Java Security Vulnerabilities
Detection with Static Analysis

Master’s Thesis

G.L. Cheng
Java Security Vulnerabilities Detection with Static Analysis

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE TECHNOLOGY

by

G.L. Cheng
born in Delft, The Netherlands
© 2007 G.L. Cheng. All rights reserved.

Cover picture: the logos of TU Delft and Software Improvement Group.
Java Security Vulnerabilities
Detection with Static Analysis

Author: G.L. Cheng
Student id: 1047280
Email: g.l.cheng@student.tudelft.nl

Abstract

Security in software plays an important role in today's society as computer networking is getting more and more important. Security measures are taken to protect private information, but bad programming practices can still cause security vulnerabilities in software systems. Source code analysis tools can be used to detect such security vulnerabilities automatically. The use of these tools helps to improve the quality and security of software systems and could prevent future problems.

The class of security vulnerabilities called input validation vulnerabilities can be detected using static taint analysis. The design and implementation of such a tool are the subject of this paper. This tool detects input validation vulnerabilities in source code written in the Java programming language. This paper also describes in detail how to deal with complexities related to the object oriented nature of Java.

The tool first derives a graph structured model from the source code. This graph structured model captures data dependency relations between important program elements. This graph model is then analyzed using taint analysis to detect potential input validation vulnerabilities.

Thesis Committee:

Chair: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor: Dr. Ing. L. Moonen, Faculty EEMCS, TU Delft
Company supervisor: Dr. P. John, Software Improvement Group B.V.
Committee Member: Drs. J.W.J. Heijnsdijk, Faculty EEMCS, TU Delft
Preface

First of all, I would like to thank the people of the Software Improvement Group for the great learning opportunity and the great time I had during this project.

Secondly, special thanks to my family and friends for being here. Their encouragement, support and love make this project a success and especially meaningful.

Thirdly, I would like to thank my supervisors Dr. ing. Leon Moonen and Dr. Per John, who gave me a hard time and a lot of work with their justified criticism. Their criticism ultimately found their way into this paper.

G.L. Cheng
Delft, the Netherlands
February 6, 2007
# Contents

Preface iii

Contents v

List of Figures ix

List of Code Fragments xi

1 Introduction 1
   1.1 Style Conventions 1
   1.2 Development Environment 2

2 About the Project 5
   2.1 Project Context 5
      2.1.1 Software Improvement Group (SIG) 5
      2.1.2 Software Evolution Research Lab (SWERL) 5
   2.2 Project Goals and Research Questions 6
   2.3 Project Motivation 6
   2.4 Project Phases 7
   2.5 Project Requirements 8

3 Java Security Vulnerabilities 9
   3.1 SQL Injection 9
   3.2 Cross Site Scripting 11
   3.3 Command Injection 12
   3.4 Path Traversal 13
   3.5 Input Validation Vulnerabilities Detection 13

4 Software Analysis Toolkit (SAT) 15
   4.1 Architecture 15
      4.1.1 Info Graph 17
   4.2 Architecture Extensions for Java Analysis 18
      4.2.1 Java Info Graph 20
   4.3 Testing 21
## 10 Related Work

10.1 Dynamic Analysis .................................. 69
   10.1.1 Runtime Monitoring .............................. 69
   10.1.2 Penetration Testing .............................. 70
10.2 Static Analysis ......................................... 70
   10.2.1 WebSSari ........................................... 71
   10.2.2 Pixy ................................................. 71
   10.2.3 The Griffin Software Security Project ............ 71

## 11 Conclusions and Future Work

11.1 Contributions ......................................... 75
11.2 Conclusions ........................................... 75
11.3 Discussion ............................................. 76
11.4 Future work ........................................... 76

## Bibliography

A List of Methods ............................................. 83
   A.1 Untrusted Methods ................................. 83
   A.2 Critical Methods ................................. 84

B Type Inference Nodes ....................................... 87
   B.1 Integer Type Inference Nodes ..................... 87
   B.2 Object Oriented Type Inference Nodes .......... 87

C Expression Operators ....................................... 89
   C.1 Expression Elements ............................. 89
   C.2 Expression Separators ........................... 89

