 } 26 } Step 4 </pre>	<pre> 17 @Override 18 protected void doGet(HttpServletRequest request, 19 HttpServletResponse response) 20 throws ServletException, IOException { 21 String bad = request.getParameter("bad"); 22 String safe = ESAPI.encoder().encodeForHTML(bad); 23 response.getWriter().print(safe); 24 response.getWriter().print(ESAPI.encoder().encodeForHTML(bad)); 25 } 26 } Step 4 </pre>

Figure 13 Early vulnerability detection from ASIDE and ESVD (section 4).

Figure 13 presents a code snippet that was analyzed by two solutions that perform early vulnerability detection. This example was created in order to illustrate the importance of high accuracy when performing early vulnerability detection. Figure 13 is organized in two parts. On the left side, a code fragment analyzed by ASIDE [Zhu 2012], which was the only tool we found that performs early detection. On the right side, the same code fragment analyzed by ESVD, our prototype (section 4). This figure highlights the problem of facing a high amount of false positives in early detection, it also mimics the steps performed by a developer when creating the code fragment above: (i) Step 1 - line 21, the developer starts writing the statement that will retrieve data sent from users; (ii) Step 2 - line 21, the statement is completed; (iii) Step 3 - line 23, the method *print* is sending the content from variable *bad* back to the user. Therefore, as it will be explained in section 3.6.3, this leaves the application opened to cross-site scripting (XSS) and should be avoided; (iv) Step 4 - lines 24 and 26, the developer fixes the code by using the *encodeForHTML* method, which is a known sanitization method that removes (if any) malicious characters. This method and dozens of other ones are available in the ESAPI [Williams 2010] library, created by OWASP, tested and used by programmers from all over the world. Developers have no need to create sanitization methods on their own and no excuse on why not use them.

The idea of early detection is to help developers avoid adding vulnerable code as soon as possible. However, as shown in step 1, the statement is not even finished and the *pattern-matching* tool is already reporting a security vulnerability. As already explained, as soon as they match a code e.g. *request.getParameter*, that is in their *knowledge base*, they report it as vulnerable. On the right side of the figure, it is possible to observe an error icon. However, this error was generated by Eclipse and not by our prototype. This error is because there is no semicolon “;” in the end of the line, besides that, the line is correct. Step 2 cannot be considered vulnerable yet, because even if variable *bad* receives a malicious content, it is not doing anything with it. ASIDE still flags the line as vulnerable and our prototype does nothing (it considers the line secure). On the other hand, in step 3 it is possible to observe a security vulnerability and in this case, both tools correctly report it. As the developer continues to write the code, in

step 4, he/she fixes the code. However, there are two interesting points that should be mentioned. First, the *pattern-matching* tool is incorrectly reporting variable *safe* (line 25) as vulnerable, because it does not “understand” that the variable has been sanitized in a previous line (23). Second, the tool considers line 26 secure, because the method *encodeForHTML* is registered as a sanitization method in its *knowledge base*. The interesting part is the fact that if we analyze this code, it is possible to notice that they are doing the same thing, just in two different ways. Our prototype can correctly identify these two lines (25 and 26) as secure and remove the warning from the previous step. ASIDE is just one from the several existing solutions we tested, in order to verify how accurate the techniques they use to find security vulnerabilities are. The description and more information about the other tools are presented on the next section.

## 2.6. Related Work

There are dozens of available solutions that are intended to perform detection of security vulnerabilities in the source code[OWASP 2003a]. Some noteworthy examples are SSVChecker [Dehlinger et al. 2006], FindBug [Pugh and Loskutov 2006], ASIDE [Zhu 2012], Lapse+ [Livshits 2006], CodePro Analytics [Google 2001], Fortify HP [HP 2002] and AppScam IBM [IBM 2001]. They can be divided in several different categories. First, the programming languages (Java, C#, PHP, etc.) they give support. Second, the type of vulnerabilities (SQL injection, Cookie Poisoning, etc.) they give support. Third, how the detection is performed (dynamic analysis or static analysis). Fourth, if they are open-source or private. Finally, which technique (pattern matching or data flow analysis) they use to find security vulnerabilities. Therefore, before we even decided to create our own tool, we downloaded and tested several of these tools in order to find the strengths and limitations of each one of them. It is important to mention that we did not test all the existing ones, because there are too many, instead, from the existing list, we tested the most popular ones (based on blogs and forums). The next sub-sections describe our findings on the best three of them.

We noticed that some existing tools that try to find security vulnerabilities, started by finding code anomalies and after achieving good results, migrated to security. However, most of them did not evolve, because in order to find if a method is a long method or has a high level of coupling, all there is to do is count the number of lines from the method and the number of references to other classes. These are simply two examples from code anomalies. However, in the context of security vulnerabilities, the idea of simply counting something does not work. For instance, to report a code as vulnerable to SQL injection, simply the fact that the developer used the *statement* object instead of the *preparedStatement*, does not suffice, because it is necessary to gather more information, such as how the query was created or if the parameters were sanitized. Therefore, the fact that a technique (pattern matching) successfully worked for code anomalies does not mean it will also work on detecting security vulnerabilities.

### **2.6.1. Lapse+**

The first analyzed tool was Lapse+. A free security scanner tool that performs source code static analysis. It was created at Stanford University for detecting vulnerabilities of untrusted data injection in Java EE applications [Livshits 2006]. The advantage of this plugin is the fact that it is able to identify a well-defined set of security vulnerabilities, including, but not limited to, Cookie Poisoning [OWASP 2013f], SQL injection [OWASP 2013b] and Cross-site Scripting (XSS) [OWASP 2013a]. The disadvantages are: First, it uses pattern matching to search for security vulnerabilities. Second, it performs late detection and every time the tool is executed it scans all files on all opened projects not focusing only on the changes made since the last scan.

### **2.6.2. ASIDE**

The second analyzed tool was ASIDE (Application Security plugin for Integrated Development Environment). ASIDE is a free open source plugin with real time verification (or early detection) created at University of North Carolina. This tool performs early detection and focus mainly on vulnerabilities that stem

from input validation. However, the fact that it uses pattern matching to find vulnerabilities, cause it to have a high rate of false positives. According to the OWASP page about ASIDE<sup>4</sup>, it is still evolving and the last version was released in early 2013.

### 2.6.3. CodePro Analytics

CodePro Analytics also known as CodePro is an Eclipse plugin, able to identify security vulnerabilities and code anomalies. Therefore, instead of having to install multiple tools, developers have the option to choose what they want to search for within the context of the same tool. This was the only plugin we found that did not use *pattern matching*. It uses *data flow analysis* with context-insensitivity (see section 3.1). This means that it is able to correctly identify vulnerabilities in Figure 16, but not the ones in Figure 17. This was the only tool that presented different results (amount of warnings and even different warnings) from one execution to another. In other words, running the tool on the same source code two or more times could result in different warnings. As Lapse+, this tool also performs late detection.

## 2.7. Lack of Knowledge on Secure Programming

The later and longer it takes to fix a security vulnerability in a program, the more it costs [Guarnieri et al. 2011]. Thus, it is important that developers receive the necessary support to perform secure programming as early during development as possible. We performed an experiment where 07 participants were asked to review a source code containing 37 security vulnerabilities. From those participants, only two participants were able to find 11 and 15 vulnerabilities, respectively. These two participants were much more experienced as they periodically receive security training at their work place. The other participants found 0,1 or 2 vulnerabilities. There was no single type of security vulnerability that was consistently detected by all or most of the novice or experienced

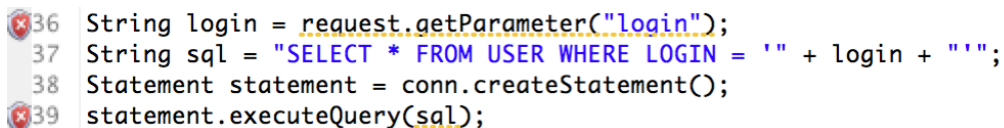
---

<sup>4</sup>[https://www.owasp.org/index.php/OWASP\\_ASIDE\\_Project](https://www.owasp.org/index.php/OWASP_ASIDE_Project)



developers. This corroborates with our claim that developers without training and even the ones who have received training would still benefit from a tooling support.

Figure 14 shows a code snippet with an example of SQL injection vulnerability. The code in Figure 15 seems to be almost the same as the one shown in Figure 14. However, it does not have the vulnerability observed in Figure 14. The main difference between them is the fact that Figure 15 is using a *PreparedStatement* object, which sanitizes the value submitted by the user. We believe that professional developers should be able to notice this problem on Figure 14 and remove it from the source code as in Figure 15. In fact, SQL injection is one of the most common security vulnerabilities. SQL injection is also a type of vulnerability that, in theory, is easy to spot. It can be noticed by analyzing just a few code statements. However, this was not the case in our experiment. Most of the participants did not notice this problem in the source code.

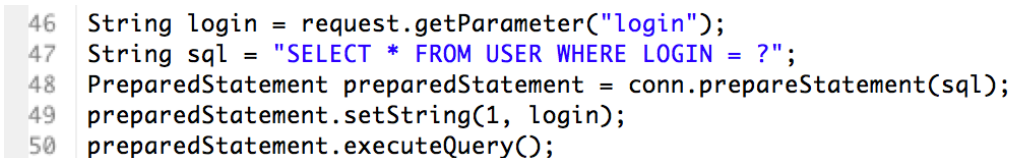


```

36 String login = request.getParameter("login");
37 String sql = "SELECT * FROM USER WHERE LOGIN = '" + login + "'";
38 Statement statement = conn.createStatement();
39 statement.executeQuery(sql);

```

Figure 14 Code snippet with SQL injection vulnerability.



```

46 String login = request.getParameter("login");
47 String sql = "SELECT * FROM USER WHERE LOGIN = ?";
48 PreparedStatement preparedStatement = conn.prepareStatement(sql);
49 preparedStatement.setString(1, login);
50 preparedStatement.executeQuery();

```

Figure 15 Code snippet without SQL injection vulnerability.

Most developers do not have the necessary knowledge to find nor to fix security vulnerabilities on their own. Even if they have knowledge, it is a daunting and error prone task. One of the main reasons for this lack of knowledge is because secure programming is almost never present in the curricula of computer science courses, and even when it is, it is usually introduced as an isolate and theory-based discipline [Lester and Jamerson 2009]. Some of these courses are of theoretical nature and may be lacking tooling support to improve student's learning. Students and developers might benefit from learning in the context of their source code which and why they should reason about vulnerabilities. Proper

assistance for supporting secure programming should be neatly integrated in the software-programming environment used by students and developers.

### 3 Data-Flow-Driven Heuristics for Vulnerability Detection

Data flow analysis hereinafter referred to simply as DFA for brevity, has the ability to follow the path of an object until its origins or to paths where it had its content changed [Hammer et al. 2006]. DFA is commonly used for optimizations on compilers [Hammer et al. 2006], because of its ability of inter and intra-procedural inspection. Our idea was to use this ability to find security vulnerabilities. For instance, if a method receives a variable as a parameter, DFA is able to trace all possible paths of this variable in order to try to identify if there is a path where this variable received untrusted data. If and only if a vulnerable path is found, the variable is flagged as vulnerable. However, in order to know what to search for and what is a *vulnerable path*, we need to provide our heuristics to the modified (to find security vulnerabilities) DFA algorithm. Our heuristics are composed of three elements, i.e. the lists of *entry-points*, *exit-points* and *sanitization-points*. They are presented in sections 3.2, 3.3 and 3.4, respectively.

Figure 16 has the same code fragment as Figure 9 with the addition of arrows indicating the data flow of the parameters being used by the *print* methods in lines 22, 23, 26, 28 and 29. The use of the DFA approach would correctly report lines 28, 35 and 37 as vulnerable as shown by the “red shield” on the left of the editor and all other lines as secure. Because DFA does not only search for methods known for being insecure, rather it searches for possible paths where these methods’ outputs (*entry-point*) reach methods that send data externally to the application (*exit-point*), without passing through any sort of validation (*sanitization-point*). Even though this use of DFA is a more complex and time-consuming task, it can drastically decrease the rate of false positives, as it will be described in the next sections. In our technique evaluation (section 5.1.5.2), we will assess whether the use of DFA significantly impacts the detection performance. The following sections present relevant concepts before the presentation of our algorithm.

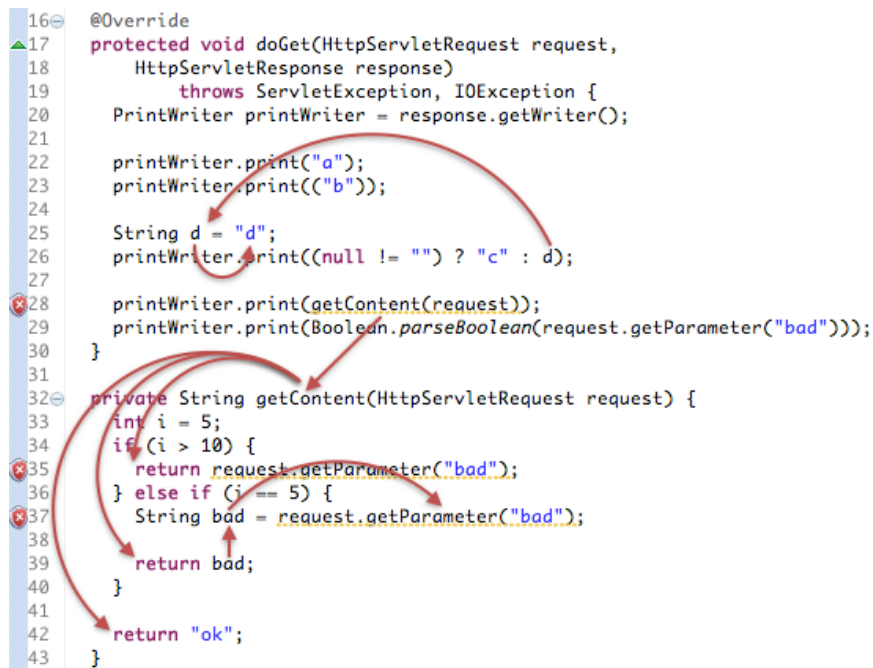


Figure 16 Data Flow Analysis representation.

### 3.1. Data Flow Analysis with Context-Sensitivity

DFA can be implemented in two different ways, namely context-insensitive and context-sensitive. When implemented with context-insensitivity, it means that every instance of a class share the same context (variable, methods, etc.). In other words, it does not differentiate one instance from another and the result from the analysis of one will be repeated to others. On the other hand, when implemented with context-sensitivity, it means that it creates a context for every instance of a class. Consider for instance the following two figures with the same simplified code fragment. When the source code is being verified, every time there is a method invocation e.g. line 59 or a new object is created e.g. lines 57 and 61, a new context is created with a copy of all the fields and methods of that class. This context handling is necessary to make sure, if one instance receives vulnerable content, it will not affect other instances of the same class. This improves the accuracy of reported warnings but also increases the amount of memory used during scan [Lhoták and Hendren 2006]. From the existing solutions for vulnerability we analyzed (section 2.6), CodePro Analytics [Google 2001] was the only one to perform some sort of data flow analysis. In our experiments (see section 5.1.5) CodePro Analytics was, in fact, able to achieve a lower rate of false

positives when compared to tools relying on pattern matching. However, the DFA of CodePro Analytics is context-insensitive, while our proposed approach relies on DFA with context-sensitivity.

Our expectation is that our approach will yield higher accuracy and, therefore, be more appropriate to be used in the context of early vulnerability detection. We present here a comparative example to illustrate why context-sensitivity can be a better choice than context-insensitivity for vulnerability detection. The code snippets in Figure 17 and Figure 18 were specifically designed to demonstrate one of the limitations of data flow using context-insensitive analysis. They consist of the same source code, excluding lines 58 and 62, which are swapped in each image. Figure 17 depicts vulnerabilities reported by a tool (i.e. CodePro Analytics) that uses context-insensitive data flow analysis and Figure 18 by a tool (our prototype) that uses context-sensitive data flow analysis. On the left side, the object *animal1* created in line 57 is receiving the content of variable *ok* in line 59. This content is not vulnerable because it is simply a *string literal*. On the other hand, the object *animal2* created in line 61 is receiving the content of variable *bad* in line 63. This content might be vulnerable because it has not been sanitized. When this code is processed by data flow analysis, using context-insensitivity, the results will be either two warnings or nothing will be flagged at all. However, if this code is manually analyzed, it is possible to notice that *animal1* has not received vulnerable content but *animal2* has. The opposite happens on the right side of the image, where *animal1* receives vulnerable content and *animal2* does not. Figure 18 depicts the expected results from this code fragment, which is *animal1* from the left side and *animal2* from to right side being considered secure and *animal2* from the left side and *animal1* from to right side being considered vulnerable. Our prototype using context-sensitivity is able to correctly differentiate these two instances from each other, and present only the expected security vulnerabilities.

```

51 @Override
52 protected void doGet(HttpServletRequest request,
53     HttpServletResponse response)
54     throws ServletException, IOException {
55     PrintWriter printWriter = response.getWriter();
56
57     Animal animal1 = new Animal();
58     String ok = "ok";
59     animal1.setName(ok);
60
61     Animal animal2 = new Animal();
62     String bad = request.getParameter("bad");
63     animal2.setName(bad);
64
65     printWriter.print(animal1.getName());
66     printWriter.print(animal2.getName());
67 }

```

```

51 @Override
52 protected void doGet(HttpServletRequest request,
53     HttpServletResponse response)
54     throws ServletException, IOException {
55     PrintWriter printWriter = response.getWriter();
56
57     Animal animal1 = new Animal();
58     String bad = request.getParameter("bad");
59     animal1.setName(bad);
60
61     Animal animal2 = new Animal();
62     String ok = "ok";
63     animal2.setName(ok);
64
65     printWriter.print(animal1.getName());
66     printWriter.print(animal2.getName());
67 }

```

Figure 17 Data Flow - Context Insensitive - CodePro Analytics [Google 2001].

```

51 @Override
52 protected void doGet(HttpServletRequest request,
53     HttpServletResponse response)
54     throws ServletException, IOException {
55     PrintWriter printWriter = response.getWriter();
56
57     Animal animal1 = new Animal();
58     String ok = "ok";
59     animal1.setName(ok);
60
61     Animal animal2 = new Animal();
62     String bad = request.getParameter("bad");
63     animal2.setName(bad);
64
65     printWriter.print(animal1.getName());
66     printWriter.print(animal2.getName());
67 }

```

```

51 @Override
52 protected void doGet(HttpServletRequest request,
53     HttpServletResponse response)
54     throws ServletException, IOException {
55     PrintWriter printWriter = response.getWriter();
56
57     Animal animal1 = new Animal();
58     String bad = request.getParameter("bad");
59     animal1.setName(bad);
60
61     Animal animal2 = new Animal();
62     String ok = "ok";
63     animal2.setName(ok);
64
65     printWriter.print(animal1.getName());
66     printWriter.print(animal2.getName());
67 }

```

Figure 18 Data Flow - Context Sensitive - ESVD.

### 3.2. Entry-Point

An *entry-point* also referred as *source* [Tripp et al. 2009][Livshits and Lam 2005], is a point in the source code where external and untrusted input enters the application. Additionally, they are not specific to a particular vulnerability. Figure 19 has a variable *login* (in line 20) receiving its content from the method *getParameter*. This method is considered an *entry-point* because it brings data from outside (data inputted by a user in a web browser) of the application into inside to be processed. When the developer created this code, she/he was expecting that users would only provide valid content related to their credentials (login and password). However, if a user provides malicious content, such as `<script>doBadThings();</script>`, and the application uses it without any sort of sanitization, the application might be exploited and damaged.

```

15 @Override
16 protected void doGet(HttpServletRequest request,
17                      HttpServletResponse response)
18                      throws ServletException, IOException {
19
20     String login = request.getParameter("login");
21     request.setAttribute("login", login);
22 }

```

Figure 19 Example of Entry-Point.

