Static code analysis of data-driven applications through common lingua and the Semantic Web technologies

By Oana-Elena Ureche

BSc Computer Science, MSc Computer Science

This thesis is submitted in total fulfillment of the requirements
for the degree of Doctor of Philosophy at the
Federation University Australia
PO Box 663
University Drive, Mount Helen
Ballarat, Victoria 3353,
Australia

May 2014

Table of Contents

List of Figures

List of Listings

List of Tables

ABSTRACT

Web applications have become increasingly popular due to their potential for businesses' high revenue gain through global reach. Along with these opportunities, also come challenges in terms of Web application security. The increased rise in the number of data-driven applications has also seen an increased rise in their systematic attacks. Cyber-attacks exploit Web application vulnerabilities. Attack trends show a major increase in Web application vulnerabilities caused by improper implementation of information-flow control methods and they account for more than 50% of all Web application vulnerabilities found in the year 2013.

Static code analysis using methods of information-flow control is a widely acknowledged technique to secure Web applications. Whilst this technique has been found to be both very effective and efficient in finding Web application vulnerabilities, specific tools are highly dependent on the programming language. This thesis leverages Semantic Web technologies in order to offer a common language through source code represented using the Resource Description Framework format, whereby reasoning can be applied to securely test Web applications.

In this thesis, we present a framework that extracts source code facts from various programming languages at a variable-level of granularity using Abstract Syntax Trees (ASTs) generated using language grammars and the ANTLR parser generator. The methodology for detecting Web application vulnerabilities implements three phases: entry points identification, tracing information-flow and vulnerability detection using the Jena framework inference mechanism and rules describing patterns of source code.

The approach discussed in this thesis is found to be effective and practical in finding Web application vulnerabilities with the limitation that it can only detect patterns that are used as training data or very similar patterns. False positives are caused by limitations of the language grammar, but they do not affect the accuracy of the security vulnerability detection method in identifying the correct Web application vulnerability.

ACKNOWLEDGEMENT

The author takes this opportunity to gratefully acknowledge the assistance and contributions of the few people who had faith in this undertaking.

Deserving special mention are my supervisors, Dr. Robert Layton, Assoc. Prof. Peter Vamplew and Dr. Paul Watters, for their guidance, patience and advice during the course of my thesis. Their experience was proven invaluable to me and this thesis would have not existed without them.

And to my family, which provided continuous moral support from a distant continent during the whole duration of my course and especially during those times when I was doubtful of my strength to finish.

And to Ahmad Azab, for having fascinating history discussions during work breaks. You have enlightened me with the knowledge of a culture that is incredibly fascinating.

And to my dearest friends, Gary Holdsworth, Boyd Williams, Ike Anyiam, Michele Catasta, Dr. Gabi Ditu and many more. Their company has made my time outside the office as equally as good as the time behind the desk. Their worries and doubts built strength in me and helped me achieve great things. For that, I am grateful. Their sense of humor is one I have never seen before. So, thank you.

And last, but not least, to all the researchers at ICSL who provided a friendly and welcoming working atmosphere. I honestly, sincerely and from the bottom of my heart, do not think I could have finished this work any place else.

The work presented in this document was supported (in part) by the Federation University, the Australian Federal Police, IBM, Westpac and the Victorian Government.

STATEMENT OF AUTHORSHIP

Except where explicit reference is made in the text of the thesis, this thesis contains no material published elsewhere or extracted in whole or in part from a thesis by which I have qualified for or been awarded another degree or diploma. No other person's work has been relied upon or used without due acknowledgement in the main text and bibliography of the thesis.

Sign: [illegible]
Candidate

Sign: [illegible]
Principal Supervisor

1. INTRODUCTION

The security of Web applications is an important issue recognized at an academic level through continuous research of new solutions to protect the confidentiality of sensitive data in Web applications (Yip, Wang, Zeldovich, & Kaashoek, 2009; Baca, Carlsson, Petersen, & Lundberg, 2013; Roy, Porter, Bond, Mckinley, & Witchel, 2009). At industry level, a recent report conducted by Ponemon Institute (Ponemon, 2013) entitled "The Post Breach Boom" shows that data breaches cause substantial losses, with 45 percent of losses exceeding 500,000 dollars. According to the "2012 Data Breach Investigative Report" by Verizon, most attacks are caused by hacking and malware (WhiteHat Security, 2013). The majority of attacks targeted the least protected areas: Web applications.

In practice, developers of Web applications that contain sensitive/hidden data will surround the implementation code with security features to protect the sensitive data. Unfortunately, it is impossible to guarantee that a piece of complex software does not contain some flaws (Ritchey & Ammann, 2000). Developers make mistakes, which results in implementation bugs that can be exploited in order to gain access to unauthorized information.

Static code analysis is a common method successfully applied by industries to improve software security by detecting code susceptible to malicious attacks (Baca, Carlsson, Petersen, & Lundberg, 2013). Static code analysis is a technique that examines an application's source code to assess its security risk (Louridas, 2006). A primary design technique of static analysis is control flow testing at the program's unit level, where source code is reviewed in order to find any patterns discordant with the non-interference (Hammer, Krinke, & Nodes, 2006) security policy. The non-interference security policy ensures the robustness of a system by guaranteeing that the application does not leak confidential information to unauthorized users of the system.

Existing academic solutions that apply static code analysis to control the flow of confidential information have a common limitation: their proposal for new scripting languages (Zdancewic, 2004). This clear shortcoming makes proposed solutions infeasible to be applied to existing systems, and consequently to traditional programming languages (i.e. Java, PHP). Yet, surprisingly, there are not many efforts made to develop a method that can be uniformly applied to secure test data-driven applications independently of their underlying code (Ureche, Layton, & Watters, 2012). We propose employing Semantic Web technologies to transform source code written in various traditional programming languages to a common format, so that reasoning using inference engines can be applied to assuring code security.

The Semantic Web is the grand vision of machines understanding the meaning of information and rests on the theory that this can be achieved by accommodating semantics of Web data (Gruber, 2008). While critics speculated the Semantic Web vision to be infeasible (Richardson, Agrawal, & Domingos, 2003), advocates validated the original concept with applications in industry, biology and human science (Feigenbaum, Herman, Hongsermeier, Neumann, & Stephens, 2007). Typically, data from various sources is represented as triples in the Resource Description Framework (RDF) format with the purpose of being uniformly queried by machines in order to solve a specific problem or answer a specific question (Arenas & Perez, 2011). In this thesis, we turn to static code analysis using RDF.

Although the literature proposes a few RDF converters for source code, these services are not suitable for our needs. For example, Keivanloo et al. (2011) proposed the SeCOLD framework with the primary focus on providing a Uniform Resource Locator (URL) generation schema for sharing source code facts. Ganapathy and Sagayaraj (2011) extracted metadata from source code (i.e. author, method name and description) with the intention of providing easy reuse of existing code. Both services consider only the Java programming language.

In light of this, it is imperative to both recognize and acknowledge that a method that represents traditional programming languages within a common lingua can possibly provide the most effective means to uniformly query for vulnerabilities in data-driven applications, thus offering a language independent approach towards securing Web systems.

1.1. Motivations

Web applications have become increasingly popular due to their potential for businesses' high revenue gain. Using Web applications, businesses can potentially reach anyone with an internet connection at any time, in order to sell their products. Online shopping is a cheaper alternative due to the low costs of running an online store. Along with these opportunities for both sellers and buyers, also come challenges in terms of Web application security. The increased rise in the number of data-driven applications has also seen an increased rise in their systematic attacks. These attacks target software bugs, design flaws and configuration defects, depicted in Figure 1 and account for major losses of revenue and user mistrust in online companies (Liu & Cheng, 2009).

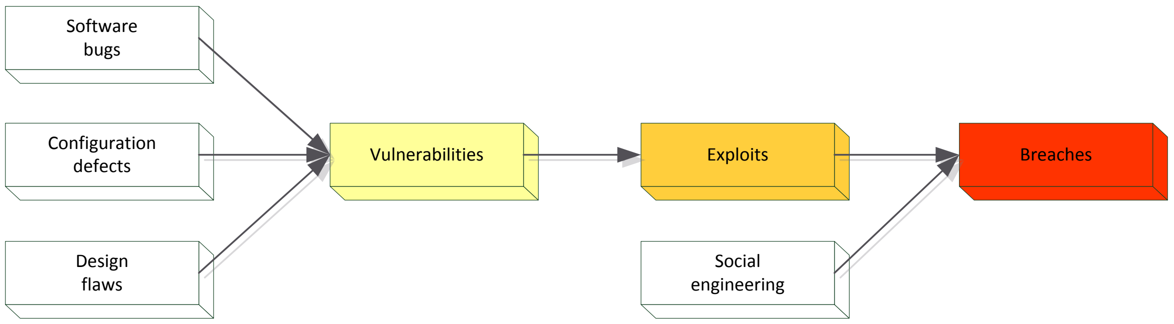


Figure 1 Causes of cyber-attacks. Risk sources are depicted using white boxes (Liu & Cheng, 2009)

Section 1.1.1 will present the trends in Web application vulnerabilities and cyber-attacks, and thus the motivation for our work.

1.1.1. Web Applications' Attacks and Trends

Web applications constitute a major target for computer "hackers". High profile organisations and companies, such as: NASA (Bloomberg, 2008), Sony (The Sydney Morning Herald, 2011) and Citigroup Inc. (Business Spectator Pty Ltd., 2011), have suffered cyber-attacks that lead to these hackers gaining access to unauthorized data and other information (such as bank account and credit card details, contact information, email addresses) that could lead to identity theft (Stabek, Watters, & Layton, 2010). According to security company McAfee, the last five years have seen the biggest series of cyber-attacks in history, including the infiltration of 72 international organisations as well as the United Nations (Guardian News and Media Limited, 2011). To further motivate the need for securing data in Web applications through information-flow control, we will present publicly available statistics conducted by several government and not-for-profit organizations focused on improving the security of software.

The National Institute of Standards and Technology (NIST) is a government organization that has gathered Web vulnerability data for many years. NIST generates statistical data through a National Vulnerability Database[1] (NVD), a U.S. government repository of standards based vulnerability management data. Figure 2 shows generated data for the number of vulnerabilities caused by insufficient input validation from year 2004 to 2013.

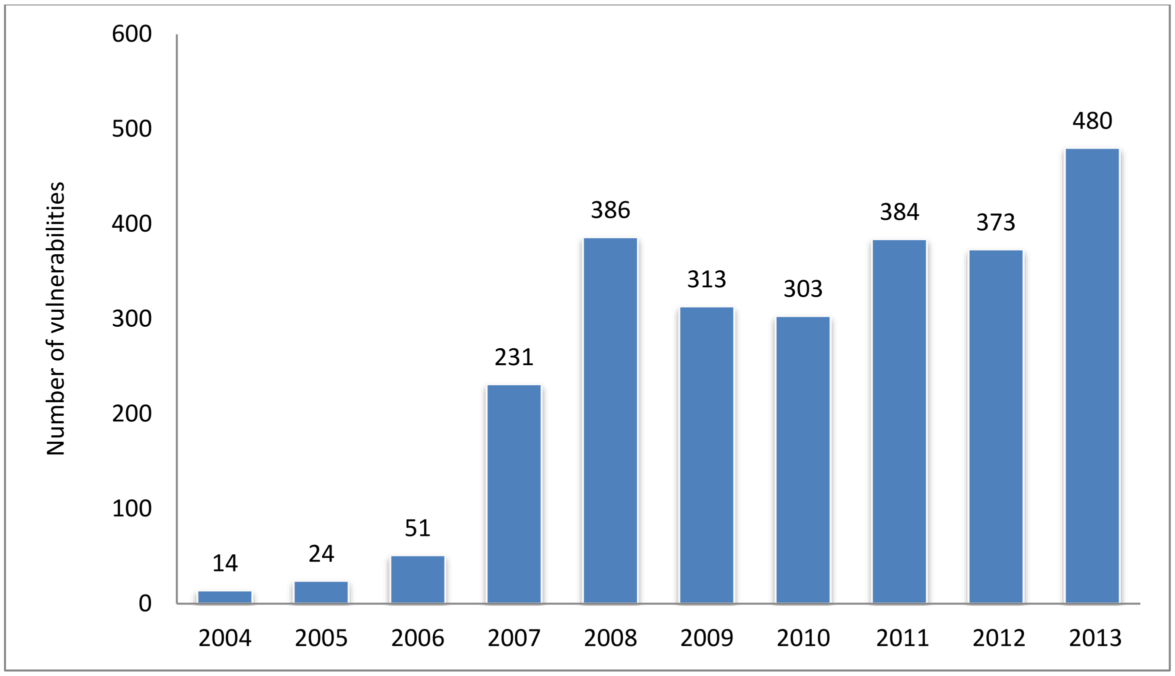


Figure 2 NIST yearly statistics of the number of vulnerabilities caused by insufficient input validation

As Figure 2 depicts, the last six years have seen a major increase in Web vulnerabilities caused by insufficient input validation, with the highest number of vulnerabilities corresponding to the year 2013. We can attribute this high number to the increased popularity of Web applications. We can thus speculate that unless remedial action is taken, this number is only going to rise in future years. Input validation vulnerabilities are caused by an application improperly validating or missing validation of user data and thereby

[1] http://web.nvd.nist.gov/view/vuln/statistics

affecting the control flow or data flow of a program (Scholte, Robertson, Balzarotti, & Kirda, 2012).

We used data generated from the National Vulnerability Database (NVD) to calculate the number of vulnerabilities caused by improper information-flow control in relation to other vulnerabilities (e.g. configuration, authentication issues, and credential management). As depicted in Figure 3 the vulnerabilities caused by improper implementation of information-flow control methods account for more than 50% of all Web vulnerabilities found in the year 2013.

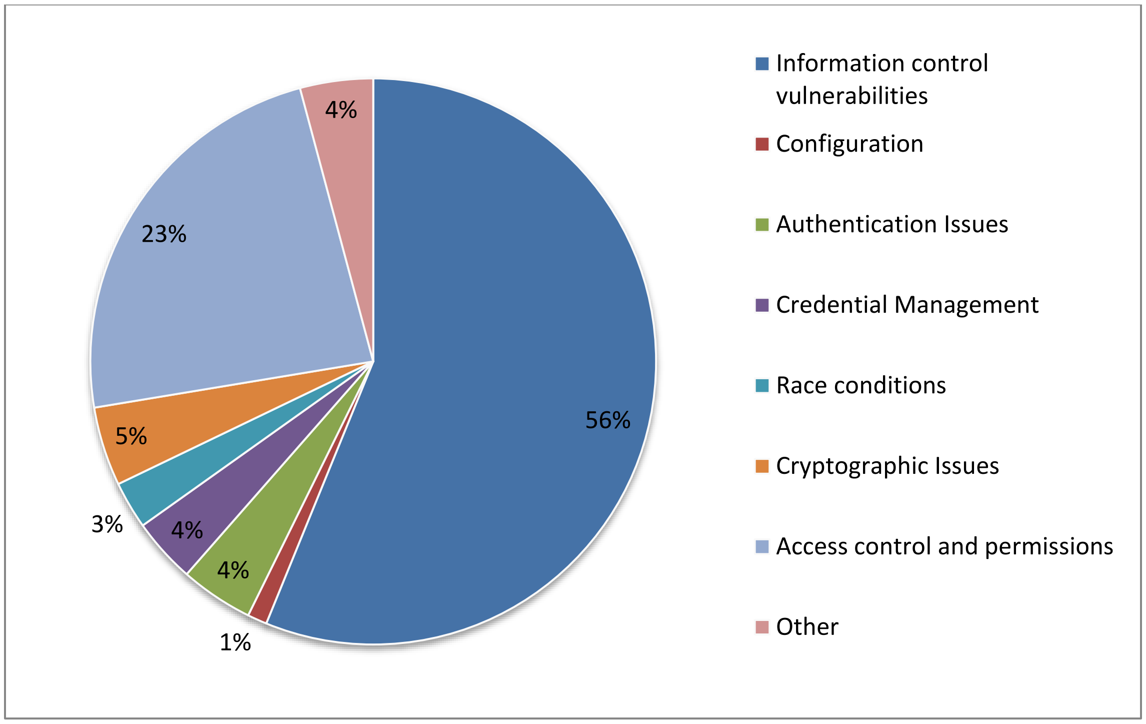


Figure 3 Number of information-flow control vulnerabilities in relation to other vulnerabilities for the year 2013

We classified vulnerabilities belonging to the information-flow control category as the ones caused by improper input validation or disclosure of information, such as cross-site scripting, SQL injection and information leakage. These vulnerabilities and others are described in Sections 1.2.1 and 1.2.2. Figure 4 depicts their proportion in relation to the total of number of vulnerabilities that are due to information-flow control issues in the year 2013. As we can see, the cross-site scripting (XSS) vulnerability is the most prevalent issue, followed by information disclosure with the SQL injection coming on the third place. In fact, according to an application vulnerability trends report conducted by Cenzic in 2013 (Cenzic, 2013), XSS has been leading the list in terms of frequency of occurrence since 2011, significantly rising each year.

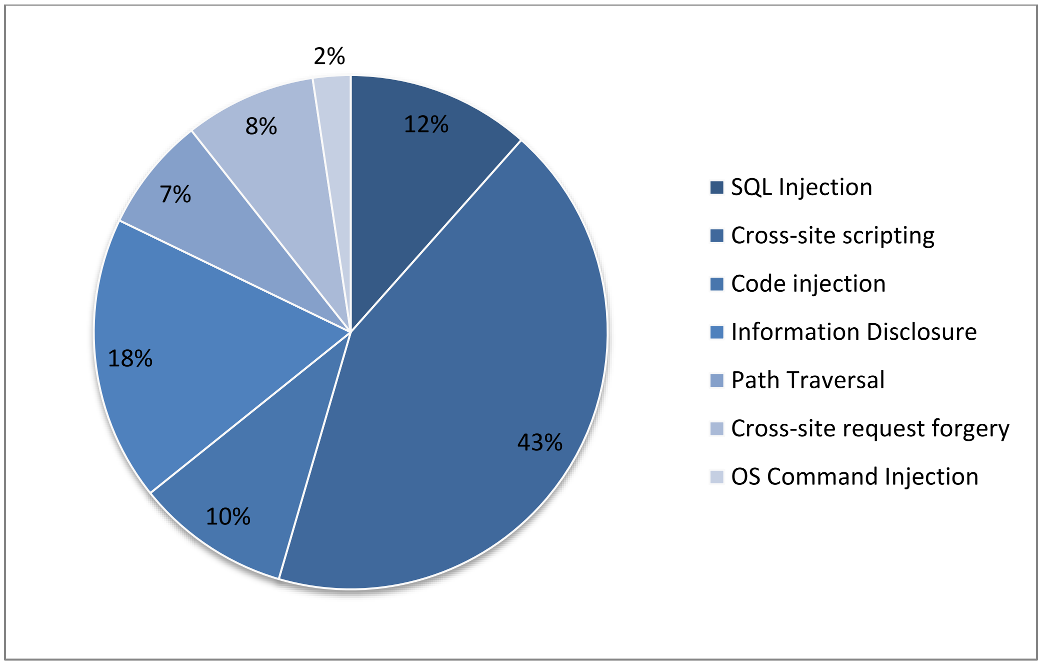


Figure 4 Proportion of Web vulnerabilities caused by improper implementation of control flow methods in 2013

In this section, we have shown the importance of a method that controls the flow of information in data-driven applications. The next section will further highlight the importance of our work, identifying the gap in literature and therefore, the need for a uniform static code analysis of programs which are potentially vulnerable to cyber-attacks.

1.1.2. An Independent Framework for Information-flow Control

Section 1.1.1 presented the real-world (financial) motivation for our work. A technical motivation is driven from a shortcoming of existing static code analysis solutions identified as a challenge in (Zdancewic, 2004).

Zdancewic identified a major limitation of existing systems that control the flow of information. These systems have focused their attention on giving better, more precise definitions of non-interference policies, which is not the *real* challenge in building secure, sound systems. The more significant challenge is to get the static code analysis application interoperable with existing systems. Existing applications that either employ ad-hoc mechanisms (e.g. Perl’s taint mode (Perl, 2014)), or implement special type systems (Li, 2005), have limited applicability (languages that satisfy specific conditions (Hammer, 2010) or they apply to a single programming language (Graf, Hecker, & Mohr, 2013)). These applications are reviewed in more detail in Chapter 2.

A static code analysis application that can interoperate with existing systems, and thus provide a language independent solution has been emphasized as important, in terms of practicality, in (Graf, Hecker, & Mohr, 2013; Rakić & Budimac, 2011 and Hammer, 2010) to name a few. Benefits, such as precision, scalability and practicality of using a language independent approach have been demonstrated in (Hammer, 2010). In this paper, the authors used program dependence graphs for information-flow control, but their application is only limited to languages that compile to Java bytecode.

1.1.3. White-box Testing versus Black-box Testing

Two common methods are used for detecting security vulnerabilities: white-box testing (Beizer, 1990) and black-box testing (Beizer, 1995). Static code analysis is usually performed as part of white-box testing (OWASP, 2013).

Whilst white-box testing methods have been proven to be effective in detecting more security vulnerabilities than black-box testing methods, testing should be complementary in order to ensure the security of a Web application (Finifter & Wagner, 2011). Static code analysis complements dynamic analysis by examining the flow of information for all execution paths and variables and not just the ones exercised at runtime (Masri, Podgurski, & Leon, 2004). This feature of static analysis is particularly useful in assuring a program’s security because most security attacks exercise an application in unforeseen and untested ways (Intel, 2012).

Although it falls out of the scope of this thesis to compare the two methods of testing a Web application, research (Finifter & Wagner, 2011) shows that some types of security vulnerabilities were not detected using a black-box method. White-box testing has been proven to detect authentication/authorization bypass Web application vulnerabilities, compared to black-box testing which was not able to detect these types of security vulnerabilities. Moreover, black-box testing detected few Stored XSS vulnerabilities, compared to white-box testing. Figure 5 illustrates the types of vulnerabilities detected using white-box and black-box methods. White-box testing is represented using the term *Manual*.

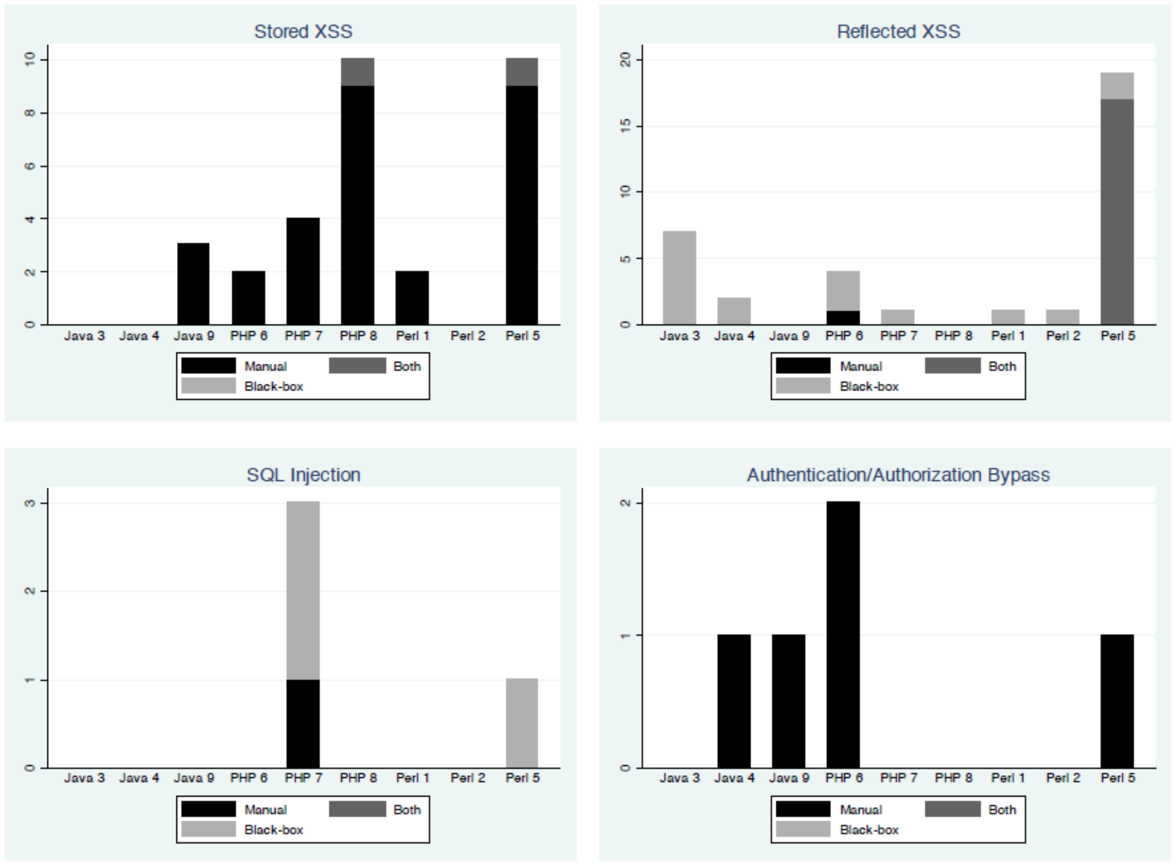


Figure 5 Vulnerabilities by vulnerability class (Finifter & Wagner, 2011)

In light of this, this thesis proposed methodology implements a static code analysis approach to detect Web application security vulnerabilities.

1.2. Overview of Web Application Vulnerabilities and Ad-hoc Solutions

As shown in Section 1.1.1, according to the study conducted by Cenzic (2013), most issues in Web application security are caused by improper input/output validation. The vulnerabilities resulting from these issues are also known as information-flow control vulnerabilities and this thesis focuses on them.

The Open Web Application Security Project (OWASP) published a study that further confirms the numbers from the Cenzic statistics. Every three years, OWASP releases its top 10 Web vulnerabilities ranking list. The top place in both 2010 and 2013 is held by vulnerabilities caused by improper input validation (OWASP, 2013). It should be mentioned that efforts to prevent the attacks caused by this type of vulnerabilities have been successfully applied in the industry by Perl's (Perl, 2014) and Ruby's (Thomas, Fowler, & Hunt, 2004) taint mode with very good outcomes (Hurst, 2004). Unfortunately, both mechanisms are dependent on the respective scripting language. More details are given in Section 2.2.1.

In order to define our problem statement we chose to classify the security vulnerabilities caused by improper input/output validation in two main categories: Input Validation Vulnerabilities and Information Leakage.

1.2.1. Input Validation Vulnerabilities

Typically, user input capable of compromising a Web application's security is received through HTML forms. If the user input is not properly sanitized, serious security problems can occur (Scholte, Robertson, Balzarotti, & Kirda, 2012). Mostly, attackers of Web applications that exploit input validation errors use cleverly crafted user input in HTML form fields in order to obtain confidential information or engage in a DoS (denial-of-service) attack.

Sanitization of user input is dependent on the context in which the user data is used and thus, difficult to be implemented correctly and relevantly. For example, while a PHP *filter_input* function is generally suitable for sanitizing input, it would not be recommended for a SQL injection attack (SQLIA). In this situation, user input sanitization should be implemented using the *mysql_real_escape_string()* or prepare statements and parameter binding, as *filter_input* cannot guarantee that it will prevent all SQL injection attacks in any situation (OWASP, 2014).

Unfortunately, when the proper sanitization is not in place, vulnerabilities must be identified and fixed. Next, we illustrate vulnerabilities caused by improper input sanitization and their context dependent solution. We aim to show that different sanitization solutions apply to different contexts and evidently, to different programming languages, making it very easy for developers to make mistakes and leave room for vulnerabilities.

Cross-site scripting. As discussed in Section 1.1.1, XSS (cross-site scripting) has been leading the list in terms of frequency of occurrence since 2011, significantly rising each year. This type of attack is typically found in Web applications and it enables attackers to inject client-side scripts into webpages. A cross-site scripting vulnerability is present where a Web application uses input data to generate a webpage's content to be displayed back to the user, without validating or sanitizing the input. Common data-driven applications vulnerable to cross-site scripting attacks are online message boards, where users can submit HTML formatted messages that will persistently appear on the board's webpage.

Cross-site scripting is mostly used for transmitting private data, like cookies and session information, with the malicious intent of using it to impersonate other users and access sensitive information. Typically, a malicious user will post a message to a board containing a `<script>`, e.g. `<script>savetofile(document.cookie);</script>`. For any user that visits the page where the script was injected, assuming that there is no prevention of a cross-site scripting attack, their cookie information is saved to the file that the attacker specified, which could be used to log in as the specific user.

There are alternative ways to the `<script>` tags in which XSS attacks can be conducted. Other HTML tags can have the same effect. For example, `<body onload=alert('document.cookie')>`. XSS attacks work because they bypass access controls, such as the same-origin policy. The same-origin policy prevents scripts from accessing content from a location other than the origin of the script. Because a cross-site injected script is running from website *X*, according to the same-origin policy, it can read website *X*'s cookie information. Access to cookie information is legitimate as the request came from a trusted source, e.g. the same website that was granted permission to the system's resources.

For example, the following PHP code is vulnerable to an XSS attack because the age parameter is read from an untrusted source (i.e. input from user) and echoed back to the user through the generated website page.

```
<?php
$age = $_REQUEST ['age'];
?>
<html>
<body>Your age is <?php echo $age; ?></body>
</html>
```

An attacker can exploit this vulnerability and instead of legitimate data, she could submit JavaScript through the `age` parameter, such as the following:

```
http://xss.vulnerable.website.com/vuln.php?age=
%3Cscript%3Ealert(%22Hello%22)%3B%3C%2Fscript%3E
```

The text "`Hello`" is echoed back to the browser, which demonstrates that the website is vulnerable to an XSS attack. There are numerous ways to prevent an XSS attack and they range from basic HTML escaping of five characters significant in XML `($, <, >, ", ')` to employing libraries specially designed for sanitizing HTML input. For example, the following PHP code uses the *HTMLPurifier*[2] library to sanitize HTML:

```
require_once '/path/to/HTMLPurifier.auto.php';

$conf = HTMLPurifier_Config::createDefault();
$purifier = new HTMLPurifier($conf);
$clean_html = $purifier->purify($vuln_html);
```

For Java, OWASP provides a similar library[3] to sanitize HTML:

```
import org.owasp.html.Sanitizers;
import org.owasp.html.PolicyFactory;
PolicyFactory sanitizer =
              Sanitizers.FORMATTING.and(Sanitizers.BLOCKS);
String clean_html = sanitizer.sanitize(vuln_html);
```

Such dedicated libraries to sanitize HTML are straightforward to use, but their policy must be updated, as valid tags could be stripped off, especially most of the modern HTML5 and CSS3 tags and attributes. If manual HTML escaping is necessary, then the disadvantage is

[2] http://htmlpurifier.org/
[3] https://www.owasp.org/index.php/OWASP_Java_HTML_Sanitizer_Project

that developers can mistakenly overlook some tags and thus still leave opportunities for attacks to occur.

Therefore, when considering the prevention of cross-site scripting attacks, developers must be aware of the context in which they can occur, whether this is a different programming language or newer markup languages.

Format string attacks. *Printf(user_input)* is a common statement in C programs. When user input is used, it is imperative that the *printf* format function uses format string parameters, such as, *printf("%s", user_input)*. If not, the user input can be treated as a format string and when an attacker exploits this vulnerability, he can submit *"%s%s%s%s%s%s%s%s%s%s"* as input, resulting in the command *printf("%s%s%s%s%s%s%s%s%s%s")* being executed. This can result in crashing the program as for every *%s* the program will retrieve data from the stack to use it as a memory address fetching whatever is located in the memory at that address. If the content fetched from the stack is not a legitimate address for which corresponding memory data exists, the program will crash.

In this case, the vulnerability can be found in C programs and to circumvent it, developers must validate the user input by using format string parameters.