D Guestbook Source Code ................................... 91

E Tainted Paths in Guestbook ................................ 95

F Glossary .................................................... 97
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Flow diagram</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Modules diagram</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Graph diagram</td>
<td>3</td>
</tr>
<tr>
<td>3.1</td>
<td>Web server Java SQL architecture</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>Cross Site Scripting (XSS)</td>
<td>11</td>
</tr>
<tr>
<td>3.3</td>
<td>Typical system architecture with input validation vulnerabilities</td>
<td>14</td>
</tr>
<tr>
<td>3.4</td>
<td>Path from untrusted method to critical method</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>Simple analysis</td>
<td>15</td>
</tr>
<tr>
<td>4.2</td>
<td>Basic architecture SAT</td>
<td>16</td>
</tr>
<tr>
<td>4.3</td>
<td>Basic Info Graph</td>
<td>18</td>
</tr>
<tr>
<td>4.4</td>
<td>Java architecture SAT</td>
<td>19</td>
</tr>
<tr>
<td>4.5</td>
<td>Java Info Graph</td>
<td>20</td>
</tr>
<tr>
<td>5.1</td>
<td>Flow diagram AssignmentExtractionVisitor</td>
<td>26</td>
</tr>
<tr>
<td>5.2</td>
<td>Flow diagram MethodCallExtractionVisitor</td>
<td>27</td>
</tr>
<tr>
<td>5.3</td>
<td>Fact Graph example</td>
<td>29</td>
</tr>
<tr>
<td>6.1</td>
<td>Flow diagram new AssignmentExtractionVisitor</td>
<td>37</td>
</tr>
<tr>
<td>6.2</td>
<td>Flow diagram new MethodCallExtractionVisitor</td>
<td>38</td>
</tr>
<tr>
<td>6.3</td>
<td>Flow diagram new ReturnExtractionVisitor</td>
<td>39</td>
</tr>
<tr>
<td>6.4</td>
<td>Security Fact Graph step 1</td>
<td>41</td>
</tr>
<tr>
<td>6.5</td>
<td>Security Fact Graph step 2</td>
<td>41</td>
</tr>
<tr>
<td>6.6</td>
<td>Security Fact Graph step 3</td>
<td>42</td>
</tr>
<tr>
<td>6.7</td>
<td>Security Fact Graph step 4</td>
<td>42</td>
</tr>
<tr>
<td>6.8</td>
<td>Security Fact Graph step 5</td>
<td>43</td>
</tr>
<tr>
<td>6.9</td>
<td>Security Fact Graph step 6</td>
<td>43</td>
</tr>
<tr>
<td>6.10</td>
<td>Security Fact Graph complete</td>
<td>44</td>
</tr>
<tr>
<td>7.1</td>
<td>Security Fact Graph of classes A, B and Container</td>
<td>51</td>
</tr>
<tr>
<td>7.2</td>
<td>Security Fact Graph consisting of only untrusted and critical methods</td>
<td>54</td>
</tr>
<tr>
<td>7.3</td>
<td>Security Fact Graph with a loop</td>
<td>56</td>
</tr>
</tbody>
</table>
8.1 Java Security Analysis architecture ........................................... 59
8.2 Security Fact Graph of Guestbook ............................................. 61
# List of Code Fragments