Before trying to create our own list of entry-points, a complete search in the literature was performed. However, only a few websites [“Searching for Code in J2EE/Java” 2010][“Secure Coding Guidelines for Java SE” 2014] were found and even those ones, contained only a small list of methods, which we knew was not complete. Therefore, our solution was to try to find and download the source code of dozens [OWASP 2003a] of open source tools and analyze which methods they considered as entry-points. The next step was to perform a compilation of all those methods. After that, the final list was submitted to the approval by the OWASP committee. However, the result has yet to be returned. For the sake of brevity, Table 3 presents just a few examples of the methods that are in our list of entry-points. For the context of Java programming language, we have identified 75 *entry-points*, the full list is available in our study website [Sampaio 2014a].

Qualified Name	Method Name	Parameters
javax.servlet.ServletRequest	getParameter	java.lang.String
javax.servlet.ServletRequest	getAttributeNames	-
javax.servlet.http.HttpServletRequest	getQueryString	-
javax.servlet.ServletConfig	getInitParameter	java.lang.String

Table 3 List of Entry-Points.

### 3.3. Exit-Point

An *exit-point* also referred as *sink* [Tripp et al. 2009][Livshits and Lam 2005], is a point in the source code where untrusted output goes out of the boundaries of the application. Additionally, they are specific to a particular vulnerability. Figure 20 depicts in line 21 that the developer is sending the content of the variable *login* back to the browser, using the method *setAttribute*. However,

the content is unknown and has not been sanitized. This leaves the application opened to cross-site scripting vulnerability.

```

15 @Override
16 protected void doGet(HttpServletRequest request,
17                      HttpServletResponse response)
18                      throws ServletException, IOException {
19
20     String login = request.getParameter("login");
21     request.setAttribute("login", login);
22 }

```

Figure 20 Example of Exit-Point.

The same process that was performed for the creation of the entry-point list was executed to create the list of exit-points. For the sake of brevity, Table 4 presents just a few examples of the methods that are in our list of exit-points. For Java, we have identified 141 *exit-points*, the full list is available in our study website [Sampaio 2014a].

SQL Injection		
Qualified Name	Method Name	Parameters
java.sql.(Prepared)?Statement	executeQuery	java.lang.String
Cross-Site Scripting		
Qualified Name	Method Name	Parameters
javax.servlet.HttpServletRequest	setAttribute	java.lang.String java.lang.Object
Cookie Poisoning		
Qualified Name	Method Name	Parameters
javax.servlet.http.Cookie	setValue	java.lang.String

Table 4 List of Exit-Points.

### 3.4. Sanitization-Point

A *sanitization-point* also referred as *sanitizer* [Tripp et al. 2009][Livshits and Lam 2005], is a point in the source code where a method or class receives an untrusted input and returns it as a trusted output. Figure 21, depicts in line 23, that the untrusted variable *unsafeLogin* is being passed into the method *encodeForHTML* that converts HTML characters that otherwise would be



considered as code, into a format that makes it become pure data. e.g. the character “<” is replaced by “&lt;” and “>” is replaced by “&gt;”. By doing these replacements, the content of variable *safeLogin* can now be safely sent back to the browser and displayed to the user. There are some special cases that more actions are required. However, for most cases, this is sufficient.

```

17 @Override
18 protected void doGet(HttpServletRequest request,
19                      HttpServletResponse response)
20                      throws ServletException, IOException {
21
22     String unsafeLogin = request.getParameter("login");
23     String safeLogin = ESAPI.encoder().encodeForHTML(unsafeLogin);
24     request.setAttribute("login", safeLogin);
25 }

```

Figure 21 Sanitization-Point.

The same process that was performed for the creation of the entry-point and exit-point lists was executed to create the list of sanitization-points. For the sake of brevity, Table 4 presents just a few examples of the methods that are in our list of exit-points. For Java, we have identified 52 *sanitization-points*, the full list is available in our study website [Sampaio 2014a].

Qualified Name	Method Name	Parameters
java.net.URLDecoder	decode	java.lang.String
org.owasp.esapi.Encoder	encodeForHTML	java.lang.String
org.owasp.esapi.Encoder	encodeForCSS	java.lang.String

Table 5 List of Sanitization-Points.

### 3.5. Algorithm

Our algorithm reports a source code as vulnerable every time an *entry-point* reaches an *exit-point* without passing through a *sanitization-point*. This will be true for all 11 vulnerabilities we support (see section 3.6). For Java, we have identified 268 methods divided in these three categories. To the best of our knowledge, no other existing solution identifies more methods. Code Fragment 1 depicts the pseudo-algorithm that performs the data flow analysis.

```

1. While thereIsCodeToProcess() {
2.   ASTNode node = getNextNode();
3.   if (isExitPoint(node)) {
4.     for each(parameter in node.getParameters()) {
5.       if (hasVulnerablePath(parameter)) {
6.         reportParameterAsVulnerable();
7.       }
8.     }
9.   }
10. }

```

Code Fragment 1 Data Flow Analysis Pseudo-Algorithm

The algorithm receives Java files as input. After that, it processes and creates the necessary contexts for all elements (variables, fields, methods and etc.) from each line. If the element is an *exit-point*, e.g. *request.setAttribute* or *printWriter.print*, then its parameters (if any) are investigated. This is the main difference from data flow analysis to pattern matching, the ability to navigate through the source code in order to find the content of an element. If the element receives its content from an *entry-point*, the element becomes *vulnerable* also referred as *tainted*. After this point, every other element that interacts with the tainted element also becomes *tainted*. This behavior is called *tainted propagation*. However, if the element receives its content from a trusted element, such as a *string literal* or an output from a sanitization method, this element is considered secure. After all possible paths of an element are investigated; the algorithm is able to safely state if the element is vulnerable or secure. In case there is at least one path that receives tainted content, a warning is reported. In order to improve the chances of helping developers remove the vulnerability, this warning contains: the file and line where the vulnerability entered the source code and where it could be exploited. It also contains the type of the security vulnerability, such as SQL injection or XSS. This can help in case the developer is interested in removing vulnerabilities related to a specific type of vulnerability. The last piece of information is called *full path*. It contains the complete path (each and all method invocations) of the identified vulnerability. This information is important because on medium and large size projects, the amount of possible paths can grow to a size that makes it nearly impossible to perform manual inspection.

The ability to follow the path of variables and methods creates some problems that do not exist when performing static analysis with pattern matching. Figure 22 depicts a code fragment that has a method named *infiniteLoop*. This

method has some statements and in line 124, it invokes itself. Although there is no compilation error in it, if this code is ever executed it will be in a loop until the JVM (Java Virtual Machine) runs out of memory. However, this should not prevent our algorithm from analyzing it. To handle this situation, our algorithm has a recursion control implemented. When it detects a recursion, it does not scan that method invocation repeatedly. Otherwise the scan would never finish. Nonetheless, the rest of the statements are normally analyzed.

```

109 @Override
110 protected void doGet(HttpServletRequest request,
111                      HttpServletResponse response)
112     throws ServletException, IOException {
113     infiniteLoop(request, response);
114 }
115
116 private void infiniteLoop(HttpServletRequest request,
117                          HttpServletResponse response)
118     throws IOException {
119     PrintWriter printWriter = response.getWriter();
120
121     String a = request.getParameter("a");
122     printWriter.print(a);
123
124     infiniteLoop(request, response);
125
126     String b = request.getParameter("b");
127     printWriter.print(b);
128
129 }

```

Figure 22 Infinite loops.

### 3.6. Supported Vulnerabilities

We were able to implement the heuristics to provide support to 11 security vulnerabilities, namely: Command Injection [OWASP 2013j], Cookie Poisoning [OWASP 2013f], Cross-Site Scripting (XSS)[OWASP 2013a], HTTP Response Splitting [OWASP 2013f], LDAP Injection [OWASP 2013k], Log Forging [OWASP 2013l], Path Traversal [OWASP 2013m], Reflection Injection [OWASP 2013n], Security Misconfiguration [OWASP 2013e], SQL Injection [OWASP 2013b] and XPath Injection [OWASP 2013o]. These vulnerabilities were selected to receive our support, because they all stem from untrusted inputs (data sent from the user) that are not properly *validated* also referred as *sanitized*. These inputs do not have their content compared to a range of expected values or they do not have malicious data removed from its content to ensure they are safe to use. We also

decided to support these types of vulnerabilities, because of all vulnerabilities identified in web applications, untrusted inputs are recognized as being the most common and capable of causing severe damage [OWASP 2013d].

With the idea to show the reader how the vulnerabilities occur in the source code and how they can be mitigated, the next sections present the security vulnerabilities supported by our heuristics. The examples that will be presented, reveal a key characteristic for most of the security vulnerabilities: these security vulnerabilities tend to affect only a few lines of code. Then, their detection and removal from the source code might be considered trivial even for inexperienced developers (section 2.3.1). However, when developers are working on large projects with dozens or hundreds of classes and undersized deadlines, this can become a daunting task. Although our examples only contain a few lines of code, the main idea is to show how to identify a specific vulnerability and what actions could be performed to remove it from the source code.

### 3.6.1. Command Injection

Command injection is an attack aimed at executing arbitrary commands on the host operating system via a vulnerable application [OWASP 2013j]. This attack is possible when an application passes unsafe user supplied data (forms, cookies, HTTP headers and etc.) to a system shell.

In line 21, the *exec* method is going to execute an operating system command with the content supplied by the application's user (variable *command* on line 20). However, there is no guarantee that the user will not provide malicious commands, such as: *cmd.exe /k rd /s /q c:\Windows\*. In case this truly happens, the user can cause severe damage to the host operating system, such as: steal information, delete files and much more.

```

15 @Override
16 protected void doGet(HttpServletRequest request,
17                      HttpServletResponse response)
18                      throws ServletException, IOException {
19
20     String command = request.getParameter("command");
21     Runtime.getRuntime().exec(command);
22 }

```

Figure 23 Command Injection vulnerability.

Figure 24 depicts some simple modifications in the code that are sufficient to mitigate the vulnerability. Most of the time, developers know which possible values a user can provide. In this case, the application is going to execute some commands on the operating system. Thus, there is a finite and probably small list of possible commands. Therefore, the developer can verify if the provided input is among that list. In line 22, the method *isValidCommand* is invoked in order to verify if the user correctly provided input as expected by the application. If that is not the case, an exception is thrown (line 25) preventing the user to cause any damage to the application or operating system.

```

16 @Override
17 protected void doGet(HttpServletRequest request,
18                      HttpServletResponse response)
19     throws ServletException, IOException {
20
21     String command = request.getParameter("command");
22     if (isValidCommand(command)) {
23         Runtime.getRuntime().exec(command);
24     } else {
25         throw new InvalidParameterException();
26     }
27 }
28
29
30 private boolean isValidCommand(String command) {
31     if ("ls".equals(command)) {
32         return true;
33     } else if ("pwd".equals(command)) {
34         return true;
35     }
36
37     return false;
38 }

```

Figure 24 Command Injection mitigation.

### 3.6.2. Cookie Poisoning

Cookie poisoning is an attack in which the content stored in the cookies is modified in order to bypass security mechanisms [OWASP 2013f]. Cookies can be used to store the price of products, user's information such as ids and passwords or any other type of information the application desires. The cookies are saved on the browser (client) and are sent back and forth upon each request (sent by the browser) and response (sent by the server). However, there are ways (tools, JavaScript, etc.) that a user can deliberately change the contents of a cookie.

As depicted in the code snippet of Figure 25, the application is storing the id and price of the product that the user wants to buy into a cookie. This may seem hard to believe. However, cases such as this exist and have been exploited in the past [Linden 2009]. There are tools available that allow users to deliberately change the values inside a cookie. For instance, if the user changes the price from U\$100,00 dollars to U\$1,00 dollar, the application will suffer financial damages.

```

15 @Override
16 protected void doGet(HttpServletRequest request,
17                      HttpServletResponse response)
18     throws ServletException, IOException {
19
20     int productId =
21         Integer.parseInt(request.getCookies()[0].getValue());
22     double productPrice =
23         Double.parseDouble(request.getCookies()[1].getValue());
24
25     chargePriceAndDeliverProduct(productId, productPrice);
26 }
27
28 private void chargePriceAndDeliverProduct(int productId,
29                                           double productPrice) {
30     // Save the purchase and deliver product.
31 }

```

Figure 25 Cookie Poisoning - Problem.

Although applications can use information stored inside cookies, they should only be used on specific situations and should never be trusted. As it can be seen in the code below, just a simple modification is sufficient to remove the vulnerability. Instead of retrieving the price of the product from the cookie, the price should be retrieved from the database, based on the product id that the user has selected. After that, even if the user selects an expensive product, then changes the product id to the id of a cheaper one, the price that will be retrieved will be the price of the cheaper product. Therefore, if the user pays less money, he/she will receive a cheaper product.

```

15 @Override
16 protected void doGet(HttpServletRequest request,
17                      HttpServletResponse response)
18     throws ServletException, IOException {
19
20     int productId =
21         Integer.parseInt(request.getCookies()[0].getValue());
22     double productPrice = getProductPriceFromDatabase(productId);
23
24     chargePriceAndDeliverProduct(productId, productPrice);
25 }
26
27 private double getProductPriceFromDatabase(int productId) {
28     return 1; // Select the price based on the informed productId.
29 }
30
31 private void chargePriceAndDeliverProduct(int productId,
32                                           double productPrice) {
33     // Save the purchase and deliver product.
34 }

```

Figure 26 Cookie Poisoning - Mitigation.

### 3.6.3. Cross-Site Scripting (XSS)

Cross-Site Scripting occurs whenever an application takes untrusted data and sends it back to a web browser without proper validation or escaping [OWASP 2013a, 2013d]. The browser then executes the data (possible malicious script) that is capable of stealing user sessions, deface web pages or redirect users to malicious sites. To avoid this, all data sent to the browser should be sanitized and properly escaped.

In Figure 27, the *login* variable is receiving the content submitted by the user in the *login* parameter. Without any sort of validation to verify the content of the parameter, this content is being sent back to the browser. However, the content is unknown and has not been sanitized, in case there is a malicious script in it, the browser will execute it and the vulnerability will be exploited.

```

15 @Override
16 protected void doGet(HttpServletRequest request,
17                      HttpServletResponse response)
18     throws ServletException, IOException {
19
20     String login = request.getParameter("login");
21     request.setAttribute("login", login);
22 }

```

Figure 27 Cross-Site Scripting - Problem.

In line 23 of Figure 28, the code uses the method *encodeForHTML* that converts HTML characters that otherwise would be considered as code, into a format that makes it become pure data. e.g. the character “<” is replaced by “&lt;,” and “>” is replaced by “&gt;.” By doing this, the content can now safely

be sent back to the browser and displayed as content (instead of html tags) to the user.

```

17 @Override
18 protected void doGet(HttpServletRequest request,
19                      HttpServletResponse response)
20                      throws ServletException, IOException {
21
22     String unsafeLogin = request.getParameter("login");
23     String safeLogin = ESAPI.encoder().encodeForHTML(unsafeLogin);
24     request.setAttribute("login", safeLogin);
25 }

```

Figure 28 Cross-Site Scripting - Mitigation.

### 3.6.4. HTTP Response Splitting

HTTP Response Splitting is a vulnerability that happens when an attacker passes malicious data to a vulnerable application, and the application includes this data in a HTTP response header [OWASP 2013f]. If the attacker adds a CRLF (carriage return and line feed) to the end of that untrusted input, he/she can start inserting his/hers own headers into the response. Among others, one consequence of this vulnerability is to make users of the application fetch contaminated pages [Howard et al. 2009] that can cause harm to them.

In Figure 29, the method *sendRedirect* (line 21) adds a 302 (Object Moved) header to the response. The response is then sent to the user, containing among other header information, the page, which the browser should redirect to. However, the content of variable *page* has not being sanitized and if it would contain data such as *Hacker\r\nHTTP/1.1 200 OK\r\n...* The browser would then receive two split responses instead of simply one. Usually attackers exploit this vulnerability to increase the options for a larger attack.

```

15 @Override
16 protected void doGet(HttpServletRequest request,
17                      HttpServletResponse response)
18                      throws ServletException, IOException {
19
20     String page = request.getParameter("page");
21     response.sendRedirect(page);
22 }

```

Figure 29 HTTP Response Splitting - Problem.

To mitigate this vulnerability, developers have to validate the input and remove CRs and LFs from it. In line 25, the method *encodeForURL* does exactly



that. The *sendRedirect* method can be used to redirect users to a local or external page. In case it is to a local page, it is also recommended to verify if the content has any extra URL information. e.g. An IP address, *www* or *http*.

```

18 @Override
19 protected void doGet(HttpServletRequest request,
20                      HttpServletResponse response)
21                      throws ServletException, IOException {
22     try {
23
24         String unsafePage = request.getParameter("page");
25         String safePage = ESAPI.encoder().encodeForURL(unsafePage);
26         response.sendRedirect(safePage);
27
28     } catch (EncodingException e) {
29         e.printStackTrace();
30     }
31 }

```

Figure 30 HTTP Response Splitting - Mitigation.

### 3.6.5. LDAP Injection

LDAP (Lightweight Directory Access Protocol) is commonly used on medium-large companies that want to provide a “single sign on” to its employees. In other words, one user name and password can be used on several different applications. A LDAP injection occurs when untrusted user input is used to construct LDAP statements, e.g. queries, searches or any other LDAP function [OWASP 2013k].

In lines 31 and 36 depicted in Figure 31, the developer is using the content of variables *a* and *b*. However, these two variables have received their content from the method *getParameter* (lines 25 and 26), which is an entry-point (see section 3.2) and all content received from it should be sanitized before being used. Consequently, creating a LDAP search query with these variables is not recommended. In case an attacker manages to provide malicious content, she/he would be able to sign in as a different person (or application) and perform actions (probably causing damage) in their name.

```

20 @Override
21 protected void doGet(HttpServletRequest request,
22                      HttpServletResponse response)
23     throws ServletException, IOException {
24     try {
25         String a = request.getParameter("a");
26         String b = request.getParameter("b");
27
28         InitialDirContext context = new InitialDirContext(null);
29
30         SearchControls ctrls = new SearchControls();
31         ctrls.setReturningAttributes(new String[] { a });
32         ctrls.setSearchScope(SearchControls.SUBTREE_SCOPE);
33         response.getWriter().print(ctrls);
34
35         NamingEnumeration<SearchResult> answers =
36             context.search("o=xx.com", "(uid=" + b + ")", ctrls);
37
38         response.getWriter().print(answers.nextElement().getName());
39     } catch (NamingException e) {
40         e.printStackTrace();
41     }
42 }

```

Figure 31 LDAP - Problem.

One possible solution could be to use regular expression to verify if there is unwanted content in the input. Then, encode all characters to make sure they are interpreted as data and not as html tags (code). Another possible solution is depicted on Figure 32, the developers created two helper methods called *sanitizeInput* and *sanitizeOutput*. As the name is saying, these methods will handle (if any) malicious content from input data and before sending it back to the browser (output data). Now, variables *a* and *b* on lines 27 and 28 have been sanitized and are safe to be used on lines 33 and 38.

```

22 @Override
23 protected void doGet(HttpServletRequest request,
24                      HttpServletResponse response)
25     throws ServletException, IOException {
26     try {
27         String a = sanitizeInput(request.getParameter("a"));
28         String b = sanitizeInput(request.getParameter("b"));
29
30         InitialDirContext context = new InitialDirContext(null);
31
32         SearchControls ctrls = new SearchControls();
33         ctrls.setReturningAttributes(new String[] { a });
34         ctrls.setSearchScope(SearchControls.SUBTREE_SCOPE);
35         response.getWriter().print(ctrls);
36
37         NamingEnumeration<SearchResult> answers =
38             context.search("o=xx.com", "(uid=" + b + ")", ctrls);
39
40         response.getWriter().print(sanitizeOutput(
41             answers.nextElement().getName()));
42     } catch (NamingException e) {
43         e.printStackTrace();
44     }
45 }
46 private String sanitizeInput(String parameter) {
47     return ESAPI.encoder().decodeForHTML(parameter);
48 }
49 private String sanitizeOutput(String parameter) {
50     return ESAPI.encoder().encodeForHTML(parameter);
51 }

```

Figure 32 LDAP - Mitigation.