SQL injection. A SQLIA is a type of injection attack, where SQL queries are injected via the input data from the Web client to the data-driven application. These types of attacks can have a wide range of consequences:

- Confidentiality: access to sensitive data from the database (e.g. customers credit card number)
- Authentication: exploit poor authentication logic and log in as a different user
- Authorization: change authorization privileges if available in the database
- Integrity: changes to the database data via SQL commands (Insert/Update/Delete)

To illustrate how a SQLIA works, we will assume that a Web application retrieves data about a person using the following SQL command in PHP. We used the reserved variable

`$_POST` to show that input data is retrieved from the client via a POST request and thus is not safe:

```
SELECT age, email, marital_status, position
FROM members
WHERE full_name = '$_POST[full_name]';
```

Next, we assume a malicious user is trying to compromise the integrity of the `members` database by deleting it entirely. Because there is no sanitization in place, it is sufficient that the attacker submits `John Doe'; DROP TABLE members; --` in place of the full name POST variable. This constructs the following SQL command, which in fact executes two SQL queries.

```
SELECT age, email, marital_status, position
FROM members WHERE full_name = 'John Doe';
DROP TABLE members; --';
```

The first one retrieves data about the person named John Doe and the second one deletes the members database entirely. The pair of hyphens is used to mark the beginning of a comment in SQL and thus everything encountered after this mark is ignored.

This example shows how little effort is required to compromise the integrity of a database when proper input validation is not implemented.

In order to handle such situations, developers must look at the context in which they occur. For example, if an email address is read via a POST request, one way to sanitize input is to make sure it only contains these characters:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
@.-_+
```

Unfortunately, this method does not apply in all situations (e.g. when an input field is used for a description of a user profile where semicolons, quotes and other characters can be present).

The recommended way (OWASP, 2008) in which user data should be handled is through prepared statements. Using prepared statements or parameterized queries enables the database to distinguish between code and data. In the example given above the database engine will in fact try to match `John Doe'; DROP TABLE members; --` against records corresponding to full name data, thus not constructing new SQL queries. In PHP, this can be achieved using the following syntax:

```
$full_name = $_POST[full_name];
$stmt = $mysqli->prepare ("SELECT age, email,
        marital_status, position FROM members
        WHERE full_name = ?")
$stmt->bind_param(1, $full_name);
$stmt->execute();
```

Java has a different syntax:

```
String full_name = request.getParameter("full_name");
String query = "SELECT age, email, marital_status, position
               FROM members WHERE full_name = ?";
PreparedStatement ps = connection.prepareStatement(query );
ps.setString( 1, full_name);
ResultSet results = ps.executeQuery( );
```

Fortunately, many programming languages support parameterized queries, but when performance becomes an issue (MSDN, 2013), stored procedures can be used instead (Ke, Muthuprasanna, & Kothari, 2006). The difference between parameterized queries and stored procedures is that the database defines and stores the SQL code for a stored procedure, thus making the programming code less portable and reusable.

As shown, SQL injection attacks can be prevented in numerous ways, but the approach used is chosen based on the context of the vulnerability. Sometimes, performance can be an important issue and thus, while prepared statements are the preferred way of implementing security, they can also cause problems and thus, alternative defenses should be considered. Moreover, every programming language uses a different syntax, creating an opportunity for developers to make mistakes.

Command injection. The essence of command injection attacks has been captured in (Su, 2006). The authors attributed the susceptibility of Web applications to this large class of attacks to two causes: improper handling of user input and a semantic gap. The semantic gap is explained by the difference in the way in which databases and Web applications interpret query strings. Whilst databases view them as well-defined, meaningful commands, Web applications typically interpret query strings as an unorganized sequence of characters.

Although security can be enforced in the database and it is recommended by Oracle (2014), enforcing security in the database can fail to make an application's security implementation modular or reusable and thus, less portable, since the same application using a different database will need a reimplementation of the security policy. Therefore, a scalable security application needs to detect if user input is improperly handled.

Although SQLIA are part of the command injections class of attacks, there are other means in which these attacks can be executed, such as through command line arguments. For example, the following C code reads a filename via the command line and displays its contents to the user:

```
int main(char* argc, char** argv) {
	char cmd[CMD_MAX] = "/usr/bin/cat ";
	strcat(cmd, argv[1]);
	system(cmd);
}
```

This program will work as expected when a legitimate file name is used. However, if an attacker uses the input "`;rm -rf /`" instead, the system call will not execute the `cat` command due the insufficient number of arguments and if run with root privileges it will continue by recursively deleting all the contents of nearly every writable mounted filesystem on the computer. Eventually, the computer will crash due to missing a crucial file or directory.

A solution to this problem is to sanitize command line user input. Note that in this case stored procedures and parameterized queries do not apply, thus the developer is faced with an entirely different context than shown in previous sections.

HTTP response splitting. HTTP response splitting is a Web application vulnerability that results from the failure to properly sanitize user input read in an HTTP request and subsequently used to generate an HTTP response. An HTTP response splitting vulnerability allows an attacker to break the HTTP response into two or more responses, which will both be handled by the server. HTTP responses are separated by a carriage return (CR, ASCII 0x0D) and a line feed (LF, ASCII 0x0A), therefore this exploit is possible if the application allows input containing these characters.

For example, the following Java code is vulnerable to the HTTP response splitting attack, assuming that the `fullName` variable's content is read from an untrusted source:

```
String name = request.getParameter(fullName);
Cookie cookie = new Cookie("name", name);
cookie.setMaxAge(cookieExpiration);
response.addCookie(cookie);
```

If the Web application reads as input `'John Doe'` and saves it as the content of the `fullName` variable, then the subsequent HTTP response will have the following content:

```
HTTP/1.1 200 OK
Content-Length: length
…
Set-Cookie: name=John Doe
…
```

Instead of valid input, a malicious attacker can use the following value for the `fullName` variable, resulting in a second HTTP response completely controlled by the attacker.

```
'John Doe

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 32
<html>Malicious code here</html>'
```

In order to prevent an HTTP response splitting vulnerability, a developer can choose a white-listing approach, where the input can be checked to only allow permissible characters. In the example above, the application should allow only characters ranging from `Aa` to `Zz`. Alternatively, if the input can contain any characters, developers can remove all carriage returns that are followed by line feeds. In addition, there are libraries that validate untrusted input, such as `Validator`[4] in Java. However, for optimal results, the chosen solution must be based on the context in which an HTTP response splitting vulnerability could exist.

[4] http://docs.oracle.com/javase/7/docs/api/javax/xml/validation/Validator.html

E-mail injection. E-mail injection, or e-mail header injection, is an attack usually performed to send spam emails. By knowing the MIME (Multipurpose Internet Mail Extensions) encoding, an attacker is able to inject extra headers, for example a 'Bcc' (Blind Carbon Copy) field to add recipients for whom their email addresses will be hidden. Some spammers do not find it necessary that the recipients email addresses be hidden and use a 'Cc' (Carbon Copy) field instead. A general description of the PHP mail function is as follows:

```
<?php mail($recipient, $subject, $message, $headers); ?>
```

A legitimate way to call this function could have the following form:

```
<?php mail("recipient@provider.com", "Hello", "Hi,\n9am
meeting tomorrow.\nJohn.", "From: sender@provider2.com"); ?>
```

We will assume that the data used in the `$headers` is read from a user of the application and thus, not trusted. If the PHP email function headers are not properly sanitized, an attacker can inject new fields, causing the application to send emails to other recipients. According to *RFC 822*[5] fields are separated by a line feed (LF, ASCII 0x0A). Therefore, if an attacker calls the PHP mail function with the following parameters:

```
<?php mail("recipient@provider.com", "Hello", "Hi,\n9am
meeting tomorrow.\nJohn.", "From:
sender@provider2.com%0ACc:recipient@provider3.com%0ABcc:vict
im1@provider4.com,victim2@provider5.com"); ?>
```

the corresponding email data will be:

```
To: recipient@provider.com
Subject: Hello
From: sender@provider2.com
Cc:recipient@provider3.com
Bcc:victim1@provider4.com,victim2@provider5.com

Hi,
9am meeting tomorrow.
John.
```

The attacker has therefore, injected the email headers successfully using the 'Cc' and 'Bcc' fields, allowing him to send spam emails to his victims.

[5] http://www.ietf.org/rfc/rfc822.txt

A solution to this problem is to check the user data for line feeds using regular expressions in PHP code: `eregi("(\r)”, $from)` or `(value.indexOf("\n") != -1)` in Java and declining to send emails that contain line feed characters in the headers under inspection. There are also third party classes that protect against this type of attack, such as PHP’s PEAR Mail[6]. The chosen solution should be based on the context and possible performance considerations.

1.2.2. Information Leakage

Section 1.2.1 focuses on poorly validated user input. However, many of the encountered concerns also apply to non-validated or poorly validated output passed from the Web application to its users.

A common Web application vulnerability caused by improperly validating output data is information leaks. An information leak commonly occurs when too much information is revealed through exception message printings, as in the following Java example:

```
try {
       /.../
} catch (Exception e) {
       System.out.println(e);
}
```

Printed debug information, stack traces or path information may be exposed to end users. Many times this leaked information can be helpful to attackers because it reveals implementation details and can be used to exploit vulnerabilities.

One solution to verify that sensitive information is not leaked to users is an automated approach (Morisset & Oliveira, 2007) that uses vulnerability scanning tools to generate error messages. Security experts can then verify the error messages and ensure that no sensitive information is disclosed to users of the application.

Manual approaches can be successfully used as well and are based on static code analysis tools that search for patterns in code that leak information or cause improper error handling (Walker, 2010). Therefore, testing for security violations due to non-validated output fits well into our source code facts extraction framework described in Chapter 3 and the same

[6] http://pear.php.net/package/Mail/

concepts of inspecting input validation vulnerabilities are also applicable to information leaks.

1.3. Problem Statement

The motivation for our research was given in Section 1.1. Web vulnerabilities clearly pose serious security problems resulting in substantial consequences. Therefore, we have identified two motivations: financial and technical, driven from a shortcoming of existing static code analysis solutions. Specifically, businesses benefit financially from using Web applications due to their anytime availability and the low cost of running an online store. However, Web applications are an easy target for systematic attacks due to software bugs, design flaws and configuration defects whilst accounting for major losses of revenue and user mistrust.

Furthermore, Section 1.2 presents an overview of the types of Web vulnerabilities targeted by our research and their current ad-hoc solutions. It shows that different vulnerabilities have different solutions and that the same vulnerability can have different solutions for different contexts, thus making it very hard for a Web application to be security proof.

We have thus identified two problems:

1. undesirable effects that Web vulnerabilities can cause when properly exploited by attackers
2. difficulties faced by programmers to write Web applications with no security flaws

According to the study conducted by Cenzic (2013), presented in Section 1.1.1, most issues in Web application security are caused by improper input/output validation. The vulnerabilities resulting from these issues are also known as information-flow control vulnerabilities and this thesis is focused on them. The introduction chapter shows that static code analysis is a common method successfully applied by industries to improve software security by detecting code susceptible to malicious attacks.

In light of this, research towards powerful static code analysis tools for detecting Web vulnerabilities caused by improper flow control represents an important step for improving the security of Web applications and the consequent societal benefits.

1.3.1. Research Questions

This research will address the limitations of applicability to existing Web application systems of previous approaches (described in Section 2.2) and determine if Semantic Web technologies can provide a language independent method using a pattern-based static code analysis approach. This research aims to answer the following question:

Can Semantic Web technologies applied to information-flow control provide a language independent method, using a pattern-based static code analysis approach, to address the limitation of previous works in terms of applicability to existing Web application systems?

Most previous approaches propose a new scripting language for developing Web application systems, so the programmers are faced with the problem of learning a new language to develop a Web application and rewriting existing applications in the proposed language.

Firstly, the proposed method aims to be applicable to existing Web application systems written in different traditional scripting languages (PHP, Java, Python) while at the same time using a pattern-based static code analysis approach. Secondly, the proposed method automates a significant proportion of the work involved to model code, by not requiring the developers to rewrite the applications using a proposed scripting language. A summary of this thesis' contributions is given below.

1.4. Summary of Thesis Contributions

In research on security testing data-driven applications using static checks, we aim to implement a solution that it is independent of the input programming language. Zdancewic identified this shortcoming as a challenge in (Zdancewic, 2004). Therefore, this thesis makes three main contributions in the area of static code analysis:

- It presents a new algorithm that converts source code into a common format providing for uniform static code analysis
- It develops a method that represents source code facts with the finest granularity, suitable for implementing information-flow control checking

- It presents a rule-based semantic reasoner that detects Web vulnerabilities in data-driven applications

1.4.1. Uniform Static Code Analysis

A common limitation of previous approaches that perform static code analysis to detect Web vulnerabilities in data-driven Web systems are their dependence on the programming language of the system under examination (Zdancewic, 2004). We propose an algorithm that converts source code written in different programming languages to a common format to allow for uniform static code analysis. Most tools that offer a standardized source code representation are limited to one programming language and their output dataset is coarse-grained and thus not suitable for our application scope. A recently proposed framework that converts source code to an independent form (Rakić, Budimac, & Savic, 2013) uses enriched Concrete Syntax Trees (eCST) (Rakić & Budimac, 2011) to uniformly represent source code, but the format is a recently proposed, non-standardized description of computer programs.

A few Semantic Web frameworks used to extract source code facts are described in Section 2.3.4, but they only provide a coarse granularity representation that does not allow the implementation of information-flow control methods and they apply to only one programming language. As no other Semantic Web frameworks solutions exist for source code representation (Keivanloo, Forbes, Rilling, & Charland, 2011), then no previous tools are effective in solving our problem. We are therefore compelled to develop a new model, in order to provide the necessary fine granularity for our application scope. A discussion on what consists coarse versus fine granularity is given in Section 3.2.

The proposed framework described in Chapter 3 uses an intermediary form obtained using Abstract Syntax Trees by employing Another Tool for Language Recognition (ANTLR), a well-established parser generator (Parr & Fisher, 2011). The result is source code in the Semantic Web's Resource Description Format (RDF) (W3C, 2004) format that allows a novel rule-based reasoning implementation that detects Web vulnerabilities independently of the input language used. Reasoning using rules to detect Web application vulnerabilities is described in Chapter 4.

1.4.2. Fine-grained Approach

In order to explain our second contribution, an example of an insecure implicit information-flow follows.

```
a = 1; //public variable
if (b == True) { //private variable
        a = 2;
}
```

We assume in this example that `a` is a public variable and `b` is a private variable. The assignment `a = 2` gives out publicly the value of `b`. Specifically, if `a` changes its value to `2` or does not change its value; an attacker can infer that the private variable is `True`, or `False`, respectively. Using static analysis, the `a` variable needs to be tracked across the source code in order to determine if it leaks any private information. A line representation of source code, such as the following, would not be sufficient in this case.

```
<http://domain.com/project/line/129>
<http://domain.com/project/hasContent> "a = 2;".
```

A suitable representation of source code needs to evaluate any assignments in the code. Thus, a fine-grained representation, at variable and assignment level, would provide sufficient details to implement methods of information-flow control. For example, the line `a = 2;` represented in RDF using our proposed algorithm results in the following graph:

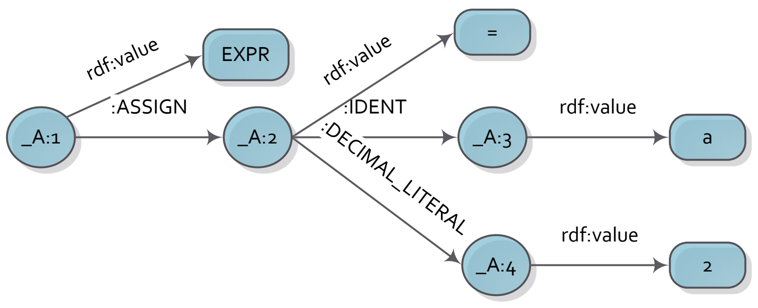


Figure 6 Source code assignment expressed in RDF

Using semantics through an RDF graph, we managed to express the assignment in a way that can be understood by machines and reasoned upon; the assignment is expressed by the `:ASSIGN` predicate; the assigned value is a decimal according to the

`:DECIMAL_LITERAL` predicate and its value equals 2. Therefore, our second contribution is to offer a method that represents source code facts with the finest granularity, suitable for implementing information-flow control. Section 3.2 describes the concepts and techniques used for achieving a fine-grained, independent representation of source code. Chapter 3 describes the source code facts extraction framework.

1.4.3. Detection of Web Application Vulnerabilities through Novel Semantic Web Reasoning

We propose a novel approach to static code analysis that employs Semantic Web technologies to convert source code to a common format where reasoning can be applied to detect Web application vulnerabilities using rules that describe patterns of code vulnerable to security exploits. To the best of our knowledge, a Semantic Web approach to static code analysis has not been proposed in the current literature. The methodology that applies Semantic Web technologies to the detection of security vulnerabilities in Web applications is described in Chapter 4.

The reason behind the choice of Semantic Web technologies is threefold. First, policy enforcement in the context of security and privacy, leveraging Semantic Web technologies, has been previously tested and applied in academia (e.g. Ashri, Payne, Marvin, Surridge, & Taylor, 2004; Kagal, Finin, & Joshi, 2003; Rao & Sadeh, 2005; to name just a few). Although these frameworks do not apply to software, they do provide a proof of concept that security can be enforced with the application of Semantic Web technologies. Second, using Semantic Web, computer programs can be published in a language specifically designed for data (RDF) and inference capabilities suffice for an implementation of a tool that reasons upon data to detect flaws in source code. Third, the Semantic Web offers a new dimension to the analysis of code through its inference capability. Therefore, we believe that a semantically enhanced static code analysis will allow for applications where machines can inference new patterns of vulnerabilities based on rules modelling patterns of code. Some examples that prove this concept are given in Sections 4.3 and 5.2.2.

1.5. Thesis Organization

The rest of this thesis is structured as follows. Chapter 2 presents the literature review from two different perspectives. First, it describes the various methods for checking the security of Web applications. Second, it discusses Semantic Web frameworks applied in the context of security, as well as efforts to convert source code to RDF, together with their shortcomings in terms of applicability to our problem statement. Chapter 3 describes the source code facts extraction framework and the algorithms used for its implementation. Chapter 4 presents the method for detecting Web application vulnerabilities using Semantic Web technologies and its implementation. Experiments and results are given in Chapter 5. Chapter 6 will discuss limitations of the Web vulnerabilities detection methodology and of the techniques employed, as well as their impact on the results. Finally, Chapter 7 will conclude the thesis, and will propose future work.

2. RELATED WORK

Standard security mechanisms, such as access control, do not prevent private information from leaking through computations, since they only control information release, not its propagation. For example, access control mechanisms can be used to restrict the execution of operations on a file only to the administrator of the system, but have no control over what the administrator will do with the information read from the file. Oracle recommends where possible, security to be enforced in the database using stored procedures, but this implementation is neither modular nor portable. Details of these limitations are given in Section 2.1.

In order to detect flaws in Web applications, a promising solution is statically checking information flows within systems that manipulate sensitive data. Both the industry and academia offer a variety of static checks implementations, but they either propose new scripting languages or are designed to operate only on one specific programming language. A discussion of these approaches is given in Section 2.2. The solution proposed in this thesis aims to use standardized representation of source code in RDF to overcome these limitations.

Although the Web of Data is steadily increasing (Parundekar, Knoblock, & Ambite, 2012) with more than 31 billion triples reported to be stored in the Linked Data Cloud in 2011 (Kaoudi & Manolescu, 2013), not many source code facts representation services are available. The few frameworks available have either limited application (e.g. they convert only Java source code) or their representation is high-level, where only metadata and descriptions of methods are made available in RDF. Section 2.3 presents these frameworks.

In addition, Section 2.3.5 discusses the frameworks that enforce security policies using Semantic Web technologies and their limitations in terms of applicability to this thesis context, namely that they apply to different fields than testing data-driven applications.

2.1. Ad-hoc Mechanisms to Enforce Security

The traditional way of enforcing security is through ad-hoc mechanisms, such as security enforced in the database and context-based solutions. While Oracle recommends that when possible security be enforced using stored procedures, most of the time this does not represent the ideal way of preventing certain attacks and thus, other approaches are considered. This section aims to show how difficult it is to implement security and therefore, how imperative it is to explore ways to detect vulnerabilities in Web applications.

2.1.1. Security Enforced in the Database

According to Oracle's security guidelines (Oracle, 2014) there are two questions to consider when creating and implementing security for database applications:

1. Do the application users have database access accounts?
2. Is it better to enforce security in the application or the database?

Oracle recommends that where possible, security should be enforced in the database, leveraging intrinsic security mechanisms, such as stored procedures and auditing; and that security implemented in this way cannot be bypassed. However, in order to enforce security mechanisms of the database, the application users must have linked database access accounts.

For many commercial database applications, the application users do not have database access accounts. For these applications users connect to the database using a single, highly privileged user with database access account; the "One big application user" model (Oracle, 2014) is used.

Apart from commercial applications using the "One big application user" model, enforcing security in the database can fail to make an application's security implementation modular or reusable and thus, less portable, since the same application using a different database will need a reimplementation of the security policy.

2.1.2. Context-based Solutions

Throughout Section 1.2, various forms of attacks and their context-based solutions were described. Most attacks are due to insufficient sanitization of input/output data. Different sanitization solutions apply to different contexts and evidently, to different programming languages, making it very easy for developers to make mistakes and leave room for vulnerabilities (SANS, 2014). When proper sanitization is not in place, vulnerabilities must be identified and fixed.

For example, sanitization of user input is dependent on the context in which the user data is used and thus is difficult to be implemented correctly and relevantly. For example, while a PHP *filter_input* function is suitable for sanitizing input that is echoed back to the webpage, it would not be suitable for a SQLIA (SQL injection attack), where user input sanitization should be implemented using the *mysql_real_escape_string()* or prepared statements and parameter binding.

Sanitization can also be dependent on the language used. In order to prevent a cross-site scripting attack, PHP offers the *HTMLPurifier* library, while in Java the *OWASP Java HTML Sanitizer* can be employed with similar effects.

Sometimes, solutions must be chosen based on performance considerations. The recommended way to prevent SQL injection attacks is through parameterized queries. Unfortunately, this approach can cause performance issues and in this situation, the developer might choose to prevent attacks using stored procedures, or through other means.

2.2. Information-flow Control Using Static Code Analysis

Static analysis complements dynamic analysis by examining the flow of information for all execution paths and variables and not just the ones exercised at runtime (Masri, Podgurski, & Leon, 2004). This feature of static analysis is particularly useful in assuring a program’s security because most security attacks exercise an application in unforeseen and untested ways (Intel, 2012). Static analysis is applied offline to assess an application’s compliance with an information-flow policy. Although current static analysis systems are efficient at finding vulnerabilities (Baca, Carlsson, Petersen, & Lundberg, 2013), their application is limited to a specific programming language. This section reviews these systems and presents their limitations in terms of applicability.

2.2.1. Taint Mode

Malicious users of data-driven applications can try to exploit security weaknesses by cleverly crafting data entered through an HTML form. This type of exploit causes the application to behave in unexpected ways, which ultimately leads to leakage of confidential data.

Taint mode (Masri, Podgurski, & Leon, 2004) is a run-time check provided by several scripting languages, such as Perl and Ruby. A developer can activate taint mode in order to check the logic of CGI scripts. This mode will treat all user input data coming from outside the application and used in potentially unsafe operations (e.g. system calls) as tainted, unless a developer specifically marks the respective data as safe or applies compiled templates to the vulnerable code, i.e. by using prepared statements (OWASP, 2014). Taint mode ignores safe operations, such as print.

SQL injection is one example of tainted data used to execute an attack on a target database. The example below illustrates code vulnerable to an SQL injection attack. In this case, the Perl programming language is used to insert data into a database.

```
#!/usr/bin/perl
my $username = $cgi->param("username"); # Get the name from
the browser
...
$dbh->execute("SELECT * FROM users WHERE username =
'$username';"); # Execute a SQL query
```

The `$username` variable is read from an HTML form and used to retrieve data from a database. If an attacker uses the value `'; DROP TABLE users;--` for the `$username` variable, then the SQL command becomes `SELECT * FROM users WHERE username = ''; DROP TABLE users;--`. The malicious user added a new SQL command with devastating effects: it deletes the entire users table. In order to fix this problem, a developer must either sanitize the `$username` variable or employ prepare and execute statements (OWASP, 2014). Because the `$username` variable is read from outside the application and used in a potentially dangerous operation, i.e. accessing a database, it cannot be trusted and thus, marked as tainted by the Perl's taint mode.

Because Perl's taint mode is a run-time check, if a variable is marked by Perl's taint mode as tainted, the program running with taint mode will fail with a similar error as the following:

```
Insecure dependency in open while running with -T switch at
./program.pl line 10.
```

To be able to run the program, the programmer has to specifically un-taint the variable marked as tainted.

Perl is not the only language that supports taint checking. Taint mode is also a feature of the Ruby scripting language. Ruby uses a variable, named `$SAFE`, to control five security levels (Thomas, Fowler, & Hunt, 2004). The security level ranges from level 0, where no data is marked as tainted, to level 4, where modification of global data is forbidden.

Whilst the taint checking method has been proven both effective and efficient at finding Web application vulnerabilities (Hurst, 2004), not all scripting languages support this mode. Two examples are Python and PHP. Furthermore, taint mode is language dependent, i.e. Perl's taint mode cannot be applied to Python code. Therefore, taint mode does not provide a language independent solution for security checking data-driven applications.

2.2.2. Language-based Information-flow Control

This section describes various methods of implementing information-flow control proposed in academia and it is divided in three parts corresponding to three category types: language extension, language runtime and new scripting language. The first part presents methods that use language extensions and achieve the control of information-flow statically and with little run-time overhead. The second part describes RESIN, a language run-time that can be used to track sensitive data that are annotated with a policy object, but incurs a 33% CPU overhead for HotCRP[7], a conference management system. The last part presents a completely new proposed scripting language that implements security. The first two categories of methods have the limitation that they cannot be applied to existing Web systems written in traditional programming languages, without modifications or added information regarding security policies. Adopting the last category of methods would necessitate rewriting all existing code in the newly proposed scripting language.

[7] http://read.seas.harvard.edu/~kohler/hotcrp/

Language Extension

In academia, researchers have implemented languages or extensions that support information flow policies. Myers (1999) and Simonet & Rocquencourt (2003) proposed extensions of existing programming languages, Java and Objective Caml, respectively. Both proposed approaches use type systems to annotate data and statically check that every information flow is legal concerning a security policy.

Myers (1999) describes JFlow, a new language that extends the Java programming language developed to protect the confidentiality and integrity of sensitive data mostly using statically checked data flow assertions. JFlow's goal is to prevent sensitive information from being leaked through computation and it achieves this using decentralized label modeling (DLM). JFlow uses static checking, allowing a detailed tracking of security classes without incurring run-time overhead. The author provides programmers with a JFlow compiler that will statically check programs written in JFlow. Furthermore, JFlow supports the development of secure applets and servers that handle sensitive data.

```
class passwordFile authority(root) {
      public boolean
            check (String user, String password)
            where authority(root) {
                  // Return whether password is correct
                  boolean match = false;
                  try {
                        for (int i = 0; i < names.length; i++) {
                              if (names[i] == user &&
                              passwords[i] == password) {
                                    match = true;
                                    break;
                              }
                        }
                  }
                  catch (NullPointerException e) {}
                  catch (IndexOutOfBoundsException e) {}
            return declassify(match, {user; password});
      }
      private String [ ] names;
      private String { root: } [ ] passwords;
}
```

Listing 1 A JFlow password file (Myers, 1999)

Although JFlow addressed some of the limitations of previous proposed approaches, (e.g. mutable objects, subclassing, exceptions) programmers are required to write applications in the JFlow language. Listing 1 shows an example of JFlow code that protects passwords using information flow controls. The method `check` accepts two arguments, a username and a password and returns a `boolean` value that indicates whether the password is a match for the username. Declassification is used for practicality reasons in order to relax policies owned by principals, as strict information-flow control is too restrictive to write real applications. The method `check` is executed with the `root` authority, which has the ability to declassify data.

The authors write that due to checking being done statically, there is little run-time overhead. Although the JFlow annotations are used only in situations where security issues may arise, and only three annotations were required to prevent leakage of information in the example from Listing 1, the proposed approach cannot be applied to traditional scripting languages such as PHP, Python or Perl, used in the development of Web information systems.

Flow Caml (Simonet & Rocquencourt, 2003) is an extension of the Objective Caml language that introduces a type system to trace information-flow control. Flow Caml first checks for information leaks using static code analysis and then translates the program under inspection to regular Objective Caml producing code that is safe from Web application attacks. Flow Caml ensures security by assigning policies to variables and expressions. For example, to assign a policy and a value to a variable in Flow Caml, the programmer can use the following expression: `let x1 : !alice int = 42;;` thus the integer `x1` has level `!alice.` The `x1` integer can legally be stored in r1:

```
r1 := x1;;
```

Furthermore, an integer `x2` is declared with the level `!bob`. A rule is defined to forbid an information flow from `!bob` to `!alice`, and thus assigning it to `r1` raises a typing error:

```
r1 := x2;;
```

This expression generates the following information flow(s):

from !bob to !alice

which are not legal. (Simonet & Rocquencourt, 2003)

Flow Caml's default security policy does not allow information to flow from one principal to another, where `stdout` and `alice` are principals. However, information flow constraints can be redefined using the `flow` keyword and resulting in a security policy relaxation. In this case, the assertion `flow !alice < !stdout;;` would allow information to flow from the principal `alice` to `stdout` and thus, printed to standard output.

A few differences exist in the way these proposed extensions implement security. Simonet's Flow Caml features polymorphism and a full type inference algorithm, while Myers' JFlow uses monomorphic types and must fully annotate, including their security level, the arguments of methods. Although both approaches are successful at checking that a program is obeying a security policy, they both require existing systems to be rewritten using the proposed languages and the added extensions, which make them infeasible to be applied to already developed Web systems in popular scripting languages, such as PHP, Python and Perl.

Language Runtime

Yip et al. (2009) introduced a new language runtime called RESIN that prevents security vulnerabilities using three concepts: policy objects, data tracking and filter objects. Developers of applications must specify the policy and filter objects in the application's programming language. At runtime, RESIN will track sensitive data annotated with a policy object and will not allow this data to cross a boundary defined in a filter object.

For example, to track passwords in programs written in the PHP programming language, programmers will use RESIN concepts to annotate the passwords with policy objects, similar to the one in Listing 2. If RESIN catches any inappropriate disclosure of passwords annotated with such policy, it will throw an exception. The runtime is a modified interpreter of the PHP language causing CPU overhead.

```
class PasswordPolicy extends Policy {
	private $email;
	function __construct($email) {
		$this->email = $email;
	}
	function export_check($context) {
		if ($context['type'] == 'email' &&
			$context['email'] == $this->email) return;
		global $Me;
		if ($context['type'] == 'http' &&
			$Me->privChair) return;
		throw new Exception('unauthorized disclosure');
	}
}

policy_add($password, new PasswordPolicy('u@foo.com'));
```

Listing 2 "Simplified PHP code for defining the HotCRP password policy class and annotating the password data. This policy only allows a password to be disclosed to the user's own email address or to the program chair." (Yip, Wang, Zeldovich, & Kaashoek, 2009)

Although RESIN is effective at preventing a series of common attacks, it does not use static checks of code. RESIN is not a method employed to verify that the code is written with safety in mind, but a mechanism that prevents leakage of confidential data by applications that contain vulnerable code. Furthermore, RESIN adds extra overhead in terms of application processing time (i.e. it adds 33% CPU over-head when generating a page for a conference management software). Programmers are still faced with the challenge of learning new concepts, writing policy objects, data flow assertions and filter objects, although the specifications are expressed using existing programming languages.

New Scripting Language

Others propose completely new Web scripting languages to enforce confidentiality and integrity of data. For example, Li (2005) and Zdancewic (2005) introduced a domain-specific Web scripting language designed for interfacing with databases. Their rationale behind this approach includes the implementation of DBMS stored procedures as not modular, nor reusable, and taint mode as only an ad-hoc mechanism. Li's proposed Web scripting language is similar to PHP.

Li's scripting language uses strongly typed queries to the application's underlying database to prevent security violations. Listing 3 shows an example of a Web script written in the newly proposed scripting language. It implements two query interfaces to enable interaction with the database.