<table>
<thead>
<tr>
<th>Section</th>
<th>Code Fragment</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Example code fragment</td>
<td>2</td>
</tr>
<tr>
<td>3.1</td>
<td>SQL injection vulnerable code</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>XSS vulnerable code</td>
<td>11</td>
</tr>
<tr>
<td>3.3</td>
<td>Command injection vulnerable code</td>
<td>12</td>
</tr>
<tr>
<td>3.4</td>
<td>Path traversal vulnerable code</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>Car class</td>
<td>21</td>
</tr>
<tr>
<td>4.2</td>
<td>Car test class</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>Method parameter use cases</td>
<td>27</td>
</tr>
<tr>
<td>5.2</td>
<td>Fact Graph example code (SampleClass)</td>
<td>28</td>
</tr>
<tr>
<td>6.1</td>
<td>Object inner code</td>
<td>34</td>
</tr>
<tr>
<td>6.2</td>
<td>Object manipulation</td>
<td>34</td>
</tr>
<tr>
<td>6.3</td>
<td>Security Fact Graph example code (NewSampleClass)</td>
<td>40</td>
</tr>
<tr>
<td>6.4</td>
<td>Security Fact Graph example code (NewStack)</td>
<td>40</td>
</tr>
<tr>
<td>6.5</td>
<td>Class instances</td>
<td>44</td>
</tr>
<tr>
<td>6.6</td>
<td>Static method calls</td>
<td>45</td>
</tr>
<tr>
<td>6.7</td>
<td>Class members</td>
<td>45</td>
</tr>
<tr>
<td>6.8</td>
<td>Data path with loop</td>
<td>46</td>
</tr>
<tr>
<td>6.9</td>
<td>Approximation library data flow</td>
<td>47</td>
</tr>
<tr>
<td>7.1</td>
<td>Container class</td>
<td>50</td>
</tr>
<tr>
<td>7.2</td>
<td>A class</td>
<td>51</td>
</tr>
<tr>
<td>7.3</td>
<td>B class</td>
<td>51</td>
</tr>
<tr>
<td>7.4</td>
<td>Caller class</td>
<td>52</td>
</tr>
<tr>
<td>7.5</td>
<td>Callee class</td>
<td>52</td>
</tr>
<tr>
<td>7.6</td>
<td>Pseudo code tainted path finding</td>
<td>55</td>
</tr>
<tr>
<td>8.1</td>
<td>Example use prepared statements 1</td>
<td>62</td>
</tr>
<tr>
<td>8.2</td>
<td>Example use prepared statements 2</td>
<td>62</td>
</tr>
<tr>
<td>D.1</td>
<td>Guestbook source code</td>
<td>91</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In today’s world where computer networking plays an important role in everyday life, computer criminals cause havoc in critical or important network environments. Common criminal activities include: tapping network traffic, tampering databases, modifying websites, disabling services and information theft [26]. These activities can cause bad publicity, data-loss and privacy problems, which could result in significant (financial) damages to companies. Systems that are secure enough to resist such attacks are therefore essential.

Security breaches are often the result of bad programming practices during development. Some of these security vulnerabilities are easily detected and fixed when the program crashes or unexpected output is given. Other security vulnerabilities will never be noticed during normal use. Automatic source code analyzers can help detecting these security vulnerabilities before deployment of a software system.

The design and implementation of such a tool is the main subject of this paper. First, the details and context of this project are described in chapter 2. Then a couple of examples of typical security vulnerabilities that have to be detected by the tool are explained and analyzed in chapter 3. The existing analysis infrastructure or framework, the tool is build on, is documented and described in chapters 4 and 5, which are followed by the chapters 6, 7 and 8, which discuss the implementation details and the design of the tool. Chapter 9 discusses the influences of the use of several popular frameworks on the analysis. In chapter 10, about related work, similar projects that try to achieve the same thing are compared with this tool. Future improvements that increase precision are discussed in chapter 11, which also presents the conclusions.

1.1 Style Conventions

This paper follows a style convention for clarity. The following style conventions are used:

- Relevant large program parts are displayed as code fragments, which are listed on its own index page. The code of a simple class is given in code fragment 1.1. The keywords that belong to the programming language are bold. The lines are numbered for easy referencing in from the text.
1.2 Development Environment

To get an impression of how the security tool is developed at the office of the Software Improvement Group (SIG), the development environment is described. To confirm to the existing software standard used by SIG, the development environment influences the way the tool is developed. The workstation is an Apple iMac running Mac OS X as Operating System, which is also connected to the Internet. The main programming

```java
public class Thesis {
    private String author;
    private String title;
    public Thesis(String author, String title) {
        this.author = author;
        this.title = title;
    }
    public String getAuthor() {
        return author;
    }
    public String getTitle() {
        return title;
    }
}
```

Code Fragment 1.1: Example code fragment

- General flow diagrams illustrate the flow of processes or modules. The flow diagram captures the global view of operations (phases) that are needed to achieve a goal. An example is given in figure 1.1, which illustrates a simple program flow. The arrows show the sequence of the phases.

- Modules of a system or application are illustrated in a modules diagram like figure 1.2. Abstract classes that have to be implemented are colored grey. Data structures are modeled as an octagon figure in the diagram.

- Nodes in a graph data structure are illustrated as a graph diagram in figure 1.3, which depicts a binary tree. The capsule shaped nodes may differ in horizontal/vertical ratio, due to the text inside. The nodes can be any kind of program element or related information, like variable or directory. The nodes are named using strict and consistent name format rules, so it can be easily determined what it represents.