### 3.6.6. Log Forging

There are several reasons why an application could store information in log files. It could be to store history of events, transactions for later review, auditing purpose and others. Whatever the reason is, developers should be aware that attackers can exploit logging vulnerabilities in order to disguise a bigger attack [OWASP 2013].

Line 23 is trying to convert the content of variable *value* from type *String* to *Integer*. However, if the variable does not contain a content that can be converted, the method *parseInt* will throw an exception of type *NumberFormatException*. In case this happens, line 27 will store the information that a parsing failed, combined with the value that caused the problem. However, attackers can provides content such as *one%0aINFO:+user+status%3dok* with the intention to make the log file more complicate for a later review.

```

15 @Override
16 protected void doGet(HttpServletRequest request,
17                      HttpServletResponse response)
18                      throws ServletException, IOException {
19
20     String value = request.getParameter("value");
21
22     try {
23         int intValue = Integer.parseInt(value);
24         System.out.println(intValue);
25     } catch (NumberFormatException e) {
26         System.out.println("Failed to parse value:" + value);
27     }
28 }
29

```

Figure 33 Log Forging - Problem.

As already described, the best way to mitigate a vulnerability is to sanitize all external (user, other applications) input. Line 23, will decode html characters (if any) into their corresponding ASCII values. This will guarantee that they are interpreted as data and not as tags (code). Besides that, the code behaves in the same way as in Figure 33. In case the method *parseInt* is not able to convert the value from variable *safeValue*, it will throw an exception that will be logged on line 30. However, in this case, even if the content of variable *safeValue* originally

contained malicious data, it will be inserted into the log file as pure data and will not be interpreted as code.

```

17 @Override
18 protected void doGet(HttpServletRequest request,
19                      HttpServletResponse response)
20     throws ServletException, IOException {
21
22     String unsafeValue = request.getParameter("value");
23     String safeValue = ESAPI.encoder().decodeForHTML(unsafeValue);
24
25     try {
26         int intValue = Integer.parseInt(safeValue);
27         System.out.println(intValue);
28     } catch (NumberFormatException e) {
29         System.out.println("Failed to parse value:" + safeValue);
30     }
31 }
32

```

Figure 34 Log Forging - Mitigation.

### 3.6.7. Path Traversal

Path traversal is a vulnerability that allows attackers to access files and directories that are stored outside the application's folder or that they were not supposed to have access [OWASP 2013m]. For instance, some web applications allow their users to download files, such as pdfs, images etc. A common way to perform this is by providing an URL (see URL 1 below). However, if an attacker provides an URL with a path to a different file or directory (see URL 2 below) and the developer does not verify if the content being requested is indeed a valid value, the application might be exploited.

1. `http://site.com/get-files.jsp?file=report.pdf`
2. `http://site.com/get-files.jsp?file=../../../../../etc/passwd`

In the current example (Figure 35), the developer is using the method `getParameter` to receive the content submitted by the user. However, without performing any type of sanitization on the variable *filename*, she/he created a new *File* object and then tried (using the *delete* method) to delete the file informed by the user. As it was mentioned on the previous section, if the user provides a path to a file outside of the domain of the application, the code will delete it, thus, probably causing damage to the application or even to the operating system.

```

16 @Override
17 protected void doGet(HttpServletRequest request,
18                      HttpServletResponse response)
19     throws ServletException, IOException {
20     String fileName = request.getParameter("fileName");
21
22     File file = new File(fileName);
23     file.delete();
24 }

```

Figure 35 Path Traversal - Problem.

In order to mitigate this problem (Figure 36), the developer can verify if the user is requesting a file that is inside the correct directory. Other types of verification might be, to verify if the extension (pdf, doc, etc.) of the file is correct and if the user is the real owner of that file. In summary, simply a few verifications can make the software more secure.

```

16 @Override
17 protected void doGet(HttpServletRequest request,
18                      HttpServletResponse response)
19     throws ServletException, IOException {
20     String fileName = request.getParameter("fileName");
21
22     File file = new File(fileName);
23     if (isValidFile(file)) {
24         file.delete();
25     }
26 }
27 private boolean isValidFile(File file) {
28     try {
29         // Verify if the file is in the correct directory.
30         if ("C:\\MyProject\\photos\\".equals(file.getCanonicalPath())) {
31             return (file.exists()) && (file.isFile());
32         }
33     } catch (IOException e) {
34         e.printStackTrace();
35     }
36     return false;
37 }

```

Figure 36 Path Traversal -Mitigation.

### 3.6.8. Reflection Injection

Reflection is commonly used by programmers that want to modify the runtime behavior of the application [Oracle [S.d.]]. Therefore, this relatively advanced feature should not be used indiscriminately. Fortunately, developers know about this and the use of reflection is most common on libraries and frameworks, such as Hibernate<sup>5</sup> and Struts<sup>6</sup>, not so much on regular programs. A

<sup>5</sup><http://hibernate.org/>

reflection injection vulnerability occurs when the developer incorrectly uses external un-sanitized input in one of the reflection construct methods [OWASP 2013n]. If an attacker is able to exploit this vulnerability, she/he can cause unexpected classes to be loaded, or change which methods or fields are accessed on an object.

Suppose the code snippet below (Figure 37) is used to create a protected connection where data can be safely transferred. The developer is expecting a concrete class, such as `HttpsURLConnection` from the `javax.net.ssl` package, which implements the interface `HttpURLConnection` and sends data in a secure mode. However, if an attacker provides the class `URLConnection` from the `java.net` package, this class also implements the interface `HttpURLConnection`. Consequently, the code will work with no error but the fact that the connection and the transfer of the data will be in an insecure mode.

```

16 @Override
17 protected void doGet(HttpServletRequest request,
18                      HttpServletResponse response)
19                      throws ServletException, IOException {
20     try {
21         String connectionClass = request.getParameter("connectionClass");
22
23         Class<?> connClass = Class.forName(connectionClass);
24         HttpURLConnection conn = (HttpURLConnection) connClass.newInstance();
25         conn.connect();
26
27     } catch (ClassNotFoundException |
28            InstantiationException | IllegalAccessException e) {
29         e.printStackTrace();
30     }
31 }

```

Figure 37 Reflection Injection - Problem.

Once again, to mitigate this problem, the programmer can restrict the values that are accepted as valid content. In Figure 38, before creating a new instance on line 27, the developer is verifying on line 26 if the user provided one of the expected class names in order to create a protected connection. In case this is not true, the connection will simply not be created. Therefore, removing any possibility of having unsecure connections that can be intercepted.

```

18 @Override
19 protected void doGet(HttpServletRequest request,
20                      HttpServletResponse response)
21                      throws ServletException, IOException {
22     try {
23         String connectionClass = ESAPI.encoder().decodeForHTML(
24             request.getParameter("connectionClass"));
25
26         if (isValidConnectionClass(connectionClass)) {
27             Class<?> connClass = Class.forName(connectionClass);
28             HttpURLConnection conn = (HttpURLConnection) connClass.newInstance();
29             conn.connect();
30         }
31     } catch (ClassNotFoundException |
32            InstantiationException |
33            IllegalAccessException e) {
34         e.printStackTrace();
35     }
36 }
37 private boolean isValidConnectionClass(String connectionClass) {
38     if ("javax.net.ssl.HttpURLConnection".equals(connectionClass)) {
39         return true;
40     } else if ("sun.net.www.protocol.https.HttpURLConnectionImpl".
41              equals(connectionClass)) {
42         return true;
43     }
44     return false;
45 }

```

Figure 38 Reflection Injection - Mitigation.

### 3.6.9. Security Misconfiguration

Security misconfiguration usually happens when default accounts are not removed, software systems or libraries are not updated or in the context of source code, when encrypted passwords are hard-coded into the source code [OWASP 2013e]. The problem with this attitude is related to the fact that there are projects, such as *Java Decompiler* [“Java Decompiler” [S.d.]], which are able to decompile java byte code, extract the content of classes and reveal information, such as login and password to unauthorized people. This type of information should be encrypted or at least stored in another file [OWASP 2013e].

The line 23 in Figure 39 is creating a *Connection* object, which will be used to connect and send commands to the database. In order to create this connection, the method *getConnection* must be invoked passing three different parameters. First, the URL of the JDBC<sup>7</sup> driver that will be used to establish and maintain the connection opened. The second and third parameters are the user name and password to authenticate into the database server. The problem with this code snippet is the fact that it has the database’s login and password hard coded and in a decrypted form, which as already mentioned, is not recommended.

<sup>7</sup><http://www.oracle.com/technetwork/java/javase/jdbc/index.html>

```

18 @Override
19 protected void doGet(HttpServletRequest request,
20                      HttpServletResponse response)
21                      throws ServletException, IOException {
22     try {
23         Connection conn = DriverManager.
24             getConnection("com.mysql.jdbc.Driver", "userName", "password");
25
26         conn.commit();
27     } catch (SQLException e) {
28         e.printStackTrace();
29     }
30 }

```

Figure 39 Security misconfiguration - Problem.

The proper way to mitigate the problem of having a hard coded user name and password, is to store them (in a encrypted form) in a configuration file. Then the developer can read this file, decrypt the content and use it in the application. The code snippet below does not show the full implementation for the sake of simplicity. However, the important lesson to learn from this vulnerability is that user names and password should always be kept encrypted.

```

18 @Override
19 protected void doGet(HttpServletRequest request,
20                      HttpServletResponse response)
21                      throws ServletException, IOException {
22     try {
23         Connection conn = DriverManager.
24             getConnection("com.mysql.jdbc.Driver",
25                         getEncrypedUserNameFromConfigFile(),
26                         getEncrypedPasswordFromConfigFile());
27
28         conn.commit();
29     } catch (SQLException e) {
30         e.printStackTrace();
31     }
32 }
33 private String getEncrypedUserNameFromConfigFile() {
34     // ...
35     return "";
36 }
37 private String getEncrypedPasswordFromConfigFile() {
38     // ...
39     return "";
40 }

```

Figure 40 Security misconfiguration - Mitigation.

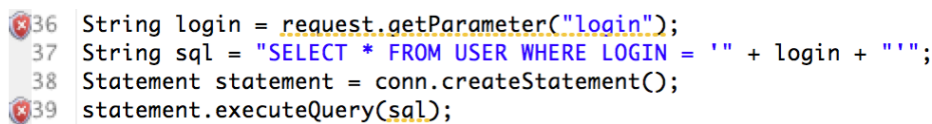
### 3.6.10. SQL Injection

SQL Injection affects programs in all programming languages and it is based on the same principle (untrusted data) as cross-site scripting (XSS) (see 3.6.3). It occurs when developers do not perform the proper validation on inputs originated from users or other applications. These inputs should not be trusted because it is not clear what are the intentions of these users and



applications [21], [19]. When these inputs are passed to an interpreter as part of a command or query and it contains malicious data, it can trick the interpreter into executing unintended commands or access data without proper authorization. To prevent this problem every input should be validated.

In Figure 41, it is possible to observe that the code has at least three mistakes. (1) The *login* variable is being used without receiving any sanitization. (2) The *sql* variable is being created by the concatenation of the "*SELECT \*...*" with the information submitted by the user and (3) the code is using the *statement* instead of the *preparestatement*, which is the recommended for interacting with the database. Therefore, if an attacker submits malicious content, she/he will be able to execute unintended actions, such as login as another person, maybe delete some content that she/he was not supposed to have access, delete tables and cause much more severe damage.



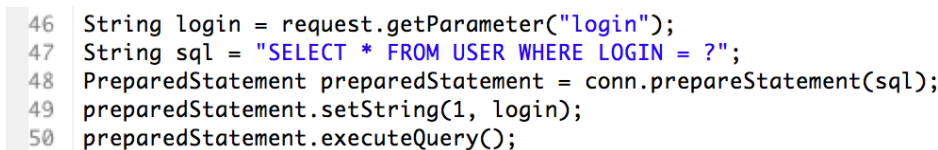
```

36 String login = request.getParameter("login");
37 String sql = "SELECT * FROM USER WHERE LOGIN = '" + login + "'";
38 Statement statement = conn.createStatement();
39 statement.executeQuery(sql);

```

Figure 41 SQL injection - Problem.

Although Figure 42 appears similar to Figure 41, it is possible to observe that the *sql* variable has a "?" (question mark, also known as *place holder*) that will be replaced with the content of the variable *login*. However, before the replacement, the content will be sanitized by the method *setString* from the *preparedStatement* object. These differences are enough to mitigate the SQL injection vulnerability and make the source code secure.



```

46 String login = request.getParameter("login");
47 String sql = "SELECT * FROM USER WHERE LOGIN = ?";
48 PreparedStatement preparedStatement = conn.prepareStatement(sql);
49 preparedStatement.setString(1, login);
50 preparedStatement.executeQuery();

```

Figure 42 SQL injection - Mitigation.

### 3.6.11. XPath Injection

XPath injection occurs in the same way as SQL injection (see section 3.6.10), which is when the XPath queries for XML data are created with user-supplied information that has not been sanitized [OWASP 2013o]. XPath

Injections might be even more dangerous than SQL Injections, because XPath does not contain access control and allows querying of the complete database (XML document) [OWASP 2013o].

Figure 43 depicts a source code containing an XPath vulnerability. The reason for that is the fact that the developer is using the *login* and *password* variables to create an XPath expression on line 30. The variables *login* and *password* have not being sanitized, thus, they might contain malicious content. If the user provides her/his expected credentials, the application will work as expected. However, if an attacker provides content, such as '*or 1=1 --*', she/he could be able to login as any user, usually the first registered user is the admin. Thus, this vulnerability is considered extreme dangerous to an application.

```

21 @Override
22 protected void doGet(HttpServletRequest request,
23     HttpServletResponse response)
24     throws ServletException, IOException {
25     String database = "database.xml";
26     String login = request.getParameter("login");
27     String password = request.getParameter("password");
28
29     try {
30         String expression = "/employees/employee[loginID/text()=' " + login
31             + " and passwd/text()=' " + password + "']";
32
33         InputSource inputSource = new InputSource(new FileInputStream(database));
34
35         XPath xPath = XPathFactory.newInstance().newXPath();
36         xPath.evaluate(expression, inputSource);
37     } catch (XPathExpressionException e) {
38         e.printStackTrace();
39     }
40 }

```

Figure 43 XPath Injection - Problem.

The same idea of how to mitigate other vulnerabilities can be used on XPath injection. User-provided information should never be trusted and before being used, they should be sanitized. The solution presented on Figure 44, was to create a method call *sanitizeInput* on line 42 and invoke it for all the input parameters, i.e. lines 27 and 28. Now, the XPath object on line 37 can safely execute the desired search.

```

22 @Override
23 protected void doGet(HttpServletRequest request,
24                      HttpServletResponse response)
25     throws ServletException, IOException {
26     String database = "database.xml";
27     String login = sanitizeInput(request.getParameter("login"));
28     String password = sanitizeInput(request.getParameter("password"));
29
30     try {
31         String expression = "/employees/employee[loginID/text()=' " + login
32                             + " and passwd/text()=' " + password + "']";
33
34         InputSource inputSource = new InputSource(new FileInputStream(database));
35
36         XPath xPath = XPathFactory.newInstance().newXPath();
37         xPath.evaluate(expression, inputSource);
38     } catch (XPathExpressionException e) {
39         e.printStackTrace();
40     }
41 }
42 private String sanitizeInput(String parameter) {
43     return ESAPI.encoder().decodeForHTML(parameter);
44 }

```

Figure 44 XPath Injection - Mitigation.

### 3.7. Current Limitations

In summary, the types of security vulnerabilities (presented in this section) reveal a key characteristic of most of the security vulnerabilities: each of them tends to affect only a few lines of code but, at the same time, they are responsible for causing severe damage on real applications. Therefore, developers should be aware of security vulnerabilities when creating their software systems. All the presented vulnerabilities are supported by our heuristics. However, there are still known vulnerabilities that are not detected. There are at least two reasons for this limitation. First, this is the first version of our heuristics and, just as any other existing solution, it has its limitations. Second, we focused on vulnerabilities that stem from untrusted input. There are other vulnerabilities that are not affected by this problem. Consequently, they are not detected by our approach.

This section also discusses other potential features that were not implemented due to time constraints. Therefore, they are limitations of our current prototype and not limitations from data flow analysis. For example, containers, such as *arrays*, *lists*, *maps* and *vectors*. *Vectors* are an important part of modern programming languages [Dillig et al. 2011]. The fact that a container might receive a tainted element does not mean that all other elements of its internal structure are also tainted. The current version of our heuristics is not able to make this distinction. In other words, as depicted in Figure 45, variable *x* (line 20) is an

array initialized with three elements. Untrusted variables *a* and *b* and *string literal* (trusted content) “*c*”. Once a container receives untrusted data, the object itself and all of its indexes are incorrectly marked as vulnerable. Every attempt to use that object will be reported as a security vulnerability, as is depicted in lines 22, 25 and 26, which in this case are false positives. In this example, only lines 23 and 24 should have been reported as vulnerable.

```

13 @Override
14 protected void doGet(HttpServletRequest request,
15                      HttpServletResponse response)
16                      throws ServletException, IOException {
17     String a = request.getParameter("a");
18     String b = request.getParameter("b");
19
20     String[] x = { a, b, "c" };
21
22     response.getWriter().print(x);
23     response.getWriter().print(x[0]);
24     response.getWriter().print(x[1]);
25     response.getWriter().print(x[2]);
26     response.getWriter().print(x[50]);
27 }

```

Figure 45 False positives on containers generated by ESVD.

For now, it is enough to know that we tested some existing solutions (see section 2.6) in order to verify how accurate our prototype was, when compared to other tools. Figure 46 depicts the results from the same code snippet (from Figure 45) being analyzed by these tools. The first image on the left is a screenshot from the vulnerabilities reported by ASIDE. The second image on the right belongs to CodePro Analytics. This tool reports two vulnerabilities on each line, because it identifies two vulnerable paths (lines 17 and 18) reaching the vulnerable code (lines 23, 24, 25 and 26). The last image on the bottom is the Lapse+ view where the vulnerabilities found are displayed. As it can be seen just as the other tools, it also incorrectly reports lines 25 and 26.

Suspicious call	Method	Category	Project	File	Line
response.getWriter().print(x[0])	java.io.PrintWriter.print(String)	Cross-site Scripting	WebDemo	Containers.java	23
response.getWriter().print(x[1])	java.io.PrintWriter.print(String)	Cross-site Scripting	WebDemo	Containers.java	24
response.getWriter().print(x[2])	java.io.PrintWriter.print(String)	Cross-site Scripting	WebDemo	Containers.java	25
response.getWriter().print(x[50])	java.io.PrintWriter.print(String)	Cross-site Scripting	WebDemo	Containers.java	26

Figure 46 False positives on containers generated by ASIDE, CodePro Analytics and Lapse+.

## 4 Early Vulnerability Detector: Implementation

After our detection heuristics were created, we designed and implemented our tool prototype. Although our heuristics are generic and can be implemented to the context of other programming languages, the first (and current) version of our prototype only provides support for the Java<sup>8</sup> programming language, which is one of the most popular programming languages [Zeichick 2012]. The prototype is a plugin for the Eclipse<sup>9</sup> IDE (integrated development environment), which is the most popular IDE used for the Java programming language[Geer 2005]. Another fact that contributed for choosing Eclipse (and not other IDE) was the amount of available tutorials [51][70][71] and engagement from the community. The plugin, called *ESVD - Early Security Vulnerability Detector*, is free of charge and can be downloaded from the Eclipse Marketplace [Sampaio and Garcia 2014].