```
<?ssp_header
     FormInputs ("UserName" => u, "Password" => p, "QueryYear" => y);

     Query GetID ( username: !tainted ) => (
          PASSWORD : {this=*},
          ID : {if (PASSWORD=*) this 0}
      );

     Query FetchRecords( index: !untainted, year:!{Integer(*)} ) => (
            ORDERID : public,
            AMOUNT : public,
            CCNUM : {tailstr(this,4)}
      );

      Variables ( pub_id: public!untainted );
 !ssp_header>
 <html><head><title>....</title>
 <meta http-equiv="content-type" content="text/html; charset=UTF-8">
 </head><body>
 ......
 <?ssp
       q1 := query GetID(u);
       if ( empty(q1) ) {
             print 'Unknown username';
       } else {
             (pwd, id) := readrow(q1);
             L_AUTH: if (pwd=p)
             {
                   print 'Username = '; print u;
                   pub_id := declassify(id, L_AUTH:(pwd=*));
                   print 'School ID ='; print pub_id;
                   q2 := query FetchRecords( pub_id, Integer(y) );
                   while (!empty(q2)) {
                         (orderid, amount, ccnum) := readrow(q2);
                         print 'Order ID = '; print orderid;
                         print 'Amount = '; print amount;
                         print 'Credit Card = XXXX-XXXX-XXXX-';
                         print tailstr(ccnum, 4);
                  }
            } else {
                  print 'Wrong password';
            }
      }
 !ssp>
```

Listing 3 Web script example of Li's proposed language (Li, 2005)

It reads the username and password as input, fetches records from the database and displays results. Public, tainted data coming from outside the application is marked with the "`public ! tainted`" security level and thus considered untrusted by the application. Using the security level `"!untainted"`, Li's scripting language can prevent passing tainted data to the index variable, marked using this security level (e.g. `index: !untainted`). Moreover, the argument `year` in the `FetchRecords` query, can only be of type `integer`, achieved by converting data received as a `string` to an `integer`. This type of enforcement can stop certain SQL injection attacks, as it does not allow reading strings that can be used as SQL queries.

Although the proposed Web scripting language provides strong enforcement of confidentiality and integrity policies, it is a completely new scripting language. This makes it infeasible to be applied to existing systems written in popular programming languages.

2.2.3. Java Bytecode Analysis

Some research only focusses on testing the security of the Java programming language (Graf, Hecker, & Mohr, 2013; Hammer, 2010 and Barthe, Pichardie, & Rezk, 2007) due to its popularity.

Hammer (2010) uses system dependence graphs (SDG) to represent the semantics of a Java program and their major contribution is the implementation of the first dependence-graph-based information-flow control application for full Java bytecode, including taking into account exceptions, constructors, or unstructured control flow. The validation of the code is achieved through graph traversal in the style of program slicing and according to the non-interference security policy, assures that no secret data is leaked to unauthorized users of the system. Program slicing considers statements that carry information used in the statement that is being analyzed for security compliance. A statement's (backward) slice contains all the statements that might semantically influence the statement under inspection. Using this method, the author reduced the annotation burden associated with type systems, such as those presented in Section 2.2.2.

Based on the above work, Graf, Hecker, & Mohr (2013) developed more recent work on an information-flow control method for Java bytecode. Their solution is also based on SDGs, but with a focus on practicality. The authors state that in contrast with other theoretical methods, their approach is the first tool that can check for both possible and probable information flow leaks. Their proposed approach can deal with sequential and multithreaded medium sized programs, in some cases, up to 100kLoC (lines of code). One of their contributions is a user interface of their IFC framework (Information-Flow Control), named IFC console, used to analyze information flow.

Barthe, Pichardie, & Rezk (2007) propose an information-flow control method operating on Java bytecode as well. They handle the unstructured nature of bytecode programs caused by jumps and exceptions using three successive phases. A PA (pre-analyzer) phase detects branches that will never be taken. A CDR (control dependence regions) analyzer reduces the size of the control graph by eliminating the branches found in the PA phase. Lastly, an IF (information flow) analyzer verifies the correctness of the program according to the non-interference policy.

Table 1 Most popular websites and their server-side languages (Rossum, 2006; Hoff, 2008; Campbell, 2010)

Website	Popularity (unique visitors)	Server-side language
Google.com	1,000,000,000	C, C++, Go, Java, Python, PHP
Facebook.com	880,000,000	PHP, C++, Java, Python, FBML, Ajax, Erlang, D, Xhp
YouTube.com	800,000,000	C, Python, Java
Yahoo	590,000,000	PHP
Live.com	490,000,000	ASP.NET
MSN.com	440,000,000	ASP.NET
Wikipedia.org	410,000,000	PHP
Blogger	340,000,000	Python
Bing	230,000,000	ASP.NET
Twitter.com	160,000,000	C++, Java, Scala, Ruby on Rails

The methods presented in this section have a common limitation; they only apply to Java bytecode. Although these approaches offer promising results, other programming languages need to be considered. Google released a list of the top 1000 visited websites in the world[8]. Part of this list is illustrated in Table 1, which includes the first 10 websites, their popularity and the server-side programming language used for implementation. This

[8] http://www.google.com/adplanner/static/top1000/

table shows that although Java and PHP seem to be more popular, a substantial number of websites are implemented using different languages.

Note that Java, PHP and Python are the most common used languages, but others, such as ASP.NET and C are also used in website development. Because industries employ many other programming languages than Java, approaches should consider a uniform way of securing code independently of the underlying code syntax.

2.3. Semantic Web Concepts and Practices

The Semantic Web is the grand vision of machines understanding the meaning of information and rests on the theory that this can be achieved by accommodating semantics of Web data (Gruber, 2008). While critics speculated the Semantic Web vision to be infeasible (Richardson, Agrawal, & Domingos, 2003), advocates validated the original concept with applications in industry, biology and human science (Feigenbaum, Herman, Hongsermeier, Neumann, & Stephens, 2007). Typically, data from various sources is represented as triples in the Resource Description Framework (RDF) format with the purpose of being uniformly queried by machines in order to solve a specific problem or answer a specific question (Arenas & Perez, 2011). In this paper, we apply RDF to static code analysis.

Semantic Web technologies can provide a solution to the limitations of existing information-flow control using the static analysis methods presented in Section 2.2. Using Semantic Web technologies, programming code can be exported to a common format, where reasoning using rules and inference engines can be uniformly implemented independently of the input language. This research aims to implement a method that finds Web application vulnerabilities for any existing Web system, regardless of the chosen language for its development. In this approach, new languages can be supported by implementing a translator to the common format: the existing analysis engine can then be applied without modification.

This section reviews different technologies that the Semantic Web offers starting with an introduction on vocabularies and the Resource Description Framework (RDF) data model, which the Semantic Web builds upon. A Semantic Web framework with flexible inference mechanisms will be described in Section 2.3.3. This framework will be used by the analysis method described in Chapter 4. For the sake of clarity, Section 2.3.3 will also describe how reasoning with rules works. Section 2.3.4 continues with presentation of current methods that extract source code facts and represent them with RDF, as well as their limitations. Section 2.3.5 will finish with the presentation of existing frameworks that use Semantic Web technologies to apply security concepts. Although these frameworks apply in a different context than static analysis of source code, they demonstrate that Semantic Web reasoning and RDF can be successfully employed to implement security.

2.3.1. Vocabularies

Vocabularies (W3C, 2013) are used on the Semantic Web to define concepts and relationships of a domain of interest. Relationships are also called "terms". If vocabularies are complex, containing several thousands of terms, they are usually referred to as ontologies.

Vocabularies are commonly used to enable applications that use data integration for solving problems. An example is the use of vocabularies to describe pharmaceutical products and patient data. Integrating the knowledge from both vocabularies and using Semantic Web technologies, decision support systems can be developed that can automatically prescribe treatments for patients.

In order to represent source code in the RDF format, as well as reasoning on top of the RDF data, we used two vocabularies: the RDF concepts vocabulary (RDF) and Flow Control (FC). Note that vocabularies are abbreviated using capital letters, but when used to represent data in the RDF format, the abbreviation uses lower case letters. The purpose of existing vocabularies is to offer concepts and relationships that are only useful for representing certain data, but that can be also reused by other applications. The RDF concepts vocabulary (RDF) was chosen to represent some relationships. The RDF vocabulary provides a couple of terms that can be used by the proposed methodology to create anonymous nodes (`rdf:parseType Resource`) and to assign values to nodes (`rdf:value`). More information about the use of these terms is given in Chapter 3. Although the authors of the RDF vocabulary chose the same abbreviation as the format itself (i.e. RDF); it should be clear from the context in which it appears, whether RDF is used as a vocabulary rather than the format for representing data. Flow Control (FC) is a vocabulary created for the purpose of this research, in order to represent concepts such as `fc:assign`, `fc:number`, `fc:variable`, in the source code representation realm.

The concepts and relationships for representing source code facts necessary for the development of this research are derived from the grammar of a programming language. Grammars are explained in Section 3.2.2. For example, if the source code facts extraction framework described in Chapter 3 extracts source code facts from a PHP program, then the FC vocabulary's concepts and relationships are derived from the description of the PHP grammar. More details are given in Chapter 3. To the best of our knowledge, vocabularies that are derived from the grammars used by the proposed methodology do not exist.

Vocabularies have namespaces that are unique on the Web. For example, the RDF vocabulary can be accessed at `http://www.w3.org/1999/02/22-rdf-syntax-ns#`. Unique namespaces are necessary to avoid name conflicts. For example, the term `lift` can be used as both a verb and a noun and can be a member of two vocabularies. One vocabulary might describe a building and use this term to represent data about an elevator. Another vocabulary could be a representation of a body builder's activities and thus, lift could be used to describe the action of lifting weights.

A unique URI was used for the source code facts namespace: `http://oanaureche.com/flowcontrol`. It should be noted that this unique URI was not necessary, as the source code facts data represented in RDF are not published on the Web and therefore there is no risk of name conflicts. However, for the sake of compliance with the Semantic Web vision and for future work that might involve publishing or linking data on the Web, a unique URI was created. Furthermore, the FC vocabulary has no contents as the concepts and relationships are created on the fly, derived from the description of the language grammar. Future work that involves publishing data should include creating a vocabulary for the source code facts representation, in order for source code represented in RDF to have meaning for machines crawling or reasoning on the Web. For the proposed methodology, this step was not necessary, as work is done internally.

2.3.2. RDF Data Model

RDF (W3C, 2004) is a W3C recommendation and a specification for defining information on the Web. RDF supports the Extensible Markup Language (XML) syntax and it is used to make statements about Web resources represented by Universal Resource Identifiers (URIs).

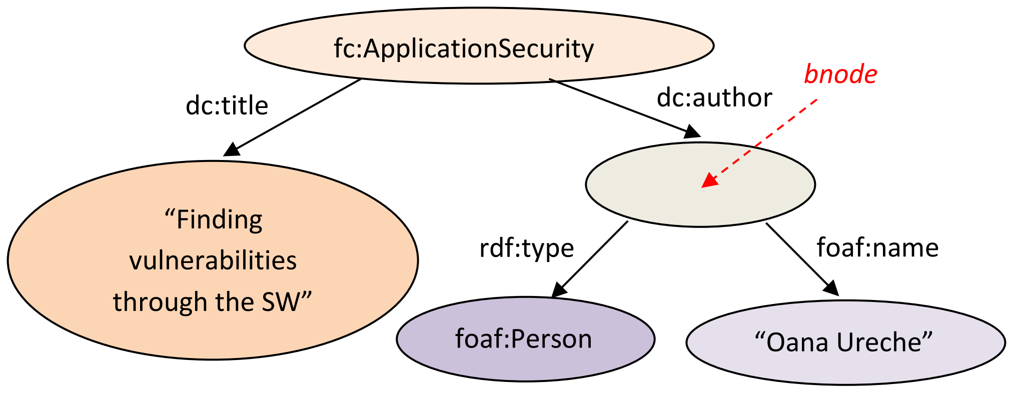


Figure 7 Example of an RDF graph

The underlying structure of an RDF statement is a directed graph representing triples in the form of subject-predicate-object. For example, the RDF graph in Figure 7 makes the following statements: “A document with the URI `fc:ApplicationSecurity` has the title ‘Finding vulnerabilities through the SW’. The author of this document is a person and her name is ‘Oana Ureche’”. RDF can use a blank node (also called an anonymous node or *bnode*) to create a statement. A blank node represents a resource for which there is no URI or literal.

The meaning of the predicates, subjects and objects is given by the vocabularies used: Dublin Core[9] (DC), RDF concepts, Friend of a Friend (FOAF) and lastly, Flow Control[10] (FC).

Commonly, two RDF serialization formats are used: RDF/XML[11] and Notation 3[12] (N3). Listing 4 and Listing 5 describe the RDF graph from Figure 7, using these serialization

[9] http://purl.org/dc/elements/1.1/
[10] http://xmlns.com/foaf/0.1/
[11] http://www.w3.org/TR/rdf-syntax-grammar/
[12] http://www.w3.org/2000/10/swap/Primer

formats. N3 was proposed as an alternative to the RDF/XML serialization, due to its readability and the ease of writing RDF statements.

```
<?xml version="1.1"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:foaf="http://xmlns.com/foaf/0.1/"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:fc="http://oanaureche.com/flowcontrol/">
<rdf:Description rdf:about="fc:ApplicationSecurity">
  <dc:title>Finding vulnerabilities through the SW</dc:title>
     <dc:author>
       <rdf:Description foaf:name="Oana Ureche">
          <rdf:type
                rdf:resource="http://xmlns.com/foaf/0.1/Person/"/>
        </rdf:Description>
     </dc:author>
</rdf:Description>
</rdf:RDF>
```

Listing 4 Representation of RDF statements using RDF/XML

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix fc: <http://oanaureche.com/flowcontrol/> .

<fc:ApplicationSecurity> dc:author _:b1;
      dc:title "Finding vulnerabilities through the SW" .
_:b1 a <http://xmlns.com/foaf/0.1/Person/>;
      foaf:name "Oana Ureche" .
```

Listing 5 Representation of RDF statements using N3

As previously mentioned, a blank node represents a resource for which there is no URI or literal. It can serve as a parent node to a grouping of data. Such nodes can be observed in the graph as well as in the two different serialization formats. In the statement

```
<fc:ApplicationSecurity> dc:author _:b1;
```

the `_:b1` node is the blank node. Notation 3 uses the `_:` namespace to identify blank nodes. There is no commitment to the chosen name `b1`, but it has to be consistent throughout the RDF serialization to identify statements that have the node `:_b1` in common. When writing Jena rules to infer statements, the `?` symbol is used to identify blank nodes. Rules and inference mechanisms are discussed in the next section.

2.3.3. Reasoning with Jena Rules

Jena is a leading Semantic Web toolkit for Java programmers (Carroll, et al., 2004). The open source Jena framework provides Java libraries for flexible inference using Semantic Web technologies. It is important to describe this framework, as it will constitute the programming toolkit for the methodology implementation phase described in Chapter 4. Jena was chosen as it is the most widespread Semantic Web framework (Lindorfer, 2010) and it is sufficient for this project. While there are other frameworks, it falls out of the scope of this research to analyze which framework is the most productive or usable.

The Jena machinery is designed to be used for general inference purposes and it includes a generic rule engine that supports user-defined rules. By Jena definition, the term *inference* represents the action of deriving additional information, while *reasoner* refers to the code function that performs the inference task.

```
Rule :=   bare-rule .
          or   [ bare-rule ]
          or   [ ruleName : bare-rule ]

bare-rule :=   term, ... term -> hterm, ... hterm    // forward rule
          or   bhterm <- term, ... term              // backward rule

hterm     :=   term
          or   [ bare-rule ]

term      :=   (node, node, node)        // triple pattern
          or   (node, node, functor)     // extended triple pattern
          or   builtin(node, ... node)   // invoke procedural primitive

bhterm      :=   (node, node, node)      // triple pattern

functor   :=   functorName(node, ... node)   // structured literal

node      :=   uri-ref              // e.g. http://foo.com/eg
          or   prefix:localname     // e.g. rdf:type
          or   <uri-ref>            // e.g. <myscheme:myuri>
          or   ?varname             // variable
          or   'a literal'          // a plain string literal
          or   'lex'^^typeURI       // a typed literal
          or   number               // e.g. 42 or 25.5
```

Listing 6 Definition of Jena rules[13]

Listing 6 shows the description of Jena rules syntax. There are two parts to a rule: the *If* clause and the *Then* clause. The rules written throughout the proposed methodology use

[13] http://jena.apache.org/documentation/inference/

the forward chaining method. Forward chaining starts with available data and infers new data when the antecedent (*If* clause) is met. Using the forward chaining method an inference engine will look for statements that match the antecedent and, when found, the consequent (*Then* clause) is inferred.

Commonly, applications that access the Jena inference machinery use the *ModelFactory* class to associate a data set with some reasoner, in order to create a new *InfModel*. The example in Listing 7 illustrates this process. It uses a generic rule reasoner to programmer-defined rules. The new resulting model (e.g. `InfModel inf`) can be queried for data. Statements that already existed in the original dataset will be found together with new inferred statements resulting from the application of programmer-defined rules.

```
String rules = "[rule1: (?a :hasFather ?b) (?b :hasBrother ?c) ->
                  (?a :hasUncle ?c)]";
Reasoner reasoner = new GenericRuleReasoner(Rule.parseRules(rules));
InfModel inf = ModelFactory.createInfModel(reasoner, rawData);
```

Listing 7 Inference with generic rule reasoner

Consider the raw data as the statements in Listing 8 together with the rule that defines what it means to be an uncle. After applying reasoning in the form of the code from Listing 7, the inferred statement will be `:John :hasUncle :James .` When queried for statements, the new model (e.g. `InfModel inf`) will return three statements, the original two statements and the inferred statement.

```
@prefix : <http://namespaceexample.org/>
:John :hasFather :Jim .
:Jim :hasBrother :James .
{ ?a :hasFather ?b . ?b :hasBrother ?c . } -> { ?a :hasUncle ?c } .
```

Listing 8 Raw data and rule defining what it means to be an uncle

This section briefly presented the syntax of a Jena rule and a small reasoning example. Anonymous nodes were discussed in the previous section. Although these examples are simplistic, these concepts will be applied in the vulnerability detection method described in Chapter 4.

2.3.4. Representation of Source Code Triples

A few Semantic Web frameworks to extract source code facts exist for different purposes. This section presents these frameworks and describes their shortcomings in terms of our application scope.

Ganapathy and Sagayaraj (2011) proposed a framework that extracts Java source code metadata and stores it in the OWL (Web Ontology Language) using the Jena framework, briefly described in Section 2.3.3. The metadata extraction process is achieved using QDox[14], a parser for extracting definitions of classes, interfaces and methods from Java source code files. Their motivation was driven by the fact that organizations spend a lot of time re-doing the same work that had been done already, work that had already been tested. Their goal was to help organizations in reusing existing code, and thus, to reduce their production time. Using the proposed framework, developers can search for methods using parameters that match the metadata stored in the OWL. Once a method is found, the source code corresponding to the respective method is retrieved from a Hadoop Distributed File System (HDFS) repository.

The limitation of this framework is that it only stores high-level source code facts in the OWL, such as function, definition, type, arguments, brief description, author etc. The code inside a method is only stored intact in a HDFS repository for future retrieval. Therefore, we cannot perform a data-flow analysis on the resulting dataset of the proposed framework.

Keivanloo et al. (2011) was concerned with two research challenges identified in (W3C, 2004): 1) design an ontology for source code ecosystems, covering a wide range of aspects, such as revisions, presentation, syntax and semantics; and 2) create a generation schema to uniquely identify source code entities. They embedded the implementation of these two requirements into a Linked Data publication framework (SeCOLD) that provides access to 1.5 billion triples of source code data. Software engineering researchers and practitioners can use this data for mining, searching and analysis purposes.

[14] http://qdox.codehaus.org/index.html

The resulting datasets, with respect to the granularity of the source code facts, have a coarse granularity that does not go deeper than the source code line level. For example, consider the following Java code, where `128`, `129` and `130` are the line numbers:

```
128       public void printData() {
129               System.out.println(this.data);
130       }
```

The SeCOLD framework would express the line with number `129` using the following triple. Note that the line content is not dissected any further. Therefore, this coarse granularity does not allow information-flow control, where the information regarding the flow of a variable is imperative. For example, if data were modified using a side channel, it would not be observed. Because the proposed model covers only high-level aspects of source code it does not meet our requirements in terms of granularity.

```
<http://domain.com/secold/resource/line/project_link/129>
<http://domain.com/ontologies/2010/11/socon/hasContent>
"System.out.println(this.data);" .
```

Furthermore, the framework only applies to Java source code, because the authors did not include source code facts extraction methods for other programming languages. This also makes this framework unsuitable for our purposes.

As no other Semantic Web framework solutions exist for source code representation (Keivanloo, Forbes, Rilling, & Charland, 2011), then no previous tools are effective at solving our problem. We are therefore compelled to develop a new model, in order to provide the necessary fine granularity for our application scope.

2.3.5. Enforcing Security and Privacy Using Semantic Web Reasoning

Policy enforcement in the context of security and privacy, leveraging Semantic Web technologies, has been previously tested and applied in academia (e.g. Ashri et al., 2004; Kagal, Finin & Joshi, 2003; Rao & Sadeh, 2005; to name just a few). Particularly, in (Rao & Sadeh, 2005) the authors implemented a Semantic Web framework and a meta-control model to control service discovery and access of information using Policy Enforcing Agents. Rao and Sadeh are concerned with the issue of enforcing security and privacy policies in pervasive computing environments. Apart from enforcing access of data, a particular type of PEA, namely an Information Disclosure Agent (IDA) is responsible for enforcing obfuscation policies concerning the accuracy or inaccuracy of the released data.

The framework exercises several Semantic Web concepts. First, it uses OWL-S (W3C, 2004) to advertise service profiles modeling sources of information. Second, services have owners that set their policies using rules, translated into CLIPS using XSLT. Third, rules related to ontologies are expressed using ROWL (Gandon & Sadeh, 2004). Finally, the IDA is implemented in JESS, one of the fastest rule engines available[15].

Similar to Rao and Sadeh's work, Kagal, Finin and Joshi (2003) used OWL-S to describe Semantic Web services and implemented an algorithm to check their policy compliance. The purpose of the work was to handle users' private information by using Web services that reason about their users' policies. Two kinds of policies are used. Privacy policies specify how private information is accessed (e.g. under what conditions). Authorization policies specify who can access the private information. In order to write the policies, the authors used Rei[16], a policy specification language.

Ashri et al. (2004) argue that in order to be able to provide dynamic and adaptive network security, conventional security mechanisms need to be combined with the openness and expressive power associated with Semantic Web technologies. Their research uses semantic reasoning to implement a security device, termed a Semantic Firewall. The Semantic Firewall would provide a safe environment supporting autonomous systems and their interactions without direct user intervention. Leveraging semantic reasoning allows services from different organization to safely communicate using information from security requirements and capabilities attached to each service. This information can allow or disallow interaction based on the security data associated with the service.

In this section, we have shown that reasoning with rules and ontologies in order to enforce security and integrity of information has been previously proposed in academia. Although these frameworks do not apply to source code analysis, they do provide the proof of concept that security can be enforced with the application of Semantic Web technologies.

[15] http://www.jessrules.com/
[16] http://rei.umbc.edu/

3. SOURCE CODE FACTS EXTRACTION FRAMEWORK

This chapter gives a description of the proposed source code facts extraction framework. Section 3.1 gives an overview of the framework and the chapter continues with detailed descriptions of the components involved in the framework.

3.1. Framework Overview

Section 2.3.4 contained a few examples of academic efforts to extract source code facts from programming code, but their limitations included a coarse-grained approach or their method only applied to one programming language (Java). While the Java language is popular, Section 2.2.3 demonstrated that other programming languages are extensively used by popular websites and thus, the focus should be on a uniform method to extract source code facts independently of the input language.

As part of this research, the aim is to offer a service that represents source code within a common format, where reasoning can be achieved, in terms of assuring code security. Therefore, two challenges were identified, which are also part of the thesis contributions, presented in Section 1.4.

1) Uniform static code analysis: to develop a framework that converts source code to RDF independently of the input language
2) Fine-grained approach: to extract source code facts with enough detail for enforcing security policies through the information-flow control

The challenge to provide a solution independent of the programming language results in the need to keep input data and business logic separate. Thus, we employed APIs from the ANTLR[17] (Another Tool for Language Recognition) library to parse external sources with their own specific syntax. Specifically, ANTLR allows for adapting the interface between the source code and the language grammar without changing the implementation. Because we separate the source code language from the reasoning part, using an independent syntax for the latter, we further prevent coupling. Figure 8 shows the translation process using the ANTLR parser generator and an XQuery processor. The translation process has two phases: a language specific phase and language independent phase.

[17] http://www.antlr.org/

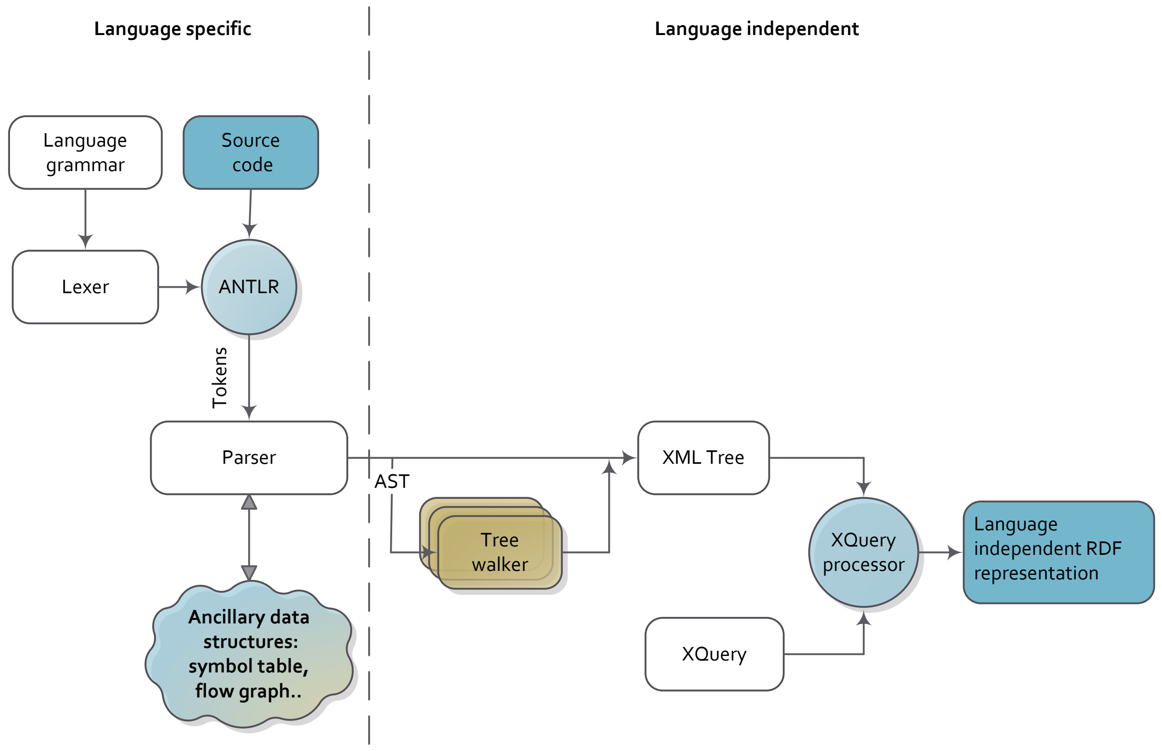


Figure 8 Source code facts extraction framework; allows for static code analysis

The language specific phase uses data specific to the programming language given as input (e.g. the source code and its language grammar) and transforms the input into an independent form. We mentioned that the representation of source code must be of the finest level of detail in order to allow the implementation of information-flow control methods. Abstract Syntax Trees (ASTs) are used to provide the lowest level of detail through nodes and edges (Appel & Ginsburg, 1998) containing information about the source program in a tree-like structure (Zou & Kontogiannis, 2001). ASTs achieve this using lexers and parsers that match symbols from the language grammar with the source program given as input and attach semantics to the matched symbols.

The language independent phase receives as input an AST. This second phase's process maps the AST into a form that allows static code analysis using Semantic Web reasoning. Our framework uses XML as an intermediary between an AST and source code represented in RDF. First, similar to an AST, an XML document provides a representation of data in a tree manner using nodes and edges to encode relationships (attributes) between objects. Second, XML can be transformed into RDF using an XQuery processor. We used a standardized open source Saxon XQuery processor[18]. It provides the functionality needed for the proposed methodology and it is available for Java, therefore easy to integrate into our implementation.

3.2. Independent Fine-grained Representation

One of the challenges encountered during this research, which is also one of the contributions of this thesis is the representation of source code facts independently of the input language used by a Web system susceptible to attacks. Another challenge and contribution is that this representation should be suitable for further information-flow control method implementation. That is, it should be fine-grained. The contributions were presented in Section 1.4. The need for a fine-grained approach was explained in Section 1.4.2. To summarize, a source code fact such as `“a = 2”` needs to be broken down into three units `“a”, “=”, “2”` to allow for static code analysis. Section 2.3.4 described a few frameworks that extract source code facts, but their limitation is that their extraction is coarse-grained, such as this above example, or even coarser, by only extracting metadata.

3.2.1. Abstract and Concrete Syntax Trees

Abstract Syntax Trees (ASTs) are used to provide the lowest level of detail through nodes and edges (Appel & Ginsburg, 1998) containing information about the source program in a tree-like structure (Zou & Kontogiannis, 2001). ASTs achieve this using lexers and parsers that match symbols from the language grammar with the source program given as input and attach semantics to the matched symbols. More information about lexers and parsers can be found in Section 3.2.3. Abstract Syntax Trees are widely used by compilers due to their ability of representing source code structure (Aho, Lam, Sethi, & Ullman, 2006). Eclipse[19] is one program that uses ASTs, not only for compiling purposes, but also for tracking function and variable names throughout the project package.

In (Aho, Lam, Sethi, & Ullman, 2006) the authors used ASTs as a starting point in designing a translator. In an AST, interior nodes represent operators and the children of the nodes represent operands. For the tree in Figure 9 the + is the root and the operator, while the union `8-4` and `1` are operands. Leveraging the same concepts, in the expression `8-4`,

[18] http://saxon.sourceforge.net/
[19] http://www.eclipse.org/

– represents an operator, while 8 and 4 are operands. In general, to represent a programming construct using ASTs, an operator for the constructor is created and the operands are represented by the semantically meaningful components of that construct. The order of the operators and operands is preserved in an AST tree by reading the tree using an in in-order traversal algorithm. However, if we are trying to match the pattern `a = b + c`, in order to trace the variable `c`, then the pattern will match the variable `b` also as a variable to trace. Therefore, the second match results in a false positive. These false positives are described in Section 5.1.1.

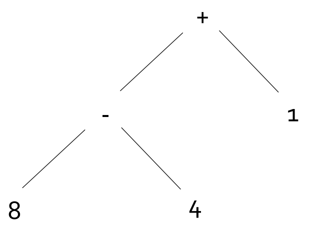


Figure 9 Abstract Syntax Tree for 8 – 4 + 1

As opposed to an AST where interior nodes represent programming constructs, in a Concrete Syntax Tree (CST), or a parse tree, the interior nodes represent nonterminals (Ranta, 2012). Nonterminals can many times be programming constructs, but they can be also represented by “helpers”, such as semicolons, parenthesis etc. These “helpers” have been dropped in the building of an AST, as they are not needed for analysing. Therefore, the syntactic clutter is removed from the parse tree by discarding all the information that is necessary for parsing the code, but irrelevant for static code analysis. This research uses ASTs to represent source code without the syntactic clutter that is irrelevant for static code analysis.

An AST for the Java code line `return a + 2;` will produce the AST from Figure 10 as generated by this thesis’ proposed methodology algorithm.

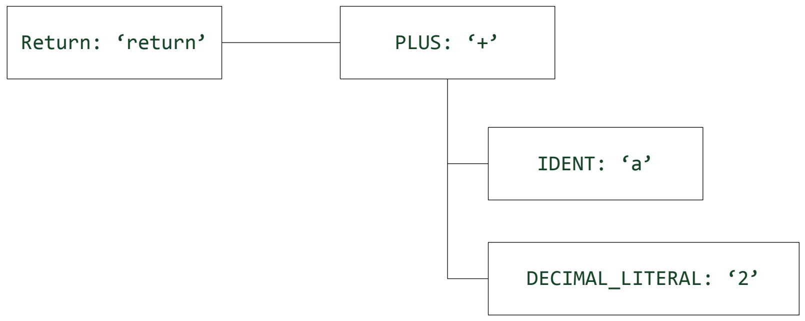


Figure 10 Generated AST for Java code `return a + 2;`

Figure 10 shows that unnecessary syntax for analysis of code has been omitted from the generated tree (e.g. the semicolon), while the line of code was broken into four parts, each enhanced with semantics and generating a fine-grained result to allow for static code analysis.