- Quotes from program source code, such as types, expressions and statements are displayed in a different font. Example: `JavaClass or var = this.getVar()`.

- Terms and abbreviations that are explained in the glossary are underlined. Example: `static analysis`. The glossary can be found in appendix F.
1.2 Development Environment

**Figure 1.1: Flow diagram**

**Figure 1.2: Modules diagram**

**Figure 1.3: Graph diagram**
language used, is the Java programming language [29]. The project source files will be centrally stored in a CVS [34] repository for source code version management. The Eclipse [44] Integrated Development Environment (IDE) is used together with a collection of plugins like Checkstyle [4]. Checkstyle checks the code for conformance of a standard code style. The inhouse framework called the Software Analysis Toolkit (SAT) is available for use in order to standardize tools and promote code reuse. The code in the repository is built daily for automated testing that includes unit testing and Clover [11] coverage report.
Chapter 2

About the Project

This chapter describes various details of this project. First, the involved organizations are described. The goals of this project and the research questions are explained in the next section, which is followed by the section that discusses the motivation of this project. The project phases and elicitated requirements are listed and explained in the last two sections.

2.1 Project Context

2.1.1 Software Improvement Group (SIG)

This project is initiated in cooperation with the Software Improvement Group (SIG), which is based in Amsterdam. SIG mainly provides services regarding the analysis of software systems [9]. The Software Risk Assessment (SRA) is an important service, in which SIG makes an assessment of the software quality of the software systems of companies. The goal is to improve the quality of the software system, to provide a better understanding of these systems, to support decision-making and to locate weak areas in the software. Qualitative aspects looked at in a SRA are, for instance, maintainability of the source code, stability, software process and documentation.

In order to do that, several tools are used to assist in analyzing the software. For every programming language specialized analyzers are needed. The results and other information are used to assess the technical quality of the software system. Together with interviews, an advice is formulated in a written report. In this report, the risks are presented and recommendations are given to mitigate those risks.

2.1.2 Software Evolution Research Lab (SWERL)

The project is supervised by the Software Evolution Research Laboratory (SWERL), which is part of the Software Engineering Research Group, department of Software Technology, faculty of Electrical Engineering, Mathematics, and Computer Science (EWI), Delft University of Technology (TU Delft). The field of research of SWERL concerns the management of change of software during its life (software evolution) [8]. Changes to existing software are needed for several reasons, such as the need for additional requirements, bug fixes, porting to other systems, etc. These changes can have a negative impact on the quality of the software. The system can become
increasingly complex, making it difficult to understand and maintain. Especially if the documents are not updated accordingly. SWERL investigates how to tackle this problem, trying to develop techniques to make software maintainable and understandable. Other research goals are developing techniques to analyze existing legacy software for reconstruction.

### 2.2 Project Goals and Research Questions

The main goal of this project is to design and implement a tool in the Java programming language that can detect the class of security vulnerabilities called input validation successfully. To answer this problem, it is necessary to find answers to the following questions:

1. What are input validation vulnerabilities?
2. How to detect these input validation vulnerabilities in Java source code?
3. How does a standard analysis look like?
4. What can be reused in the current analysis framework?
5. Which existing analysis can be used or modified?
6. What are the problems caused by the complexities of object oriented features in Java?
7. How can these problems be dealt with?
8. Is the tool capable of detecting the mentioned security vulnerabilities?
9. What are the influences of the use of popular frameworks on the analysis?

These questions are answered in this paper. The first two questions concerning input validation vulnerabilities are answered in chapter 3. The next question related to the software standard used to implement an analysis, is answered in chapter 4. In chapter 5 the analysis found to be suitable to be extended is described, including the limitations. Chapter 5 concerns the fifth question. The extended features needed for the new type of analysis are discussed in chapters 6 and 7, which provide answers for the sixth and seventh question. The final results are presented in chapter 8, which is related to the eighth question. The last question is answered in chapter 9, where several popular frameworks are selected for inspection.

### 2.3 Project Motivation

As information network technology is more and more integrated into everyday life and everything is connected to the Internet, security concerns are raising. Many security incidents have occurred that proved the existence of weaknesses in software systems that are considered and needed to be secure. Therefore companies have to make sure private data is shielded correctly from the outside world by taking security measures
in their software. Sometimes these measures are not enough or not implemented correctly due to human mistakes, making it breakable by outsiders. A need for security validation tools that can check the existence of security vulnerabilities in software arises.