### 4.1. Architecture

Although this prototype was created to find security vulnerabilities in the source code of Java programs, the architecture was designed to flexibly allow the incorporation of detectors for other programming flaws. Some examples of these additional programming flaws are: detection of code anomalies (*code smells* [Fowler et al. 1999]), detection of empty or poor exception handlers, and others. The reason is that our early vulnerability detector could be used in conjunction with other early detectors, intended to support other practices for modular and robust programming. For instance, our research group has other students studying these subjects [Albuquerque et al. 2014; Barbosa et al. 2012]. These studies are probably going to result in the creation of other Eclipse early detector plugins. Thus, the architecture foundation for promoting the detector's integration is

---

<sup>8</sup><https://www.oracle.com/java/>

<sup>9</sup><https://www.eclipse.org>

already prepared. Figure 47 depicts the plugin's architecture, which consists of four main modules: Manager, Analyzer, Verifier and Reporter. Each of them will be described in the next sections. This section also presents the mechanism (call graph) that made our prototype feasible. In order to be constantly running in the background, it would not be possible to process all files from all projects all the time. Therefore, the call graph object is able to identify modified elements (classes and methods), its interactions and to inform to the DFA algorithm, what are the elements that should be processed.

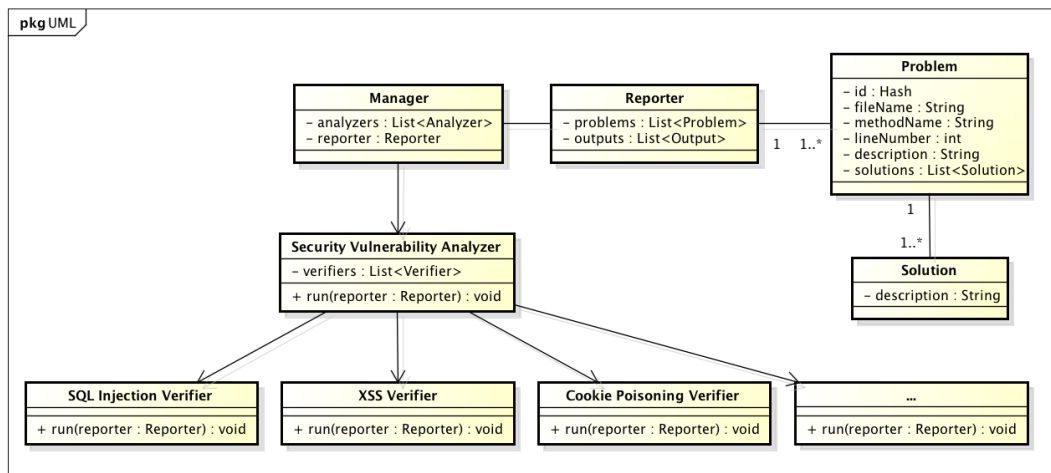


Figure 47ESVD Plugin Architecture.

#### 4.1.1. Verifier

The verifier is the module that has all the necessary knowledge to find the vulnerabilities in the source code. It combines the knowledge from lists of entry-point, exit-point and sanitization-point with the data flow analysis of the analyzed code. By dividing the architecture into several layers, when a new vulnerability appears it will only be necessary to create a new verifier. In case there are false positives in the cookie poisoning verifier, it is known exactly where to search for the problem. Developers must understand that each verifier will only detect one security vulnerability. Therefore, if they unselect one or more verifiers and the source code has the vulnerability of that verifier, the plugin will not be able to detect and report it. Thus, unless there is a specific reason to unselect them, they should be left selected. Figure 48 depicts the already 11 implemented verifiers.

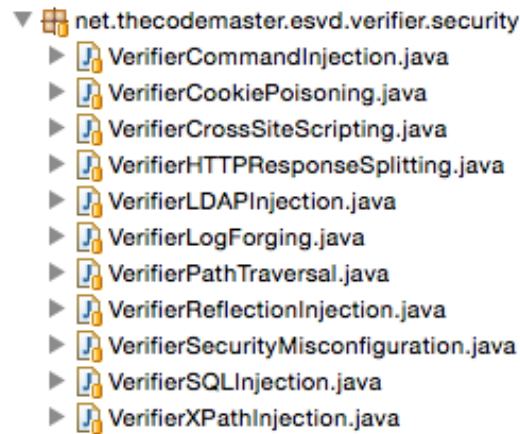


Figure 48 Implemented Verifiers.

#### 4.1.2. Analyzer

The analyzer is the module responsible to aggregate all the verifiers of its type. Thus, the Security Vulnerability Analyzer has several verifiers: SQL injection Verifier, Cookie Poisoning Verifier, and so forth. When the user selects an analyzer, it automatically selects all of its verifiers. However, if the user does not want to execute a specific verifier, he/she can unselect it, as can be seen in Figure 49 and the verifier will not be executed.

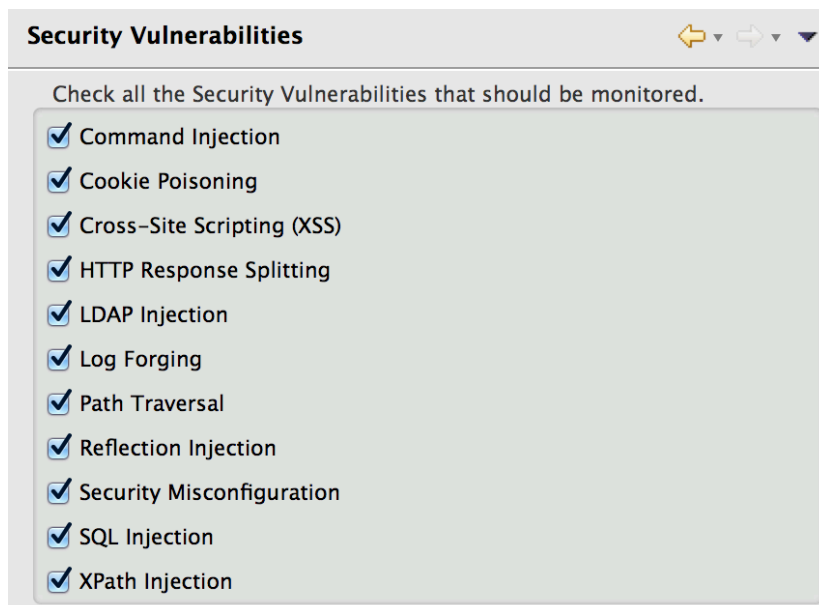


Figure 49 Verifiers of the Security Vulnerability Analyzer.

#### 4.1.3. Manager

The Manager is the main module of the plugin, which is responsible to interact with the user (in our context, the developer) by providing an interface where the user can define basic settings. First, the developer selects what will be executed when the plugin runs. These *options* that the user can select to run or not, are called Analyzers (section 4.1.2). In the current version of our plugin, there is only one analyzer, which is the Security Vulnerability Analyzer. Second, the developer selects where the results should be displayed. These *options* where the results can be displayed are called Reporters. Currently, there is only one reporter, which is the Security View Reporter. In the future, other analyzers and reporters can be added.

In order to make the plugin perform early detection, it is necessary to monitor all files (\*.java) that are being modified from a selected project. Fortunately, Eclipse already provides the ability to attach one or more classes into its compilation process. Therefore, the user selects which analyzers are going to be executed and the manager attaches them into Eclipse's compilation process. After that, the analyzers are invoked every time Eclipse compiles a class. The Eclipse's infrastructure already handles multiple threads, how to update the user interface (UI) and other aspects that the plugin does not need to worry about.

#### 4.1.4. Reporter

Finally, the last entity is the reporter; it is the module responsible to receive a list with all found problems (vulnerabilities, code anomalies and etc.) and writes them out on the output selected by the developer. This output can be (1) Eclipse Console, (2) Eclipse Problems View, (3) Text file, (4) Xml file or (5) ESVD Security View (see Figure 50). The idea of allowing the writing of the results into other places, such as a file, is that other tools can read this file and perform other operations that we do not perform, or present them in a different way. An example is a chart showing the most found vulnerabilities or who is the developer that is adding more vulnerabilities into the source code.



Description	Location	Vulnerability	Resource	Path
▼ a has 1 vulnerable path.	21	Cross-Site Scripting	TestingUnit.java	
⊛ The method 'request.getParameter("a")' must be sanitized before being used.	20	Input not sanitized	TestingUnit.java	doGet - a - request.getParameter("a")
▼ b has 1 vulnerable path.	24	Cross-Site Scripting	TestingUnit.java	
⊛ The method 'request.getParameter("b")' must be sanitized before being used.	23	Input not sanitized	TestingUnit.java	doGet - b - request.getParameter("b")
▼ a + b has 2 vulnerable paths.	26	Cross-Site Scripting	TestingUnit.java	
⊛ The method 'request.getParameter("a")' must be sanitized before being used.	20	Input not sanitized	TestingUnit.java	doGet - a + b - a - request.getParameter("a")
⊛ The method 'request.getParameter("b")' must be sanitized before being used.	23	Input not sanitized	TestingUnit.java	doGet - a + b - b - request.getParameter("b")

Figure 50 Security Vulnerability View.

## 4.2. Call Graph

In order to make our prototype constantly running on the background, it was not possible to process all files from all projects all the time. Therefore, the goal of the *Call Graph* object is to store all the interactions between the classes of the project being analyzed. Thus, these interactions will guide the data flow algorithm when the analysis is performed. As it can be seen in Figure 51, we have three types of class interactions: (i) Forward connection occurs when class *A* invokes one or more methods of any other class(es), (ii) Backward connection occurs when class *C* has one or more methods being invoked by any other class(es), and (iii) Bi connection occurs when class *B* invokes one or more methods of class *C* and has one or more methods being invoked by any other class.

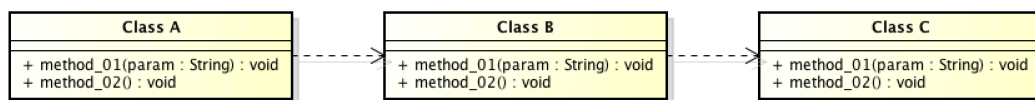


Figure 51 Types of Interactions.

### 4.2.1. Clean Call Graph

The first time the source code is analyzed, it is necessary to process all resource (\*.java) files. In a process called *Call Graph Construction*. Figure 52 depicts a project containing six classes with names Class A - F. The names of the classes and their methods are simply illustrative. The idea here is to show that all these classes and their methods will be scanned. The meaning of *scan*, *process*, *analyze* in this dissertation is that our algorithm will follow the data flow of variables, fields and method invocations in all methods of the current class being processed.

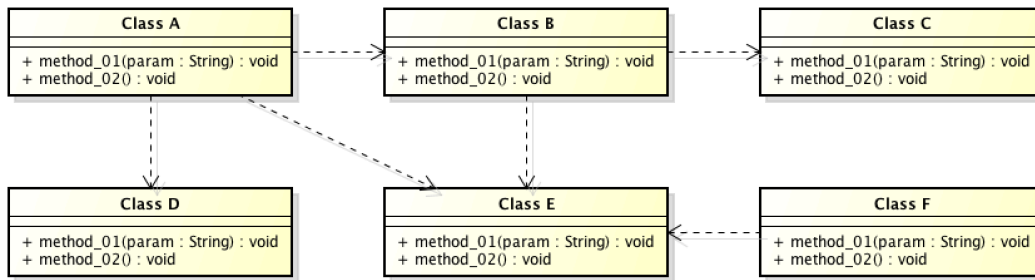


Figure 52 Clean Call Graph.

#### 4.2.2. Prime Call Graph

The next time the code is analyzed, only the modified resources since the last analysis and the classes that have connection with them are scanned, because the *Call Graph* already has the interactions of the other classes that do not interact with the current modified resources. In Figure 53, it is possible to observe that if a developer is working on class F, only classes F and E will be processed. Classes A, B, C and D (from Figure 52) cannot affect the result of the analysis. This ability decreases the amount of time, memory required to perform a full verification and makes it feasible to perform early vulnerability detection using DFA on large projects.

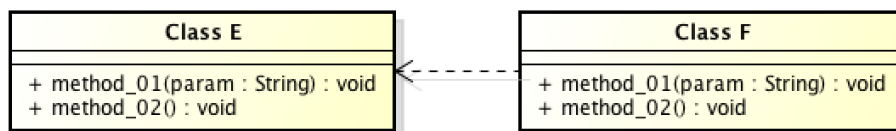


Figure 53 Prime Call Graph.

#### 4.3. Features

The developers are the end users of our prototype and not security specialists. Thus, we avoided adding features in our prototype that no average programmer will ever use, or that are so complex that only security specialists are able to use. We planned carefully to only add features that make sense to the developer's point of view. These features will be described next.

1. *Early Detection*: The developer will not need to activate the tool execution (i.e. press a "Run" button) for starting the scanning of files and searching for

security vulnerabilities in the program. As default configuration, the tool will always be running in the background. As soon as it finds a vulnerability, a warning containing information about the vulnerability is displayed to the developer.

2. *Data Flow Analysis with Context-Sensitivity*: In order to reduce the rate of false positives that are normally present in most existing solutions (section 1.2), we implemented our DFA-based algorithm to detect vulnerabilities with context sensitivity, aiming at obtaining higher accuracy.
3. *Description of the vulnerability*: When a security vulnerability is found in the source code, a complete description is presented to the developer. This description contains: the name of the vulnerability, the file name, the line number, and the path showing where the vulnerability entered the source code and where it can be exploited.
4. *Improved Performance*: Eclipse provides all necessary information related to the files that are currently being edited by the developer. We believe there is no reason to re-scan all the other unchanged files. The plugin performs this verification before performing a full code analysis. This is necessary to compensate the constant processing of the plugin. Otherwise, it would be impossible to perform this kind of detection on all files in the projects.

## 5 Evaluation

We performed two empirical studies in order to evaluate if secure programming can be improved through: (i) context-sensitive data flow analysis (section 3.1), and (ii) early vulnerability detection (section 2.5). The first study was intended to verify if and to what extent data flow analysis could reduce the rate of false positives in comparison to other techniques (supported by other existing static analysis tools). This verification was carried out by comparing the accuracy of our technique against other automated techniques (section 2.6). To this end, we ran our tool prototype and the other techniques on five open-source software projects and one custom-made project (section 5.1.2). These projects are representative of different domains and exhibit different trends on security vulnerability.

The second study focused on assessing the accuracy of early detection. We designed and executed a controlled experiment, where participants were asked to implement some functionalities of a program. Each participant was assigned to one of two groups, *early detection group* and *late detection group*. The former group was equipped with continuous detection support for identifying vulnerabilities (early detection). The latter group of participants was only able to trigger the vulnerability detection analysis at the 40 minutes mark of their programming session. Therefore, this group was actually employing a late detection procedure, which is stimulated by the conventional techniques for program security analysis. We then compared which of the groups of participants produced code with higher security. This comparison was based on the analysis of the amount of security vulnerabilities created, fixed and found in the source code they produced (section 5.2.2).

## 5.1.

### Study 1: Accuracy Benchmarking

This section reports the procedures and results of the first exploratory study. The first study was aimed at assessing if and to what extent our approach – i.e. supported by context-sensitive data flow analysis – could reduce the rate of false positives in comparison to other techniques (supported by other existing static analysis tools). In this study, we were not concerned with assessing the influence of early and late detection on the produced code’s security. The evaluation of accuracy in this study was based on three metrics presented in section 5.1.4.

In order to conduct this first study, we executed our prototype and the three existing solutions (Lapse+, ASIDE and CodePro) selected to participate on our study (section 2.6). However, it is important to mention that our focus is on their underlying techniques employed to find security vulnerabilities and not in anything else, such as user interface or usability. In particular, we intend to compare: (i) the accuracy of *pattern matching* against *data flow analysis*. Table 6 presents the underlying techniques supported by each tool, i.e. if they use pattern matching or data flow analysis and if they perform late detection or early detection.

	Pattern Matching	Data Flow Analysis	Late Detection	Early Detection
Lapse+	X		X	
ASIDE	X			X
CodePro		X (context-insensitive)	X	
ESVD		X (context-sensitive)		X

Table 6 Characteristics of the tools used in our evaluation.

We ran these four tools on six open-source projects, namely BlueBlog [Burén 2003], PersonalBlog [Payne 2003], WebGoat [OWASP 2006], Roller [Johnson 2002], Pebble [Brown 2006] and NCO [Sampaio 2013]. We recorded the security vulnerability reports for each one of them. Almost all of these projects were selected because they were also used on previous studies [Livshits 2005], [Tripp et al. 2009] for evaluating security analysis tools. The only exception is NCO, a personal project created by the author of this dissertation. This decision will be justified later.

The original motivation in using many projects in common with previous studies was to compare our results against theirs. However, this comparison was not possible because the aforementioned studies did not make their results

publicly available. In any case, we still used all these projects as independent researchers considered them good benchmarks for assessing the accuracy of vulnerability detection (section 5.1.2). Therefore, we had to derive our own results without a direct comparison with the vulnerabilities found in those previous studies. We then analyzed whether each reported warning was actually a security vulnerability. This analysis was based on an oracle (or ground truth) containing a list with all known security vulnerabilities of each project. As the original developers and the main authors of the aforementioned studies were not available, the oracle was produced through a careful manual inspection of each possible case of security vulnerability. The author of this dissertation has carefully examined if each of the possible security vulnerabilities was a true or false positive. The comparison of the ground truth and the reported warnings enabled us to validate or invalidate each warning produced by a particular tool. The final report containing all reported warnings and the ground truth of each of the open-source projects can be downloaded from our study website[Sampaio 2014b].

#### **5.1.1. Testing Environment**

The testing environment comprised an Apple laptop running OS X Mavericks, with a 2,7 GHz Intel Core i7 processor and 16GB of RAM. The plugins ran on top of Java Standard Edition Runtime Environment (JRE), build 1.7.0\_55-b13. The maximum amount of memory granted to each plugin was set to 1GB of RAM. In order to improve our confidence on the results, we performed each test five times for each plugin. These multiple tests were necessary because we noticed differences in time, memory usage and the list of vulnerabilities reported for some of the tools when executed on the same source code. The results presented on the next sections are the average of these five executions. The average was used to compensate any possible external process running on the background of the used laptop.

### 5.1.2. Open-Source Projects

This section describes the six applications used in our study. All the selected projects are open source in order to allow others to replicate our study in the future. The selection of these projects was also based on three main criteria. First, they have been recently used as benchmark applications for evaluating security analysis tools, in studies such as SecuriBench [Livshits 2005] and TAJ [Tripp et al. 2009]. Second, previous studies have reported a wide range of security vulnerabilities in these programs, and finally, they rely on different programming technologies and have different sizes (Table 7). As aforementioned, satisfying the first criterion above was also originally intended to enable us to compare our results against the results from previous empirical studies.

Table 7 presents some details about these six applications. The first row presents the names of each application. The second one describes the number of the program version used in our study. Although some of the applications have newer versions, we used the same versions from previous empirical studies in order to contrast the results. Rows 3, 4, 5 and 6 contain the number of packages, classes, methods and lines of code respectively. The size of the program varies from close to 2 thousand lines of code to more than 36 thousand. The inclusion of both small and large programs in our evaluation was important as execution time of the detection algorithm can be influenced by the program size. Our proposed detector is continuously running when the programmer is editing the code statements. Therefore, we needed to check to what extent the execution time significantly increases (or not) for larger programs.

BlueBlog [Burén 2003] is a blogging software, based on the use of Java<sup>10</sup> and Servlet<sup>11</sup> technologies. It was designed for non-professional users, in other words, with easy installation and extreme flexibility. PersonalBlog [Payne 2003] is a lightweight application for personal blogging. This project is written in Java and in a variety of J2EE technologies, including: Ant<sup>12</sup>, Servlets, JSP<sup>13</sup>, JDBC<sup>14</sup>,

---

<sup>10</sup><https://www.oracle.com/java/index.html>

<sup>11</sup><http://www.oracle.com/technetwork/java/index-jsp-135475.html>

<sup>12</sup><http://ant.apache.org/>

<sup>13</sup><http://www.oracle.com/technetwork/java/javaee/jsp/index.html>

Hibernate<sup>15</sup>, Struts<sup>16</sup>, Tiles<sup>17</sup> and Log4J<sup>18</sup>. WebGoat [OWASP 2006] is a deliberately insecure web application maintained by OWASP, designed to mimic recurring security problems of web applications. Users of this application can demonstrate their understanding of relevant secure programming issues by exploiting real vulnerabilities in the WebGoat application. Roller [Johnson 2002] is the open source Java blog server used by blogs.oracle.com, the Apache Software Foundation and many others. Pebble [Brown 2006] is a lightweight, open source, Java EE blogging tool. NCO (Nova Clínica Odontológica) is a custom-made J2EE application created for a dental clinic, developed by the author of this dissertation. The benefit of using this application was the fact that it follows a MVC[Deacon 2009] architecture and has hundreds of method invocations and class instantiations intertwined. Thus, this program can expose data flow analysis to its limits. The source code of this application can be downloaded from our study website [Sampaio 2013].