To generate an AST from programming code, the ANTLR library was employed. A brief introduction of the ANTLR library was given in Section 3.1. The reason for using a parser generator library was also mentioned in Section 3.1 and it is driven by the need to keep the input data and business logic separate. Due to its ability to parse external sources with their own specific syntax and to generate ASTs, ANTLR can transform input into an independent form, used later by the methodology for uniform reasoning.

Thus, the methodology uses ANTLR and ASTs to transform programming code into a form independent from the input syntax. This output is detailed enough to allow for uniform static code analysis.

3.2.2. Language Grammar

Language description files, also called language grammars, are available from the ANTLR website’s grammar list[20]. A comprehensive list includes grammars for Java, PHP, C#, Pascal and many other programming languages. The methodology uses grammars from this list and the ANTLR APIs to generate lexers and parsers for building and traversing ASTs. Lexers and parsers will be discussed in the next section. An example of a PHP grammar file is given in the APPENDIX.

[20] http://www.antlr3.org/grammar/list.html

To generate a parser that outputs an AST from a grammar, the methodology includes the command `output = AST;` inside the `options` section of the grammar file.

It should be noted that the contents of grammar files are changeable. Although there was flexibility in changing contents of grammar files, which may have provided better results, this was not possible due to time restrictions. Although existing grammars caused limitations in experimental results, it was still possible to obtain a proof of concept. The limitations will be discussed in Chapter 6. Therefore, future work should include rewriting language grammars.

3.2.3. Lexers and Parsers

The source code facts extraction framework uses ANTLR's tool `org.antlr.Tool` to generate source files for the lexer and parser. The generated files are copied in the development environment to be compiled. The compiled files are called using the ANTLR's library API to generate the intermediate tree representation (AST) for further processing.

The lexer attaches meanings to the lexemes read from an input stream (i.e. source code). For example, in the PHP grammar from the APPENDIX, the lexemes: `'=='`, `'!='`, `'==='`, `'!=='` are classified as the `EqualityOperator` token by the PHP lexer. The lexer outputs a token stream, which is fed into the parser. Considering the line of code:

```
a = 1; //assign
```

the stream of tokens from a lexer to a parser will look as depicted in Figure 11.

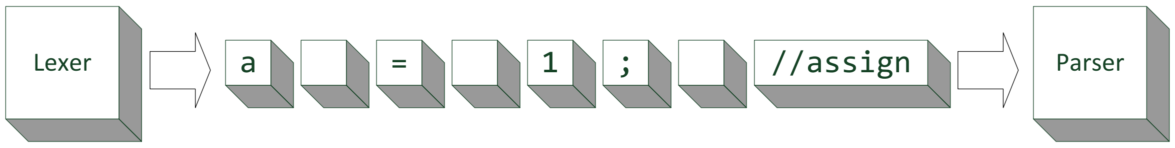


Figure 11 Stream of tokens from a lexer to a parser for the line of code `a = 1; //assign`

The parser reads the token stream and matches patterns according to the specified rules. Consider the following lexer and parser rules:

```
/*------------------------------------------------------------------
 * PARSER RULES
 *------------------------------------------------------------------*/

add:  NUMBER PLUS NUMBER;

/*------------------------------------------------------------------
 * LEXER RULES
 *------------------------------------------------------------------*/

NUMBER: ('0'..'9')+ ;

PLUS: '+';
```

For the `1 + 2` token stream the parser will match it as an `add` operation between two lexemes of type `NUMBER`. The action performed by the source code facts extraction framework is to generate an AST based on the matched pattern. This action results in the AST depicted in Figure 12.

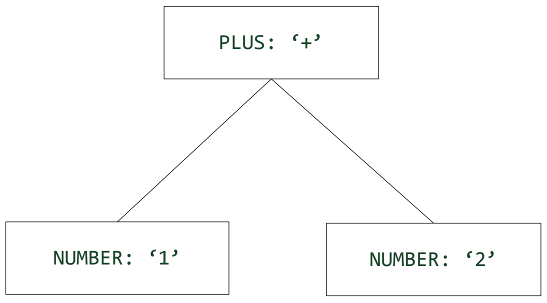


Figure 12 Generated AST for the 1 + 2 token stream

The generated AST is a Java object of type `CommonTree`[21] from the Java ANTLR API and it is used to manually traverse the AST by getting the children, token types etc. The algorithm for tree traversal is given in Section 3.3.1. The objective of the methodology is to obtain an independent form where reasoning about security vulnerabilities is possible. Section 3.2 focused on how to provide a fine-grained representation of source code. The framework achieves this using ASTs and the ANTLR’s library APIs. Section 3.3 describes the process of transforming source code represented by ASTs into a format (i.e. RDF) where static code analysis independent of the input format can be implemented.

[21] http://www.antlr3.org/api/Java/org/antlr/runtime/tree/CommonTree.html

3.3. Semantic Data

An AST is an intermediate representation. Further processing is usually necessary (Aho, Lam, Sethi, & Ullman, 2006), whether it is for compiling purposes or in the case of this research, static code analysis. It was mentioned and demonstrated in Section 3.2 that an AST provides a fine-grained representation of source code facts, necessary for static code analysis, whilst also removing irrelevant information from parse trees. Next, the AST representation needs to be converted into a format where it is possible to detect patterns of code susceptible to malicious attacks independently of the format of the input data.

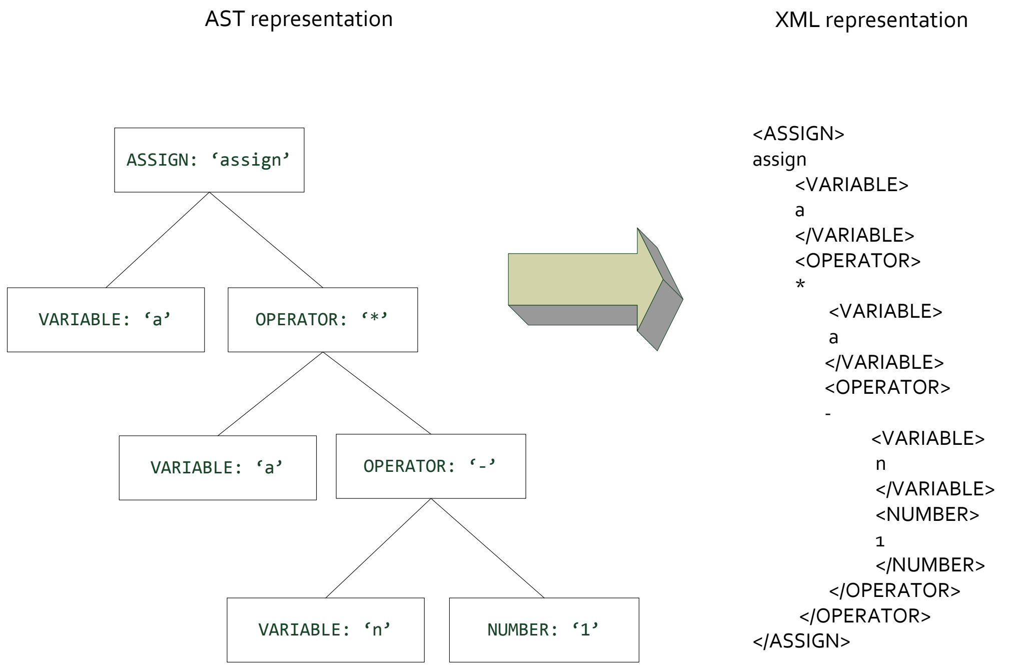
Existing solutions as discussed in Section 2.2 have a common limitation, in that their solution is dependent on the input format. In this thesis, we turn to static code analysis using RDF. Data from potentially vulnerable programs is represented as triples in the Resource Description Framework (RDF) format with the purpose of being uniformly queried by machines in order to solve the problem of independent static code analysis. Programming code is exported to RDF, where reasoning using rules and inference engines can be uniformly implemented independently of the input language.

The methodology uses Extensible Markup Language (XML) (W3C, 2013) as an intermediate representation between an AST and RDF. The algorithms that convert an AST to XML and XML data to the RDF format are given in Section 3.3.2 and Section 3.3.4, respectively. This section also explains the reasoning behind choosing XML as an intermediate representation. First, information from an AST can be represented losslessly in XML using the same tree structure. Second, existing query languages can be employed to transform XML to RDF, whereas tools for direct conversion of an AST to RDF are not available. Sections 3.3.1 and 3.3.3 explain the motivation for choosing XML as an intermediate format between an AST and source code represented in RDF.

3.3.1. Representation of an AST Using XML

Similar to an AST, XML documents are represented using a tree-like structure. The tree starts with a *root* node and branches to the *leaves*. Figure 13 shows an AST represented using XML. Both representations have the node `ASSIGN: 'assign'` as the root node and `VARIABLE: 'a'`, `VARIABLE: 'n'` and `NUMBER: '1'` as leaves, with node branches in between.

The source code facts extraction framework manually traverses the AST (obtained using the ANTLR APIs) recursively. For every node in the AST, a corresponding XML node is created. The XML node name corresponds to the token name of the AST node. The XML node content corresponds to the AST node string content. Details of the algorithm used follow in the next section.



```
<ASSIGN>
assign
    <VARIABLE>
    a
    </VARIABLE>
    <OPERATOR>
    *
        <VARIABLE>
        a
        </VARIABLE>
        <OPERATOR>
        -
            <VARIABLE>
            n
            </VARIABLE>
            <NUMBER>
            1
            </NUMBER>
        </OPERATOR>
    </OPERATOR>
</ASSIGN>
```

Figure 13 Representation of an AST using XML

3.3.2. Tree Traversal Algorithm

We used the following algorithm to output an AST into an XML document. Using a top-down approach, each node of the AST is traversed and its children type and value are printed as `<type>value</type>`. For each child the process is repeated recursively, until the algorithm visits all nodes. The XML indentation is achieved using `print " ";` times the given node's depth level. Please note that the indentation is only for the purpose of improving readability; functionality wise it is of no importance.

```
void function outputXMLTree(tree, indent) {
   if tree not empty {
      for (i=0; i<=indent; i++)
         print " ";
      for (i=0; i<number of tree children; i++){
         print "<" + child i type + ">";
         print child value;
         outputXMLTree(child, indent+1);
         print "<" + child i type + ">";
      }
   }
}
```

Listing 9 Tree traversal algorithm

The previous section and this section showed that due to the structural similarity of an AST and an XML document, XML was chosen to first represent the information contained in an AST, because existing query languages can then be employed to convert XML to RDF. The next section describes this process.

3.3.3. XML Query to Obtain RDF

The Extensible Stylesheet Language Transformation (XSLT) provides one of the standard methods of querying XML documents (W3C, 1999). However, extracting RDF data from existing XML documents using XSLT has been acknowledged to be a non-trivial task (Akhtar, Kopecky, Krennwallner, & Polleres, 2008), by the Gleaning Resource Descriptions from Dialects of Languages (GRDDL) working group (W3C, 2007). Due to the flexibility of an RDF representation and its several serialization formats, using XSLT to handle RDF data is greatly complicated (Akhtar, Kopecky, Krennwallner, & Polleres, 2008). XSLT was optimized to handle a fixed hierarchy in the form of a tree-like structure found in XML documents. The structure is conceptually different in an RDF document that does not contain a simple, well-known hierarchy.

XQuery[22] was introduced in 2007 to assist in processing XML, when it also became a W3C recommendation. Studies show that XQuery's usability is greater than XSLT (Graaumans, 2005), introducing fewer concepts and less verboseness. Therefore, while

[22] http://www.w3.org/TR/xquery/

either XML query language could have been used, XQuery was chosen based on its usability.

Some tools are available for converting XML to RDF. However, these tools are not easy to integrate, as most are available as Web services[23] or they need to be accessed via the command line[24]. Moreover, the available tools use only XSLT and as mentioned above, due to usability concerns, XQuery was chosen to extract data from XML.

In light of this, the source code facts extraction framework uses XQuery to extract data from XML and to render it using RDF. The algorithm used for querying the XML data is given in the next section.

3.3.4. XQuery Algorithm

This section describes the process of converting XML data resulted from traversing an AST as described in Section 3.3.1, into RDF data. The reason for using XQuery to extract information from XML was explained in the previous section. Next, XML to RDF mapping steps are developed based on previous research. Based on the resulting mapping conventions, an XQuery algorithm is implemented.

Mapping XML to RDF

In querying and extracting data from XML in order to create an RDF document containing the same information, a top-down approach was used, starting with the mapping of an XML document into an RDF document. The mapping process is based on previous research on a simplified syntax for RDF (Melnik, 1999 and W3C, 2007). This research is suitable for our needs, as the XML data resulted from traversing the source code's AST follows a simple structure, containing only tags and text content and no attributes (e.g. `<tag noattribute=”attributevalue”>text content</tag>`).

When converting XML to RDF, every tag encountered in the XML document is considered a property name (or predicate) in RDF. The structure of the XML and its corresponding RDF representation as proposed in (Melnik, 1999) is shown in Figure 14. The nodes `_:A1`, `_:A2` and `_:A3` are anonymous nodes. Section 2.3.2 introduces the

[23] http://rhizomik.net/html/redefer/#XML2RDF
[24] http://sourceforge.net/projects/xmltordf/

concept of an anonymous node, or a blank node. A blank node is an RDF resource for which an URI or a literal does not exist. The Notation 3 convention is used for representing blank nodes, i.e. Notation 3 uses a special `_:` namespace prefix to define blank nodes.

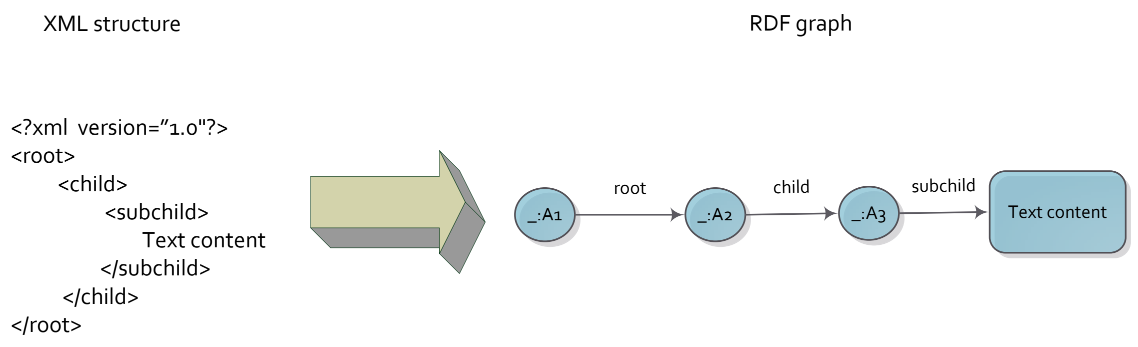


Figure 14 XML data structure and proposed conversion to RDF (Melnik, 1999)

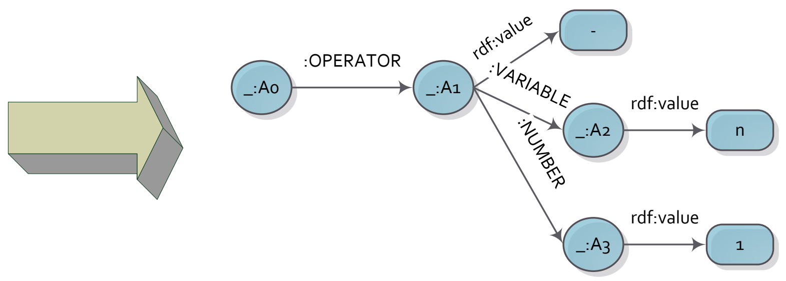
As opposed to the XML structure analysed in previous research, when traversing a source code's AST to obtain XML, the resulting nodes can contain both text content and sub children. For example, in Figure 15, the `<OPERATOR>` node contains both the "`-`" text content and two children, `<VARIABLE>` and `<NUMBER>`. As previously proposed, the tag can be represented by an RDF property. However, text content cannot be considered a predicate, as it has no sub children and thus, no RDF objects to create an RDF statement. Therefore, the text content must be considered an RDF object and thus, an RDF predicate is needed to create an RDF triple. The RDF concepts vocabulary defines the `value` predicate as "Idiomatic property used for structured values." (W3C, 2004). Based on the discussion from Section 2.3.1, the predicate can be reused to create RDF statements.

In light of this, Figure 15 shows the corresponding representation of an XML document resulted from the AST traversal, into an RDF graph. The "`:`" prefix corresponds to the `http://oanaureche.com/flowcontrol/` namespace, discussed in Section 2.3.1. The Notation 3 representation from Figure 15 aims to provide clarification in terms of full predicate names, including namespaces.

XML document from
AST traversal

```
<?xml  version=”1.0"?>
<OPERATOR>
 -
      <VARIABLE>
       n
       </VARIABLE>
       <NUMBER>
        1
       </NUMBER>
</OPERATOR>
```

RDF graph



RDF triples (N3)

```
_:A0        <http://oanaureche.com/flowcontrol/OPERATOR>              _:A1
_:A1        <http://www.w3.org/1999/02/22-rdf-syntax-ns#value>        "-"
_:A1        <http://oanaureche.com/flowcontrol/VARIABLE>              :_A2
_:A2        <http://www.w3.org/1999/02/22-rdf-syntax-ns#value>        "n"
_:A1        <http://oanaureche.com/flowcontrol/NUMBER>                _:A3
_:A3         <http://www.w3.org/1999/02/22-rdf-syntax-ns#value>        "1"
```

Figure 15 Representation of XML data using RDF graph

Algorithm

The previous section details the mapping process between the source code facts XML representation and the corresponding RDF representation. In order to translate source code from an XML format to an RDF format, anonymous nodes and the `rdf:value` predicate were used.

XQuery is used in this research to query an XML document and to build a new RDF document. The reason for choosing the XQuery language was given in the introduction of Section 3.3.4. In order to build new documents, XQuery provides constructors that can create XML structures (W3C, 2010). Because XQuery provides constructors for an XML structure, the output RDF format after conversion will be RDF/XML and thus, XQuery cannot be used to create N3 (Notation 3) documents. Introduced first in Section 2.3.2, RDF/XML is a syntax used to express an RDF graph as an XML document. Therefore, before designing the XQuery algorithm, an RDF/XML representation of the RDF graph is needed. Figure 16 shows the RDF/XML representation of the RDF document given in Notation 3 format. The RDF document in Notation 3 is the result of mapping XML to

RDF, explained in the previous section. To convert from Notation 3 to RDF/XML, the *RDF Validator and Converter*[25] tool was used.

RDF document (N3)

```
_:A0    <http://oanaureche.com/flowcontrol/OPERATOR>          _:A1
_:A1    <http://www.w3.org/1999/02/22-rdf-syntax-ns#value>    "-"
_:A1    <http://oanaureche.com/flowcontrol/VARIABLE>          :_A2
_:A2    <http://www.w3.org/1999/02/22-rdf-syntax-ns#value>    "n"
_:A1    <http://oanaureche.com/flowcontrol/NUMBER>            _:A3
_:A3    <http://www.w3.org/1999/02/22-rdf-syntax-ns#value>    "1"
```

RDF document (RDF/XML)

```
<rdf:RDF
   xmlns:rdf=”http://www.w3.org/1999/02/22-rdf-syntax-ns#”>
<rdf:Description>
   <OPERATOR xmlns="http://oanaureche.com/flowcontrol/"
             rdf:parseType="Resource">
       <rdf:value>-</rdf:value>
       <VARIABLE rdf:parseType="Resource">
           <rdf:value>n</rdf:value>
       </VARIABLE>
       <NUMBER rdf:parseType="Resource">
           <rdf:value>1</rdf:value>
       </NUMBER>
    </OPERATOR>
</rdf:Description>
</rdf:RDF>
```

Figure 16 Notation 3 RDF document and corresponding RDF/XML representation

The algorithm that is required to produce the RDF/XML representation is outlined as follows. A recursive method visits all nodes in the XML document, and depending on whether the XML node currently processed has text content or it is an XML element two actions are taken. If the XML node has text content, then the XQuery algorithm creates an `rdf:value` RDF/XML node. Otherwise, the XQuery algorithm creates an `rdf:parseType= “Resource”` RDF/XML node and the recursive method is called on the children of the currently processed element. The recursive method stops when there are no more XML nodes to process.

The XQuery function that implements the algorithm is given in Listing 10. An XML node has an element and an optional attribute. The query traverses every element from the XML DOM tree, and uses the XQuery `element` and `attributes` constructors to create an

[25] http://www.rdfabout.com/demo/validator/

RDF statement. The element part is created with the `element` XQuery constructor and the attribute part is created with the `attribute` XQuery constructor. `QName` is used to create the name of the element, i.e. for the `<OPERATOR>` element, `OPERATOR` is the name. The `QName` function also provides the possibility to add a namespace to the local name of the XML element processed. In the function from Listing 10, `'http://oanaureche.com/flowcontrol/'` is a namespace that it added to every newly created RDF/XML element. Therefore, if the XML element to convert is `<OPERATOR>` then its corresponding RDF/XML element is `<OPERATOR xmlns="http://oanaureche.com/flowcontrol/">`.

```
declare function local:extract($element as element()) as element() {
   element {QName('http://oanaureche.com/flowcontrol/',
            local-name($element))}
      { attribute { 'rdf:parseType' } { 'Resource' },
      for $child in $element/node()
            return
            if ($child instance of element())
               then local:extract($child)
            else
               if (normalize-space($child) = '')
                   then()
                   else element { 'rdf:value'}
                                 { normalize-space($child) }
      }
}
```

Listing 10 XQuery function to convert XML representation of source code to RDF format

The algorithm performs two steps:

1. Creates an XQuery element in the form of:

   `element {computedname} {content}`

   where `{computedname}` is an XQuery QName and `{content}` is an XQuery attribute element. In an RDF statement, the QName will represent a predicate for the XML element nodes with a blank node as the subject. Specifically, to create a QName a namespace is added to the local name of the element. We used a domain name for the namespace to uniquely identify the source code entity:

```
QName('http://oanaureche.com/flowcontrol/',
local-name($element))
```

2. Creates an XQuery attribute element in the form of:

```
attribute {computedname} {content}
```

where `{computedname}` is `rdf:parseType= "Resource"` and the value of `{content}` depends whether the XML node is an element node or a text node.

(a) If the XML node is an element node then its children are traversed starting from step 1.;

(b) If the XML node is a text node then the text will be represented as the RDF object part of an `rdf:value` predicate. In XQuery a new element is constructed for the `{content}` part of the attribute element:

```
element {'rdf:value'}
        {normalize-space($child)}
```

The algorithm uses blank nodes to create the RDF document. As mentioned in Section 2.3.2 a blank node is an RDF resource for which an URI or a literal does not exist. Blank nodes are created with the attribute `rdf:parseType="Resource"` and a set of properties are attached to these anonymous resources. Notation 3 uses a special `_:` namespace prefix to define blank nodes. The blank node `_:A1` has three properties: an `rdf:value`, a `:VARIABLE` and a `:NUMBER` property. As mentioned in Section 2.3.2 RDF is commonly represented using Notation 3 and RDF/XML. Notation 3 is used for improving its readability.

When converting source code to RDF, the properties of a node (e.g. `VARIABLE` and `NUMBER`) correspond to the lexemes defined by the lexer and the operations defined by the parser of the applied grammar.

3.4. Summary

Chapter 3 describes the methodology and the algorithms for extracting source code facts and representing them using a common format, specifically RDF. Extracting source code facts had two objectives. First, the representation needs to have the fine-granularity for permitting a static code analysis, whilst also removing unnecessary information that would cause performance issues. This was achieved using ASTs and the ANTLR library. Second, the representation must be in a format independent of the source code programming language that will also allow for detection of security vulnerabilities. In order to leverage Semantic Web technologies and reason about security vulnerabilities, an AST was used as an independent, intermediate representation. Therefore, an AST was converted to RDF as the final representation, where reasoning techniques can be applied to find code susceptible to malicious attacks using Semantic Web technologies. The motives for using Semantic Web technologies are supported by existing frameworks that employ these technologies to implement security, as described in Section 2.3.5.

The next chapter will describe the methodology, including algorithms and implementation for finding Web application vulnerabilities by reasoning with Semantic Web technologies on top of the source code in RDF format. This methodology is applied after extracting the source code facts as described throughout Chapter 3.

4. DETECTION OF WEB APPLICATION VULNERABILITIES USING SEMANTIC WEB REASONING

This chapter focuses on the methods used to detect security vulnerabilities in source code represented in RDF, after being translated from original input. The framework and algorithms employed to extract source code facts from programs are described in the previous chapter.

The objectives of the methods presented in this chapter are to detect code susceptible to malicious attacks independently of the input language. The focus is on detecting input validation vulnerabilities and information leaks, as they account for the majority of problems related to Web application attacks, as illustrated in Sections 1.1 and 1.2.

As shown in Section 1.1.1, the last six years have seen a major increase in security vulnerabilities caused by insufficient input validation, with the highest number of vulnerabilities corresponding to the year 2013. Furthermore, the vulnerabilities caused by improper implementation of information-flow control methods account for more than 50% of all Web application vulnerabilities found in the year 2013.

This research focuses on SQL injection, cross-site scripting, HTTP splitting, e-mail injection and information leak attacks, but it is not limited to only these attacks. This chapter will describe the methodology for detecting Web application vulnerabilities in code, which can be applied to a broader list of attacks than the ones mentioned for the proof of concept.

4.1. Detecting Web Application Security Vulnerabilities Using Code Patterns

Section 2.2 mentions that a viable method for securing programming code is the analysis of its information flow. In determining for example, whether a potentially dangerous function call can cause a security problem if exploited by an attacker, in general, it can be answered by analysing its arguments and determining if their value originates from a trusted source. These arguments, however, need to be traced in the source code to find out where their value originates. If their value originates from an untrusted source, and the value was not properly sanitized, it can be concluded that the potentially dangerous function call is a vulnerable piece of code. Tracing the origin of the argument of the function call represents the process of analysing the information flow in a program.

A recently found bug in the OpenSSL[26] implementation of the Transport Layer Security (TLS) protocol, called the Heartbleed bug[27], allows an attacker to read up to 64KB of memory in a single request, which if done a sufficient number of times, will expose the Web application server's main private keys. The vulnerability is caused by not properly checking the value of an argument given as input to a potentially dangerous operation. A static code analysis method of the program's information flow control could have detected the vulnerability. In this case, the potentially dangerous operation used was `OPENSSL_malloc` that allocates memory for the server's response. See code snippet below.

```
/* Allocate memory for the response, size is 1 byte
 * message type, plus 2 bytes payload length, plus
 * payload, plus padding
 */
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
```

The value of one of the arguments, `payload`, is not checked and thus, it can contain any value, causing the function call to allocate the maximum size of the server's private memory and ultimately, to return its contents to the user. Tracing the `payload` variable for its value's origin shows that the `payload` value is created by copying `payload` length bytes from the pointer `pl`. See lines 1 and 2 of the code snippet below. Thus, the value of `payload` is tainted which makes the code vulnerable to a Web application attack.

```
1. n2s(p, payload);
2. unsigned char *p = &s->s3->rrec.data[0], *pl;
```

Using information flow control static code analysis, the `OPENSSL_malloc` function call could have been marked as a potentially dangerous operation and thus, its arguments would have been required to be checked against the possibility of being tainted. Tracing

[26] https://www.openssl.org/
[27] http://heartbleed.com/

the origin of the `payload` argument shows that its value is tainted and therefore, that the program contains a vulnerable piece of code.

In determining what operations are considered as potentially dangerous, a viable approach is to determine the entry points into a Web application and train the code review method with patterns of code vulnerable to Web application attacks (Shah, 2006). For example, vulnerable entry points into a Web application are the HTTP variables. Developers use the HTTP Request variable to read information from a user of the Web application through an HTTP form. Therefore, in a ASP.NET application, the code review needs to be trained that the pattern of code `".*.[Rr]equest.*[^\n]\n"` expressed as a regular expression, constitutes an entry point to a Web application that if not guarded can open up potential vulnerabilities in the application. Once the patterns are expressed and the code scanned for these patterns, their arguments need to be traced to determine if their value is tainted. We have shown using the Heartbleed bug example, how this approach can help to determine if a program contains snippets of vulnerable code.

This thesis' proposed method will use inference rules to express patterns of code vulnerable to malicious attacks and will apply reasoning to determine vulnerable entry points and to trace arguments for their value's origin, independently of the application's underlying programming language. The rest of this chapter will describe the creation of the inference rules and the reasoning algorithm.

4.1.1. Entry Point Identification

In securing a Web application's underlying code, entry points into the Web application must be identified, as they can also serve as entry points for attacks (Microsoft Corporation, 2003). It was shown throughout Section 1.2 that an attacker can potentially use specially crafted data to cause the application to behave in an unexpected way and gain access to sensitive information. These issues can in general be prevented by sanitizing user input. However, when the proper sanitization is not in place, a testing tool should detect the code that is vulnerable to attack. Furthermore, the previous section reiterates that a phase to identify entry points is needed early in the process of testing an application for vulnerabilities. After the identification of entry points, the variables that contain the tainted data coming from entry points need to be traced throughout the application in order to determine if the tainted data is used in a potentially dangerous operation. Once the entry points are identified, using this thesis' proposed methodology, Jena rules will be created from patterns of code that allow untrusted data to enter the application through these entry points. Reasoning with rules over a Web application's RDF translation will infer whether the application is using any information coming from an untrusted source.

This section identifies commonly used entry points into a Web application. According to Microsoft Corporation (2003) a Web application can have the following entry points:

1. Logical application entry points that are available to the client through the Web application user interface, such as:
    - Query strings are part of a uniform resource locator (URL), in the following form: `http://server/program/path/?query_string`, containing data that needs to be passed to a Web application. An attacker can easily manipulate the query string values, as they are displayed in the browser's address bar, due to the use of the HTTP GET protocol.
    - Form field values are sent to the server using the HTTP POST protocol. Form fields can be visible or hidden and the values of both types of fields can be used to for example, inject code, such as in the case of SQL injection attacks.
    - Cookie values can be changed by a client of a Web application using tools, such as Edit Cookies Firefox add-on[28] and used to gain unauthorized access to a website.
2. Physical or platform entry points include ports and sockets. Programming constructs can be used to access and read information from these resources and information read from these entry points is considered untrusted. One Java example is reading data using an outbound TCP connection using the Socket[29] class: `sock = new Socket("host.example.org", 39544);`

Once the entry points are identified, the next step is to find patterns of code that allow entry point data to enter an application. Together with test cases[30] that are made available

[28] https://addons.mozilla.org/en-US/firefox/addon/edit-cookies/
[29] http://docs.oracle.com/javase/7/docs/api/java/net/Socket.html
[30] http://samate.nist.gov/SRD/testsuite.php

by the National Institute of Standards and Technology (NIST), we identified these patterns by manually analyzing the source code in the test cases. The test cases contain examples of various injection issues, including input triggering the vulnerability. Some examples of these patterns of code are given in Listing 11.

```
/* Read data using an outbound tcp connection */
Socket sock = new Socket("host.example.org", 39544);

/* Read data from an HTML form */
String user = request.getParameter("user");

/* Read data from console */
InputStreamReader instrread = null;
instrread = new InputStreamReader(System.in);

/* Get environment variable ADD */
data = System.getenv("ADD");
```

Listing 11 Examples of Java patterns of code for input triggering security vulnerabilities in a Web application

Jena rules are created from these patterns of code. An inference engine will use these rules to reason on top of the code in RDF and to check if data from untrusted sources enters the application under inspection. If the application reads data from an untrusted source, then the data will be traced throughout the application's code to determine if tainted data is used in a potentially dangerous operation. This approach was illustrated using the Heartbleed bug example in the beginning of Section 4.1. The next section describes the methodology for tracing information flow in a program.