In line with the quality assessment services provided by the SIG, a security analysis tool that can assess the security quality of software would be valuable. With this analysis tool, SIG can extend their software assessment services with a specialized software security assessment. The security assessment will give companies that have to protect their data, insight into the security of their software. Found security vulnerabilities can be fixed, preventing future security incidents. The development of such a security tool for future use by the SIG is therefore desirable.

As a requirement of this project, is that it has to be related to the work of SWERL. SWERL main area of research focuses on recovering lost knowledge of large software systems. One method is using compiler construction related tools to automate the knowledge recovery. This project also uses compiler construction techniques to detect potential security vulnerabilities in software, which makes this project closely related to the field of research of SWERL.

### 2.4 Project Phases

The project consists of several main phases of the project, which are:

1. *Requirements Elicitation:* The requirements are determined in this phase. An initial proposal is made and adjusted by interviewing people of the software development department. The elicited requirements are listed in the next section.

2. *Analysis:* The existing software infrastructure and data structures are analyzed in this phase. Important relevant parts and their internal workings are documented in this paper. The SAT chapter is based on this phase.

3. *Design:* The design and working of the analyzer are described here. Deliverables of this phase are the algorithm, the data structure model and the architecture model.

4. *Implementation:* Building the analyzer is done in this phase. The analyzer will be built using Eclipse and the Java Standard Edition 5 framework. During the development, automated unit tests are written to test the functionality.

5. *Assessment and Testing:* Testing the analyzer on various software systems is done in this phase. The performance of the analyzer is assessed and validated.

6. *Refinement:* Based on the assessment and testing, a refinement of the analyzer might be needed. This iterative process starts with the design phase, although the requirements can be adjusted as well.
2.5 Project Requirements

The requirements are elicited after interviewing the major stakeholders, which include the head of development (Dr. Per John) and the university supervisor (Dr. ing. Leon Moonen). The analyzer should conform to the following requirements:

1. The analyzer should detect the general class of security vulnerabilities called input validation vulnerabilities by detecting paths taken by dangerous user input values. This should be done by a variant of type qualifier inference and taint analysis.

2. The analyzer should analyze sources written in the Java programming language of version 1.5.

3. The analyzer should use the Spring Framework [10] for its configuration management. This framework makes managing configurations easier, by using standardized XML files to set parameters for the analyzer.

4. The analyzer should be integrated in the Software Analysis Toolkit (SAT). The SAT has a standardized infrastructure for starting the analysis and processing results. The analyzer should conform to this standard.

5. The analyzer should make use of existing data structures and classes as much as possible. This is to prevent reinventing the wheel and reduce redundant code. Reuse will also speed up the development, because some classes do not have to be implemented from scratch. Some classes are required to be used, to satisfy the previous requirement.

6. The analyzer should be able to run in different configurations modes, which have to be configurable through the Spring XML configuration file. Configuration settings should be centralized in the Spring XML configuration file and the user should have the option to change or customize settings. Changing the sensitivity by changing options is required. More sensitivity is desired to increase precision, at the expense of time resources.
Chapter 3

Java Security Vulnerabilities

In this chapter a selection of common security vulnerabilities is discussed. These security issues are not Java specific, so it is certainly not a programming language design flaw. This type of security vulnerabilities can also be found in different languages like PHP or Perl. Careful programming can prevent these issues from occurring.

3.1 SQL Injection

A popular application of the Java programming language is the use of Java servlets to handle web server requests. The underlying pattern in the architecture of a potential vulnerable system is depicted below in figure 3.1 as an example. The architecture consists of three modules or components, which are interacting with each other. The web server component is responsible for handling requests initiated by the user. When a HTTP request is received from the user, the web server delegates the request to the Java servlet. The Java servlet may interact with a SQL database by querying, which depends on the user input. In short, a system may be vulnerable to SQL injection attacks, when SQL input by an application depends on the user input.