	BlueBlog	PersonalBlog	WebGoat	Roller	Pebble	NCO
Version	1.0	1.2.6	5.4	0.9.9	2.6.4	1.0
# of packages	22	10	24	70	100	49
# of classes	38	38	159	283	743	84
# of methods	227	253	1.453	2.704	3.445	517
Lines of Code	2.200	2.933	24.483	34.301	36.709	6.048

Table 7 Benchmark applications.

### 5.1.3. Supported Vulnerabilities

A list with dozens of recommended tools to find security vulnerabilities can be found at the OWASP website [OWASP 2003a]. However, each one of them focuses on the detection of only a few specific types of vulnerabilities. The tools selected for our study focus on vulnerabilities that stem from program input and

---

<sup>14</sup><http://www.oracle.com/technetwork/java/javase/jdbc/index.html>

<sup>15</sup><http://hibernate.org/>

<sup>16</sup><http://struts.apache.org/>

<sup>17</sup><https://tiles.apache.org/>

<sup>18</sup><http://logging.apache.org/log4j/2.x/>



output not being properly validated. The reason was twofold. First, from all vulnerabilities identified in web applications, untrusted inputs and outputs are recognized as being the most common ones [OWASP 2013d]. Second, these vulnerabilities are not dependent on how developers create their code or which technologies are being used. For instance, when developers pass an unsafe content to a log method, it is guaranteed that there is a vulnerable code, as opposed to other types of vulnerabilities such as: unauthorized access. The reason is that each program can implement different types of authorization, e.g. using sessions, cookies, URL rewriting, and so forth. In other words, vulnerabilities that stem from input and output not being properly validated can be detected without the need of any input or configuration from developers. In addition, in case they are exploited, they can cause a great amount of damage.

Table 8 presents the vulnerabilities supported by all three external tools and our prototype. The first column is simply a number for the row. The second one is the vulnerability's name and columns 3, 4, 5 and 6 are the names of the tools that participated on the benchmark. Even though ASIDE only supports four vulnerabilities, it was used on our study because of two reasons. First, ASIDE is the only available tool we found that performs early detection of security vulnerabilities for Java. Second, when we were first testing several tools from the OWASP list, trying to decide which of them would we add to the benchmark experiment, ASIDE showed decent accuracy results when compared to some others.

Nr	Vulnerability	ASIDE	Lapse+	CodePro	ESVD
1	Command Injection	-	✓	✓	✓
2	Cookie Poisoning	✓	✓	✓	✓
3	Cross-Site Scripting (XSS)	✓	✓	✓	✓
4	HTTP Response Splitting	-	✓	✓	✓
5	LDAP Injection	-	✓	✓	✓
6	Log Forging	✓	✓	✓	✓
7	Path Traversal	-	✓	✓	✓
8	Reflection Injection	-	-	✓	✓
9	Security Misconfiguration	-	-	✓	✓
10	SQL Injection	✓	✓	✓	✓
11	XPath Injection	-	✓	✓	✓
	<b>Total</b>	<b>4</b>	<b>9</b>	<b>11</b>	<b>11</b>

Table 8 Supported vulnerabilities.

#### 5.1.4. Precision, Recall and F-measure

The precision, recall and f-measure metrics are frequently used to evaluate the accuracy of static analysis techniques, and they are also particularly used to assess techniques for vulnerability detection [Sasaki 2007]. These metrics are composed by three other metrics, namely number of true positives, number of false positives and number of false negatives. *True positive* is when a security vulnerability is reported and it is an actual vulnerable code. *False positive* is when a security vulnerability is reported and it is not an actual vulnerable code. *False negative* is when a security vulnerability is not reported but it is an actual vulnerable code.

The figures below depict the equations used to calculate precision, recall and f-measure. The symbols  $tp$ ,  $fp$  and  $fn$  in the figures below represent respectively: number of true positives, number of false positives and number of false negatives.

$$\text{Precision} = \frac{tp}{tp + fp}$$

Figure 54 Equation of Precision.

Precision quantifies the rate of security vulnerabilities correctly identified by the number of detected vulnerabilities [Sasaki 2007].

$$\text{Recall} = \frac{tp}{tp + fn}$$

Figure 55 Equation of Recall.

Recall quantifies the rate of security vulnerabilities correctly identified by the number of existing vulnerabilities [Sasaki 2007].

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Figure 56 Equation of F-measure.

F-measure or balanced f-score can be interpreted as a weighted average of the precision and recall, where the highest value is one and the lowest is zero [Sasaki 2007]. F-measure is not frequently used as the other metrics. However, it was used in our study because our technique achieved a lower rate of false positives when compared to the other solutions, but in some specific situations, it achieved a higher rate of false negatives. Therefore, the f-measure was responsible to balance these two metrics and produce a better final score. False negatives are also considered very harmful to secure programming, because it might give the wrong impression that the source code is secure when in reality is not.

### 5.1.5. Study 1: Results

This section addresses the results related to our first research question. We compute, collect and discuss the results of the accuracy metrics, which are presented in section 5.1.5.1. We also present the results concerning memory and time spent (section 5.1.5.2). The goal is to support our understanding if the best accuracy results of one technique do not negatively lead to a significantly higher use of computation resources.

### 5.1.5.1. Accuracy Results

The histograms presented in this section follow a similar structure: each of the three sets of bars respectively represent: the percentage of true positives, false positives and false negatives on each of the analyzed applications. Each of these sets have bars representing the results of the four tools: the first bar represents ASIDE, the second represents CodePro, the third represents Lapse+ and fourth represents ESVD. On the right side, there is a legend with the name of the tool and the number of vulnerabilities it reported on the analyzed application. In the following paragraphs, we report the comparative results individually for each project as there was some variation across the three accuracy metrics being computed. We explicitly mention when there are similarities and divergences in the results.

The first analyzed project was BlueBlog. This project contains 2.200 lines of code and the total amount of found vulnerabilities was 18. ASIDE and Lapse+, which use pattern matching, reported 43 and 32 warnings respectively. From those warnings, 32 warnings (or 74%) from ASIDE and 19 warnings (or 59%) from Lapse+ were false positives. CodePro and ESVD, which use data flow analysis, reported five and eight warnings, respectively. From those warnings, 1 warning (or 20%) from CodePro and 0 warnings (or 0%) from ESVD were false positives. The DFA tools were able to achieve lower rates of false positive, because as mentioned earlier (section 3.1), the data flow technique only reports warnings when it is able to identify a path where an entry-point reaches an exit-point, without passing through a sanitization-point, thereby decreasing the rate of false positives.

However, as complementary results demonstrate, the rate of false negatives was higher than the ones from the pattern matching tools. ASIDE and Lapse+ had 7 (or 39%) and 5 (or 28%) false negatives while CodePro and ESVD had 14 (or 78%) and 10 (or 56%) false negatives respectively. This is explained because pattern-matching tools flag anything they are not able to verify. On the other hand, data flow analysis tools, only flag vulnerable paths. However, as already explained (section 3.5), there are some types of code, such as containers, that if

the data flow algorithm is not able to properly identify, will generate a false negative.

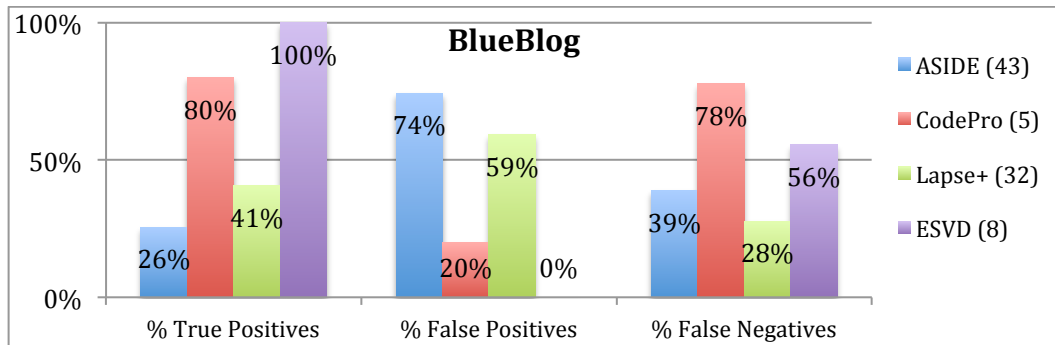


Figure 57. BlueBlog.

The PersonalBlog project contains 2.933 lines of code and the amount of found vulnerabilities was 148. ASIDE and Lapse+ reported 68 and 42 warnings, respectively. From those warnings, 9 (or 13%) warnings from ASIDE and 7 (or 17%) warnings from Lapse+ were false positives. CodePro and ESVD reported 4 and 119 warnings; from those warnings, 1 (or 25%) from CodePro and 3 (or 3%) from ESVD were false positives. On this project, ESVD achieved the lowest/best rate of false positive and false negative from all tools, 3% and 22% respectively.

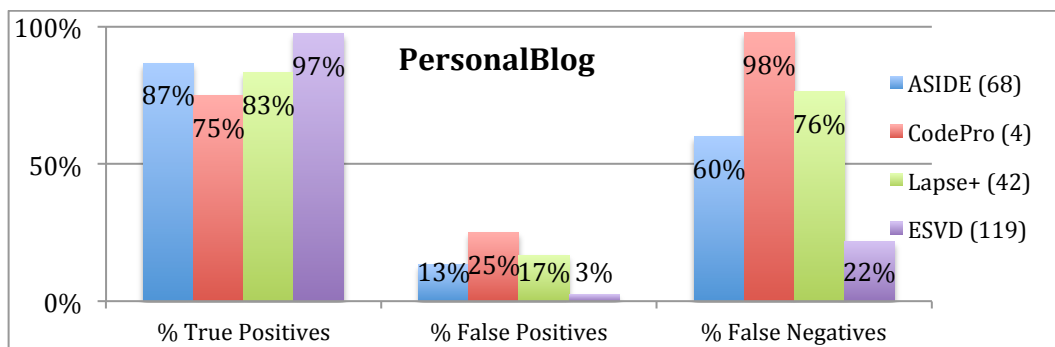


Figure 58 PersonalBlog.

One of the most important benefits from the benchmark was to have enabled us to investigate the false positives in order to understand what was the reason our technique did not reach 100% of true positive. Figure 59 depicts a code snippet where our technique flagged lines 866 and 867, because as explained in section 3.6.9, a user name and password should not be hardcoded. However, the problem on this code is the fact that the developer concatenated *strings literals* and our heuristics were not able to differentiate these strings from the actual user name and password. Because of this, it incorrectly flagged the code as vulnerable.

```

861
862
863
864
865
866
867
868
try {
    // Load the database driver
    Class.forName("org.gjt.mm.mysql.Driver");

    // Get a connection to the database
    Connection conn = DriverManager.getConnection(dburl + "?user=" +
        dbuser + "&password=" + dbpassword);
}

```

Figure 59 PersonalBlog false positive.

The WebGoat project contains 24.483 lines of code and the amount of found vulnerabilities was 488. ASIDE and Lapse+ reported 702 and 465 warnings, respectively. From those warnings, 355(or 51%) warnings from ASIDE and 148(or 32%) warnings from Lapse+ were false positives. CodePro and ESVD reported 86 and 253 warnings; from those warnings, 43(or 50%) from CodePro and 53 (or 21%) from ESVD were false positives. Once again, ESVD achieved the lowest/best rate of false positives.

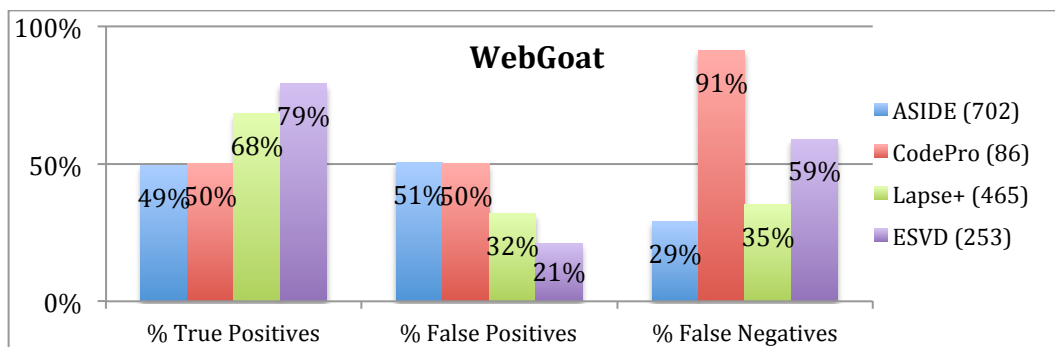


Figure 60 WebGoat.

Although ESVD achieved the lowest rate of false positives compared to the other tools when analyzing the WebGoat project, there were still several cases of them. As depicted in Figure 61, the code correctly used *preparedStatement* on line 279. However, there is a string concatenation on line 273, and a string concatenation as a query to the database is not allowed by our heuristics. Thus, flagging the code as vulnerable. However, on this particular case even with the string concatenation, there is no possible way to inject a malicious code because the type of variable *nextId* is *int* and numbers cannot hold malicious code. Although we believe our heuristics should be prepared to identify these special cases, string concatenation is not recommended even in cases like this.

```

272     int nextId = getNextUID(s);
273     String query = "INSERT INTO employee VALUES ( " + nextId + " , ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? )";
274
275     // System.out.println("Query: " + query);
276
277     try
278     {
279         PreparedStatement ps = WebSession.getConnection(s).prepareStatement(query);
280
281         ps.setString(1, employee.getFirstName().toLowerCase());
282         ps.setString(2, employee.getLastName());
283         ps.setString(3, employee.getSsn());
284         ps.setString(4, employee.getTitle());
285         ps.setString(5, employee.getPhoneNumber());
286         ps.setString(6, employee.getAddress1());
287         ps.setString(7, employee.getAddress2());
288         ps.setInt(8, employee.getManager());
289         ps.setString(9, employee.getStartDate());
290         ps.setString(10, employee.getCcn());
291         ps.setInt(11, employee.getCcnLimit());
292         ps.setString(12, employee.getDisciplinaryActionDate());
293         ps.setString(13, employee.getDisciplinaryActionNotes());
294         ps.setString(14, employee.getPersonalDescription());
295
296         ps.execute();

```

Figure 61 False positive on WebGoat.

The Roller project contains 34.301 lines of code and the amount of found vulnerabilities was 521. ASIDE and Lapse+, reported 209 and 212 warnings respectively. From those warnings, 147 warnings (or 70%) from ASIDE and 125 warnings (or 59%) from Lapse+ were false positives. CodePro and ESVD reported 58 and 466 warnings, respectively. From those warnings, 13 (or 22%) warning from CodePro and 3 (or 1%) warnings from ESVD were false positives.

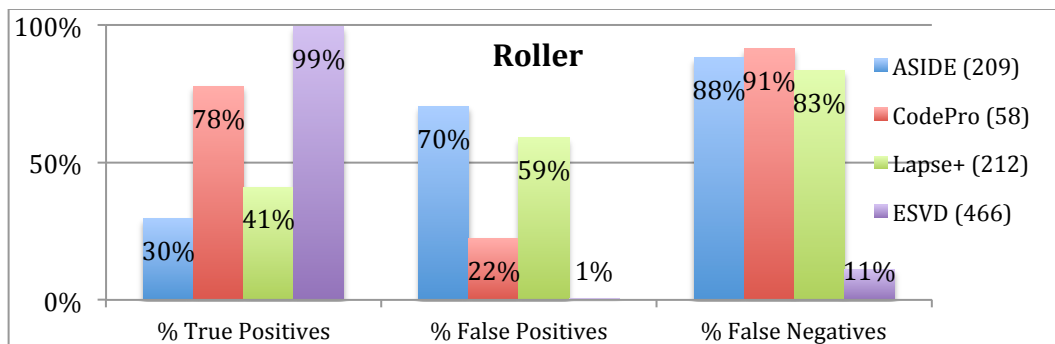


Figure 62 Roller.

Once again, our heuristics was close to but did not achieve 100% of true positives, the reason this time is shown in Figure 63. Our heuristics is prepared to identify a path that comes from an entry-point (line 93) and goes to an exit-point (line 115) without being properly sanitized. This is what happened in this example. However if we take a closer look at it, it is possible to observe that even if the user provides malicious content, the constructor *new Locale(...)* on line 112 can only return a valid locale object or the default one in case the content of variable *newLang* is not recognized as a valid option for a language. In other

words, there is no vulnerability in this case. Line 116 is a case of true positive of log forging.

```

93 String newLang = request.getParameter("language");
94 if (newLang==null || newLang.length()==0) {
95     // add error message
96     ctx.put("languageError", "Unable to switch language: no new language specified.");
97     // proceed with request serving
98     return super.handleRequest(request, response, ctx);
99 }
100
101 // verify if new language is supported
102 if (!LanguageUtil.isSupported(newLang, servletContext)) {
103     // add error message
104     ctx.put("languageError", "Unable to switch language: new language '"+newLang+"' is not supported.");
105     // proceed with request serving
106     return super.handleRequest(request, response, ctx);
107 }
108
109 // by now, all should be fine: change Locale,
110 // but preserve existing country
111 Locale existingLocale = LanguageUtil.getViewLocale(request);
112 Locale newLocale = new Locale(newLang, existingLocale.getCountry());
113
114 HttpSession session = request.getSession();
115 session.setAttribute(Globals.LOCALE_KEY, newLocale);
116 mLogger.debug("Changed language to: "+newLang);
117

```

Figure 63 Roller false positive.

The Pebble project contains 36.709 lines of code and the amount of found vulnerabilities was 440. ASIDE and Lapse+ reported 315 and 258 warnings, respectively. From those warnings, 158 (or 50%) warnings from ASIDE and 139 (or 54%) warnings from Lapse+ were false positives. CodePro and ESVD reported 38 and 289 warnings; from those warnings, 11(or 29%) from CodePro and 14(or 5%) from ESVD were false positives. Once again, ESVD achieved the lowest/best rate of false positives.

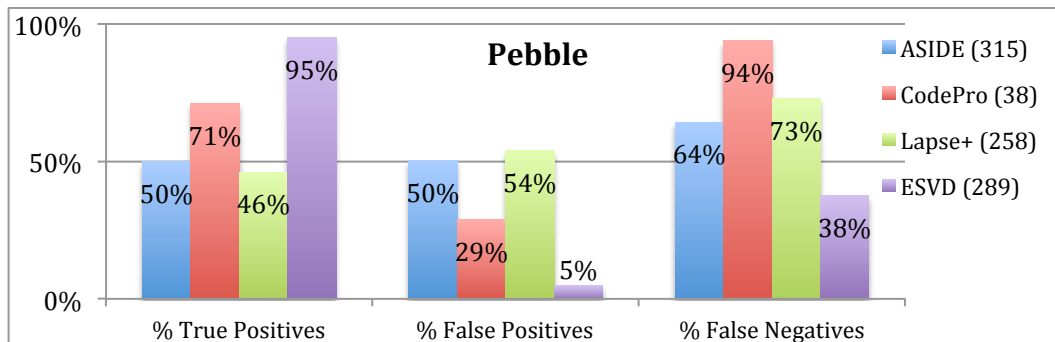


Figure 64 Pebble.