4.1.2. Tracing Information Flow

In the context of static code analysis, information-flow control methods trace the flow of data throughout the application's code to ensure no leakage of sensitive data. Several types of information-flow control methods were described in Section 2.2.

In this thesis, the tracing of information flow takes place after patterns of code have indicated that unsafe data coming from untrusted sources has entered the application. The previous section describes how to determine if the code under inspection uses data from outside the application.

Using variable assignments in the code, unsanitized and unsafe data could travel inside the application and ultimately be used in a potentially dangerous operation causing a security breach. A simple example is a vulnerable segment of C# code shown in Listing 12. Through the `Request` object in line 1, unsafe data enters the application. Using the variable assignments in lines 2, 3 and 4, this data travels inside the application, propagating itself as the code runs. Because the tainted data contained in the `pro_id` variable is used in dangerous operations, lines 5 and 6 represent vulnerable code. Line 5 can cause a SQL injection attack and line 6 can cause an XSS attack. These attacks were explained in Section 1.2.1.

```
1. NameValueCollection nvc = Request.QueryString;
2. String[] arr1 = nvc.AllKeys;
3. String[] sta2 = nvc.GetValues(arr1[0]);
4. pro_id = sta2[0];
5. String qry = "select * from items where product_id=" +
   pro_id;
6. response.write(pro_id);
```

Listing 12 Vulnerable code caused by assignment of unsafe data read from an untrusted source

In tracing information flow, patterns of code that represent assignments need to be identified and Jena rules need to be created to express these patterns of code. Using the same method as the previous section, reasoning using the resulting Jena rules will infer whether data is propagated in the code and this reasoning will be applied until the end of the trace is reached. The end of the trace is reached when there are no more assignments. In Listing 12, the end of the trace is reached when the inference engine finds the assignment of tainted data to the `pro_id` variable. Because no new variables use the value of `pro_id`, the tracing of information flow phase ends.

Examples of patterns of code that propagate data inside the application can be classified under two categories: direct assignments and indirect assignments using programming language library objects. An example of a direct assignment is `pro_id = sta2[0]`, while an indirect assignment could be: `instrread = new InputStreamReader(sock.getInputStream())`. Although the second type of assignment is dependent on the programming language used, Section 4.2.2 shows that wildcards can be used inside the Jena rules in order to reason on top of the programming code, independently of the language used.

This phase is the most tedious one as there are numerous patterns of code that cause possible propagation of tainted data. As mentioned in Section 4.1.1, we chose the test cases[31] that are made available by the National Institute of Standards and Technology (NIST) to identify these patterns. This website includes a collection of test cases in several programming languages: Java, PHP, C/C++, JSP as well as several Web applications containing CVEs (Common Vulnerabilities and Exposures), such as chrome-5.0.375.54 and wireshark-1.2.0. It contains examples of CVEs, for example, various injection issues and tainted data mishandling, including input triggering the vulnerability. The current release contains 32 test suites, while 11 test suites reside in the archive. The number of test cases contained in every archive varies from three in the jspwiki-2.5.124 Web application to 61,387 in the Juliet Test Suite for C/C++.

4.1.3. Vulnerability Detection

The previous section describes how to trace tainted data inside a Web application. When the end of the trace is reached, the last variable that contains tainted data is checked, in this phase, to determine if it is used in a potentially dangerous operation. This phase is the vulnerability detection phase. In the example from Listing 12, the end of the information flow trace is reached with the `pro_id` variable, as there are no more assignments that use this variable.

The vulnerability detection phase will match patterns of code that may cause a security breach and that use as arguments variables found in the previous section to contain tainted data. This phase finds that the `pro_id` variable from Listing 12 is used in two operations that may cause different Web application attacks, SQL injection and cross-site scripting. The patterns of code are the building of a SQL query using tainted data:

```
String qry="select * from items where product_id=" + pro_id,
```

and throwing non-validated input back to the browser using `response.write(pro_id)`. Section 1.2.1 explains why these operations may cause

[31] http://samate.nist.gov/SRD/testsuite.php

security breaches. In the former case, because the `pro_id` is passed non-validated to the SQL `SELECT` statement, an attacker can manipulate its value and inject SQL payload. Similarly, an attacker, in the latter case, can inject client-side script into the user's webpage and steal session cookies for user authentication, for example.

The patterns that can cause security breaches if used in conjunction with non-validated data are identified from the NIST website mentioned in the previous sections. Some examples of patterns of code and the potential flaws taken from the website are given in Listing 13.

```
/* POTENTIAL FLAW: absolute path traversal */
IO.writeLine(new BufferedReader (new
                 FileReader(fIn)).readLine());

/* POTENTIAL FLAW: command injection */
Process p = Runtime.getRuntime().exec(osCommand + data);

/* POTENTIAL FLAW: HTTP response splitting, input not
verified before inclusion in the cookie */
response.addCookie(cookieSink);

/* POTENTIAL FLAW: use of hard-coded password */
conn2 = DriverManager.getConnection("data-url","root", pw);
```

Listing 13 Examples of potential flaws from the NIST website

This section describes when it is necessary to identify patterns of code susceptible to Web attacks. Identifying vulnerable patterns alone is not sufficient. These patterns of code need to contain as input non-validated data in order to cause a security breach. Input from previous phases, described in Sections 4.1.1 and 4.1.2, will be used to determine if potentially dangerous operations represent a security breach.

The next section describes how to create rules from the patterns identified throughout Section 4.1 and the methodology used to reason on top of the code in RDF.

4.2. Methodology Overview

Section 4.1 describes how to detect Web application vulnerabilities using code patterns. This section illustrates how to implement this approach using Semantic Web technologies. In Section 4.1, this approach is divided into three main phases: entry point identification, tracing information flow and vulnerability detection. It was explained that Jena rules are created from patterns of code identified in these three phases. Figure 17a illustrates the process of creating rules from patterns of code. The end of the rule creation phase will result in three Jena rule sets corresponding to the patterns identified in the entry points identification, tracing variable and vulnerability detection phases. Section 4.2.2 describes how Jena rule sets are created.

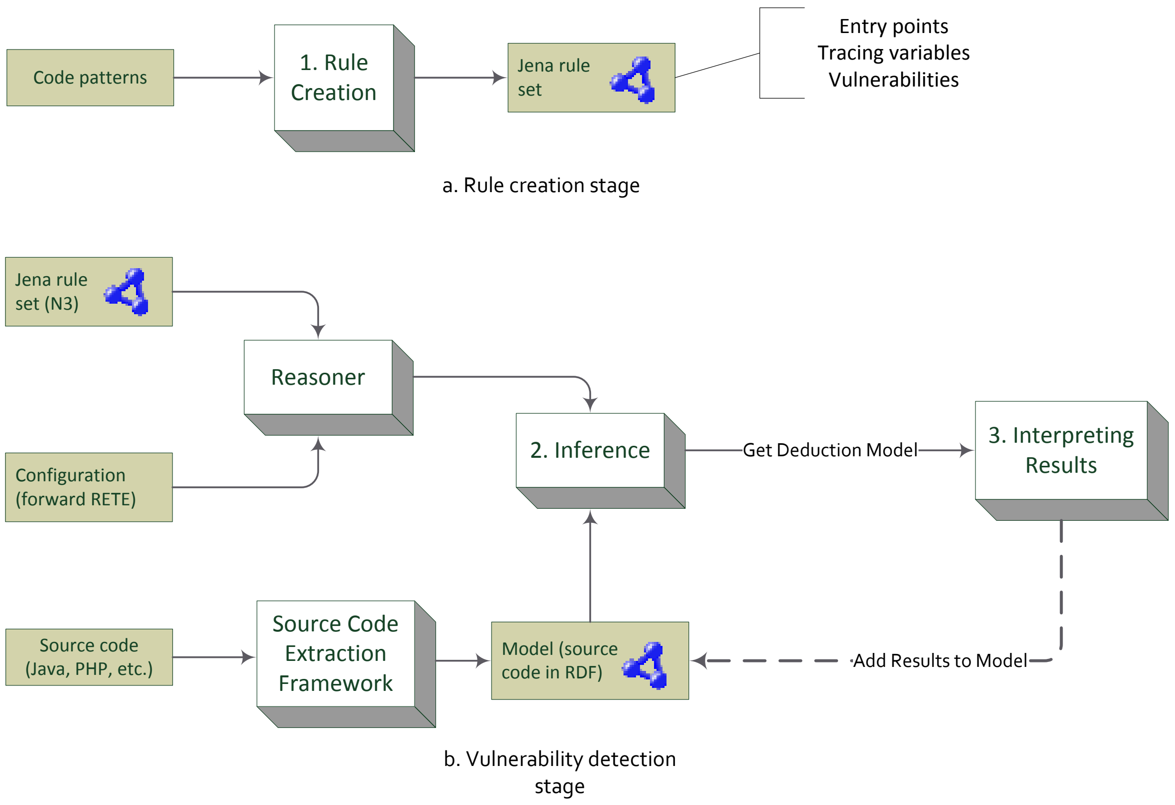


Figure 17 Web application vulnerabilities detection methodology using code patterns and Semantic Web technologies

The three main phases described in Section 4.1 follow the same path, which is illustrated using a graph in Figure 17b. A Semantic Web reasoner is created using the Jena rule set corresponding to the entry point patterns and a chosen configuration for the inference engine. The proposed methodology uses a forward chaining engine with the RETE algorithm (Forgy, 1982). Because the Jena framework offers two types of inference engines, Section 4.2.1 explains why this configuration was chosen.

The RDF data extracted from the source code of the Web application under test is available in the form of a Jena RDF Model (a set of RDF statements), after the initial source code facts extraction phase. The source code facts extraction framework is described in Chapter 3. An inference model is created by associating the reasoner with the source code model. If additional RDF assertions are derived, then they are added to the initial source code model and used as input for the next phase. The same steps are followed in the next phase.

For example, if the entry points identification phase finds that the application uses data from untrusted sources, then the deduction model will contain new RDF statements indicating the variables that contain data from outside the application. These statements will be added to the source code's RDF model and the new deduction model from the information flow trace phase will infer new statements if there are assignments from the entry points variables to new variables. The dotted line in Figure 17b indicates the flow of information from the output of one phase into the input of the next phase.

The vulnerability detection phase is the last phase with its output only being interpreted, and not used as input for another phase. At the end of this phase, the interpretation of the results will conclude if the source code is vulnerable to attacks or not. In the former case, the output of the last phase is the types of attack that can be performed successfully on the input program. The deduction model at the end of every phase and its interpretation is described in Section 4.2.4.

Therefore, three major stages are identified in the methodology of vulnerability detection using Semantic Web technologies. These are illustrated using numbers in Figure 17:

1. Rule creation
2. Inference
3. Interpreting the deduction model

These three major stages are described in more detail in the rest of this chapter. In Section 4.2.1, the motives for using the forward chaining engine with the RETE algorithm are given.

Section 4.2.2 describes how patterns of code are represented using Jena rules. For every type of pattern (e.g. entry point identification, tracing information flow and vulnerability detection) a set of Jena rules are created using the source code facts extraction framework. This represents the rule creation phase. In this case, the *Source code* item from Figure 8 in Section 3.1 is a code pattern: an entry point, an assignment or a vulnerability pattern. Therefore, the source code facts extraction framework is used to create the patterns of code in RDF (represented by Jena rules). Section 4.2.2 describes this process in detail.

After the patterns of code are represented in RDF, an inference engine is employed to find these patterns in the source code. The source code facts extraction framework described in Section 3.1 is also used to represent the source code of the Web application under test in RDF. The inference mechanism is described in detail in Section 4.2.3. Figure 17b illustrates that the patterns represented in RDF (e.g. the Jena rule set) are an input to the reasoner used by the inference mechanism. The output from applying the inference mechanism is a set of statements (e.g. RDF triples) that are further interpreted in a last phase to determine if the RDF patterns were found in the source code. Section 4.2.4 describes how the result from the inference process (e.g. the deduction model) is interpreted.

The proposed methodology does not use ontologies for the inference process. Although the use of ontologies has been considered for the implementation of the methodology, there are published works describing the limitations of ontologies and the need to add rules to overcome these limitations (Eiter, Ianni, Polleres, Schindlauer, & Tompits, 2006; Bavarian & Wohner, 2001). Moreover, the limitations of the proposed methodology described in Chapter 6 could not be overcome by using an ontology in addition to rules.

4.2.1. Forward and Backward Chaining

For reasoning purposes, the Jena framework includes a general rule-based reasoner that can be used to implement an RDFS[32] or an OWL[33] reasoner, but which can be also used for general purposes (Apache Software Foundation, 2014). The Jena framework was first introduced in Section 2.3.3 and the reasons for choosing this framework for processing data were given.

RDFS and OWL are Semantic Web formats, commonly used to process information. However, they have limitations and cause performance issues (Gua, Punga, & Zhang, 2005). Reasoning using user-defined rules has been proven to be more flexible than reasoning using OWL and RDFS (Clark & Parsia, 2014). Therefore, this methodology employs the Jena rule-based reasoner for its general use.

The general use reasoner supports rule-based inference over RDF graphs and provides functionality through two internal engines; a forward chaining engine and a backward chaining engine. The forward chaining engine is based on the standard RETE algorithm (Forgy, 1982). The RETE algorithm will be discussed in Section 6.1, as it has a limitation that affects the performance of the proposed methodology for detecting Web application vulnerabilities. These engine configurations are accessible through the `GenericRuleReasoner`[34] class. To create a `GenericRuleReasoner` object the minimum required is a list of Jena rules, also called a rule set. Section 4.2.2 describes the creation of rules that will be given as a parameter to the generic rule reasoner constructor. This section aims to compare the two different engine configurations and to outline the motives for this methodology's choice for forward chaining.

For the detection of Web application vulnerabilities using reasoning, data consists of source code extracted from a Web application. Given this data, the goal is to find Web application vulnerabilities by searching for patterns of code susceptible to Web application attacks. Often, to conclude that a line of code causes a security breach, the line of code itself is not enough to make a final decision and more information is needed. Section 1.2.1

[32] http://www.w3.org/TR/rdf-schema/
[33] http://www.w3.org/TR/owl-features/
[34] http://jena.apache.org/documentation/javadoc/jena/com/hp/hpl/jena/reasoner/rulesys/GenericRuleReasoner.html

contains an example of a command injection vulnerable code. The code below illustrates this example.

```
int main(char* argc, char** argv) {
        char cmd[CMD_MAX] = "/usr/bin/cat ";
        strcat(cmd, argv[1]);
        system(cmd);
}
```

The line of code `system(cmd);` displays a file's content to the user. Analysing the line of code alone will not result in a definite conclusion, whether it is susceptible to an attack or not. However, by analysing the lines of code above it, it can be concluded that command line input that was not sanitized is used for building the filename for which the contents will be displayed to the user. As mentioned in Section 1.2.1, if an attacker uses the input "`;rm -rf /`" instead, the system call will not execute the cat command due the insufficient number of arguments and if run with root privileges it will continue by recursively deleting all the contents of nearly every writable mounted filesystem on the computer. Eventually, the computer will crash due to missing a crucial file or directory. In this instance, more data needs to be extracted before inferring that the line of code `system(cmd);` is susceptible to a command injection attack.

In (Feigenbaum, McCorduck, & Nii, 1988) forward chaining was described as a reasoning method employed by inference engines that starting with known facts uses inference rules to extract more facts until a goal is reached. The example above shows that in order to reach the goal that the inspected code is susceptible to a command injection attack, new facts were asserted, i.e. command line arguments were not properly sanitized and used in a potentially dangerous operation. As opposed to forward chaining, the backward chaining method (Feigenbaum, McCorduck, & Nii, 1988) does not start with known facts. Backward chaining starts with a goal and tries to find facts that satisfy that goal. Therefore, forward chaining was used as the preferred method for reasoning.

The Jena framework provides two forward chaining engines for its inference support, an earlier non-RETE engine and a RETE engine. The non-RETE implementation can be more efficient in some circumstances (Apache Software Foundation, 2014), but according to the Jena framework documentation, this alternative engine is likely to be removed in a future release. Specifically, there is an issue where the non-RETE engine multi-triggers when it should only trigger once and the issue will not be fixed, as explained in the Jena mailing list[35]. The non-RETE engine functionality is only available because of backward compatibility. Therefore, the RETE engine was chosen for the implementation of the inference mechanism.

4.2.2. Rule Creation

In the rule creation stage, the rules are created automatically from source code patterns given as input, corresponding to the three phases mentioned in Section 4.1: entry points identification, tracing information flow and vulnerability detection. Examples of these code patterns were given in Section 4.1.1, 4.1.2, and 4.1.3, respectively.

During the rule creation stage, we encountered a limitation when working with language grammars. More details of this limitation will be given in Section 6.2. In order to be able to parse Java code, for example, to create an AST, the code must have a proper Java syntax (e.g. package name, class name, method declaration etc.). Therefore, although a rule is created for a single line of code, several other lines of code needed to be added first, in order to have a proper Java file that can be parsed and converted to RDF. This results in extra RDF statements, which need to be removed after the rule creation process. During this stage, most statements that are irrelevant for reasoning (e.g. the extra RDF statements) are removed. More detail about this will be given in Section 6.2.

The result of the rule creation stage is three Jena rule sets corresponding to the three phases described in Section 4.1 and used by the Jena reasoner to match patterns in a Web application's source code. The same process is followed for all three rule sets. A rule set is simply a list of rules. This stage is the most tedious as several different patterns may exist for the same type of attack or entry points. Furthermore, there are numerous patterns of code for entry points, propagation of data and dangerous operations. Examples for these patterns were given in Section 4.1. We chose the test cases[36] that are made available by the National Institute of Standards and Technology (NIST) to identify these patterns. This website includes a collection of test cases in several programming languages. It contains

[35] http://sourceforge.net/p/jena/mailman/message/1070814/
[36] http://samate.nist.gov/SRD/testsuite.php

examples of various injection issues and tainted data mishandling, including input triggering the vulnerability.

It was explained that the rules written throughout the proposed methodology use the forward chaining method. Forward chaining starts with available data and infers new data when the antecedent (*If* clause) is met. Using the forward chaining method, an inference engine will look for statements that match the antecedent and when found, the consequent (*Then* clause) is inferred. The reason for using forward chaining as opposed to backward chaining was given in Section 4.2.1.

The definition of a Jena rule was discussed in Section 2.3.3. The format used for rules throughout the detection of vulnerabilities is given in Listing 14.

```
Rule := [ ruleName : bare-rule ]

bare-rule :=   term, ... term -> hterm    // forward rule

hterm     :=   term

term      :=   (node, node, node)       // triple pattern

node      :=   uri-ref            // e.g. http://foo.com/eg
          or   prefix:localname   // e.g. rdf:type
          or   <uri-ref>          // e.g. <myscheme:myuri>
          or   ?varname           // variable
          or   'a literal'        // a plain string literal
          or   'lex'^^typeURI     // a typed literal
          or   number             // e.g. 42 or 25.5
```

Listing 14 Code pattern rule format

Variables and Wildcards

When creating Jena rules from code patterns two terms are defined: *Variables* and *Wildcards*. *Variables* are any programming code identifiers and they are used to trace information throughout the process of finding Web application vulnerabilities. *Wildcards* are any programming code variables, object names or method names that can contain any value. Wildcards in the generated Jena rules can be substituted with any values. Their numbers are assigned randomly as there is no need for tracing their value.

For example, consider the Java code in Listing 15 that reads input from a socket. The entry point in this case is through the `sock` variable and the entry point pattern is `sock = new`

`Socket("host.example.org", 39544);` According to the proposed methodology, a Jena rule needs to be generated. The `sock` variable is considered to contain tainted data, as it contains information from outside the application, and therefore tracing the information flow is imperative. The tainted data is propagated through the `instrread` variable, using the second assignment in the code.

```
sock = new Socket("host.example.org", 39544);

/* read input from socket */
instrread = new InputStreamReader(sock.getInputStream());
```

Listing 15 Java code that reads input from socket

To generate a Jena rule for the entry point pattern `sock = new Socket("host.example.org", 39544);`, `sock` is considered a *Variable*, whilst `"host.example.org"` and `39544` are considered *Wildcards*. The `sock` object is a *Variable* because it requires tracing. In this instance, it does propagate the data from the `sock.getInputStream()` object method via the second line of code. The two *Wildcards* can take any value.

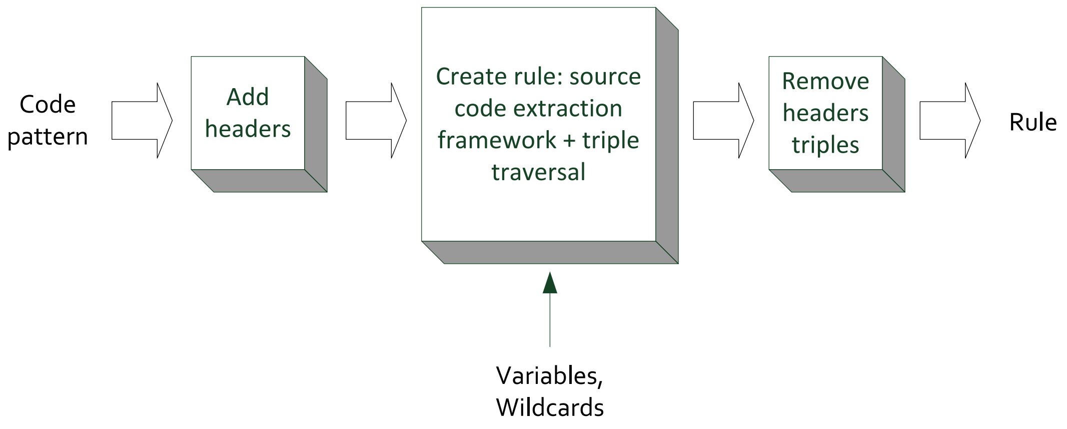


Figure 18 Rule creation flow diagram

Figure 18 shows the flow diagram involved in creating any rule for any type of pattern. This stage, as well as the Web application's source code translation to RDF, requires the source code facts extraction framework. This framework is used to convert the code pattern to RDF after the headers for proper syntax have been added. An input to this step is the array of *Variables* and *Wildcards*, for which the content is decided manually, by looking at the code pattern. The source code facts extraction framework converts the code pattern to an RDF/XML representation, which is then traversed to build the Jena rule, as it requires a specific syntax. This syntax is given at the beginning of this section, in Listing 14.

For the `sock = new Socket("host.example.org", 39544);` code pattern, the source code facts extraction framework has the output given in Listing 16. It was mentioned that RDF/XML is a verbose format, and thus we have previously refrained from listing examples using this format, although for the sake of this example, the output of the translation process was printed. Section 2.3.2 mentions that N3 was proposed as an alternative to the RDF/XML serialization, due to its readability and the ease of writing RDF statements. Note that the bold font is used to highlight the extra statements that were added automatically through the conversion of the class (e.g `class SQLInjection_example1`) and method (e.g. `public method`) Java definitions to RDF, in order to create a proper Java program.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <rdf:Description rdf:about="http://www.oanaureche.com/unit">
      <unit xmlns="http://oanaureche.com/flowcontrol/"
                  rdf:parseType="Resource">
         <ANNOTATION_LIST rdf:parseType="Resource">
            <rdf:value>ANNOTATION_LIST</rdf:value>
         </ANNOTATION_LIST>
         <CLASS rdf:parseType="Resource">
            <rdf:value>class</rdf:value>
            <MODIFIER_LIST rdf:parseType="Resource">
               <rdf:value>MODIFIER_LIST</rdf:value>
            </MODIFIER_LIST>
            <IDENT rdf:parseType="Resource">
               <rdf:value>SQLInjection_example1</rdf:value>
            </IDENT>
            <CLASS_TOP_LEVEL_SCOPE rdf:parseType="Resource">
               <rdf:value>CLASS_TOP_LEVEL_SCOPE</rdf:value>
               <VOID_METHOD_DECL rdf:parseType="Resource">
                  <rdf:value>VOID_METHOD_DECL</rdf:value>
                  <MODIFIER_LIST rdf:parseType="Resource">
                     <rdf:value>MODIFIER_LIST</rdf:value>
                     <PUBLIC rdf:parseType="Resource">
                        <rdf:value>public</rdf:value>
                     </PUBLIC>
                  </MODIFIER_LIST>
                  <IDENT rdf:parseType="Resource">
                     <rdf:value>method</rdf:value>
                  </IDENT>
                  <FORMAL_PARAM_LIST rdf:parseType="Resource">
                     <rdf:value>FORMAL_PARAM_LIST</rdf:value>
                  </FORMAL_PARAM_LIST>
                  <BLOCK_SCOPE rdf:parseType="Resource">
                     <rdf:value>BLOCK_SCOPE</rdf:value>
                     <EXPR rdf:parseType="Resource">
                        <rdf:value>EXPR</rdf:value>
                        <ASSIGN rdf:parseType="Resource">
                           <rdf:value>=</rdf:value>
                           <IDENT rdf:parseType="Resource">
                              <rdf:value>sock</rdf:value>
                           </IDENT>
                           <CLASS_CONSTRUCTOR_CALL rdf:parseType="Resource">
                              <rdf:value>STATIC_ARRAY_CREATOR</rdf:value>
                              <QUALIFIED_TYPE_IDENT rdf:parseType="Resource">
                                 <rdf:value>QUALIFIED_TYPE_IDENT</rdf:value>
                                 <IDENT rdf:parseType="Resource">
                                    <rdf:value>Socket</rdf:value>
                                 </IDENT>
                              </QUALIFIED_TYPE_IDENT>
                              <ARGUMENT_LIST rdf:parseType="Resource">
                                 <rdf:value>ARGUMENT_LIST</rdf:value>
                                 <EXPR rdf:parseType="Resource">
                                    <rdf:value>EXPR</rdf:value>
                                    <STRING_LITERAL rdf:parseType="Resource">
                                      <rdf:value>"host.example.org"</rdf:value>
                                    </STRING_LITERAL>
                                 </EXPR>
                                 <EXPR rdf:parseType="Resource">
                                    <rdf:value>EXPR</rdf:value>
                                    <DECIMAL_LITERAL rdf:parseType="Resource">
                                       <rdf:value>39544</rdf:value>
                                    </DECIMAL_LITERAL>
                                 </EXPR>
                              </ARGUMENT_LIST>
                           </CLASS_CONSTRUCTOR_CALL>
                        </ASSIGN>
                     </EXPR>
                  </BLOCK_SCOPE>
               </VOID_METHOD_DECL>
            </CLASS_TOP_LEVEL_SCOPE>
         </CLASS>
      </unit>
   </rdf:Description>
</rdf:RDF>
```

Listing 16 Code pattern: `sock = new Socket("host.example.org", 39544);` in RDF/XML

The output in RDF/XML is traversed using the NxParser Java library[37], a parser for the N3 format. While traversing the output, the nodes in triples are checked against the contents of the *Variables* or *Wildcards* arrays. The output of this step, concerning the code pattern in the given example, after the extra statements removal is shown in Listing 17. The ?

[37] https://code.google.com/p/nxparser/

symbol in the Jena syntax is used to create anonymous nodes. The numbers (i.e. `A744806`, `A744807`) are automatically generated by the Java parser libraries. There are two wildcards: `?wildcard56` and `?wildcard77` and one variable: `?entry_point`. The Jena rule corresponds to the code pattern: `?entry_point = new Socket(?wildcard56, ?wildcard77);`, where the variables and wildcards are explicitly shown.

The RDF predicates, e.g. `fc:EXPR`, `fc:DECIMAL_LITERAL`, `fc:IDENT` etc., where the namespace `fc=http://oanaureche.com/flowcontrol`, are created on the fly, derived from the description of the language grammar. A discussion on vocabularies and the FC vocabulary was given in Section 2.3.1. The process of representing source code using RDF and the ANTLR language grammars was described in Section 3.3.

```
@prefix fc: http://oanaureche.com/flowcontrol/
@prefix rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
[rule:
      (?A744806   fc:CLASS_CONSTRUCTOR_CALL   ?A744807 ),
      (?A744807   fc:ARGUMENT_LIST   ?A744808 ),
      (?A744808   fc:EXPR   ?A744809 ),
      (?A744810   rdf:value   ?wildcard56 ),
      (?A744809   fc:DECIMAL_LITERAL   ?A744810 ),
      (?A744808   fc:EXPR   ?A744811 ),
      (?A744812   rdf:value   ?wildcard77 ),
      (?A744811   fc:STRING_LITERAL   ?A744812 ),
      (?A744808   rdf:value   "ARGUMENT_LIST" ),
      (?A744807   fc:QUALIFIED_TYPE_IDENT   ?A744813 ),
      (?A744814   rdf:value   "Socket" ),
      (?A744813   fc:IDENT   ?A744814 ),
      (?A744815   rdf:value   ?entry_point ),
      (?A744806   fc:IDENT   ?A744815 ),
      (?A744806   rdf:value   "=" ) ->
      (fc:unit   fc:socket_entry_point   ?entry_point)]
```

Listing 17 Jena rule for the `sock = new Socket("host.example.org", 39544);` code pattern

The inferred statement in a situation where a match is found is: `(fc:unit fc:socket_entry_point "sock")`, because the Jena variable `?entry_point` matched the Java variable `sock` that contains the tainted data. This statement is analyzed in the interpreting the deduction model stage and additional steps are taken. More details about this stage in the process of finding vulnerabilities is given in 4.2.4.

4.2.3. Inference

It was mentioned that the three main phases described in Section 4.1 follow the same path, which is illustrated using a graph in Figure 17. All phases, entry points identification, tracing information flow and vulnerability detection have three stages: rule creation, inference and interpreting the deduction model.

The second stage is the inference stage. This stage tries to match patterns of code using the rules created in the first stage. In case of a found match, the result is the inference of new statements. On the contrary, if no match was found, there are no new inferred statements.

A Semantic Web reasoner is created using the Jena rule set corresponding to the phase that is being processed and a chosen configuration for the inference engine. For this thesis approach, the forward chaining engine with the RETE algorithm is used and Section 4.2.1 explains why this configuration was chosen. Listing 18 shows the Java code for creating the reasoner used in the proposed methodology.

```
Model m = ModelFactory.createDefaultModel();
Resource configuration =  m.createResource();
configuration.addProperty(ReasonerVocabulary.PROPruleMode,
        "forwardRETE");
configuration.addProperty(ReasonerVocabulary.PROPruleSet,
        rules_filename);

// Create an instance of such a reasoner
Reasoner reasoner = GenericRuleReasonerFactory.
                    theInstance().create(configuration);
```

Listing 18 Creating a generic rule reasoner using the Jena framework

Two inputs are required for the inference process: the reasoner and the data. An inference model is created by associating the reasoner with a Jena Model. For the entry points identification phase the data represents the Web application’s source code in RDF. For the next two phases, the data represents the Web application’s source code in RDF plus inferred statements from the previous phase. For example, for the tracing information flow phase, the data represents the Web application’s source code in RDF plus the statements that were inferred in the entry points identification phase. This flow is illustrated using the *“Add Results to Model”* dotted line in Figure 17b.

This section follows the example from the previous section. The code is given below. After running the entry points identification phase and associating the data with the entry points rules, during the inference stage, a new statement is inferred, according to the rule written for the entry point: `(fc:unit fc:socket_entry_point "sock").` The rule is available in the previous section.

```
sock = new Socket("host.example.org", 39544);
```

```
/* read input from socket */
instrread = new InputStreamReader(sock.getInputStream());
```

The name of the predicate `socket_entry_point` is used for interpreting the results in the next stage. For tracing the `sock` variable, this predicate is changed to `trace_var` and a new RDF triple is created and added to the data. The new statement to add to the model is: `(fc:unit fc:trace_var "sock").` The rule generated for the `instrread = new InputStreamReader(sock.getInputStream());` code line is given in Listing 19, which corresponds to the assignment `?new_trace_var = new InputStreamReader(?trace_var.getInputStream());` , where *Variables* and *Wildcards* are used. There are no *Wildcards* used in this instance.