SQL injection [43, 31, 32, 13, 19, 38] occurs when the semantics of a SQL query that is embedded in the source code is changed due to specially crafted user input. The bad query can do things not allowed or intended by the application. The syntax of the embedded may be correct, but the semantics is changed. This query can disclose

![Figure 3.1: Web server Java SQL architecture](image-url)
private data, alter data or elevate privileges. The example code in code fragment 3.1 is syntactically and semantically correct, but a SQL injection vulnerability exists, which makes it possible to alter the SQL query semantically.

```java
// Getting the login information.
String userName = request.getParameter("username");
String passWord = request.getParameter("password");

// Composing the query string.
String query = "SELECT * FROM AuthorizedUsers WHERE Username='" + userName + "' AND Password='" + passWord + ";"

// Executing the query.
Statement stmt = connection.createStatement();
ResultSet resultSet = stmt.executeQuery(query);

// Output according to query execution results.
if (resultSet) {
    return (true);
} else {
    return (false);
}
```

Code Fragment 3.1: SQL injection vulnerable code

The code is meant to verify username/password combinations. It does so by checking the database for the existence of this specific pair, supplied by the user. Suppose there is a valid user named "nick" and his password is "secret". A valid SQL query string is composed by the Java program:

```
SELECT * FROM AuthorizedUsers WHERE Username='nick'
    AND Password='secret'
```

This query only succeeds if the password of "nick" equals "secret". If not, it fails. If a cracker knows that the user ‘nick’ exists, but he does not know his password, he can still circumvent this check. He does that by supplying a special user name: "nick' --" and a random password: "magic". The "--" denotes the beginning of a SQL comment. Everything behind this string is ignored by the SQL server. Now, the query is:

```
SELECT * FROM AuthorizedUsers WHERE Username='nick' --''
    AND Password='magic'
```

The last part of this query is ignored by the SQL server and is interpreted as:

```
SELECT * FROM AuthorizedUsers WHERE Username='nick'
```

In essence, this SQL statement checks for the existence of the user "nick" and not the username/password combination. If the user ”nick” exists, this statement will succeed, falsely authorizing nick with any supplied password. Note that this is just one example of exploitation techniques. This hole is large enough to gain control of the database, limited by the rights of the Java application.
3.2 Cross Site Scripting

In cross site scripting (XSS) [43, 31, 32, 19, 38] a vulnerability exists in a web application that makes it possible to trick users of the website to execute arbitrary code using the website as a relay. The code appears to be originated from the website that may be a website that is trusted by the users. The trust of the user in a website with a XSS vulnerability and the website itself are abused to trick the computer/browser of the user to execute arbitrary code, which steals private information from the user. No entry is gained in the website itself.

The attacker who wants to abuse an XSS vulnerable website, first crafts a special hyperlink that has hidden code embedded. The code is meant to steal information from the user. The second problem is to get the user to click on this link. One way to do it is to post it on a forum that is known to be regularly visited by users of the XSS vulnerable website, another way is to email it directly to the users. If an user has clicked on the link, the code in the link is relayed and echoed back to the user by the website. The browser of the user starts executing the code, which can do things like stealing cookie information that contains login information. Stealing cookie information can be done by letting the code dump this information at a specially installed drop site.

This security vulnerability has its similarities with phishing [35], but the difference is that the XSS vulnerable website is relaying malicious code in this case and is therefore a key component in the exploitation process. In the case of phishing, the appearance of a website is imitated and the real website plays no active role.

An example of vulnerable code is shown in figure 3.2. The servlet retrieves a name parameter of the user, who supplied it through a form initially. The name parameter is directly used without checking to print a welcome message, which will be displayed in the browser of the user.
3.3 Command Injection

Command Injection [43, 31, 32, 38] tricks the application into executing another program. This can be used by an outsider to gain entry into a web server or to execute something with the same privileges as the application. It can also be used by an user of a stripped down computer. A stripped down computer is intentionally restricted in accessibility, so the only use of the computer is through a particular program or interface.

```java
1 // Getting the user input.
2 String name = request.getParameter("name");
3
4 // Use it for display in browser.
5 out.println("Welcome " + name + "!");
```

Code Fragment 3.2: XSS vulnerable code

A specially crafted link to the page where the code above is exploited may be posted on a forum. The link may look like this:

http://www.yourvulnerablesite.com/servlet?name=
Albert<script>x=document.cookie;alert(x)</script>

In this example the name parameter has a malicious tail: a line of JavaScript code that is enclosed between the script tags. The code in this example is fairly innocent. Only the content of the cookie is displayed in an alert window. The cookie content can be gathered by the attacker by incorporating a drop site in the link in more serious cases. This can be done without the user knowing. Like this:

```html
<script>document.location='www.dropsite.com/drophere.cgi?'
+document.cookie</script>
```

The attacker can simply gather the cookie information by looking in the log files. Abusing the information is the next step.