The source code of the Pebble project is the perfect example of the possible consequences of having different developers working on different tasks without having a well-defined security police. Figure 65 presents a code snippet, which the developer created and used his own sanitization method, called *filterHTML*. There is no problem with that. However, s/he only used it on some variables, leaving the others insecure. For instance, on line 68, variable *name* was not



sanitized and used on line 79. After that on line 72, variable *newName* was sanitized and used on line 79. If we analyze this code, it is possible to observe that it has a possible vulnerable path. However, the line 72 should not be flagged because the variable was indeed validated. Our heuristics failed to identify the method *filterHTML* because it was a custom made sanitization method not known by our heuristics.

```

66 public View process(HttpServletRequest request, HttpServletResponse response)
67     Blog blog = (Blog)getModel().get(Constants.BLOG_KEY);
68     String name = request.getParameter("name");
69     String oldName = StringUtils.filterHTML(name);
70     String type = request.getParameter("type");
71     String path = request.getParameter("path");
72     String newName = StringUtils.filterHTML(request.getParameter("newName"));
73     String submit = request.getParameter("submit");
74
75     FileManager fileManager = new FileManager(blog, type);
76     FileMetadata directory = fileManager.getFileMetadata(path);
77     try {
78         if (submit.equalsIgnoreCase("rename")) {
79             fileManager.renameFile(path, name, newName);
80

```

Figure 65 Pebble false positive.

The NCO project contains 6.048 lines of code and the amount of found vulnerabilities was 77. ASIDE and Lapse+ reported 38 and 82 warnings, respectively. From those warnings, 11(or 29%) warnings from ASIDE and 50(or 61%) warnings from Lapse+ were false positives. CodePro and ESVD reported 11 and 121 warnings; from those warnings, 7(or 64%) from CodePro and 74(or 61%) from ESVD were false positives. On the NCO project, ESVD achieved one of the worst rates of false positives, 61%.

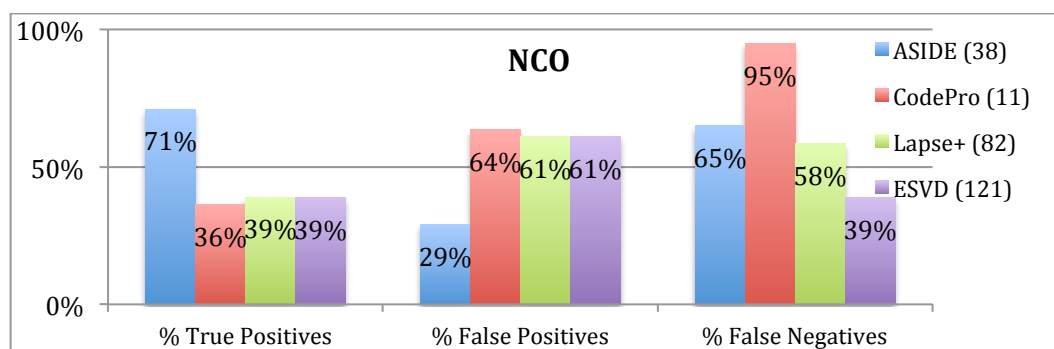
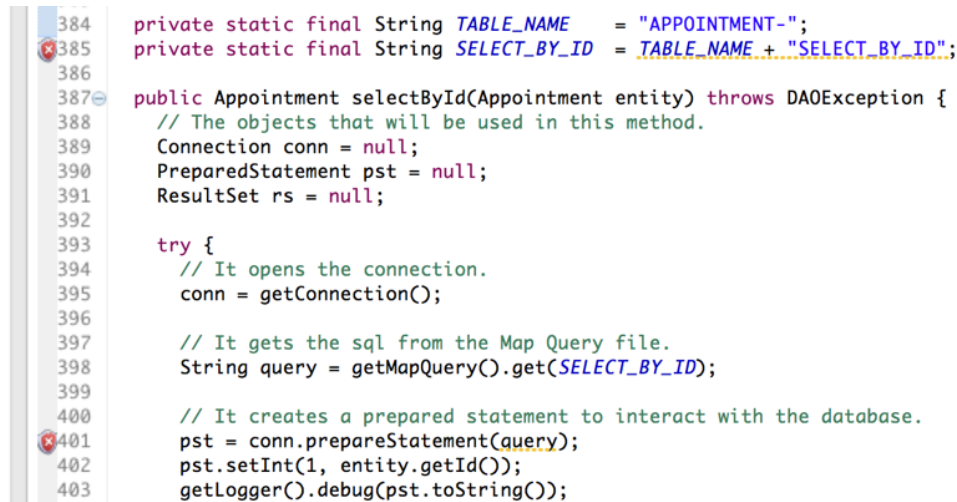


Figure 66 NCO.

NCO was created using the MVC design and one of the ideas of the author was to reuse as much code as possible. In order to do that, he created a series of constants that could be configured on each class and after that could be used always in the same way. As presented in Figure 67, every query was previously

created and whenever the developer wanted, s/he simply needed to invoke it. The problem with this approach was the string concatenation (line 385). Our heuristics search for concatenated queries that reach the *preparedStatements* or *statements* objects. Whenever that happens, the source code is flagged as vulnerable, because as explained in section 3.6.10, queries should not be concatenated. However, this is a case of false positive, because the code is concatenating two string literals created by the developer and that should not be considered vulnerable.



```

384 private static final String TABLE_NAME = "APPOINTMENT-";
385 private static final String SELECT_BY_ID = TABLE_NAME + "SELECT_BY_ID";
386
387 public Appointment selectById(Appointment entity) throws DAOException {
388     // The objects that will be used in this method.
389     Connection conn = null;
390     PreparedStatement pst = null;
391     ResultSet rs = null;
392
393     try {
394         // It opens the connection.
395         conn = getConnection();
396
397         // It gets the sql from the Map Query file.
398         String query = getMapQuery().get(SELECT_BY_ID);
399
400         // It creates a prepared statement to interact with the database.
401         pst = conn.prepareStatement(query);
402         pst.setInt(1, entity.getId());
403         getLogger().debug(pst.toString());

```

Figure 67 NCO false positive.

Figure 68 and the Table 9 present the compiled results from all the analyzed projects. Our research question RQ2, asked if *data flow analysis* could decrease the rate of false positives when compared to *pattern matching*. According to our experiments, our prototype using DFA with context sensitivity was able to achieve the final rate of 11,70% of false positives. CodePro using DFA with context insensitivity achieved 37,62% of false positives. Finally, the closest value from a pattern matching tool, was from Lapse+ with 44,73% rate of false positives.

Although we were able to successfully decrease the rate of false positives, this fact alone is not sufficient to state that DFA is a better approach than pattern matching, taking the CodePro numbers for example. It also had a lower rate of false positive when compared to the pattern matching tools. However, it had the lowest/worst rate of Recall and F-measure. This means that, although it did not have too many false positives it did not find a minimum amount of vulnerabilities either. The compilation of all results state that our prototype achieved 0,88 of precision, 0,66 of recall, resulting in a 0,75 score for the F-measure metric. In

other words, our prototype achieved the best results from all tools. The final and most important result was the 11,70% of false positive, that being the best from all tools and successfully answering our RQ2.

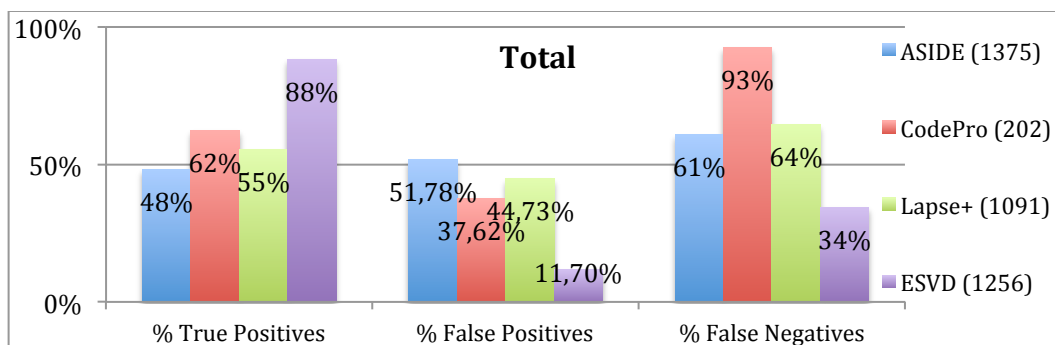


Figure 68 Compilation of results from all analyzed projects.

	Precision	Recall	F-Measure
ASIDE	0,48	0,39	0,43
CodePro	0,62	0,07	0,13
Lapse+	0,55	0,36	0,43
ESVD	0,88	0,66	0,75

Table 9 Compilation of results from all analyzed projects.

### 5.1.5.2. Memory and Time

Data flow analysis with context-sensitivity, as the name states, has the ability to analyze the flow of each object creation or method invocation. We created and implemented a special algorithm to compute these data flow properties. The algorithm has to *remember* what methods have been analyzed and what is the current reference or value of an object. This algorithm is much more complex than pattern matching algorithms, which only need to scan the source code once and compare if the code structure (being analyzed) matches a code template that usually represents a security vulnerability.

Our prototype was conceived with the impact of such difference on resources usage in mind. The impact of using a high amount of memory could be to slow down the IDE or even worst, shut down the whole system. Figure 69 presents the average (from the five executions) memory usage by all tools when analyzing the open-source projects. This information was collected using the *memory profiler* plugin from the Eclipse IDE. Although CodePro also performs data flow analysis, its DFA is context-insensitive. In other words, it does not remember the context of object and methods. This is the reason its memory usage did not increase alongside with the project size.

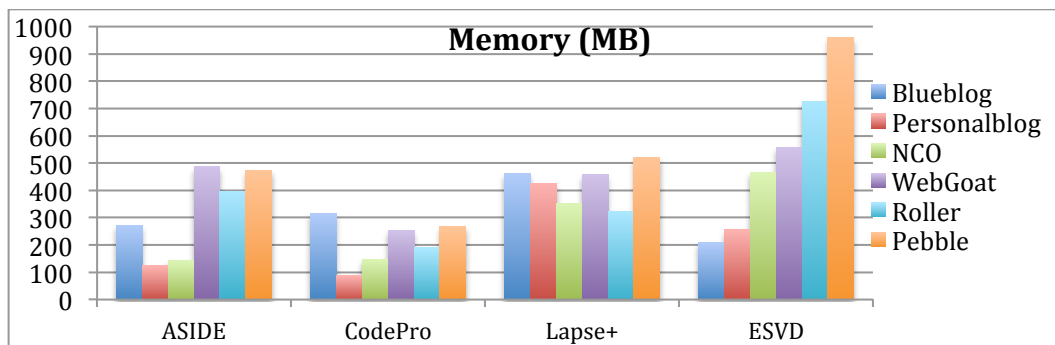


Figure 69 Memory Usage.

As presented in Figure 70, both tools using pattern matching performed the analysis in just a couple milliseconds, even in large projects such as Roller and Pebble. This information was collected using the *memory profiler* plugin from the Eclipse IDE, which besides showing the memory that is being currently used, it also show the time of one (or more) specific thread. The main benefit of using this technique is the fact that even if a method is invoked several times, it is analyzed once only. In other words, all classes and methods are scanned one by one in no particular order and just once. On the other hand, DFA needs to follow method invocation in order to find if a possible *entry-point* can reach an *exit-point* without being properly *sanitized*. Based on this characteristic of DFA, as the analyzed applications grew in size (lines of code, number of classes and methods) so did the time spent to analyze them. CodePro and ESVD, which implement DFA, went from a few milliseconds to proximally 10 and 6 minutes respectively.

On the first experiment, the largest (amount of classes and lines of code) project was Pebble containing 743 classes with 36.709 lines of code. However, there are real life projects that can be much bigger than this. Therefore, this might be a problem for DFA solutions. In our prototype, a mechanism was implemented

in order to ease this problem. The mechanism made use of the clean call graph and the prime call graph. As explained in section 4.2, the first time our prototype runs it creates the call graph (method interactions) of the source code. This call graph is then passed to the data flow analysis algorithm. After that, every time the developer changes one class, only this class and the classes it interacts to are re-scanned. Thereby, significantly decreasing the total time of analysis.

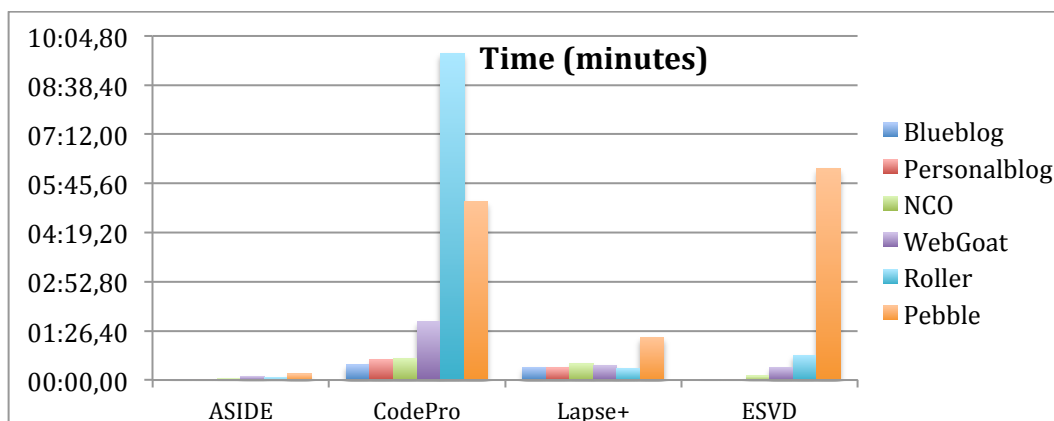


Figure 70 Execution time.

## 5.2.

### Study 2: Late vs. Early Detection–A Quasi-Experiment

We also performed a controlled experiment to observe if developers, who receive support for early vulnerability detection, are able to produce more secure software than those who receive late detection support. We analyzed if the use of early detection reduced (or increased) the number of security vulnerabilities when compared with the use of late detection. To do that, we could not simply inject vulnerabilities into a source code and ask participants to search for them. Thus, the experiment was carefully designed to increase the chances of making developers create security vulnerabilities on their own, while developing typical programming tasks (section 5.2.1). The complete description of the programming tasks, questionnaires and results from the experiment can be download from our study website [Sampaio 2014c].

### **5.2.1. Methodology**

We designed and executed a quasi- experiment in order to observe if early detection outperforms (or not) late detection in terms of encouraging developers to address vulnerabilities in their source code. The hypothesis tested in the experiment was the following: “H1: Early detection helps developers to produce more secure code when compared to the late detection approach”.

In order to confirm or refute this hypothesis, we performed a controlled experiment to observe if developers, who receive early support in code editing, were able to produce more secure software than those developers who receive late support. In order to avoid biased results from using different tools that performs early detection and late detection, we decided to use our prototype (i.e. the same tool) on both parts of the experiment. The first step of the experiment was the creation of the coding exercise (Section 5.2.1.1). After that, we created two groups where the participants were assigned. Each participant was assigned to one of the two groups (Section 5.2.1.2).

#### **5.2.1.1. Coding Exercise**

The coding exercise was composed of five programming tasks (see Table 10). Each task was specifically created with the intention of exposing participants to situations where they would introduce (by their own) security vulnerabilities in their source code. The tasks were also independent from each other and participants could choose which tasks they would like to implement first. As we did not want participants to waste time creating HTML pages, we already provided a project containing all basic files. The participants were only requested to create the Java source code to handle these pages.

The first task was to create a login page where a user could provide her/his credentials (e.g. login and password) and then login in the application. This task opens the opportunity for several security vulnerabilities. Because the developer has to handle credentials (security misconfiguration), connection with the database (sql injection), data being sent back and forth from the server to the browser (cross-site scripting) and so forth.

The second task was to create a page where all the comments stored in the database from the application could be displayed to the user. To perform the task the participant had to handle connections with the database, handle sessions or cookies to store information about the logged in user. Therefore, several vulnerabilities could be created.

The third task was to create a page where the user could see all comments created by her/him, and then select one or more and delete them. Usually developers create delete pages by passing the *ids* on the URL. In this situation is common that if the developer does not properly verify the identity of the user who is deleting the data, an attacker can provide other *ids* and delete data from other users.

The fourth task was to create a page where comments could be added by the user and saved into the database. If not properly implemented, this page could allow attackers to insert malicious content into the database, perform SQL injection, cross-site scripting and so forth.

The fifth and final task was to create a page where a user logged in as an administrator could see all stored comments and could select and delete any of them. To perform this task, the participant would have to create an access control check in order to verify if the user is a normal user or an administrator. This leads to several types of vulnerabilities, such as unauthorized access, cookie poisoning, SQL injection and so forth.

Nr.	Description
1	Create a login page.
2	Create a page where all comments stored in the database are displayed.
3	A page where each user can delete its own comments.
4	A page where a user can add comments.
5	Create a special user (administrator), who can delete any comment.

Table 10 Description of the tasks of the coding exercise.

The author of this dissertation tried himself to execute all the five tasks. He needed 37 minutes to complete all these tasks. A task was considered completed, only if the page was working as expected. Therefore, we estimated that 90 minutes for each subject would be more than enough to participate in our experiment. However, the six companies, which allowed their developers to

participate, explicitly constrained the time their employees could spend in our experiment. They mentioned that 90 minutes was too much and they allowed their developers to participate in our experiment for the maximum of one hour. Therefore, we recalculated the amount of time dedicated for each part of the experiment. As each participant could spend one hour, we could not design our experiment in a way the same subject would participate in both groups, i.e. performing some programming tasks with early detection and other programming tasks with late detection. This experiment design would be more complex and would also require more time from the participants to complete all the tasks. An important information is that some participants stated that they would not mind to stay until they could finish all tasks. Therefore, some participants performed the experiment for more than one hour. More information about that will be provided on section 5.2.2.

#### **5.2.1.2. Early and Late Detection Groups**

The participants were divided in two groups, namely *early detection group* and *late detection group*. The main difference between them was how the participants were allowed to use our prototype. For the early detection group, they would have one hour to complete all the tasks, they were allowed to activate our plugin at the beginning of the experiment and receive its support throughout the whole time. For the late detection group, we enforced they were following the typical behavior of late detection, which is encouraged by most of the existing security detection tools (section 2.5). Then, they had 40 minutes for the programming tasks; after that, they had 20 minutes to run the prototype and fix or ignore the vulnerability warnings.

We tried to find participants with at least some basic knowledge of secure programming. The distribution of the 27 developers, from which 11 are professionals and 16 are undergrad students (or novice programmers) in these two groups was not random. Instead, we wanted to have an equal number of professionals and students on each group. As the number of subjects was not very high, we did not want to favor one group over the other in terms of programming



expertise. Therefore, in order to characterize the expertise of the participants, they had to answer a questionnaire [Sampaio 2014c] related to their background.

The selection of the participants for the experiment relied on two explicit criteria. First, the participant had to have some degree of knowledge on Java web development (JSP, JSF or any other Java web technology), because the tasks were based on these technologies. Second, they should have worked with Eclipse before, because we did not want to have participants wasting time trying to find how to do something on Eclipse and our prototype is built as an Eclipse plugin. These developers were found by different ways, such as: companies contacted by the author of the dissertation, Twitter<sup>19</sup> and LinkedIn<sup>20</sup>. We contacted dozens of companies, from which six allowed their developers to participate on our experiment. Through Twitter and LinkedIn, it was possible to find seven international developers: two of them to participate in the experiment, and five of them to provide feedback about our tool. Some developers did not want to or had time to spend on the experiment, but watched our online video on how to use the ESVD plugin [Sampaio 2014d] and provided us valuable feedback.

#### **5.2.1.3. Experiment setup**

The participants were informed they were participating on a study to evaluate a prototype plugin that would check the quality of the source code. We also informed them they could choose to read the warnings and try to fix them (if any) or simply ignore them. The participants did not receive any training on how to interact with our plugin because every message or page of the plugin has security keywords on them. The training would tell them we were analyzing security vulnerabilities and it was not our intention to change their normal behavior. In other words, we did not want to *force* them to think about security only because of their participation in our experiment.