```
@prefix fc: http://oanaureche.com/flowcontrol/
@prefix rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
[assignment_rule:
       (fc:unit fc:trace_var ?trace_var),
       (?A747941   fc:CLASS_CONSTRUCTOR_CALL   ?A747942 ),
       (?A747942   fc:ARGUMENT_LIST   ?A747943 ),
       (?A747943   fc:EXPR   ?A747944 ),
       (?A747944   fc:METHOD_CALL   ?A747945 ),
       (?A747945   fc:DOT   ?A747946 ),
       (?A747947   fc:value   "getInputStream" ),
       (?A747946   fc:IDENT   ?A747947 ),
       (?A747948   fc:value   ?trace_var ),
       (?A747946   fc:IDENT   ?A747948 ),
       (?A747946   fc:value   "." ),
       (?A747942   fc:QUALIFIED_TYPE_IDENT   ?A747949 ),
       (?A747950   rdf:value   "InputStreamReader" ),
       (?A747949   fc:IDENT   ?A747950 ),
       (?A747951   rdf:value   ?new_trace_var ),
       (?A747941   fc:IDENT   ?A747951 ),
       (?A747941   rdf:value   "=" ) ->
       (fc:unit fc:new_trace_var ?new_trace_var)]
```

Listing 19 Jena rule for the `instrread = new InputStreamReader(sock.getInputStream());` code assignment

The new RDF triple that was added in the entry points identification phase (e.g. `(fc:unit fc:trace_var "sock")`) is going to match the triple `(fc:unit fc:trace_var ?trace_var)`, in the rule from Listing 19, and in case of a match using the assignment rule, the new inferred statement will be `(fc:unit fc:new_trace_var "instrread")`. Therefore, the new variable `instrread` needs tracing. In order to trace the new variable, the same inference process is used. As mentioned in Section 4.1.2 this process is repeated until the end of the trace is reached. If there are no more variables to trace, the last variable traced is checked to see if it is used in a potentially dangerous operation. This last phase will apply the same inference mechanism, by associating the source code in RDF with the rules created for the dangerous operations patterns and the new statements inferred in the tracing information flow phase.

4.2.4. Interpreting the Deduction Model

At the end of every phase, (e.g. entry points identification, tracing information flow and vulnerability detection) decisions are made based on the deduction model resulting from applying the inference mechanism. Using the Jena framework, the deduction model is accessed by employing the `InfModel` interface's `getDeductionsModel`[38] method. This deduction model is a graph containing the triples added to the base graph due to rule firings and thus, only inferred statements are included in this graph without the raw data used to derive information.

Depending on the size of the deduction model and the phase that is currently processed, different actions are taken. If the deduction model is empty, then no actions are taken. For example, if the entry points identification phase does not find a match against any entry points rule, then it is concluded that the Web application under test does not use any data from outside the application, and thus, it is a secure application with regards to the proposed methodology. If the deduction model includes statements such as `(fc:unit fc: socket_entry_point "sock")` then the action taken will be to trace these variables during the next phase (i.e. tracing information flow).

[38] http://jena.apache.org/documentation/javadoc/jena/com/hp/hpl/jena/rdf/model/InfModel.html#getDeductionsModel()

The deduction model is also used for printing out messages to the tester. For example, Listing 20 shows a typical console output of the testing method during the tracing information flow phase:

```
Inferred statement: [http://www.oanaureche.com/flowcontrol/unit,
http://www.oanaureche.com/flowcontrol/socket_entry_point, "sock"]
Tracing variable: sock
Inferred statement: [http://www.oanaureche.com/flowcontrol/unit,
http://www.oanaureche.com/flowcontrol/new_trace_var, "instrread"]
Tracing variable: instrread
Inferred statement: [http://www.oanaureche.com/flowcontrol/unit,
http://www.oanaureche.com/flowcontrol/new_trace_var, "buffread"]
Tracing variable: buffread
Inferred statement: [http://www.oanaureche.com/flowcontrol/unit,
http://www.oanaureche.com/flowcontrol/new_trace_var, "data"]
Tracing variable: data
Reached the end of trace with variable: data
```

Listing 20 Console output example for the tracing information flow phase

In the vulnerability detection phase, the deduction model from the tracing information flow phase will be used to determine which variable needs to be checked against usage in a potentially dangerous operation. In the example from Listing 20, the end of the trace was reached with the `data` variable and thus, the next phase will use this variable to match dangerous operations pattern rules against the raw model. If matches are found, the console messages will be similar to the ones from Listing 21.

```
Use of variable in dangerous operations:

Type of vuln: sqlinjection
SQL_statement_name: "sqlstatement"
Variable_used: "data"

End of use in dangerous operations
```

Listing 21 Console output example for the vulnerability detection phase

In this case, vulnerable patterns were found and thus, the proposed methodology will conclude that the code is vulnerable to the type of attack for which the pattern had a match.

The deduction model is thus used as input for the next phase, as well as for printing messages to the console in order to provide input to users of the testing application.

4.3. Sub-graph Generalization

Jena rules can be used to infer additional statements that can match more patterns than the ones identified in the training data using sub-graph generalization, i.e. by training the methodology to not distinguish between two argument lists with different formats, for example. We will next consider the Java code lines in Listing 22 used for tracing information flow. The Java code lines were taken from the NIST website previously mentioned in Section 4.2.2 for generating Jena rules.

```
Code patterns for tracing information flow:
   1. instrread = new InputStreamReader(sock.getInputStream());
   2. fIn = new File(root + data);
```

Listing 22 Code patterns for tracing information flow

In line 1, there is a flow of data from the `sock` variable to the `instrread` variable. During the tracing information flow phase, the `instrread` variable will be the new variable to trace. In line 2, the flow of data is from the `data` variable to the `fIn` variable and in this case, `fIn` becomes the new variable to trace. These lines of code are modelled using the RDF graphs from Figure 19 and Figure 20, respectively. The graphs are using the Notation 3 format for the anonymous node (e.g. the `_:` namespace).

When generating the Jena rules, it was mentioned that the wildcards and variables are assigned manually. In Listing 23 the patterns, including wildcards and variables, used for generating the corresponding Jena rules are listed. We chose the same name for wildcards in order to highlight the similarity between the two code patterns.

```
Jena rules patterns for tracing information flow:
   1. ?new_trace_var = new ?wildcard2(?trace_var.?wildcard1());
   2. ?new_trace_var = new ?wildcard2(?wildcard1 + ?trace_var);
```

Listing 23 Jena rules patterns for tracing information flow

The value of constructor names and method names can be replaced with wildcards because when tracing information flow we need to determine only if there is a flow of data from one variable to another and not the methods employed to achieve this. Figure 21 depicts the RDF graph used to model the Jena rules for the analyzed code patterns.

Note that the graphs are similar, with the exception of the RDF graph part that models the called method's argument list. However, using reasoning, a Jena rule can be used to infer additional statements that can match both patterns using only the Jena rule for matching the `instrread = new InputStreamReader(sock.getInputStream());` code pattern.

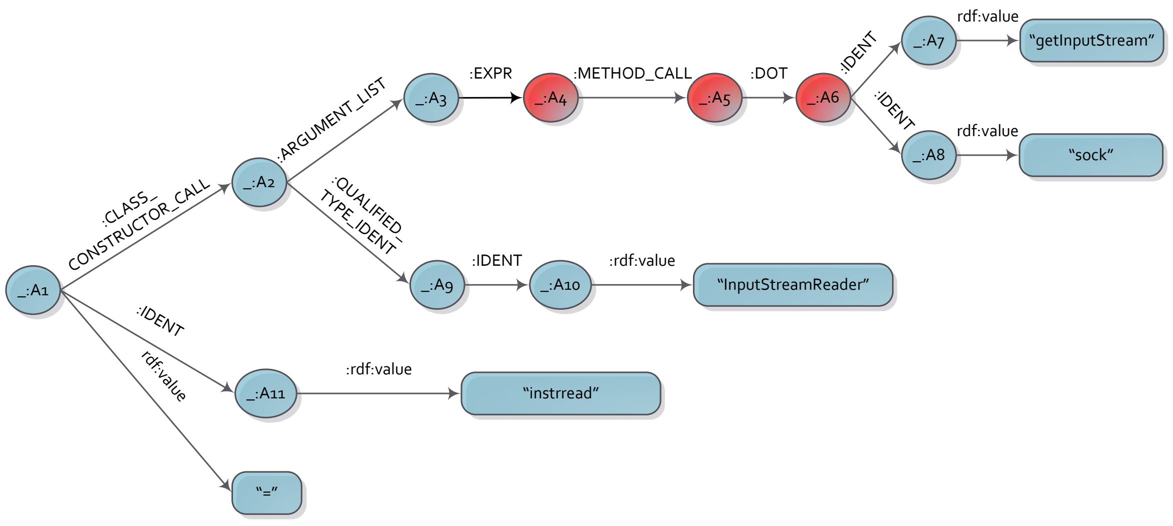


Figure 19 `instrread = new InputStreamReader(sock.getInputStream());` modelled using an RDF graph

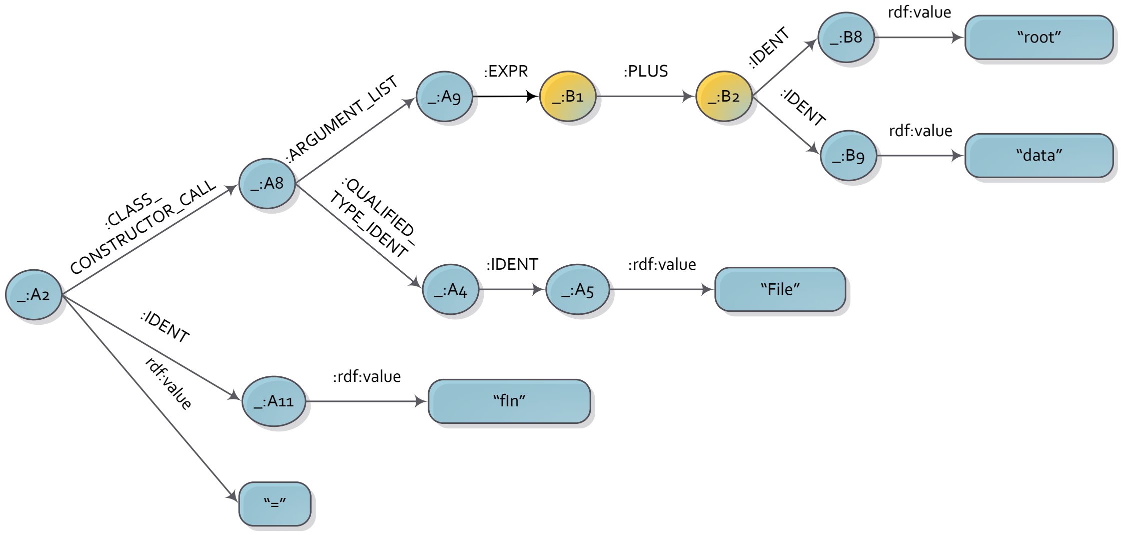


Figure 20 `fIn = new File(root + data);` modelled using an RDF graph

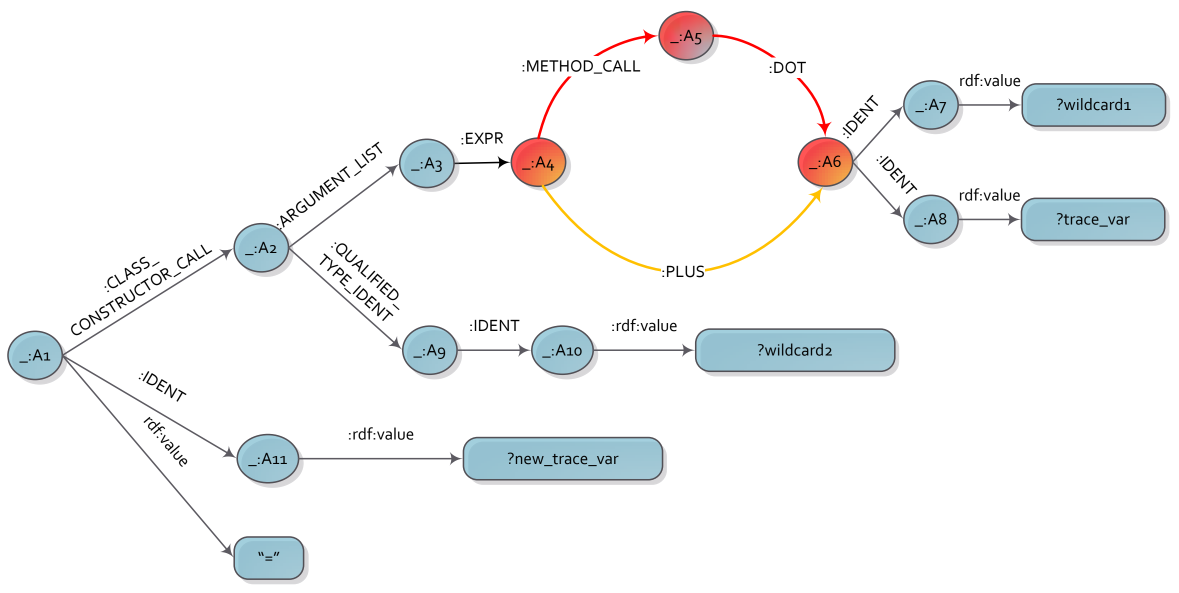


Figure 21 RDF graph modelling the Jena rule pattern for matching `instrread = new InputStreamReader(sock.getInputStream());` and `fIn = new File(root + data);`

We assume that tracing information flow contains only one Jena rule matching the pattern `fIn = new File(root + data);` and that the Web application under test uses the `instrread = new InputStreamReader(sock.getInputStream());` code to assign values from the `sock` variable to the `instrread` variable. When testing the Web application for security vulnerabilities the proposed methodology must match the `instrread = new InputStreamReader(sock.getInputStream());` code as an information flow pattern. However, the training data does not include a Jena rule to match this pattern.

```
[rule:
(?A1  <http://oanaureche.com/flowcontrol/METHOD_CALL> ?A2 ),
(?A2  <http://oanaureche.com/flowcontrol/DOT>   ?A3 ) ->
(?A1   <http://oanaureche.com/flowcontrol/PLUS> ?A3)]
```

Listing 24 Jena rule to enable sub-graph generalization

Next, consider the Jena rule in Listing 24. Jena rules use the `?` character for anonymous nodes. When this Jena rule is associated with the raw data that includes the `instrread = new InputStreamReader (sock.getInputStream());` code it will infer a new statement, as it finds a match for the rule antecedent. The inferred rule (e.g. `?A1 <http://oanaureche.com/ flowcontrol/PLUS> ?A3`) will be added to the raw model, thus allowing for matching the `instrread = new`

`InputStreamReader(sock. getInputStream());` pattern with the Jena rule generated for the `fIn = new File(root + data);` pattern.

This sub-graph generalization capability was employed throughout the methodology in order to enable not straightforward matching. The results of writing Jena rules to enable sub-graph generalization will be discussed in Section 5.2.2.

This chapter described the methodology for detecting Web application vulnerabilities by employing Semantic Web reasoning and Jena rules over the source code represented in RDF. The results of this methodology are described in Chapter 5. The limitation of this methodology is described in Section 6.3. Other limitations, caused by the RETE algorithm and the language grammars are described in Section 6.1 and 6.2. However, they affect only the creation of rules stage.

5. EXPERIMENTAL RESULTS

This chapter summarizes experimental results for the static analysis method described in Chapter 4. The analysis is applied to a set of real-world open source Java and PHP applications. The applications were chosen based on the availability of the source code, publicity of their security issues and the language used for implementation (PHP and Java). Prior to applying the proposed method to real-world applications, we used collections of test cases in the Java and PHP language as training data for our system. This chapter starts with the description of the training data.

Comparing the proposed methodology with existing work in terms of accuracy and efficiency has been considered. This comparison was not a viable solution, due to the related works either not being applied to real-world applications, such as JFlow (Myers, 1999) and Flow Caml (Simonet & Rocquencourt, 2003) or not explicitly stating which vulnerabilities were detected or not detected, such as RESIN (Yip, Wang, Zeldovich, & Kaashoek, 2009). In the first case, the authors focused on how to implement assertions to detect vulnerabilities and compared their solutions based on improved functionality over previous works. In the latter case, the authors applied their solution to existing Web applications, but they did not include specific references to the Web vulnerabilities found (e.g. using links from the CVE website).

Our methodology is applied to existing Web applications and explicitly states the CVE number for every Web vulnerability, whether discovered or not discovered. As opposed to other woks, the experimental results use direct links to the discovered vulnerabilities. The ones that are not discovered are also linked to published vulnerabilities and reasons why they were not detected are individually given.

5.1. Training Data

It was mentioned in Section 4.2.2 that the Jena rules were created from patterns of code. We chose the test cases[39] that are made available by the National Institute of Standards and Technology (NIST) to identify these patterns. This website includes a collection of test

[39] http://samate.nist.gov/SRD/testsuite.php

cases in several programming languages. It contains examples of various injection issues and tainted data mishandling, including input triggering the vulnerability.

These test cases provided patterns for all three types of rules: entry points identification, tracing information flow and vulnerability detection. Examples of these types of patterns were previously given in Section 4.1. The resulting Jena rule sets are used by the inference mechanism to match patterns in a Web application's source code. The reasoning methodology description is described in Section 4.2.

Table 2 gives information about the training data used for the static code analysis. Although the NIST website includes test cases for other types of vulnerabilities, for the training data we chose the most common vulnerabilities caused by improper implementation of control flow methods. The most common vulnerabilities were identified in Section 1.1.1. Table 2 is sorted according to the proportion of the vulnerabilities identified as the most common. The chart illustrating their proportion was given in Figure 4 from Section 1.1.1. For clarity, the chart is duplicated below.

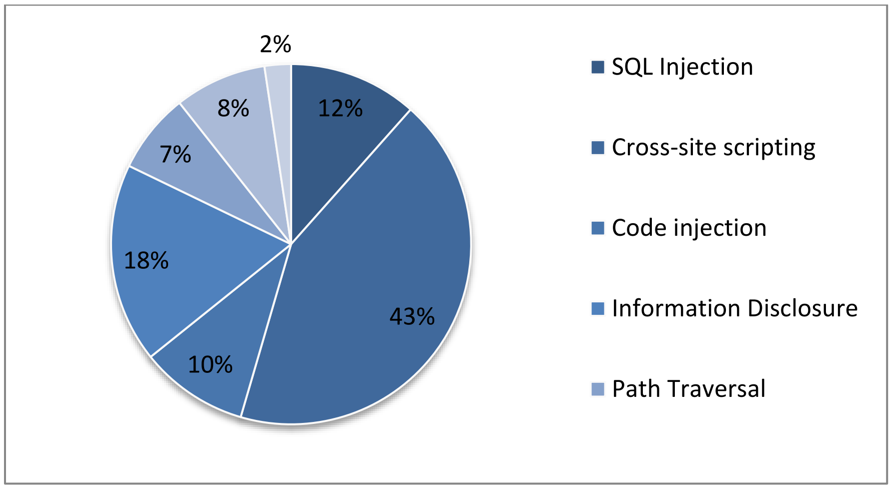


Table 2 Training data for static code analysis

Vulnerability type	Files	Blank lines	Lines of Comments	Lines of Code
SQL Injection (CWE89)	2,024	76,134	79,544	695,620
XSS (CWE80)	506	9,779	13,653	46,298
XSS Error Message (CWE81)	506	9,779	14,500	46,298
XSS Attribute (CWE83)	506	9,779	13,653	46,298
HTTP Response Splitting (CWE113)	2,024	54,564	79,544	305,274
Information Leak Error (CWE209)	508	13,351	17,806	90,253
Absolute Path Traversal (CWE36)	506	9,779	13,653	47,651
Relative Path Traversal (CWE23)	506	9,779	13,653	48,498
Cross Site Request Forgery (CWE352)	140	4,002	6,814	18,792
OS Command Injection (CWE78)	506	10,626	15,347	52,227
Total	7,728	207,337	268,015	1,396,395

The Common Weakness Enumeration (CWE) (community-developed dictionary of software weakness types)[40] classifies software weaknesses using numbers and Table 2 includes the CWE number classification for the Web application vulnerabilities listed. Note that a cross-site scripting (XSS) attack can have several causes and they are classified under different CWE types. A cross-site scripting attack can be caused by an improper neutralization of scripts in attributes in a webpage, corresponding to the CWE-83 software weakness and one possible code pattern is `response.getWriter(). println("<br>bad() - <img src=\"" + data + "\">");` where input is not verified before used in an image tag. Another cause for a cross-site scripting attack can

[40] http://cwe.mitre.org/

be an error message sent to the client containing input that is not sanitized, such as `response.sendError(404, "<br>bad() - Parameter name has value " + data);`.

The CLOC[41] tool was employed to count the number of files, blank lines, comments and lines of code, as it provides the tools necessary to analyze a Web application's source code.

5.1.1. False Positives

During the training process, false positives were observed, however with no impact over the accuracy of the methodology. The correct type of vulnerability was still identified.

```
[sqlinjection:
      (fc:unit fc:trace_var ?trace_var),
      (?A1 :METHOD_CALL ?A2),
      (?A2 :DOT ?A3),
      (?A3 :IDENT ?A4),
      (?A3 :IDENT ?A5),
      (?A4 rdf:value ?sqlStatement_name),
      (?A5 rdf:value "execute"),
      (?A2 :ARGUMENT_LIST ?A6),
      (?A6 :EXPR ?A7),
      (?A7 :PLUS ?A8),
      (?A8 :PLUS ?A9),
      (?A9 :IDENT ?A10),
      (?A10 rdf:value ?trace_var) ->
      (fc:unit fc:sql_injection_statement ?sqlStatement_name)
      (fc:unit fc:sql_injection_with_var ?trace_var)]
```

Listing 25 Jena rule for a SQL injection dangerous operation pattern

We will demonstrate false positives using an example. In the case of a SQL injection with the following dangerous operation code pattern: `bResult = sqlstatement.execute("insert into users (status) values ('updated') where surname = '"+data+"'");` the corresponding generated Jena rule is given in Listing 25.

For clarity, the generated numbers for the anonymous nodes have been reduced to one and two digits size numbers and the RDF graph for this rule is given in Figure 22. Note the paths that are highlighted using the colors red and green.

41 http://cloc.sourceforge.net/

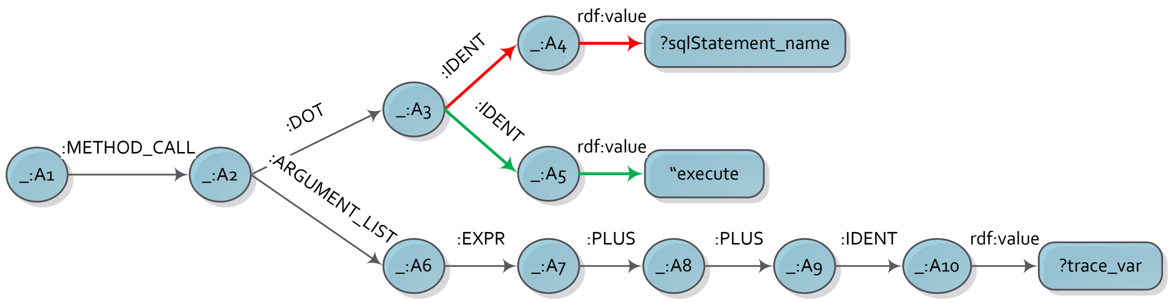


Figure 22 RDF graph for SQL injection dangerous operation code pattern

The inference engine makes no distinction between these paths as they use the same triple structure (`?Anon1 predicate ?Anon2`), where the predicate value is the same. Thus, when inferring new data the Jena inference mechanism will infer two statements for the (`fc:unit fc:sql_injection_statement ?sqlStatement_name`) part of the Jena rule antecedent, given in the listing below.

```
Inferred statement: [http://www.oanaureche.com/flowcontrol/unit,
http://www.oanaureche.com/flowcontrol/sql_injection_statement, "execute"]

Inferred statement: [http://www.oanaureche.com/flowcontrol/unit,
http://www.oanaureche.com/flowcontrol/sql_injection_statement, "sqlstatement"]
```

Listing 26 Application output illustrating false positives

Although the inference engine infers an extra statement that represents a false positive, it does however identify the correct name for the SQL statement variable.

The issue that causes false positives in this case can be overcome by modifications to the language grammar and making the distinction that `sqlstatement` is an object, whilst `execute` is a method. The `:IDENT` predicate could then be replaced by `:OBJECT`, and `:METHOD` respectively. However, the modification of language grammars falls out of the scope of this thesis.

5.2. Static Analysis Results

Experimental results were obtained by applying our proposed methodology to real-world applications with known vulnerabilities. These Web application security vulnerabilities are published for the application's older versions. For example, the Apache Tomcat website publishes its older version's vulnerabilities on their "Reporting security problems" page[42]. It was possible to download previous versions of these applications and run our methodology on their source code. Results were compared against their known security vulnerabilities.

5.2.1. Summary of Discovered Vulnerabilities

The real-world applications used for the experimental phase were: Apache Tomcat[43], OpenEMR[44], phpMyAdmin[45], Jetty[46], Quick & Dirty PHPSource Printer[47], WebCollab[48] and Moodle[49]. These applications offer access to their source code and they are developed in Java and PHP. Not all applications' websites contain the details of their security vulnerabilities. However, it was still possible to obtain their list of security vulnerabilities using an interface to CVE vulnerability data[50]. The data are taken from the NVD (National Vulnerability Database) XML feeds provided by NIST. For example, Moodle's SQL injection security vulnerability list is available here[51].

The proposed methodology was only able to identify types of vulnerabilities and patterns used in the training data for entry points identification, tracing information flow and dangerous operations. The pattern-based limitation is explained in Section 6.3. It was however possible to identify vulnerabilities with similar patterns of code using wildcards and sub-graph generalization. The impact of using wildcards and sub-graph generalization is described in Section 5.2.2. The sub-graph generalization concept is explained in Section 4.3.

Table 3 shows the number and type for each identified vulnerability, as well as the entry points source for the tainted data. The rows specify the entry point source. The

[42] http://tomcat.apache.org/security.html
[43] http://tomcat.apache.org/index.html
[44] http://www.open-emr.org/
[45] http://www.phpmyadmin.net/home_page/index.php
[46] http://www.eclipse.org/jetty/
[47] http://guff.szub.net/quick-and-dirty-phpsource-printer/
[48] http://webcollab.sourceforge.net/
[49] https://moodle.org/
[50] http://www.cvedetails.com/
[51] http://www.cvedetails.com/vulnerability-list/vendor_id-2105/product_id-3590/opsqli-1/Moodle-Moodle.html

vulnerability type is shown in columns. The table's data represents the number of vulnerabilities found for each type of entry point.

Table 3 Classification of vulnerabilities found

	Cross-site scripting	SQL injection	Directory traversal	HTTP response splitting
HTML form fields	17	4	1	1
URL manipulation	0	0	1	0
Parameter tampering	1	1	0	0
Non-Web input	4	0	1	0

Vulnerabilities caused by improper sanitization of HTML form fields are the most common ones. An example of a SQL injection vulnerability found in OpenEMR[52] allowed malicious users to execute arbitrary SQL commands via the `u` parameter. The vulnerable code is given in Listing 27.

```
$user = $_GET['u'];

$authDB = sqlQuery("select password,length(password) as
passlength from users where username = " + $user + "'");
```

Listing 27 OpenEMR vulnerable code snippet

Web applications were also found to use non-Web input that can trigger certain attacks. For example, phpMyAdmin was found to be vulnerable to a cross-site scripting attack through importing a file, the name of which was used later for output in a webpage. By using a crafted name for the file name, it is possible to trigger a cross-site scripting attack. However, only someone logged into phpMyAdmin can trigger this vulnerability. The same application was used to `echo` a file's content without sanitizing it. A cross-site scripting attack via parameter tampering could have been triggered in earlier versions of phpMyAdmin, through a crafted logo URL in the navigation panel. These vulnerabilities are assigned the same CVE identifier[53].

[52] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-2115
[53] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4996

URL manipulation was able to be performed on the Jetty Web server by making HTTP requests as the following: `http://jetty-server:8080/cgi-bin/..\..\..\..\..\..\winnt/notepad.exe`, causing a directory traversal attack[54]. However, URL manipulation was commonly implemented through HTTP parameters. For example, a SQL vulnerability[55] in Moodle was caused by not sanitizing the `id` URL parameter.

Although other types of vulnerabilities found in the applications under test were not detected, due to the training data not containing patterns for these types of vulnerabilities, we found that our proposed methodology could have been successfully applied in other instances. For example, older versions of the Jetty HTTP server allow attackers to cause a denial of service using HTTP requests with a large Content-Length. In order to detect this vulnerability, input read through HTTP requests should be checked if used unsanitized in a denial of service dangerous operation.

Details of discovered/not discovered vulnerabilities

This section provides detailed information on the type of vulnerabilities found, together with the vulnerabilities that were not detected. Reasons why some vulnerabilities were not detected are given next.

The Web application phpMyAdmin, version 3.5.5 contains nine cross-site scripting vulnerabilities. The proposed methodology was able to detect eight. The vulnerability that was not detected is assigned the CVE identifier CVE-2013-5002[56] and it uses a non-sanitized public variable, specifically `public $pageNumber;`, to `echo` a message on the Web page. However, the value for this variable is assigned in a different script and the methodology does not trace information flow across different programming scripts. PhpMyAdmin 3.5.5 includes one SQL injection vulnerability, assigned the CVE number CVE-2013-5003[57], which has been detected by the proposed methodology. There are no directory traversal or HTTP response splitting vulnerabilities reported and the other

[54] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-1178
[55] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-6538
[56] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-5002
[57] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-5003

reported vulnerabilities (e.g. bypass authentication, DoS) were not used in the training data.

OpenEMR 4.1.0 contains one SQL injection and one cross-site scripting vulnerability. These vulnerabilities were detected. No path traversal or HTTP response splitting vulnerabilities were reported. A code execution vulnerability, assigned the CVE identifier CVE-2011-5161[58], was reported, but the methodology could not detect it, due to the code pattern missing from the training data. The vulnerability is caused by an unrestricted file upload. This pattern of entry points identification is not currently available as a Jena rule, although it is possible to generate a Jena rule for matching a file upload entry point pattern.

Three vulnerabilities were reported for the Jetty Web server, version 4.0.0. The directory traversal vulnerability was detected by the proposed methodology. The other two vulnerabilities were not detected, as their types, (e.g. bypass authentication and DoS) were not included in the training data. It is possible to detect the DoS[59] vulnerability as it allows attackers to cause a denial of service using HTTP requests with a large Content-Length. In order to detect this vulnerability, input read through HTTP requests should be checked if used unsanitized in a denial of service dangerous operation.

In Apache Tomcat 6.0.0, eight cross-site scripting vulnerabilities were reported. The proposed methodology was able to identify four of them. Of the others, three cross-site scripting vulnerabilities (CVE-2009-0781[60], CVE-2007-2449[61], CVE-2007-1355[62]) were found in JSP files. Although possible to detect vulnerabilities in other scripting language, to the best of our knowledge there are no ANTLR grammars available for parsing JSP source code. The fourth non-detected vulnerability[63] was due to an unsanitized variable used in a dangerous operation. However, the value is read in another class and current methodology does not trace information flow across different programming scripts. One

[58] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-5161
[59] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-2381
[60] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-0781
[61] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-2449
[62] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1355
[63] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-1232

directory traversal was accurately identified. The remaining two directory traversal vulnerabilities (CVE-2009-2902[64], CVE-2009-2693[65]) used cross-script information.

There is no SQL injection and HTTP response splitting vulnerability reported for Apache Tomcat 6.0.0. Most vulnerabilities are of type DoS, bypass authentication and gain privileges which were not used in the training data.

Quick & Dirty PHPSource Printer 1.1 reported only one directory traversal vulnerability[66], which was correctly identified by the proposed methodology.

WebCollab 3.30 reported one security vulnerability: an HTTP response splitting[67]. This vulnerability allows attackers to conduct response splitting attacks via an item parameter read through an HTML form.

Moodle version 1.9.1 was reported to be vulnerable to four SQL injection attacks, out of which three were identified by the proposed methodology. The fourth vulnerability[68] was not discovered due to the entry points pattern (e.g. file upload) not available as a Jena rule. Most cross-site scripting vulnerabilities reported in Moodle 1.9.1 are due to unsanitized input read through HTML forms and used on output[69]. This information flow is typical for a cross-site scripting vulnerability and thus, the proposed methodology was trained for this type of situation. Although the code follows a typical flow for an XSS vulnerability, the methodology could not detect vulnerabilities[70] caused by unsanitized input read using a `for` loop (e.g. `foreach($_GET as $var => $val)`) as this pattern of entry point was not included in the training data. The proposed methodology identified 56% (9 out of 16) of the XSS vulnerabilities reported for Moodle 1.9.1.

[64] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2902
[65] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2693
[66] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-2169
[67] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2652
[68] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-4305
[69] https://moodle.org/mod/forum/discuss.php?d=108590
[70] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-2362

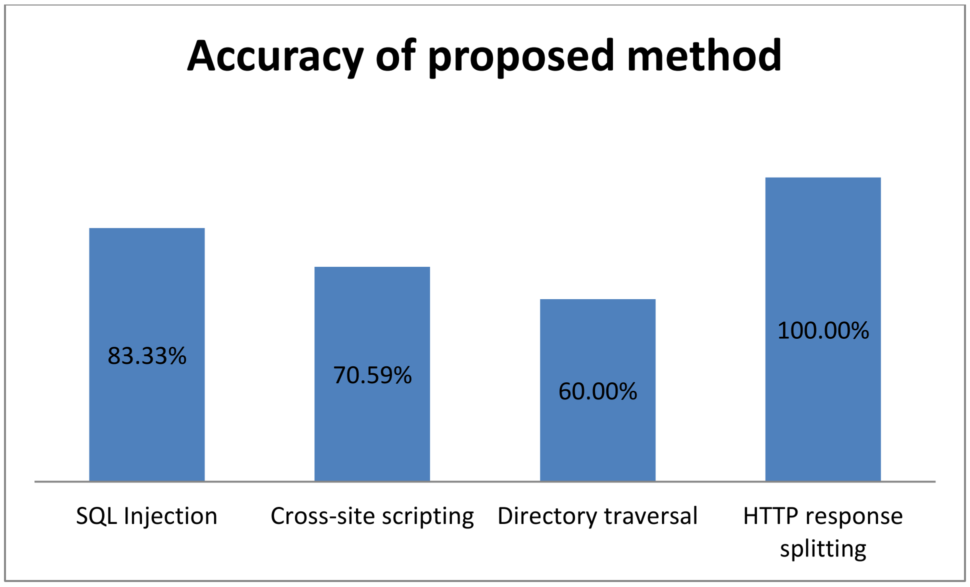


Figure 23 Accuracy of proposed method

Figure 23 illustrates the accuracy of the proposed method. As described throughout this section, most of the vulnerabilities that were not discovered are due to the methodology not previously being trained with appropriate rules for vulnerability patterns and entry points, as well as vulnerabilities found in different programming/scripting languages (e.g. JSP) than the ones used for the proof of concept. The pattern-based limitation is explained in Section 6.3. In Apache Tomcat 6.0.0, another limitation of the proposed methodology was found. Specifically, the current methodology cannot trace information flow across different programming scripts.

5.2.2. Impact of Using Wildcards and Sub-graph Generalization

Section 4.2.2 introduces the concept of wildcards and variables used when generating Jena rules. Section 4.3 describes how employing Jena rules can enable the generalization of RDF sub-graphs. This section illustrates the impact of using wildcards and sub-graph generalization capabilities on the results of the methodology.

The methodology's training data included the dangerous operation code pattern in Listing 28. This operation can enable a cross-site scripting attack if the `data` parameter is not properly sanitized. This code pattern can be modelled using the graph in Figure 24. We intentionally omitted the statements that model the message `"<br>bad() - Parameter name has value"` as they do not offer any value in matching the code pattern.

Moreover, by omitting these statements, other patterns using different messages can be matched.