3.3 Command Injection

Command Injection [43, 31, 32, 38] tricks the application into executing another program. This can be used by an outsider to gain entry into a web server or to execute something with the same privileges as the application. It can also be used by an user of a stripped down computer. A stripped down computer is intentionally restricted in accessibility, so the only use of the computer is through a particular program or interface.

```java
1 String parameter = textField.getText();
2 String command = "SomeExecutable " + parameter;
3 Process child = Runtime.getRuntime().exec(command);
```

Code Fragment 3.3: Command injection vulnerable code

The command string variable can contain arbitrary strings like: "delete document.txt". Running the code in figure 3.3 deletes document.txt. Other scripting or programming languages spawns a command shell to execute the command, which is very dangerous, because it is possible to inject more commands using command separators like ";". A typical command string for UNIX like systems would be: "rm -rf document.txt; mail cracker@evil.org < /etc/passwd", which deletes the intended document.txt first and then the second command after the command separator is executed. In this case, the password file is emailed to "cracker@evil.org".
Luckily, the Java developers understood the problem of giving direct access to a shell, therefore the intended program is executed without the shell and with the intended parameters. So using command separators to inject additional commands is not possible in the Java case. This limits the risk, but it can still be dangerous if “SomeExecutable” does not validate its input correctly. Imagine “SomeExecutable” being a program that accepts a scripting language as parameter. An attacker can use this language to gain control of “SomeExecutable”.

### 3.4 Path Traversal

This flaw, called path traversal [38], is exploited by including special path traversal strings in the input, to display private directory contents or to manipulate (opening/reading/deleting) files outside the intended public directory. Unlike command injection, this vulnerability is used to manipulate file access. An example of directory listing in Java source code can be found in code fragment 3.4.

```java
// Retrieving directory name from user.
String dirName = request.getParameter("dirname");

// Open directory.
File dir = new File("C:\Public_directories" + dirName);

// List content of directory code here...
```

**Code Fragment 3.4: Path traversal vulnerable code**

By including strings like: ‘.’ or ‘..\fileName’ it is easy to traverse the directory structure of the whole system and access files or directories outside public directories.

Against UNIX systems, this security flaw is popular for retrieving the /etc/passwd file, which contains local user account information.

### 3.5 Input Validation Vulnerabilities Detection

SQL injection, cross site scripting, command injection and path traversal vulnerabilities have fundamental properties in common. The commonality is that user input is trusted and not validated before it is used in a subsystem. A subsystem is an independent system that is used by the application. This subsystem is accessed by its interface, which is commonly a collection of methods. The subsystems of the vulnerable programs discussed earlier, are the SQL database for applications vulnerable to SQL injections, the browser for cross site scripting vulnerable websites and the underlying operating system for the last two vulnerabilities. These vulnerabilities would not exist if user input is properly checked and sanitized before being used, eliminating dangerous input. The vulnerability is exploited when the attacker tricks the subsystem into doing something not intended by the application.

The definitions of two classes of security methods are presented next. The security methods play an important role in the next chapters, therefore definitions are needed to avoid confusions.
3.5 Input Validation Vulnerabilities Detection

User
Application
Subsystem
Database

Figure 3.3: Typical system architecture with input validation vulnerabilities

Untrusted Method
Critical Method

Figure 3.4: Path from untrusted method to critical method

**Definition 1 (Untrusted Method)** A method used to retrieve user input. If this input is not validated before use, it may pose a threat to the system.

**Definition 2 (Critical Method)** A method that requires validated input. This is often a method used to interact with a subsystem.

The vulnerabilities mentioned earlier are generally called input validation vulnerabilities, because they share the same vulnerability principle and can be detected using the same detection technique [39, 40, 32]. The key idea is to detect the use of values obtained from untrusted input methods as input of critical subsystem methods. If such a vulnerability exists, there has to be an execution path between these methods. This path is taken by the value through data propagation. By using analysis methods, such as data flow analysis [22, 53, 25], it is possible to detect such paths from untrusted input methods to critical methods. In this way, it is possible to detect input validation vulnerabilities.