When the experiment was being designed, there were two main concerns in mind: the quality of the resulting data and how to increase the number of participants. Based on this, we created a list of necessary software that remote

---

<sup>19</sup> <http://www.twitter.com>

<sup>20</sup> <http://www.linkedin.com>

participants could install in order to participate. The installation process was performed by the participants prior the experiment. In other words, the time they took to install all the required software systems was not part of the time from the experiment. These required systems were Eclipse<sup>21</sup> Kelper (4.3) or Luna (4.4 - the latest one), our plugin (ESVD<sup>22</sup> - Early Security Vulnerability Detector), and any DBMS<sup>23</sup> where data from the experiment could be stored. We recommended MySQL<sup>24</sup> and provided all initial scripts in order to create the schema, users and tables. The last software was a screen recorder. For Mac, we used and recommended a trial version of ScreenFlow<sup>25</sup>; for Windows, we used a free version of CamStudio<sup>26</sup>. The video of the participants provided a wide variety of valuable information. For instance, we could observe: (i) how the participants interacted with the warnings, and (ii) how long they took to remove vulnerabilities from the source code and much more information.

### **5.2.2. Study 2: Results**

This section presents the results observed from the second experiment and is structured as follows. Section 5.2.2.1 presents the number of participants on each group and their years of experience. Section 5.2.2.2 describes the amount of time spent by each group in the experiment. Section 5.2.2.3 presents the amount of finished tasks by each participant. Section 5.2.2.4 discusses the number of vulnerabilities added, removed and ignored by the participants. Section 5.2.2.5 presents the average amount of time until a vulnerability is inserted into the source code. Section 5.2.2.6 discusses the threats to the validity of our study. Finally, section 5.3 presents our conclusions about results gathered in the experiment.

---

<sup>21</sup><https://www.eclipse.org>

<sup>22</sup><https://marketplace.eclipse.org/content/early-security-vulnerability-detector-esvd>

<sup>23</sup>Data Base Management System

<sup>24</sup><http://www.mysql.com>

<sup>25</sup><http://www.telestream.net/screenflow/overview.htm>

<sup>26</sup> <http://camstudio.org>

### 5.2.2.1. Participant Characterization

The experiment was open to participation for a period of 30 days. In that time, we were able to find 27 developers, from which 11 are professionals and 16 are undergrad students (or novice programmers). As already mentioned (section 5.2.1.2), we wanted to have a similar number of professionals and students on each group. Table 11 presents the number of participants and the average of years of experience on development (not necessarily on secure programming) for the corresponding participants in each group. After they answered the questionnaire about their expertise, they were assigned to either the *early detection* or *late detection group*. We managed to have six professionals in the early group and five professionals in the late group. Although the number of participants in each group was also the same, the average of years of experience on the late detection group was almost double than the early detection group. Therefore, the strategy we used to divide the participants was not equitable from this perspective. On the other hand, this division somehow favored the late detection group rather the early detection group. In addition, we managed to have eight students in both groups and their average of experience were almost the same..

	Early		Late	
	Quantity	Years of experience	Quantity	Years of experience
Professional	6	5,8	5	9,4
Student	8	2,0	8	1,75

Table 11 Number of participants and average of years of experience.

After the division, all participants received the written instructions. These instructions mainly described the five tasks they had to perform. After the period of one hour, they were informed about the end of the time. However, we decided we would allow the participant to continue in case he/she explicitly requested, independently if he/she was in the early detection or late detection group. Our understanding was that they were still eager and motivated to perform their programming and vulnerability detection tasks. Then, when they were satisfied with their tasks, the produced source code and screen recording were requested from the participant. The time spent by each participant will be presented later. Several (8) participants stated they would like to have more time in order to try to finish all tasks and 7 participants finished the experiment in one hour or less.

On the other hand, some (9) other participants stated they were not comfortable delivering their source code to us, because “*there was not much done*” or because “*the code contained too many bugs*”. We reminded them that the experiment was private and no source code would be shared. This argument was not enough to convince them and we had to discard those participants. Therefore, Table 12 presents the final numbers, containing only participants that, at the end of the experiment, delivered the source code to us. The total amount of participants went from 27 to 18 developers, from which 10 are professionals and 8 are students (or novice programmers). From the participants who decided not to deliver the source code, one was a professional and eight were students.

	Early		Late	
	Quantity	Year of experience	Quantity	Year of experience
Professional	6	5,8	4	10,8
Student	2	3,5	6	1,7

Table 12 Final number of participants and average of years of experience.

Table 13 presents the final distribution of the participants on each group. Column 1 represents the unique id of each participant, because no names were requested during the experiment. Column 2 presents, for the corresponding participant (in Column 1), which of the two groups of the experiment – i.e. early and late detection group – he/she took part. Column 3 informs if the participant was a professional or a student. The final numbers were: (i) 8 participants in the early group, from which 6 are professionals and 2 are students, and (ii) 10 participants in the late group, from which 4 are professionals and 6 are students. Therefore, the difference in number of participants between the two groups was not significant.

Part. Id	Group	Type
1	Early	Professional
2	Late	Professional
3	Early	Professional
4	Late	Professional
5	Early	Professional
6	Late	Professional
7	Late	Student
8	Early	Student
9	Late	Student
10	Late	Student
11	Late	Student
12	Late	Student
13	Late	Student
14	Early	Professional
15	Early	Professional
16	Early	Professional
17	Early	Student
18	Late	Professional

Table 13 Distribution of the participants on each group.

#### 5.2.2.2. Programming Time per Group

Table 14 presents the total amount of hours spent by participants on the experiment. Column 2 represents the total amount of hours spent by professionals and students on the early detection group. Similarly, Column 3 presents the total amount of hours spent by professionals and students on the late detection group. The total amount of hours was 18 hours and 34 minutes. From those hours, 11 hours and 16 minutes were spent by the early detection group, while 7 hours and 17 minutes were spent by the late detection group. The difference in between the two groups is approximately 4 hours. Therefore, it was expected that the early detection group would produce more source code and vulnerabilities than the other group. However, in order to be fair and equitable, we will consider, for instance, the proportion of vulnerabilities introduced and fixed rather than absolute measures. Professionals on both groups worked several hours more than students.

	Early	Late
Professional	9:32:40	4:31:07
Student	1:44:18	2:46:15
Partial	<b>11:16:58</b>	<b>7:17:22</b>
Total	<b>18:34:20</b>	

Table 14 Programming time (hours) per group.

### 5.2.2.3.

#### Programming Time per Participant and Performed Tasks

Table 15 presents the list of participants, their experiment time and the number of tasks completed by each one of them. From this table, the first observation is that three participants (7, 11 and 13) had their experiment time as 00:00:00. Unfortunately, this happened because the screen recorder crashed when they tried to save the file, thereby losing all evidence of the amount of time they spent on the experiment and how they interacted with our prototype. Additionally, because they were remote participants, we were not able to help them. On the other hand, they executed either one or two programming tasks.

Another observation is the fact that, although the experiment was supposed to last one hour, some participants decided to end it earlier. When we asked them why, some stated that developing for one hour straight was exhausting. On the other hand, participants 3, 5, 6, 10, 14, 15, 17 and 18 asked for more time. All 18 participants were able to finish at least one task. Additionally, eight participants finished two tasks, four participants finished three tasks, two participants finished four tasks and finally only two participants were able to finish all five tasks.

Part. Id	Experiment Time	Task 1	Task 2	Task 3	Task 4	Task 5
1	00:48:00	1	0	0	0	0
2	00:52:43	1	0	0	0	0
3	02:04:10	1	1	0	0	0
4	00:54:51	1	0	0	0	0
5	01:20:15	1	0	0	0	0
6	01:17:59	1	1	1	0	0
7	00:00:00	1	0	0	0	0
8	00:32:21	1	0	0	0	0
9	00:32:18	1	0	0	0	0
10	01:16:43	1	1	0	0	0
11	00:00:00	1	1	0	0	0
12	00:57:14	1	1	0	0	0
13	00:00:00	1	0	0	0	0
14	01:04:54	1	0	0	0	0
15	03:18:42	1	1	1	1	1
16	00:56:39	1	1	0	0	0
17	01:11:57	1	0	1	0	0
18	01:25:34	1	1	1	1	1
	<b>18:34:20</b>	<b>18</b>	<b>8</b>	<b>4</b>	<b>2</b>	<b>2</b>

Table 15 Experiment time (hours) and tasks performed by each participant.

#### 5.2.2.4.

#### Vulnerabilities Added, Removed and Left

Table 16 presents the number of vulnerabilities added, removed and left by each participant. What is important to notice is the fact that all participants but one added at least one vulnerability into their source code. Participant 2 did not add vulnerabilities because of two factors. First, he created a few lines of source code only. Second, he knew that *PreparedStatement* should be used when interacting with the database. Therefore, he avoided the creation of a SQL injection. There was also a trend that more vulnerabilities were introduced by participants that spent more time programming. For instance, participants 3 and 15 worked for two and three hours, respectively, and both created eight vulnerabilities each in their source code.

Part. Id	Experiment Time	Added	Removed	Left
1	00:48:00	1	1	0
2	00:52:43	0	0	0
3	02:04:10	8	2	6
4	00:54:51	2	0	2
5	01:20:15	5	2	3
6	01:17:59	2	0	2
7	00:00:00	2	0	2
8	00:32:21	2	0	2
9	00:32:18	2	0	2
10	01:16:43	2	0	2
11	00:00:00	2	0	2
12	00:57:14	2	0	2
13	00:00:00	3	1	2
14	01:04:54	5	0	5
15	03:18:42	8	4	4
16	00:56:39	4	1	3
17	01:11:57	2	2	0
18	01:25:34	5	1	4
-	<b>18:34:20</b>	<b>57</b>	<b>14</b>	<b>43</b>

Table 16 Number of vulnerabilities added, removed and left.

The amount of vulnerabilities that were added, removed and left in the delivered source code is presented on Table 17. During the 18 hours and 34 minutes of experiments, the plugin was able to detect 57 security vulnerabilities in the source code of the participants, from which 35 (or 61,4%) were added by participants from the early group and 22 (or 38,6%) from the late group. Although the participants from the early group added more vulnerabilities than the other group, this is partially justified by the fact that they worked for 11 hours and 16 minutes while the late group worked only for 7 hours and 17 minutes.

Another important information from Table 17 is that although 57 vulnerabilities were detected by our prototype, only 14 (or 24%) vulnerabilities were removed (fixed) from the source code. This is a quite low percentage. If this was a real (and single) software project, the application would be under serious risks. From the vulnerabilities that were removed, developers receiving early support were able to remove 12 (or 34,2%), while developers receiving late support only removed 2 (or 9,09%). Therefore, the early detection approach was able to encourage programmers to remove more vulnerabilities than the late detection approach.



The final observation from the Table 17 is the amount of vulnerabilities left unhandled in the source code. The total amount was 43, from which 23 (or 53,5%) were left by participants from the early group and 20 (or 46,5%) from the late group. However, we cannot confirm or refute our hypothesis H1 based only on this column. In order to be fair with both groups, it is more appropriate to analyze the amount of added vulnerabilities divided by the amount of left vulnerabilities of each group. In other words, the early detection group added 35 vulnerabilities and left 23 unhandled (or 65,7%). On the other hand, late group added 22 vulnerabilities and left 20 unhandled (or 90,9%). Based on these observations, there is an indication that our hypothesis H1 that states that early detection helps developers to produce more secure code when compared to the late detection approach is true.

	Added		Removed		Left	
	Early	Late	Early	Late	Early	Late
Professional	31	9	10	1	21	8
Student	4	13	2	1	2	12
Partial	35	22	12	2	23	20
Total	<b>57</b>		<b>14</b>		<b>43</b>	

Table 17 Vulnerabilities added, removed and left during the experiment.

Table 18 describes in detail the vulnerabilities that were added, removed and left on the delivered source code during the experiment. Misconfiguration was the most common vulnerability, appearing 27 times. Several participants stated they hardcoded the username and password of the database in the source code, because it would not be possible to create the complete infrastructure to store this information in an encrypted file during the experiment time. Although this is a reasonable explanation, we did not consider this as a false positive, because indeed it is how this security vulnerability typically finds its way in mainstream software projects [OWASP 2013e]. Log forging was the vulnerability that was removed more times. The screen recording helped us understand the reason behind this trend. The reason was debugging, in other words, developers usually log what they are doing in order to know if the code is working properly. In some of these occasions, the code was actually logging untrusted data and the plugin correctly identified the vulnerability. However, after verifying that the code was working as expected, the developers just deleted the code statement and, therefore, removed the vulnerability.

Vulnerability	Added	Removed	Left
HTTP Response Splitting	1	1	0
Cookie Poisoning	2	0	2
SQL Injection	3	1	2
Log forging	10	6	4
Cross-Site Scripting	14	3	11
Misconfiguration	27	3	24
<b>Total</b>	<b>57</b>	<b>14</b>	<b>43</b>

Table 18 Security vulnerabilities reported during the experiments.

#### 5.2.2.5.

#### Average Time for New Vulnerabilities

The total amount of hours spent by participants on the experiment and the total amount of vulnerabilities that were added during the experiment were presented on Table 14 and Table 17, respectively. Therefore, if we divide the amount of hours by the amount of added vulnerabilities, we will obtain the results presented on Table 19, which means the average time it took for a vulnerability to be inserted into the source code. For this measure, both groups have an average close to 22 minutes. In other words, a new vulnerability was added into the source code every 22 minutes. If we imagine that developers work 8 hours per day, we could estimate 21 vulnerabilities are added every single day by a single developer. Even though this is simply an estimate, we believe this information corroborates with our claim that developers should received tooling support as early as possible. Otherwise, vulnerabilities could be left unhandled in the code, eventually reaching the production environment.

When we were analyzing the screen recording of the participants from the late detection group, it was possible to observe several cases where by the time developers discovered that their code contained one or more vulnerabilities, they had already finished one or more tasks. Therefore, they had to return to previous tasks, remove the vulnerable code and re-implement the requested functionality. Based on this fact, it became clear to us that, if developers receive early vulnerability detection support, they no longer will have to waste time redoing work that could have been done potentially quicker and once.

	Early	Late
Professional	0:18:28	0:30:07
Student	0:26:05	0:12:47
Average	<b>0:22:16</b>	<b>0:21:27</b>

Table 19 Average time (hours) until a vulnerability was added.

#### 5.2.2.6. Threats to Validity

Every empirical study has threats to validity that need to be addressed. This section discusses key threats to validity relevant for this study, and actions we have taken in order to reduce their impact.

**Total time for the experiment:** When we designed the experiment and its tasks, we had in mind that developers would have enough time available to finish all tasks. However, most companies only allowed their developers to participate for the maximum of one hour. As a consequence, only a few participants were able to complete all tasks from the experiment in one hour, what could reduce opportunities for introducing vulnerabilities and/or discouraging their handling (due to lack of time). However, in order to try to mitigate this problem, we decided to accept the request of several participants for more time. Then, several participants could naturally complete more tasks.

**Number of participants:** For a period of 30 days, we tried to find developers through the Internet and our personal contacts. However, our population in the experiment was constrained to 18 subjects, out of 27 subjects that started to participate in the experiment. Even though someone could consider our population is limited, we did our best to involve both experienced and novice programmers in both groups (using early and late detection). In addition, many experiments in software engineering have even fewer participants than in our experiment due to the difficulty in finding volunteers. Finally, we run our experiment remotely in order to increase the participation of professional developers (but this leads to other threats discussed in the following).

**Remote participants:** In order to increase the number of participants, we prepared the experiment in a way it could be applied remotely. Therefore, we did not have full control over the subjects. As they are developing their tasks at a distance, the participants could do other things not related to the experiment. As a consequence, we cannot fully ensure our time-spent measures were fully reliable.

In order to try to mitigate this threat, we recorded the video and participants' actions. As far as we could observe, all the participants were fully dedicated to execute only the experiment tasks. A few remote participants had problems that nobody else had, e.g. the screen recorded crashing, slow machines and so forth. However, fortunately, only a few participants faced these problems.

**Skills of the participants:** When we were selecting the participants, we tried to find developers with at least some basic knowledge of secure programming. Otherwise, for instance, if our tool reported a SQL injection, the participant would not know anything about it. However, it was possible to watch on the screen recording that a considerable number of participants tried to find basic information at the Internet about the reported vulnerabilities, so that they could fix them. On the other hand, we observed the frequency of this behavior was similar across the early and late detection groups.

### 5.3. Concluding Remarks

During our second study, it was possible to notice that even developers with several years of development experience had very limited knowledge about security vulnerabilities. This idea was further observed by (i) the amount (57) of vulnerabilities added into the source code, and (ii) the number (43) of vulnerabilities left unhandled on the delivered source code. Even though our prototype correctly identified all 57 vulnerabilities, some participants tried to find a solution for a couple minutes and then gave up. When these participants were asked why they left vulnerabilities unresolved in the source code, a few of them justified their negligence because they knew they were participating in an experiment and, therefore, the vulnerabilities would not cause any actual harm. However, these same participants stated that if that had happened on their workplace, they would have tried harder.

Fortunately, as observed in our first study, our DFA-based approach presented a much higher accuracy than other existing DFA and pattern-matching approaches. This means that, at least, our approach would be more effective if developers were deeply concerned in addressing security vulnerabilities. In addition, our DFA-based approach did not cause significant overhead in terms of

memory and execution time. In addition, in the second experiment, no participant reported about such overhead problems. Even though the produced programs were not large, the programs used in the first study were of reasonable size.

In the controlled experiment, the participants, who received the detection support from our tool, were constantly aware of the vulnerabilities that they were adding into the source code. On the other hand, participants from the late group were unconscious about the security vulnerabilities emerging in their code until the 40 minutes mark. Only after that, they started searching for help and removing vulnerable code. Amongst many observations, we noticed that developers receiving early detection support were able to remove 12 (or 34,2%) vulnerabilities, while developers receiving late support only removed 2 (or 9,09%). Based on the reported results, the early detection approach tended to encourage programmers to remove more vulnerabilities than the late detection approach. For the sample of our, it is possible to observe our hypothesis H1 (in the second study), which stated, “Early detection helps developers to produce more secure code when compared to the late detection approach”.

## 6 Conclusion

One of the main goals of software engineering is to create dependable software, i.e. software that users can trust. To achieve this goal, it is necessary that software systems provide their services properly even if being under attack by malicious people or programs. To this end, developers should be aware of security vulnerabilities when creating their software. Developers without accurate tooling support struggle to perform secure programming and, in particular, are not able to find neither to fix security vulnerabilities from their source code.

In this context, we studied the state-of-the-art on security vulnerability detection in source code. We also proposed the combination of two ideas. First, we proposed to support a change from the default behavior of late detection to early detection. We believed and were able to observe that early detection can provide better support for secure programming. Second, we also proposed a vulnerability detection solution based on a particular variant of data flow analysis. We evaluate two static analysis techniques – i.e. *pattern matching* and *data flow analysis*– to support vulnerability detection in several industry-strength projects.

According to the results of our first study, our prototype achieved the lowest rate of false positives, i.e. 11,70%. For this measure, the best result from a pattern-matching tool was 44,73%, which was much worse than in our DFA approach. From the results obtained in the study, it was also possible to observe that the strict use of false positives is not enough to assess the real support of a tool for secure programming. For instance, when analyzing the CodePro measures, for example, it also had a much lower rate of false positives when compared to the pattern matching tools. However, it had the lowest/worst rate of Recall and F-measure. Although it did not have too many false positives, it did not find a minimum amount of vulnerabilities either. The compilation of all results state that our prototype achieved 0,88 of precision, 0,66 of recall, resulting in a 0,75 score for the F-measure metric. The best results from all other tools.

The second part of our proposal evaluation wanted to observe if early detection could encourage (or not) developers create more secure software. We were able to observe that, during the 18 hours and 34 minutes of experiments, our prototype was able to detect 57 security vulnerabilities in the source code of the participants. Developers receiving early support added 35 vulnerabilities and were able to remove 12 (or 34,2%) vulnerabilities, while developers receiving late support added 22 and only removed 2 (or 9,09%). Therefore, the early detection approach was able to help developers remove more vulnerabilities than late detection.