```
res.getWriter().println("<br>bad() - Parameter name has value" +
                        data);
```

Listing 28 Dangerous operation that might enable an XSS vulnerability

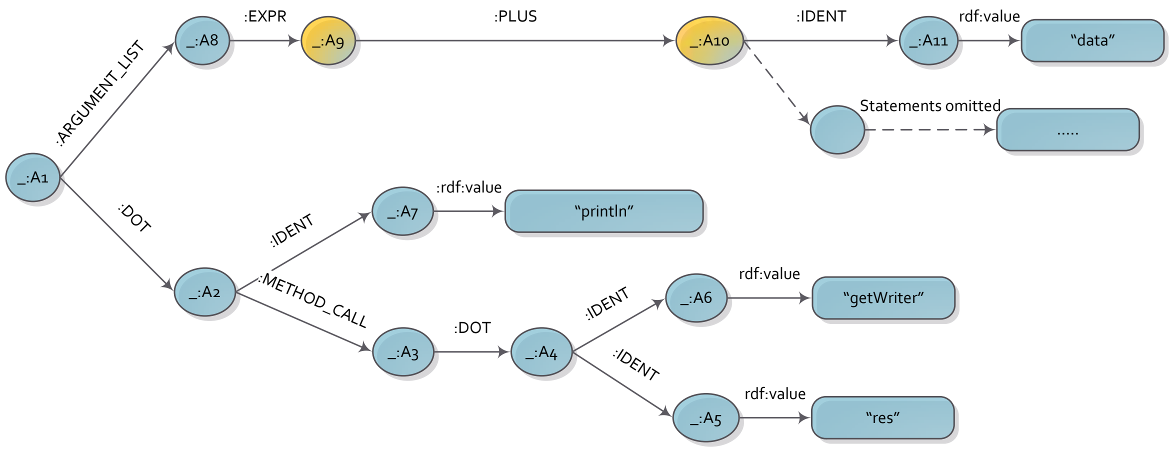


Figure 24 `res.getWriter().println("<br>bad() - Parameter name has value" + data);` modelled using an RDF graph

This code pattern will be matched using the Jena rule modelled by an RDF graph in Figure 25. The `?trace_var` node matches the `data` parameter, used in a potentially dangerous operation that can enable a cross-site scripting attack. For the object name and the print method, we used wildcards.

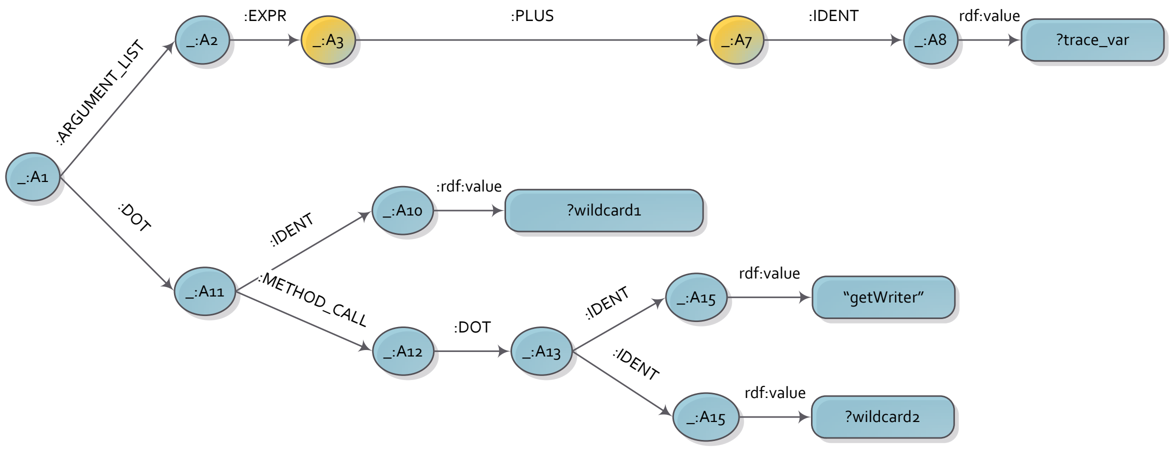


Figure 25 Jena rule modelling the code pattern `res.getWriter().println("<br>bad() - Parameter name has value" + data);`

The Jena rule in Listing 29 was written during the training phase to enable sub-graph generalization. Using this rule, the pattern shown in Figure 26 can be matched along with the pattern used in the training data.

```
[rule:
      (?A1   <http://oanaureche.com/flowcontrol/METHOD_CALL>     ?A2 ),
      (?A2   <http://oanaureche.com/flowcontrol/ARGUMENT_LIST>    ?A3 ),
      (?A3   <http://oanaureche.com/flowcontrol/EXPR>   ?A4 ) ->
      (?A1    <http://oanaureche.com/flowcontrol/PLUS>   ?A4)]
```

Listing 29 Jena rule enables sub-graph generalization

Therefore, by associating the raw model with the Jena rule that generalizes sub-graphs, the operation in Listing 30 was identified as a potentially dangerous operation in a real-world application. The `args` argument is traced and echoed unsanitized to the user of the application. We intentionally omitted the statements that include the `MessageFormat` settings, as these statements are not necessary in matching the dangerous operation pattern.

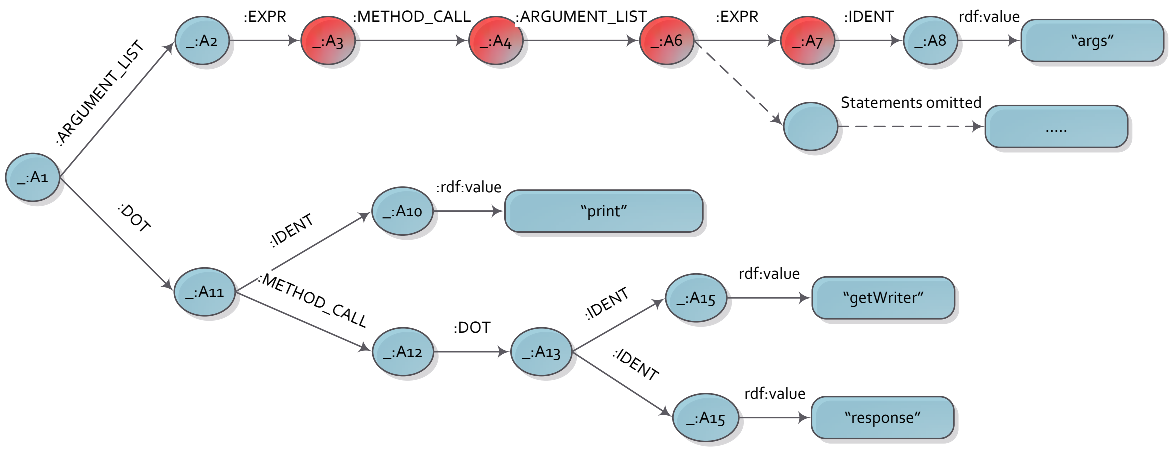


Figure 26
`response.getWriter().print(MessageFormat.format(MANAGER_HOST_ROW_BUTTON_SECTION, args));` modelled using an RDF graph

```
response.getWriter().print(MessageFormat.format(
            MANAGER_HOST_ROW_BUTTON_SECTION, args));
```

Listing 30 Dangerous operation found in real-world application

The code from Listing 30 was identified as a dangerous operation in Apache Tomcat version 6.0.18, CVE-2008-1947[71]. A similar pattern was identified as a cross-site scripting

[71] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-1947

vulnerability in Apache Tomcat, versions 6.0.0-6.0.29, with the identifier CVE-2011-0013[72].

5.2.3. Pattern-based Limitation

This section describes a few concrete examples that we encountered in the experimental phase, during which the methodology was not able to match vulnerable code, including types of vulnerabilities that were part of the training data.

The first example consists of a local path disclosure vulnerability in phpMyAdmin, identified internally using the security identifier PMASA-2012-2[73].

The revision code in Listing 31 shows the code lines that were removed, as well as the code lines that were added in order to mitigate this vulnerability. Removed lines are marked with a "-" sign and added lines are marked with a "+" sign.

```
   /* Read config file.
   */
-require CONFIG_FILE;
+if (is_readable(CONFIG_FILE)) {
+    require CONFIG_FILE;
+}
```

Listing 31 Revision code for path disclosure in phpMyAdmin

The vulnerability appears in the `show_config_errors.php`. An error message can show the full path of the `CONFIG_FILE`, leading to possible further attacks. In order to avoid this, the presence of the configuration file must be validated.

This snippet of vulnerable code does not follow an information flow type pattern, e.g. it does not contain a pattern for an entry point into the Web application and a method that uses tainted data, and thus, the methodology proposed cannot match this type of pattern.

Another limitation of the methodology was encountered during the testing of the Moodle open-source learning platform. The disclosed vulnerability is assigned the identifier CVE-

[72] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0013
[73] http://www.phpmyadmin.net/home_page/security/PMASA-2012-2.php

2012-2363[74]. It is a SQL injection vulnerability in `calendar/event.php` that allows attackers to execute arbitrary SQL commands via crafted form input. However, the form input is read using a method that is imported: `$form = data_submitted();`. Although the data contained in the `$form` is tainted and coming from untrusted sources outside the application, the vulnerable assignment is not matched by the proposed methodology. Currently, the proposed methodology does not trace user defined methods in order to detect vulnerabilities.

5.2.4. Analysis Times

This section shows the time necessary to analyze the real-world Web applications used in our experimental phase. Table 4 shows the duration, in seconds, for each Web application analyzed using the proposed methodology. The machine used for running the methodology uses an Intel Core i5 processor @ 2.67GHz and 4.00GB RAM (2.99GB usable). We used the same tool as the one mentioned in Section 5.1 (e.g. CLOC) for counting the lines of code, blank lines, comments and number of files.

Table 4 Analysis duration of methodology

Web application	Files	Blank lines	Lines of Comments	Lines of Code	Time (seconds)
Apache Tomcat 6.0.0	1,056	52,756	103,687	154,602	4,429
OpenEMR 4.1.0	2,130	71,542	84,075	451,352	10,651
phpMyAdmin 3.5.5	338	11,862	36,325	116,958	976
Jetty 4.2.27	440	14,665	31,022	58,131	779
WebCollab 3.30	252	7,885	7,943	28,278	524
Moodle 2.0.1	4,496	124,544	310,309	623,420	17,654

[74] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-2363

The analysis time data is plotted using a chart in Figure 27. The red, dashed line illustrates the ideal progression for duration in relation to the size of the application. For plotting the processing time, we chose the number of files in a Web application for the x-axis, as this number provides a good representation of the size of the data analyzed. Although blank lines are ignored by the proposed methodology, comments are modeled in the source code RDF graph and thus, the number of lines of code alone does not provide a good representation of the size of the data that is being analyzed. As shown in Figure 27 the processing time is in proportion with the size of the Web application being analyzed, resulting in an almost linear type graph.

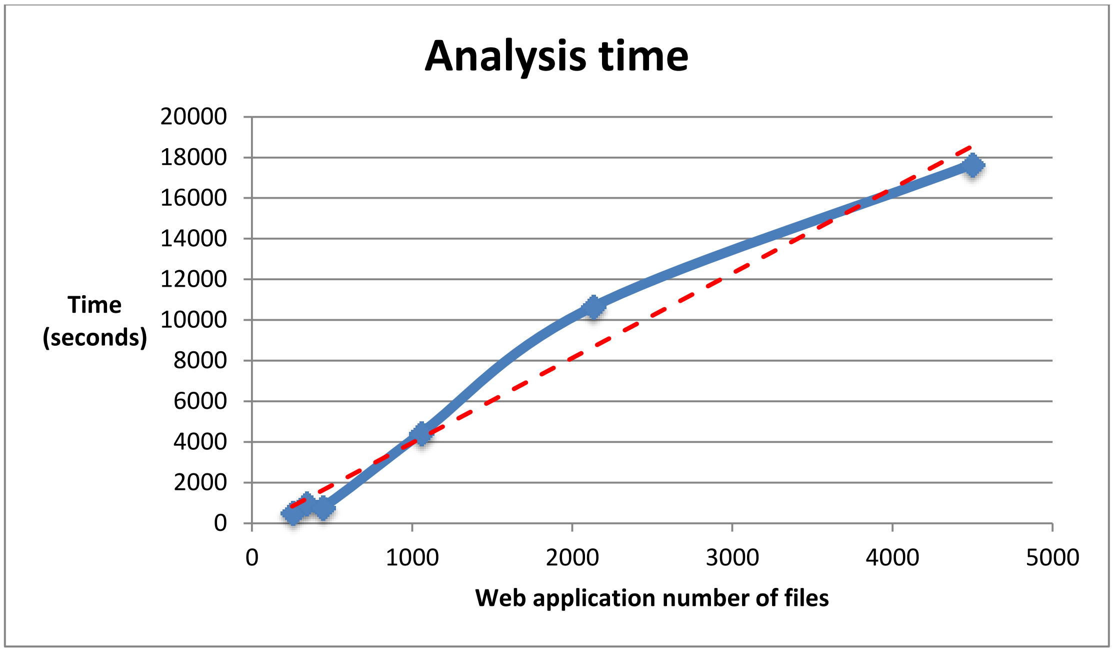


Figure 27 Analysis time in seconds for the number of files in a Web application

Although, the analysis time grows linearly with the size of the Web application, there are exceptions, as described below. If more applications are to be considered for running the proposed methodology, the chart may not result to be linear. Whilst XML is time consuming, only one Web application file at the time is analyzed and thus, the analysis time does not grow exponentially for Web applications that contain significantly more files than other Web applications. Furthermore, the Jena rule sets representing patterns for entry points, information flow assignments and Web vulnerabilities are created prior to the application/detection phase, and thus not influencing the overhead of the work.

We have identified an exception concerning the linearity of the analysis time graph. For a Web application containing 338 files, the processing time was equal to 976 seconds, while for a Web application containing 440 files, there were required 779 seconds to analyze it.

During the experimental phase, we have observed that, for example, to process the `interface/main/calendar/includes/pntables.php` file from the OpenEMR Web application, 70 seconds were required. This file's size is 44KB. However, for the `import.php` file in the same folder, the time required to process it was one second. This file's size is 21KB. However, the `pntables.php` file contains only arrays declarations, with no PHP code implementing functionality. Therefore, the structure of the contents of the source code files can affect the proposed methodology processing time.

6. LIMITATIONS IMPACT

This chapter presents the impact of the limitations of this thesis' approach to static code analysis. The limitations described in Sections 6.1 and 6.2 are caused by dependencies of our methodology, specifically the Jena forward engine using the RETE algorithm and the language grammars used by the ANTLR parser generator to extract source code facts. However, these issues only affect the rule creation process. During the detection of security vulnerabilities, runtime performance is not affected. The third limitation, described in Section 6.3 is a common limitation of methodologies that employ matching algorithms, in that they fail to match patterns that were not used in the training stage. However, adding more training data will result in fewer security vulnerabilities not being discovered. Furthermore, limitations of white-box testing compared to black-box testing are discussed in the last section.

6.1. Unoptimized RETE Network

During the development of the methodology, significant delays (approximately 10 seconds and more to reason using a single rule) and Java heap size memory errors in processing the inference model were observed. Further testing concluded that the order of the statements in the Jena rule affects the performance of the reasoning engine.

These delays are caused by the compiling of rules into a crude RETE network within Jena (Reynolds, 2013). The shape of the network and distance between nodes depends on the order or the statements in the rule and thus, different intermediate joins with different queue lengths are expected, causing different processing times. Research is being conducted to improve the performance of a RETE-based inference engine (Özacar, Öztürk, & Ünalir, 2007).

As an example, consider the rules in Listing 32 and Listing 33. Both Jena rules infer the name of the Java Connection[75] object from the Java pattern: `Connection conn = IO.getDBConnection()`. Reasoning on a sample dataset using the Jena rule in Listing 33 takes approximately 2 seconds, compared to the Jena rule in Listing 32 that causes a delay of approximately 10 seconds on the same dataset. The Jena rule in Listing

[75] http://docs.oracle.com/javase/7/docs/api/java/sql/Connection.html

32 was generated during the rule creation phase described in Section 4.2.2. The order of the RDF statements in the Jena rule from Listing 33 was optimized for better performance.

```
[connection:
        (?A3 :METHOD_CALL ?A4 ),
        (?A2 rdf:value ?variable_conn ),
        (?A1 :EXPR ?A3 ),
        (?A6 rdf:value "getDBConnection" ),
        (?A4 :DOT ?A5 ),
        (?A5 :IDENT ?A6 ),
        (?A5 :IDENT ?A7 ),
        (?A7 rdf:value "IO" ),
        (?A1 :IDENT ?A2 ),
        (?A8 rdf:value "Connection" ) ->
        (fc:unit fc:variable_conn ?variable_conn)]
```

Listing 32 Generated Jena rule

```
[connection:
        (?A1 :IDENT ?A2 ),
        (?A2 rdf:value ?variable_conn ),
        (?A1 :EXPR ?A3 ),
        (?A3 :METHOD_CALL ?A4 ),
        (?A4 :DOT ?A5 ),
        (?A5 :IDENT ?A6 ),
        (?A5 :IDENT ?A7 ),
        (?A7 rdf:value "IO" ),
        (?A6 rdf:value "getDBConnection" ),
        (?A8 rdf:value "Connection" ) ->
        (fc:unit fc:variable_conn ?variable_conn)]
```

Listing 33 Optimized Jena rule

The statements in the optimized Jena rule have the smallest distance between them. For example, the first statement is linked to the second statement using the `?A2` blank node, the first statement is linked to the third statement using the `?A3` node and so on. In the generated Jena rule from Listing 32 the second statement is not linked to any adjacent RDF statement; the `?A2` node from the second statement, appears again only in the eighth statement, causing distance between nodes in the compiled RETE network and thus delays in the processing time.

The optimization process was achieved using a service[76] provided by the World Wide Web Consortium (W3C). This service translates an RDF/XML representation into an N3

[76] http://www.w3.org/RDF/Validator/documentation#intro

representation and outputs the RDF statements in an optimized order suitable for processing by the RETE network.

This limitation affected the time required to generate Jena rules and caused overhead in terms of implementation design. However, using optimized Jena rules increases the Web application vulnerability's runtime performance. It is worth mentioning that un-optimized Jena rules commonly resulted in an "out of memory" error, thus un-optimized Java rules was not a viable option.

6.2. Language Grammars Requirement for Proper Language Syntax

It was briefly mentioned in Section 4.2.2 that during the rule creation stage, we encountered a limitation when working with language grammars. In order to be able to parse Java code, for example to create an AST, the code must have a proper Java syntax (e.g. package name, class name, method declaration etc.). Therefore, although a rule is created for a single line of code, several other lines of code needed to be added first, in order to have a proper Java file that can be parsed and converted to RDF. This results in extra RDF statements, which need to be removed after the rule creation process. During this stage, most statements that are irrelevant for reasoning (e.g. the extra RDF statements) are removed. This limitation is illustrated next using a simple example. We assume that a Jena rule needs to be generated for the assignment `a = 1;`. Using the current Java grammar when employing the ANTLR parser generator for the code `a = 1;` throws the exception in Listing 34.

```
org.xml.sax.SAXParseException: The content of elements must consist of
well-formed character data or markup.
2014-05-12 12:04:37 ERROR RDFDefaultErrorHandler:59 - (line 1 column 1):
Content is not allowed in prolog.
```

Listing 34 Exception when parsing code without proper syntax

```
//proper Java syntax
class Generic_class_name{
        public void generic_method_name() {
                a = 1;
        }
}
```

Listing 35 Code pattern with proper Java syntax

```xml
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <rdf:Description rdf:about="http://www.oanaureche.com/unit">
      <unit xmlns="http://oanaureche.com/flowcontrol/"
              rdf:parseType="Resource">
         <ANNOTATION_LIST rdf:parseType="Resource">
            <rdf:value>ANNOTATION_LIST</rdf:value>
         </ANNOTATION_LIST>
         <CLASS rdf:parseType="Resource">
            <rdf:value>class</rdf:value>
            <MODIFIER_LIST rdf:parseType="Resource">
               <rdf:value>MODIFIER_LIST</rdf:value>
            </MODIFIER_LIST>
            <IDENT rdf:parseType="Resource">
               <rdf:value>Generic_class_name</rdf:value>
            </IDENT>
            <CLASS_TOP_LEVEL_SCOPE rdf:parseType="Resource">
               <rdf:value>CLASS_TOP_LEVEL_SCOPE</rdf:value>
               <VOID_METHOD_DECL rdf:parseType="Resource">
                  <rdf:value>VOID_METHOD_DECL</rdf:value>
                  <MODIFIER_LIST rdf:parseType="Resource">
                     <rdf:value>MODIFIER_LIST</rdf:value>
                     <PUBLIC rdf:parseType="Resource">
                        <rdf:value>public</rdf:value>
                     </PUBLIC>
                  </MODIFIER_LIST>
                  <IDENT rdf:parseType="Resource">
                     <rdf:value>generic_method_name</rdf:value>
                  </IDENT>
                  <FORMAL_PARAM_LIST rdf:parseType="Resource">
                     <rdf:value>FORMAL_PARAM_LIST</rdf:value>
                  </FORMAL_PARAM_LIST>
                  <BLOCK_SCOPE rdf:parseType="Resource">
                     <rdf:value>BLOCK_SCOPE</rdf:value>
                     <EXPR rdf:parseType="Resource">
                        <rdf:value>EXPR</rdf:value>
                        <ASSIGN rdf:parseType="Resource">
                           <rdf:value>=</rdf:value>
                           <IDENT rdf:parseType="Resource">
                              <rdf:value>a</rdf:value>
                           </IDENT>
                           <DECIMAL_LITERAL rdf:parseType="Resource">
                              <rdf:value>1</rdf:value>
                           </DECIMAL_LITERAL>
                        </ASSIGN>
                     </EXPR>
                  </BLOCK_SCOPE>
               </VOID_METHOD_DECL>
            </CLASS_TOP_LEVEL_SCOPE>
         </CLASS>
      </unit>
   </rdf:Description>
</rdf:RDF>
```

Listing 36 RDF/XML representation with statements for proper Java syntax

In order to fix the code for generating a Jena rule, the assignment needs to be enclosed in a proper Java program. A basic syntax shown in Listing 35, lines 1 to 5, will suffice. In this case, however, when extracting source code facts necessary for generating the Jena rule, additional RDF statements are added to the RDF representation. The RDF statements necessary for generating a Jena rule corresponding to the pattern `a = 1;` are highlighted using the bold style in Listing 36. The remaining RDF statements represent the surrounding Java basic syntax. It is necessary for these statements to be removed as real world applications will have different names and access modifiers for classes and methods than the ones we used for generating the Jena rule (e.g. `Generic_class_name` and `public generic_method_name`).

Although this limitation affects the overall performance when creating rules and caused overhead in terms of implementation design and time, it does not affect the runtime performance of the security vulnerability detection methodology, as this limitation's issues are only dealt with in the rule creation phase.

6.3. Pattern-Based Modelling and Static Code Analysis

A common limitation of methods that employ matching techniques based on patterns (Parasoft, 2013 and Costanza, 2004) is the fact that they can only detect patterns that were used as training data or very similar to the training data when wildcards and sub-graph generalization are used. The impact of using wildcards and sub-graph generalization is discussed in Section 5.2.2. However, as the training data grows, the number of patterns that these methods cannot detect becomes less significant.

During the experimental phase, when testing our methodology against real-world applications, patterns that our proposed methodology could not detect were observed. These patterns included Web application vulnerabilities that were not included in the training data. Concrete examples are given in Section 5.2.3. The training data consisted of the most common security vulnerabilities, mentioned in Section 1.1.1. Section 5.1 contains details of the training data used.

Security vulnerabilities found in Web application's security reports, such as Frame injection in Javadoc documentation discovered in Apache Tomcat 6.0.39, Digest Authentication weakness found in Apache Tomcat 6.0.36, and Local file inclusion vulnerability in phpMyAdmin versions prior to 4.0.0, were not detected by our methodology, as the training data did not include patterns for these types of vulnerabilities.

Furthermore, this thesis approach is a white-box testing approach and although the proposed methodology can detect a wide variety of security vulnerabilities, it is not possible to be trained to secure all aspects of a Web application. Section 1.1.3 shows that although white-box testing was able to detect more types of security vulnerabilities than black-box testing, some security vulnerabilities were only detected using a black-box testing approach. Static code analysis is also only possible when the source code of the Web application is available. Therefore, the methods proposed in this thesis should be used in conjunction with other approaches, such as black-box testing.

Furthermore, we mentioned that only the last variable traced was used for testing against use in a potentially dangerous operation. Although it constitutes a limitation of the proposed methodology, it was not found necessary to check intermediary variables in the tracing information flow phase, as these were not used in dangerous operations during the training phase. This situation was also not encountered during the experimental phase.

7. CONCLUSIONS AND FUTURE WORK

This thesis provided a novel methodology for finding security vulnerabilities in Web applications. The methodology addresses a current limitation of previously proposed approaches: their dependence on the programming language of the Web application under test.

To address this limitation, we proposed reasoning over source code represented independently of its programming language. The motives for using a Semantic reasoner were supported by previous research in the area of security. We have shown that reasoning with rules and ontologies in order to enforce security and integrity of information has been previously proposed in academia. Although the proposed frameworks do not apply to source code analysis, they provided the proof of concept that security can be enforced with the application of Semantic Web technologies.

The representation chosen was the format the Semantic Web is built upon, specifically RDF. This format allows the methodology to employ Semantic Web technologies, and therefore, its inference capabilities. Although previous research proposed the conversion of source code to RDF, their result was a coarse-grained representation of source code facts, and static code analysis could have not been implemented. We thus, provided the following contributions:

- An algorithm that converts source code into a common format that allows static code analysis independently of the input programming language
- A method that represents source code facts with the finest granularity, suitable for implementing static code analysis
- A novel approach that employs a Semantic reasoner to detect security vulnerabilities in Web applications

The implementation of the source code extraction framework employs the ANTLR parser generator and language grammars to represent the source code using Abstract Syntax Trees. The ASTs are traversed into an XML document that is converted to RDF using XQuery. The language grammars caused limitations during the Jena rule creation stage, as it required the code patterns to be enclosed in a program with proper syntax. Although this limitation caused delays in the implementation duration, it does not affect the performance of the vulnerability detection methodology.

Jena rules were generated from three types of patterns of code, corresponding to the phases of the methodology: entry points identification, tracing information flow and vulnerability detection. These rules are associated with the source code model by the Semantic reasoner during testing of the Web application for security vulnerabilities. Additional data is inferred if the patterns converted into Jena rules match code in the Web application turned into RDF. The pattern-based approach had a common limitation identifiable in static code analysis methods that use patterns to match data. Specifically, the methodology cannot match patterns that were not included in the training data. Although a known limitation of a pattern-based approach, using sub-graph generalization and wildcards we were able to match not only precise patterns, but also similar patterns, with no discovered false positives.

False positives were caused when additional statements were inferred. These statements did not add any value to the results of the methodology. However, they did not affect the accuracy of the methodology. We were still able to identify the correct type of vulnerability.

The patterns for generating Jena rules were selected from the test cases made available by the National Institute of Standards and Technology (NIST). The patterns were selected for the most common vulnerabilities caused by improper implementation of control flow methods. These types of vulnerabilities were identified in the introduction chapter.

In order to test the proposed methodology against real-world applications, we chose Web applications with known vulnerabilities. Although the methodology could not identify all the published security vulnerabilities, due to the pattern-based approach limitation and patterns that do not follow an information flow model (e.g. source-data flow-sink), we were still able to identify a substantial amount of Web application vulnerabilities. Detection of vulnerabilities was also limited due to variables values assigned through method calls, as this feature was not implemented.

Nevertheless, similar patterns not included in the training data were matched due to the implementation of reasoning using sub-graph generalization and wildcards. Furthermore, the objective of this research, e.g. a language independent method that detects security vulnerabilities in Web applications using a pattern-based approach, was met.

7.1. Future work

As a pattern-based approach, the methodology will be limited to detecting only the patterns for which the Jena rules were generated. Therefore, future work should include more training data that contains a larger variety of security vulnerabilities patterns and types. However, this may result in CPU overhead, as the methodology will need to find a match against a potentially large amount of data. Therefore, future work should also include ways to minimize the CPU overhead or to find a compromise between the size of the Jena rule set and the CPU processing power.

Larger coverage of code can be achieved through analysis of method calls. For example, if the value of variable `file` is assigned using a method call, such as `file = read_contents()`, the methodology should analyze the contents of the `read_contents()` method. In an entry points identification phase, this method could return tainted data and thus, the `file` variable would need to be traced inside the application. However, this feature can cause CPU overhead as well. Assuming the `read_contents()` method is not defined in the current script file which includes dozens of imports. The static code analysis method would need to analyze the contents of every imported file for declaration of the `read_contents()` method. In this case, in order to save the CPU processing power, the methodology may choose not to trace this method.

A useful feature of the proposed methodology is the capability to detect vulnerabilities across a system written in different programming languages. Although the methodology could be currently applied to such system, considering that the sub-systems interoperate and that the current methodology does not implement cross-script analysis, the proposed methodology is limited for a multi-language system. Future work should include a feature to enable the possibility of cross-script and cross-subsystem analysis.

BIBLIOGRAPHY

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc.

Akhtar, W., Kopecky, J., Krennwallner, T., & Polleres, A. (2008). XSPARQL: Traveling Between the XML and RDF Worlds - and Avoiding the XSLT Pilgrimage. *Proceedings of the 5th European Semantic Web Conference on The Semantic Web: Research and Applications* (pp. 432-447). Tenerife, Canary Islands, Spain: Springer-Verlag.

Apache Software Foundation. (2014). *Reasoners and rule engines: Jena inference support.* Retrieved from Apache Jena: http://jena.apache.org/documentation/inference/

Appel, A. W., & Ginsburg, M. (1998). Modern Compiler Implementation in C. Cambridge University Press.

Arenas, M., & Perez, J. (2011). Querying semantic web data with SPARQL. *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (pp. 305-316). Athens, Greece: ACM.

Ashri, R., Payne, T., Marvin, D., Surridge, M., & Taylor, S. (2004). Towards a Semantic Web Security Infrastructure. *In Proceedings of Semantic Web Services Symposium.* Standford.

Baca, D., Carlsson, B., Petersen, K., & Lundberg, L. (2013). Improving software security with static automated code analysis in an industry setting. *Software: Practice and Experience, 43*(3), 259–279.

Barthe, G., Pichardie, D., & Rezk, T. (2007). A Certified Lightweight Non-interference Java Bytecode Verifier. In *Programming Languages and Systems* (pp. 125-140). Springer Berlin Heidelberg.

Bavarian, W. W. & Wohner, W. (2001). A Modest Proposal: Reasoning Beyond the Limits of Ontologies. In *Proceedings of IJCAI-01 Workshop on Ontologies and Information Sharing* (pp. 4-5).

Beizer, B. (1990). *Software Testing Techniques.* Boston: International Thompson Computer Press.

Beizer, B. (1995). *Black Box Testing.* New York: John Wiley & Sons, Inc.

Benantar, M. (2010). *Access Control Systems: Security, Identity Management and Trust Models.* Springer.

Bloomberg. (2008). *Bloomberg L.P. Network Security Breaches Plague NASA.* Retrieved from Bloomberg - Business & Financial News: http://www.businessweek.com/stories/2008-11-19/network-security-breaches-plague-nasa

Brumley, D., & Song, D. (2006). Towards attack-agnostic defenses. *Proceedings of the 1st USENIX Workshop on Hot Topics in Security* (pp. 57-62). USENIX Association.

Business Spectator Pty Ltd. (2011). *Citigroup cyber security breached.* Retrieved from http://www.businessspectator.com.au/bs.nsf/Article/UPDATE-3-Citi-says-hackers-access-bank-card-data-HNAQP

Campbell, S. (2010, February 27). *How Does Facebook Work? The Nuts and Bolts.* Retrieved from http://www.makeuseof.com/tag/facebook-work-nuts-bolts-technology-explained/

Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., & Wilkinson, K. (2004). Jena: implementing the semantic web recommendations. *Proceedings of the 13th international World Wide Web conference* (pp. 74-83). New York.

Cenzic. (2013). *Application Vulnerability Trends Report.* Retrieved from http://info.cenzic.com/rs/cenzic/images/Cenzic-Application-Vulnerability-Trends-Report-2013.pdf

Clark & Parsia. (2014). *OWL & Rule Reasoning.* Retrieved from Stardog Docs: http://docs.stardog.com/owl2/

Costanza, P. (2004, March 13). *Dynamic vs. Static Typing - A Pattern-Based Analysis.* Retrieved from http://www.p-cos.net/documents/dynatype.pdf

Eiter, T., Ianni, G., Polleres, A., Schindlauer, R. & Tompits, H. (2006). Reasoning with rules and ontologies. *In Reasoning Web 2006. Springer* (pp. 93-127).

Feigenbaum, E. A., McCorduck, P., & Nii, H. P. (1988). *The Rise of the Expert Company.* Macmillan.

Feigenbaum, L., Herman, I., Hongsermeier, T., Neumann, E., & Stephens, S. (2007). *The Semantic Web in Action.* Scientific American, vol. 297 (pp. 90-97).

Finifter, M., & Wagner, D. (2011). Exploring the Relationship Between Web Application Development Tools and Security. *In USENIX Conference on Web Application Development (WebApps). USENIX Association*.

Forgy, C. (1982). Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. In *Artificial Intelligence* (pp. 17–37).

Ganapathy, G., & Sagayaraj, S. (2011). To Generate the Ontology from Java Source Code. *International Journal of Advanced Computer Science and Applications (IJACSA), Vol. 2* (pp. 111-116).

Gandon, F., & Sadeh, N. (2004). Semantic web technologies to reconcile privacy and context awareness. *Web Semantics: Science, Services and Agents on the World Wide Web, 1*(3), 241-260.

Graaumans, J. P. (2005). *Usability of XML Query Languages.* Utrecht University.

Graf, J., Hecker, M., & Mohr, M. (2013). Using JOANA for Information Flow Control in Java Programs. *Proceedings of the 6th Working Conference on Programming Languages (ATPS 2013).*

Gruber, T. (2008). Collective knowledge systems: Where the Social Web meets the Semantic Web. *6*(1).

Gua, T., Punga, H. K., & Zhang, D. Q. (2005). A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications, Volume 28, Issue 1,* , 1-18.

Guardian News and Media Limited. (2011). *Biggest series of cyber-attacks in history uncovered.* Retrieved from http://www.theguardian.com/technology/2011/aug/03/biggest-series-cyber-attacks-uncovered

Hammer, C. (2010). Experiences with PDG-Based IFC. *ESSoS'10 Proceedings of the Second international conference on Engineering Secure Software and Systems*, (pp. 44-60).

Hammer, C., Krinke, J., & Nodes, F. (2006). Intransitive Noninterference in Dependence Graphs. *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation* (pp. 119-128). IEEE Computer Society.

Hoff, T. (2008, November 22). *Google Architecture.* Retrieved from http://highscalability.com/google-architecture

Hoff, T. (2008, March 12). *YouTube Architecture.* Retrieved from http://highscalability.com/youtube-architecture

Hurst, A. (2004). *Analysis of Perl's Taint Mode.* Retrieved from http://hurstdog.org/papers/hurst04taint.pdf

Intel. (2012). Improve C++ Code Quality with Static Analysis. Retrieved February 25, 2013, from http://software.intel.com/sites/products/evaluation-guides/docs/studioxe-evalguide-SSA-with_C++_020812.pdf

Kagal, L., Finin, T., & Joshi, A. (2003). A policy language for a pervasive computing environment. *In Collection of IEEE 4th International Workshop on Policies for Distributed Systems and Networks.*

Kaoudi, Z., & Manolescu, I. (2013). *Triples in the clouds.* Retrieved from http://web.imis.athena-innovation.gr/~zoi/rdfcloud_icde13.pdf

Ke, W., Muthuprasanna, M., & Kothari, S. (2006). Preventing SQL injection attacks in stored procedures. *Software Engineering Conference, 2006. Australian*, 18-21.

Keivanloo, I., Forbes, C., Rilling, J., & Charland, P. (2011). Towards sharing source code facts using linked data. *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation* (pp. 25-28). Waikiki, Honolulu: ACM.

Kuruvila, S. (2011, December 13). *PHP Parser - An antlr grammar for parsing php source files.* Retrieved from https://code.google.com/p/phpparser/

Li, P. (2005). Practical information-flow control in web-based information systems, In Proceedings of 18th IEEE Computer Security Foundations Workshop. IEEE Computer (pp. 2-15). Society Press.

Lindorfer, F. (2010). *Semantic Web Frameworks.* Basel: Basel University.

Liu, S., & Cheng, B. (2009). Cyberattacks: Why, What, Who, and How. *IT Pro*, pp. 14-18.

Louridas, P. (2006, July). Static Code Analysis. *IEEE Software, 23*(4).

Masri, W., Podgurski, A., & Leon, D. (2004). Detecting and Debugging Insecure Information Flows. *In ISSRE'04: the 15th International Symposium on Software Reliability Engineering*, (pp. 198-209).

Melnik, S. (1999, November 19). *Simplified Syntax for RDF.* Retrieved from http://infolab.stanford.edu/~melnik/rdf/syntax.html

Microsoft Corporation. (2003, September 2). *Improving Web Application Security: Threats and Countermeasures.* Microsoft Press.

Morisset, C., & Oliveira, A. S. (2007). Automated Detection of Information Leakage in Access Control. *Preliminary Proceedings of the 2nd International Workshop on Security and Rewriting Techniques (SecReT'07).* Paris.

Microsoft Developer Network (MSDN). (2013, September 3). *Forced Parameterization Can Lead to Poor Performance.* Retrieved from http://blogs.msdn.com/b/sql_pfe_blog/archive/2013/09/03/forced-parameterization-can-lead-to-poor-performance.aspx

Myers, A. (1999). JFlow: Practical Mostly-Static Information Flow Control. *In Proc. 26th ACM Symp. on Principles of Programming Languages.*, (pp. 228-241).

Noonan, W., & Dubrawsky, I. (2006). *Firewall Fundamentals.* Cisco Press.

Oracle. (2014, January). *Security guide.* Retrieved from http://download.oracle.com/docs/cd/B28359_01/network.111/b28531.pdf

Open Web Application Security Project (OWASP). (2008, January 14). *Preventing SQL Injection in Java.* Retrieved from https://www.owasp.org/index.php/Preventing_SQL_Injection_in_Java

Open Web Application Security Project (OWASP). (2013). *2013 OWASP Top 10 Most Dangerous Web Vulnerabilities.* Retrieved from http://www.port80software.com/support/articles/2013-owasp-top-10

Open Web Application Security Project (OWASP). (2013, March 19). *Static code analysis.* Retrieved from https://www.owasp.org/index.php/Static_Code_Analysis

Open Web Application Security Project (OWASP). (2014). *SQL Injection Prevention Cheat Sheet*. Retrieved May 8, 2013, from https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

Özacar, T., Öztürk, Ö., & Ünalir, M. O. (2007). Optimizing a Rete-based Inference Engine using a Hybrid Heuristic and Pyramid based Indexes on Ontological Data. *Journal of Computers, Vol 2, No 4*, 41-48.

Paar, C., Pelzl, J., & Preneel, B. (2010). *Understanding Cryptography: A Textbook for Students and Practitioners.* Springer.

Parasoft. (2013). *Static Analysis Best Practices.* Retrieved from http://www.parasoft.com/printables/staticanalysis.pdf?path=/products/article.jsp

Parr, T., & Fisher, K. (2011). LL(*): The Foundation of the ANTLR Parser Generator. *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 425-436). San Jose: ACM.

Parundekar, R., Knoblock, C. A., & Ambite, J. L. (2012). Discovering concept coverings in ontologies of linked data sources. *ISWC'12* (pp. 427-443). Boston: Springer-Verlag.

Perl. (2014). *Perl Security.* Retrieved from http://perldoc.perl.org/perlsec.html

Ponemon. (2013, February 26). *The Post Breach Boom.* Retrieved from http://www.ponemon.org/blog/the-post-breach-boom

Rakić, G., & Budimac, Z. (2011). Introducing Enriched Concrete Syntax Trees. *In Proc. of the 14th International Multiconference on Information Society (IS), Collaboration, Software And Services In Information Society (CSS)*, 211-214.

Rakić, G., Budimac, Z., & Savic, M. (2013). Language Independent Framework for Static Code Analysis. *Proceedings of the 6th Balkan Conference in Informatics* (pp. 236-243). Thessaloniki: ACM.

Ranta, A. (2012). *Implementing Programming Languages. An Introduction to Compilers and Interpreters.* London: College Publication.

Rao, J., & Sadeh, N. (2005). A Semantic Web Framework for Interleaving Policy Reasoning and External Service Discovery. *Proceedings of RuleML*, (pp. 56-70).

Reynolds, D. (2013, July 18). Retrieved from Jena-users Mailing List Archives: http://mail-archives.apache.org/mod_mbox/jena-users/201307.mbox/%3C51E7A0DE.5000801@gmail.com%3E

Richardson, M., Agrawal, R., & Domingos, P. (2003). Trust Management for the Semantic Web. *In Proceesings of the Second International Semantic Web Conference*, (pp. 351-368).

Ritchey, R. W., & Ammann, P. (2000). Using model checking to analyze network vulnerabilities. *IEEE Symposium on Security and Privacy*, (pp. 156 -165).

Rossum, G. v. (2006, January 10). *Python Status Update.* Retrieved from http://www.artima.com/weblogs/viewpost.jsp?thread=143947

Roy, I., Porter, D. E., Bond, M. D., Mckinley, K. S., & Witchel, E. (2009). Laminar: Practical Fine-Grained Decentralized Information Flow Control. *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (pp. 63-74). ACM.

Sabelfeld, A., & Myers, A. C. (2003). Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 5-19.

SANS. (2014, May 26). *AppSec - Protecting Your Web Apps: Two Big Mistakes and 12 Practical Tips to Avoid Them.* Retrieved from http://www.sans.org/reading-room/whitepapers/application/appsec-protecting-web-apps-big-mistakes-12-practical-tips-avoid-33038

Scholte, T., Robertson, W., Balzarotti, D., & Kirda, E. (2012). Preventing Input Validation Vulnerabilities in Web Applications through Automated Type Analysis. *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual* (pp. 233-243). Izmir: IEEE.

Shah, S. (2006, February 11). *Detecting Web Application Security Vulnerabilities.* Retrieved from O'Reilly SysAdmin: http://www.onlamp.com/pub/a/sysadmin/2006/11/02/webapp_security_scans.html

Simonet, V., & Rocquencourt, I. (2003). Flow Caml in a Nutshell. Proceedings of the first APPSEM-II workshop, (pp. 152-165).

Stabek, A., Watters, P., & Layton, R. (2010). The Seven Scam Types: Mapping the Terrain of Cybercrime. *Second Cybercrime and Trustworthy Computing Workshop*, (pp. 41-51).

Stallings, W. (2010). *Cryptography and Network Security: Principles and Practice.* Upper Saddle River, NJ, USA: Prentice Hall Press.

Su, Z. (2006). The essence of command injection attacks in web applications. (pp. 372-382). ACM Press.

The Sydney Morning Herald. (2011). *PlayStation hacking scandal: police chief says contact your bank now.* Retrieved from http://www.smh.com.au/digital-life/games/playstation-hacking-scandal-police-chief-says-contact-your-bank-now-20110427-1dvts.html

Thomas, D., Fowler, C., & Hunt, A. (2004). *Programming Ruby: The Pragmatic Programmer's Guide.* Pragmatic Bookshelf.

Thomson-Smith, L. D. (2011). *Anti-Virus Software: Guarding systems from the malware pandemic.* FastBook Publishing.

Ureche, O., Layton, R., & Watters, P. (2012). Towards an Implementation of Information Flow Security using Semantic Web Technologies. *Cybercrime and Trustworthy Computing Workshop.* Ballarat, Australia: IEEE.

World Wide Web Consortium (W3C). (1999, November 16). *XSL Transformations (XSLT) Version 1.0.* Retrieved from http://www.w3.org/TR/xslt

World Wide Web Consortium (W3C). (2004). OWL-S: Semantic Markup for Web Services. Retrieved from http://www.w3.org/Submission/OWL-S/

World Wide Web Consortium (W3C). (2004, February 10). *Resource Description Framework (RDF): Concepts and Abstract Syntax.* Retrieved November 5, 2011, from http://www.w3.org/TR/rdf-concepts/

World Wide Web Consortium (W3C). (2007, March 22). *A strawman Unstriped syntax for RDF in XML.* Retrieved from http://www.w3.org/DesignIssues/Syntax

World Wide Web Consortium (W3C). (2007, September 11). *Gleaning Resource Descriptions from Dialects of Languages (GRDDL).* Retrieved from http://www.w3.org/TR/grddl/

World Wide Web Consortium (W3C). (2010). *XQuery 1.0: An XML Query Language (Second Edition).* Retrieved from http://www.w3.org/TR/xquery/

World Wide Web Consortium (W3C). (2013, October 29). *Extensible Markup Language (XML).* Retrieved from http://www.w3.org/XML/

World Wide Web Consortium (W3C). (2013). *Vocabularies.* Retrieved from http://www.w3.org/standards/semanticweb/ontology

Walker, B. (2010). *Using Static Code Analysis to Find Bugs Before They Become Failures.* Retrieved from http://www.uploads.pnsqc.org/2010/slides/p39_Walker_slides.pdf

WhiteHat Security. (2013). *Web Application Security: Identifying appropriate investments for positive ROI.* Retrieved from https://www.whitehatsec.com/assets/presentations/13PPT/PPTroi0513.pdf

Yip, A., Wang, X., Zeldovich, N., & Kaashoek, M. (2009). Improving application security with data flow assertions. *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (pp. 291-304). New York: ACM.

Zdancewic, S. (2004). Challenges for information-flow security. *In Proc. Programming Language Interference and Dependence (PLID).*

Zdancewic, S. (2005). Practical Information-flow Control in Web-Based Information Systems. *18th IEEE Computer Security Foundations Workshop CSFW05* (pp. 2-15). IEEE.

Zou, Y., & Kontogiannis, K. (2001). Towards A Portable XML-based Source Code Representation. In International Conference on Software Engineering (ICSE) 2001.

APPENDIX

PHP Grammar for ANTLR (Kuruvila, 2011)

```
grammar Php;

options {
    backtrack = true;
    memoize = true;
    k=2;
    output = AST;
    ASTLabelType = CommonTree;
}

tokens{
    SemiColon = ';';
    Comma = ',';
    OpenBrace = '(';
    CloseBrace = ')';
    OpenSquareBrace = '[';
    CloseSquareBrace = ']';
    OpenCurlyBrace = '{';
    CloseCurlyBrace = '}';
    ArrayAssign = '=>';
    LogicalOr = '||';
    LogicalAnd = '&&';
    ClassMember = '::';
    InstanceMember = '->';
    SuppressWarnings = '@';
    QuestionMark = '?';
    Dollar = '$';
    Colon = ':';
    Dot = '.';
    Ampersand = '&';
    Pipe = '|';
    Bang = '!';
    Plus = '+';
    Minus = '-';
    Asterisk = '*';
    Percent = '%';
    Forwardslash = '/';
    Tilde = '~';
    Equals = '=';
    New = 'new';
    Clone = 'clone';
    Echo = 'echo';
    If = 'if';
    Else = 'else';
    ElseIf = 'elseif';
    For = 'for';
    Foreach = 'foreach';
    While = 'while';
    Do = 'do';
    Switch = 'switch';
    Case = 'case';
    Default = 'default';
    Function = 'function';
    Break = 'break';
    Continue = 'continue';
    //Goto = 'goto';
    Return = 'return';
    Global = 'global';
    Static = 'static';
    And = 'and';
    Or = 'or';
    Xor = 'xor';
    Instanceof = 'instanceof';

    Class = 'class';
    Interface = 'interface';
    Extends = 'extends';
    Implements = 'implements';
    Abstract = 'abstract';
    Var = 'var';
    Const = 'const';
    Modifiers;
    ClassDefinition;

    Block;
    Params;
    Apply;
    Member;
    Reference;
    Empty;
    Prefix;
    Postfix;
    IfExpression;
    Label;
    Cast;
    ForInit;
    ForCondition;
    ForUpdate;
    Field;
    Method;
}

@header{
package net.kuruvila.php.parser;
}
@lexer::header{
package net.kuruvila.php.parser;
}
@lexer::members{
    // Handle the first token, which will always be a BodyString.
    public Token nextToken(){
        //The following code was pulled out from super.nextToken()
        if (input.index() == 0) {
            try {
                state.token = null;
                state.channel = Token.DEFAULT_CHANNEL;
                state.tokenStartCharIndex = input.index();
                state.tokenStartCharPositionInLine =
input.getCharPositionInLine();
                state.tokenStartLine = input.getLine();
                state.text = null;
                mFirstBodyString();
                state.type = BodyString;
                emit();
                return state.token;
            } catch (NoViableAltException nva) {
                reportError(nva);
                recover(nva); // throw out current char and try
again
            } catch (RecognitionException re) {
                reportError(re);
                // match() routine has already called recover()
            }
        }
        return super.nextToken();
    }
}


prog : statement*;

statement
    : simpleStatement? BodyString
    | '{' statement '}' -> statement
    | bracketedBlock
    //| UnquotedString Colon statement -> ^(Label UnquotedString
statement)
    | classDefinition
    | interfaceDefinition
    | complexStatement
    | simpleStatement ';'!
    ;

bracketedBlock
    : '{' stmts=statement* '}' -> ^(Block $stmts)
    ;

interfaceDefinition
    : Interface interfaceName=UnquotedString interfaceExtends?
        OpenCurlyBrace
        interfaceMember*
        CloseCurlyBrace
        -> ^(Interface $interfaceName interfaceExtends?
interfaceMember*)
    ;

interfaceExtends
    : Extends^ UnquotedString (Comma! UnquotedString)*
    ;
interfaceMember
    : Const UnquotedString (Equals atom)? ';'
        -> ^(Const UnquotedString atom?)
    | fieldModifier* Function UnquotedString parametersDefinition
';'
        -> ^(Method ^(Modifiers fieldModifier*) UnquotedString
parametersDefinition)
    ;

classDefinition
    :   classModifier?
        Class className=UnquotedString
        (Extends extendsclass=UnquotedString)?
        classImplements?
        OpenCurlyBrace
        classMember*
        CloseCurlyBrace
        -> ^(Class ^(Modifiers classModifier?) $className
^(Extends $extendsclass)? classImplements?
            classMember*
        )
    ;

classImplements
    :  Implements^ (UnquotedString (Comma! UnquotedString)*)
    ;

classMember
    : fieldModifier* Function UnquotedString parametersDefinition
        (bracketedBlock | ';')
        -> ^(Method ^(Modifiers fieldModifier*) UnquotedString
parametersDefinition bracketedBlock?)
    | Var Dollar UnquotedString (Equals atom)? ';'
        -> ^(Var ^(Dollar UnquotedString) atom?)
    | Const UnquotedString (Equals atom)? ';'
        -> ^(Const UnquotedString atom?)
    | fieldModifier* (Dollar UnquotedString) (Equals atom)? ';'
        -> ^(Field ^(Modifiers fieldModifier*) ^(Dollar
UnquotedString) atom?)
    ;

fieldDefinition
    : Dollar UnquotedString (Equals atom)? ';'-> ^(Field ^(Dollar
UnquotedString) atom?)
```

```
    ;

classModifier
    : 'abstract';

fieldModifier
    : AccessModifier | 'abstract' | 'static'
    ;


complexStatement
    : If '(' ifCondition=expression ')' ifTrue=statement
conditional?
        -> ^('if' expression $ifTrue conditional?)
    | For '(' forInit forCondition forUpdate ')' statement ->
^(For forInit forCondition forUpdate statement)
    | Foreach '(' variable 'as' arrayEntry ')' statement ->
^(Foreach variable arrayEntry statement)
    | While '(' whileCondition=expression? ')' statement ->
^(While $whileCondition statement)
    | Do statement While '(' doCondition=expression ')' ';' ->
^(Do statement $doCondition)
    | Switch '(' expression ')' '{'cases'}' -> ^(Switch expression
cases)
    | functionDefinition
    ;

simpleStatement
    : Echo^ commaList
    | Global^ name (','! name)*
    | Static^ variable Equals! atom
    | Break^ Integer?
    | Continue^ Integer?
    //| Goto^ UnquotedString
    | Return^ expression?
    | RequireOperator^ expression
    | expression
    ;


conditional
    : ElseIf '(' ifCondition=expression ')' ifTrue=statement
conditional? -> ^(If $ifCondition $ifTrue conditional?)
    | Else statement -> statement
    ;

forInit
    : commaList? ';' -> ^(ForInit commaList?)
    ;

forCondition
    : commaList? ';' -> ^(ForCondition commaList?)
    ;

forUpdate
    : commaList? -> ^(ForUpdate commaList?)
    ;

cases
    : casestatement*  defaultcase
    ;

casestatement
    : Case^ expression ':'! statement*
    ;

defaultcase
    : (Default^ ':'! statement*)
    ;

functionDefinition
    : Function UnquotedString parametersDefinition bracketedBlock
->
        ^(Function UnquotedString parametersDefinition
bracketedBlock)
    ;

parametersDefinition
    : OpenBrace (paramDef (Comma paramDef)*)? CloseBrace ->
^(Params paramDef*)
    ;

paramDef
    : paramName (Equals^ atom)?
    ;

paramName
    : Dollar^ UnquotedString
    | Ampersand Dollar UnquotedString -> ^(Ampersand ^(Dollar
UnquotedString))
    ;

commaList
    : expression (','! expression)*
    ;

expression
    : weakLogicalOr
    ;

weakLogicalOr
    : weakLogicalXor (Or^ weakLogicalXor)*
    ;
```

```
weakLogicalXor
    : weakLogicalAnd (Xor^ weakLogicalAnd)*
    ;

weakLogicalAnd
    : assignment (And^ assignment)*
    ;

assignment
    : name ((Equals | AsignmentOperator)^ assignment)
    | ternary
    ;

ternary
    : logicalOr QuestionMark expression Colon expression ->
^(IfExpression logicalOr expression*)
    | logicalOr
    ;

logicalOr
    : logicalAnd (LogicalOr^ logicalAnd)*
    ;

logicalAnd
    : bitwiseOr (LogicalAnd^ bitwiseOr)*
    ;

bitwiseOr
    : bitWiseAnd (Pipe^ bitWiseAnd)*
    ;

bitWiseAnd
    : equalityCheck (Ampersand^ equalityCheck)*
    ;

equalityCheck
    : comparisionCheck (EqualityOperator^ comparisionCheck)?
    ;

comparisionCheck
    : bitWiseShift (ComparisionOperator^ bitWiseShift)?
    ;

bitWiseShift
    : addition (ShiftOperator^ addition)*
    ;

addition
    : multiplication ((Plus | Minus | Dot)^ multiplication)*
    ;

multiplication
```

```
    : logicalNot ((Asterisk | Forwardslash | Percent)^
logicalNot)*
    ;

logicalNot
    : Bang^ logicalNot
    | instanceOf
    ;

instanceOf
    : negateOrCast (Instanceof^ negateOrCast)?
    ;

negateOrCast
    : (Tilde | Minus | SuppressWarnings)^ increment
    | OpenBrace PrimitiveType CloseBrace increment -> ^(Cast
PrimitiveType increment)
    | OpenBrace! weakLogicalAnd CloseBrace!
    | increment
    ;

increment
    : IncrementOperator name -> ^(Prefix IncrementOperator name)
    | name IncrementOperator -> ^(Postfix IncrementOperator name)
    | newOrClone
    ;

newOrClone
    : New^ nameOrFunctionCall
    | Clone^ name
    | atomOrReference
    ;

atomOrReference
    : atom
    | reference
    ;

arrayDeclaration
    : Array OpenBrace (arrayEntry (Comma arrayEntry)*)? CloseBrace
-> ^(Array arrayEntry*)
    ;

arrayEntry
    : (keyValuePair | expression)
    ;

keyValuePair
    : (expression ArrayAssign expression) -> ^(ArrayAssign
expression+)
    ;
```

```
atom: SingleQuotedString | DoubleQuotedString | HereDoc | Integer
| Real | Boolean | arrayDeclaration
    ;

reference
    : Ampersand^ nameOrFunctionCall
    | nameOrFunctionCall
    ;

nameOrFunctionCall
    : name OpenBrace (expression (Comma expression)*)? CloseBrace
-> ^(Apply name expression*)
    | name
    ;

name: staticMemberAccess
    | memberAccess
    | variable
    ;

staticMemberAccess
    : UnquotedString '::'^ variable
    ;

memberAccess
    : variable
        ( OpenSquareBrace^ expression CloseSquareBrace!
        | '->'^ UnquotedString)*
    ;

variable
    : Dollar^ variable
    | UnquotedString
    ;

BodyString
    : '?>' (('<' ~ '?')=> '<' | ~'<' )* ('<?' ('php'?))?
    ;

fragment
FirstBodyString
    : (('<' ~ '?')=> '<' | ~'<' )* '<?' ('php'?)
    ;

MultilineComment
    : '/*' (('*' ~ '/')=>'*' | ~ '*')* '*/' {$channel=HIDDEN;}
    ;

SinglelineComment
    : '//'  (('?' ~'>')=>'?' | ~('\n'|'?'))* {$channel=HIDDEN;}
    ;
```

```
UnixComment
    : '#' (('?' ~'>')=>'?' | ~('\n'|'?'))* {$channel=HIDDEN;}
    ;


Array
    : ('a'|'A')('r'|'R')('r'|'R')('a'|'A')('y'|'Y')
    ;

RequireOperator
    : 'require' | 'require_once' | 'include' | 'include_once'
    ;

PrimitiveType
    : 'int'|'float'|'string'|'array'|'object'|'bool'
    ;

AccessModifier
    : 'public' | 'private' | 'protected'
    ;

fragment
Decimal
     :('1'..'9' ('0'..'9')*)|'0'
     ;
fragment
Hexadecimal
     : '0'('x'|'X')('0'..'9'|'a'..'f'|'A'..'F')+
     ;

fragment
Octal
     : '0'('0'..'7')+
     ;
Integer
     :Octal|Decimal|Hexadecimal
     ;

fragment
Digits
     : '0'..'9'+
     ;

fragment
DNum
     :(('.' Digits)=>('.' Digits)|(Digits '.' Digits?))
     ;

fragment
Exponent_DNum
     :((Digits|DNum)('e'|'E')('+''-')?Digits)
     ;
```

```
Real
    : DNum|Exponent_DNum
    ;

Boolean
    : 'true' | 'false'
    ;

SingleQuotedString
    : '\'' (('\\' '\'')=>'\\' '\''
    |         ('\\' '\\')=>'\\' '\\'
    |         '\\' | ~ ('\'' | '\\'))*
      '\''
    ;

fragment
EscapeCharector
    : 'n' | 'r' | 't' | '\\' | '$' | '"' | Digits | 'x'
    ;

DoubleQuotedString
    : '"'  ( ('\\' EscapeCharector)=> '\\' EscapeCharector
    | '\\'
    | ~('\\'|'"') )*
      '"'
    ;

HereDoc
    : '<<<' HereDocContents
    ;

UnquotedString
   : ('a'..'z' | 'A'..'Z' | '_')  ('a'..'z' | 'A'..'Z' | '0'..'9'
| '_')*
   ;

fragment
HereDocContents
    : {
        StringBuilder sb = new StringBuilder();
        while(input.LA(1)!='\n'){
            sb.append((char)input.LA(1));
            input.consume();
        }
        input.consume();
        String hereDocName = sb.toString();
        int hdnl = hereDocName.length();
        while(true){
            boolean matchEnd = true;
            for(int i = 0; i<hdnl; i++){
                if(input.LA(1)!=hereDocName.charAt(i)){
                    matchEnd=false;
                    break;
                }
                input.consume();
            }
            if(matchEnd==false){
                while(input.LA(1)!='\n'){
                    input.consume();
                }
                input.consume();
            }else{
                break;
            }
        }
    }
    ;

AsignmentOperator
    : '+='|'-='|'*='|'/='|'.='|'%='|'&='|'|='|'^='|'<<='|'>>='
    ;

EqualityOperator
    : '==' | '!=' | '===' | '!=='
    ;

ComparisionOperator
    : '<' | '<=' | '>' | '>=' | '<>'
    ;

ShiftOperator
    : '<<' | '>>'
    ;

IncrementOperator
    : '--'|'++'
    ;


fragment
Eol : '\n'
    ;

WhiteSpace
@init{
    $channel=HIDDEN;
}
     :    (' '| '\t'| '\n'|'\r')*
     ;
```