## 6.1. Contributions

The fact that novice and experienced developers need support to perform secure programming is no secret. Thus, in our study we observed that if this support is provided as early during programming (rather than posterior analysis) as possible, the chances of security vulnerabilities reaching the deployed software system may decrease substantially. However, in order to be truly successful on this support, we confirmed that early detection by itself is not sufficient if the technique used to find security vulnerabilities has a high rate of false positives. In this context, the main contributions of our study were:

1. The heuristic strategies capable of finding 11 security vulnerabilities that stem from input and output not being properly sanitized. Each heuristic has the knowledge on how to identify a source code as vulnerable according to a specific vulnerability. The advantage on using this approach is the fact that heuristics can be added or removed without interfering with the other heuristics. Additionally, our proposed heuristics can be adapted and implemented to the context of other programming languages.
2. Proposal and implementation of the algorithm of *data flow analysis* with context sensitivity to find security vulnerabilities (section 3.1). The proof-of-concept is a free Eclipse plugin for the Java programming language that performs the detection of security vulnerabilities while the developer is adding/editing the source code. The plugin can be downloaded from the Eclipse Marketplace [Sampaio and Garcia 2014].

3. The last contribution is a complete list with known security vulnerabilities (ground truth) for each of the analyzed open-source projects. We consider this reference list of all the vulnerabilities found in a benchmark as a contribution for other researchers and tool developers. They can rely on this reference list to replicate our studies and produce new experimental evaluations. We used the same versions of the open-source projects on our benchmark as previous studies did. Because we wanted to be able to compare our results against theirs. However, as stated in section 5.1.2, those results were found nowhere. Because of this project, we had to create our own list of known vulnerabilities for each project. We made our list available and it can be downloaded from our study website [Sampaio 2014b]. Thus, future studies and benchmarks can use our results instead of having to perform manual inspection, as we had to perform in our research.

## 6.2.

### Future work

The results obtained and contributions presented were only a first step towards the goal of helping developers create more secure software. Although our proposed solution presented the lowest rate of false positives (section 5.1.5) when compared to the other analyzed solutions (section 2.6), there are still several aspects to be improved on it. Some of these aspects are the following ones:

1. Our technique of *data flow analysis* supports 11 types of security vulnerabilities (section 3.6) and in the benchmark experiment presented better results when compared to other existing solutions. However, there are still elements, such as: *Containers*, *Reflection* and *InnerClasses*, that our technique does not explicitly take into consideration. Therefore, generating false negatives. More research and development is still necessary to further improve the accuracy of our technique.
2. The memory usage of our prototype was already expected to be higher than the other existing solution. One of the reasons is the nature of *data flow analysis* that follows every method invocation and the ability (context sensitivity) to distinguish different instances of the same class. However, we believe that optimizations on the algorithm of our prototype can be made in



order to consume less memory. On our current version, every time a new instance from a class is created, a new context containing all the fields and methods from that class is also created. One possible optimization could be instead of creating contexts with all fields and methods, create with only the ones that have actually been used in the code. Thus, the amount of memory used on contexts would be decreased.

3. The possibility to allow developers to add, edit or remove methods from the lists of *entry-points*, *exit-points* and *sanitization-points*. Although our heuristics already contains 268 methods (more than any of the other compared tools) in these lists. Each developer or company might have other methods they consider vulnerable or have implemented their own sanitization methods. As these lists must be often updated in a vulnerability detection tool, we consider the presence of this extension feature as very important. From all the analyzed tools (section 2.6), CodePro Analytics was the only one to have this feature. The implementation of our prototype is already prepared to this functionality. However, because of lack of time the user interface where developers would be able to add new methods or remove old ones was not created.
4. The current version of our prototype does not prioritize one vulnerability over another. In other words, the vulnerabilities are presented in the same order as they are found in the source code. However, developers usually have a limited amount of time and would not be able to remove all vulnerabilities on the first time. Therefore, a ranking system where vulnerabilities could be organized or prioritized based on some criteria, would greatly help developers decide which types of vulnerabilities they want to remove first. OWASP already has a ranking system that we could use [OWASP 2013p]. Their ranking system sorts the types of vulnerabilities from the most critical to the least critical, in terms of estimates of exploitability, detectability and impact.
5. Currently, our prototype is only able to inform developers that there are security vulnerabilities in her/his software system. However, we believe the next step towards helping developers produce more secure software is to help them remove the vulnerabilities. In order to do that, it would be necessary to add support for (semi-)automatically fixing the source code whenever vulnerabilities are found.

## 7 References

Albuquerque, D., Garcia, A., Oliveira, R. and Oizumi, W. (2014). Detecção Interativa de Anomalias de Código: Um Estudo Experimental. *WMOD2014*,

An Overview Of Vulnerability Scanners (2008). *The Government of the Hong Kong Special Administrative Region*, p. 16.

Apple (2013). Introduction to Secure Coding Guide. <https://developer.apple.com/library/mac/documentation/security/conceptual/SecureCodingGuide/Introduction.html>, [accessed on Nov 10].

Artho, C. and Biere, A. (2005). Combined static and dynamic analysis. In *Electronic Notes in Theoretical Computer Science*. <https://staff.aist.go.jp/c.artho/papers/466.pdf>.

Baca, D. (2009). Automated Static Code Analysis - A tool for early vulnerability detection. Blekinge Institute of Technology.

Baca, D., Carlsson, B. and Lundberg, L. (2008). Evaluating the cost reduction of static code analysis for software security. *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security - PLAS '08*, p. 79.

Barbosa, E. A., Garcia, A. and Mezini, M. (jun 2012). A recommendation system for exception handling code. In *2012 5th International Workshop on Exception Handling (WEH)*. IEEE. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6226601>, [accessed on Dec 9].

Bennetts, S. (2012). OWASP Zed Attack Proxy Project.

[https://www.owasp.org/index.php/OWASP\\_Zed\\_Attack\\_Proxy\\_Project](https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project).

Blyth, A. (jul 2004). Innocent code: a security wake-up call for web programmers. *Infosecurity Today*, v. 1, n. 4, p. 249.

Bolour, A. (2003). Notes on the Eclipse Plug-in Architecture.

- [https://www.eclipse.org/articles/Article-Plug-in-architecture/plugin\\_architecture.html](https://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html), [accessed on Nov 21].
- Brown, S. (2006). Pebble. <http://pebble.sourceforge.net/>, [accessed on Apr 3].
- Burén, R. (2003). BlueBlog. <https://sourceforge.net/projects/blueblog/>, [accessed on Apr 3].
- Burp Suite ([S.d.]). <http://www.portswigger.net/burp/>.
- Chess, B. and McGraw, G. (2004). Static analysis for security. *Security & Privacy, IEEE*, v. 2, n. 6, p. 76–79.
- Cowley, S. (2001). Code red costs could top \$2 billion. <http://www.pcworld.com/article/57744/article.html>, [accessed on Nov 17].
- Deacon, J. (2009). Model-view-controller (mvc) architecture. ... *de 2006.* <http://www.jdl.co.uk/briefings/MVC.pdf>, p. 1–6.
- Dehlinger, J., Feng, Q. and Hu, L. (2006). SSVChecker. <http://ssvchecker.sourceforge.net/>, [accessed on Nov 10].
- Dillig, I., Dillig, T. and Aiken, A. (2011). Precise reasoning for programs using containers. *ACM SIGPLAN NoticesEclipse* ([S.d.]). <https://eclipse.org/home/index.php>.
- Eclipsesource, M. K. (2009). The Eclipse Packaging Project and its Usage Data Collector in RAP and RCP Applications Agenda ... n. April.
- Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. (1999). Refactoring: Improving the Design of Existing Code. *Xtemp01*, p. 1–337.
- Geer, D. (2005). Eclipse becomes the dominant Java IDE. *Computer*, v. 38, n. 7, p. 16–18.
- Google (2001). CodePro Analytix. <https://developers.google.com/java-dev-tools/codepro/doc/>, [accessed on Oct 11].
- Grossman, J. (2013). Whitehat website security statistics report. [https://www.whitehatsec.com/assets/WPstatsReport\\_052013.pdf](https://www.whitehatsec.com/assets/WPstatsReport_052013.pdf), [accessed on Nov 26].
- Guarnieri, M., El Khoury, P. and Serme, G. (2011). Security Vulnerabilities Detection and Protection Using Eclipse. In *Proceedings of ECLIPSE-IT 2011*.

Halfond, W. G. J., Viegas, J. and Orso, A. (2008). A Classification of SQL Injection Attacks and Countermeasures. *PREVENTING SQL CODE INJECTION BY COMBINING STATIC AND RUNTIME ANALYSIS*, p. 53.

Hammer, C., Krinke, J. and Snelting, G. (2006). Information flow control for java based on path conditions in dependence graphs. *IEEE International Symposium on ...*,

Howard, M., Leblanc, D. and Viega, J. (2009). *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw Hill Professional, 2009. p. 432

HP (2002). Fortify. <http://www8.hp.com/us/en/software-solutions/software-security/index.html>, [accessed on Nov 15].

IBM (2001). IBM Rational AppScan Developer Edition. <http://www-03.ibm.com/software/products/en/appscan>, [accessed on Nov 10].

Java Decompiler ([S.d.]). <http://jd.benow.ca/>, [accessed on Dec 1].

Johnson, D. (2002). Apache Roller. <http://rollerweblogger.org/project/>, [accessed on Apr 3].

Kessler, G. C. and Levine, D. E. (2009). Denial-of-Service attack. *Computer Security Handbook*. p. 1–28.

Kohli, N. and Joshi, R. (2013). Implementation of Rabin Karp String Matching Algorithm Using MPI. <http://www.sjsu.edu/people/robert.chun/courses/CS259Fall2013/s3/I.pdf>, [accessed on Dec 27].

Kuhn, T. and Olivier, T. (2006). Abstract Syntax Tree. [https://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation\\_AST/index.html](https://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html), [accessed on Sep 17].

Kupsch, J. A. and Miller, B. P. (2009). Manual vs. Automated vulnerability assessment: A case study. In *CEUR Workshop Proceedings*.

Lester, C. Y. and Jamerson, F. (2009). Incorporating Software Security into an Undergraduate Software Engineering Course. *2009 Third International Conference on Emerging Security Information, Systems and Technologies*,

Lhoták, O. and Hendren, L. (2006). Context-sensitive points-to analysis: is it worth it? *Compiler Construction*,

Linden, M. A. Van der (2009). Vulnerability Case Study: Cookie Tampering. [http://www.infosectoday.com/Articles/Cookie\\_Tampering.htm](http://www.infosectoday.com/Articles/Cookie_Tampering.htm), [accessed on Aug 16].

Livshits, V. B. (2005). Stanford SecuriBench. <http://suif.stanford.edu/~livshits/securibench/intro.html>, [accessed on Apr 3].

Livshits, V. B. (2006). Lapse+. <http://evaluates.es/?q=node/14>, [accessed on Oct 10].

Livshits, V. B. and Lam, M. S. (2005). Finding Security Vulnerabilities in Java Applications with Static Analysis. *Architecture*, p. 18.

Meier, J. D., Mackman, A., Wastell, B., et al. (2005). How To: Perform a Security Code Review. <http://msdn.microsoft.com/en-us/library/ff649315.aspx>, [accessed on Jul 21].

Nadeem, M., Williams, B. J. and Allen, E. B. (2012). High false positive detection of security vulnerabilities. In *Proceedings of the 50th Annual Southeast Regional Conference on - ACM-SE '12*. ACM Press. <http://dl.acm.org/citation.cfm?doid=2184512.2184604>, [accessed on Jan 28].

NetBeans ([S.d.]). <https://netbeans.org/>.

Oracle ([S.d.]). Uses of Reflection. <http://docs.oracle.com/javase/tutorial/reflect/>, [accessed on Mar 14].

Organization for Internet Safety (2004). Guidelines for Security Vulnerability Reporting and Response Organization for Internet Safety. [http://www.symantec.com/security/OIS\\_Guidelines\\_for\\_responsible\\_disclosure.pdf](http://www.symantec.com/security/OIS_Guidelines_for_responsible_disclosure.pdf), [accessed on Apr 1].

OWASP (2003a). OWASP Projects. [https://www.owasp.org/index.php/Category:OWASP\\_Project](https://www.owasp.org/index.php/Category:OWASP_Project), [accessed on Apr 17].

OWASP (2003b). OWASP.org. [https://www.owasp.org/index.php/Main\\_Page](https://www.owasp.org/index.php/Main_Page), [accessed on Jul 10].

- OWASP (2006). WebGoat Project. [https://www.owasp.org/index.php/Category:OWASP\\_WebGoat\\_Project](https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project), [accessed on Apr 3].
- OWASP (2013a). Cross-site Scripting (XSS). [https://www.owasp.org/index.php/Cross-site\\_Scripting\\_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)), [accessed on Oct 12].
- OWASP (2013b). SQL Injection. [https://www.owasp.org/index.php/SQL\\_Injection](https://www.owasp.org/index.php/SQL_Injection), [accessed on Oct 12].
- OWASP (2013c). Broken Access Control. [https://www.owasp.org/index.php/Broken\\_Access\\_Control](https://www.owasp.org/index.php/Broken_Access_Control), [accessed on Oct 12].
- OWASP (2013d). OWASP Top 10 - 2013. [http://owasptop10.googlecode.com/files/OWASP Top 10 - 2013.pdf](http://owasptop10.googlecode.com/files/OWASP_Top_10_-_2013.pdf), [accessed on Nov 25].
- OWASP (2013e). Security Misconfiguration. [https://www.owasp.org/index.php/Top\\_10\\_2013-A5-Security\\_Misconfiguration](https://www.owasp.org/index.php/Top_10_2013-A5-Security_Misconfiguration), [accessed on Oct 12].
- OWASP (2013f). HTTP Response Splitting. [https://www.owasp.org/index.php/HTTP\\_Response\\_Splitting](https://www.owasp.org/index.php/HTTP_Response_Splitting), [accessed on Oct 12].
- OWASP (2013g). HttpOnly. <https://www.owasp.org/index.php/HttpOnly>, [accessed on Oct 12].
- OWASP (2013h). Secure Flag. <https://www.owasp.org/index.php/SecureFlag>, [accessed on Oct 12].
- OWASP (2013i). Information Leakage. [https://www.owasp.org/index.php/Information\\_Leakage](https://www.owasp.org/index.php/Information_Leakage), [accessed on Oct 12].
- OWASP (2013j). Command Injection. [https://www.owasp.org/index.php/Command\\_Injection](https://www.owasp.org/index.php/Command_Injection), [accessed on Oct 12].
- OWASP (2013k). LDAP injection. [https://www.owasp.org/index.php/LDAP\\_injection](https://www.owasp.org/index.php/LDAP_injection), [accessed on Oct 12].

OWASP (2013l). Log Forging. [https://www.owasp.org/index.php/Log\\_Forging](https://www.owasp.org/index.php/Log_Forging), [accessed on Oct 12].

OWASP (2013m). Path Traversal. [https://www.owasp.org/index.php/Path\\_Traversal](https://www.owasp.org/index.php/Path_Traversal), [accessed on Oct 12].

OWASP (2013n). Reflection injection. [https://www.owasp.org/index.php/Reflection\\_injection](https://www.owasp.org/index.php/Reflection_injection), [accessed on Oct 12].

OWASP (2013o). XPATH Injection. <http://www.soapui.org/Security/xpath-injection.html>, [accessed on Oct 12].

OWASP (2013p). OWASP Ranking Risk Factors. [https://www.owasp.org/index.php/Top\\_10\\_2013-Details\\_About\\_Risk\\_Factors](https://www.owasp.org/index.php/Top_10_2013-Details_About_Risk_Factors), [accessed on Apr 27].

Payne, M. (2003). PersonalBlog. <https://sourceforge.net/projects/personalblog/>, [accessed on Apr 3].

Pugh, B. and Loskutov, A. (2006). Findbug. <http://findbugs.sourceforge.net/>, [accessed on Nov 10].

Sampaio, L. (2013). NCO. <http://www.inf.puc-rio.br/~lsampaio/nco/nco.zip>, [accessed on Jul 17].

Sampaio, L. (2014a). Which methods should be considered “Sources”, “Sinks” or “Sanitization”? <http://thecodemaster.net/methods-considered-sources-sinks-sanitization/>, [accessed on Sep 30].

Sampaio, L. (2014b). Benchmark Results. <http://www.inf.puc-rio.br/~lsampaio/plugin/benchmark/Results.zip>, [accessed on Jul 17].

Sampaio, L. (2014c). Controlled Experiment. [http://www.inf.puc-rio.br/~lsampaio/plugin/controlled\\_experiment/controlled\\_experiment.zip](http://www.inf.puc-rio.br/~lsampaio/plugin/controlled_experiment/controlled_experiment.zip), [accessed on Jul 17].

Sampaio, L. (2014d). How to use the ESVD plug-in on Eclipse. <https://www.youtube.com/watch?v=pNr38gMWvHQ>.

Sampaio, L. and Garcia, A. (2014). ESVD - Early Security Vulnerability Detector. <https://marketplace.eclipse.org/content/early-security-vulnerability-detector-esvd/>.

Sasaki, Y. (2007). The truth of the F-measure. *Teach Tutor mater*, p. 1–5.

Searching for Code in J2EE/Java (2010).  
[https://www.owasp.org/index.php/Searching\\_for\\_Code\\_in\\_J2EE/Java](https://www.owasp.org/index.php/Searching_for_Code_in_J2EE/Java), [accessed on Mar 9].

Secure Coding Guidelines for Java SE (2014).  
<http://www.oracle.com/technetwork/java/seccodeguide-139067.html>, [accessed on Mar 9].

Telang, R. and Wattal, S. (2005). Impact of Software Vulnerability Announcements on the Market Value of Software Vendors – an Empirical Investigation. *Available at SSRN 677427*, n. February, p. 1–34.

Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M. and Weisman, O. (2009). TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation - PLDI '09.*, PLDI '09. ACM Press. <http://doi.acm.org/10.1145/1542476.1542486>, [accessed on Aug 5].

Tsipenyuk, K., Chess, B. and McGraw, G. (nov 2005). Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors. *IEEE Security and Privacy Magazine*, v. 3, n. 6, p. 81–84.

Williams, J. (2010). OWASP Enterprise Security API. <https://www.owasp.org/index.php/ESAPI>, [accessed on Sep 10].

Willis, G. W., White, G. B., Marti, W. and Huson, M. L. (2006). Incorporating Security Issues Throughout the Computer Science Curriculum.

Xie, J., Lipford, H. R. and Chu, B. (sep 2011). Why do programmers make security errors? In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6070393>, [accessed on Nov 25].

Zeichick, A. (2012). Zeichick's Take: Java, Java everywhere. <http://sdtimes.com/zeichicks-take-java-java-everywhere/>, [accessed on Feb 9].

Zhu, J. (2012). ASIDE. [https://www.owasp.org/index.php/OWASP\\_ASIDE\\_Project](https://www.owasp.org/index.php/OWASP_ASIDE_Project), [accessed on Nov 10].



## Appendix 1 - Participant Profile Questionnaire

### Participant Profile

The purpose of this form is to gather more information about the participant' profile.

**\* Required**

**Participant Id \***

**What is your job's title ? \***

**How many years of experience do you have in software development ? \***

**What programming languages do you work with ? \***

Have worked in the past and/or are currently working

☐ Java

☐ C#

☐ PHP

☐ C/C++

☐ Javascript

☐ Python

☐ Lua

☐ Ruby

☐ Other:

**Have you ever had training on :**

☐ Pair programming

☐ Secure programming

☐ Agile development

☐ Other:

**Submit**

*Never submit passwords through Google Forms.*

## Appendix 2 - System Requirements

### Scenario

You work in a company as a software developer and code reviewer. You are asked to implement some features and verify if they meet the system requirements.

### System Requirements

1. There should be a page where users can sign into the system.
  - a. The user should use a login and password.
  - b. The user also has the option to select "Keep me online", which will let the user stay signed in even if he/she closes the browser.
2. There should be a page where all the comments are displayed.
  - a. Only signed in user can access this page.
3. The user can add comments.
  - a. The user must be signed in.
  - b. The comment cannot be empty.
4. The user can delete his/hers comments.
  - a. The user must be signed in.
  - b. Only his/hers comments.
  - c. More than one comment can be deleted at once.
5. There should be an administrator user.
  - a. The user must be signed in.
  - b. The administrator can access all the pages.
  - c. The administrator can delete any comment.