Before these paths can be detected it is necessary to identify all untrusted input methods and critical methods, which can be used as input by the analyzer. These methods are often found in standard libraries, but custom methods can also be included. A list of these methods should be composed manually using specific knowledge of the system for better results.
Chapter 4

Software Analysis Toolkit (SAT)

This section describes the Software Analysis Toolkit (SAT) framework used by the SIG to perform Software Risk Assessments (SRAs). The SAT is a collection of software analysis programs used to analyze all kinds of software systems. Each program does so by looking at the source code of the system, which is better known as static analysis. Implementing a new analysis for the SAT requires the use of standard classes and interfaces provided by the SAT software framework. SAT makes it possible to perform the analysis in a standardized way and it prevents source code duplication. The standard classes that are described below form the basis of SAT. To understand the existence of these classes and why they are standardized, a typical source code analysis will be described.

Figure 4.1 illustrates the main phases of a simple source code analysis. First the analysis is started, which can be done using the command line or in an Integrated Development Environment (IDE) like Eclipse. When it is started, it reads the configuration file for the right settings or options. The next step is to locate the source directory, which is specified in the configuration file. The following step is opening and parsing every source file. The actual analysis is performed in the following step, which analyzes every source file. The next step is to process the results of the analysis for visualization or storage in a database or file. The final step is to present the results, which can be done in graphs, diagrams or tables. Most analyses are the same in the first two phases, which means it is possible to standardize these phases in classes or interfaces. These classes and interfaces, which are part of the SAT framework, are described in the next section.

4.1 Architecture

The architecture of an analysis based on the SAT framework is depicted in figure 4.2. The illustrated classes will be discussed in detail later on. The Info Graph is not a
4.1 Architecture

The network class, but it is a graph data structure. This architecture can be mapped to the first couple of analysis steps figure showed earlier. The NetworkRunner is responsible for starting the analysis process with the right parameters, so it can be mapped to both the Start and Configuration steps. The SourceGraphMaker and SourceContext can be mapped to the Parse Source Files step. The SourceContext class reads the files and the SourceGraphMaker uses the directory structure and sources to build the initial Info Graph, populating it with source code. The ObservationVisitor can be mapped to the Analysis step, because this is where the real analyses are performed. The last phases cannot be found in the architecture, because these phases are analysis dependent.

The NetworkRunner class contains the Main method, which is called when the program is started. It instantiates an implementation of the Network interface. The Spring Framework [10] is used for configuration management. Configuration files are in the XML format, where settings and options for the analysis are defined. These XML files are handled by the Spring Framework.

The Network interface specifies a single method called run(). Every Network implementation is created and started by the NetworkRunner object using the run() method. For every parameter in the XML configuration file, a set method is required, which is used by the Spring Framework to set parameters. The Spring Framework is used for configuration and component management of the analysis. Every specific analysis has its own Network implementation, where the analysis is started.

The SourceGraphMaker class is responsible for creating the initial graph data structure called System Graph, which contains an unique node storing some details of the system, like the name of the system and a time stamp. The SourceGraphMaker extends the System Graph with nodes that represent files and directories. The extended System Graph is now called the File Graph. The File Graph reflects the directory/file structure of the software system. This tree data structure is also used to extend it
with other types of nodes and edges storing other information for a various range of purposes. The File Graph, on its turn, can be extended with all kinds of other nodes and edges by other classes, which turns this graph into a heterogeneous data structure. This heterogeneous multi purpose graph is now dubbed the Info Graph, which plays an important role in the architecture of the SAT. The structure and purpose of the Info Graph are described later on.

To accommodate various programming languages, various implementations of the SourceContext interface can be used by the SourceGraphMaker to create the Info Graph. For every program source environment there is a specific implementation of the SourceContext interface. These implementations separate source code files of a specific environment from other files, for further processing. File separation is achieved using special file filter classes. It is possible to analyze systems consisting of sources written in different programming languages, like a system partly written in Cobol and partly written in Java. This class is convenient for adding another analysis for the same programming language, because source code files separation is delegated to this class. Another task is storing the sources in the File Graph.

Implementations of the ObservationVisitor interface are used to process every source file or source file intermediate stored in the File Graph. The SourceContext object creates the ObservationVisitor objects and the File Graph is given to it for further processing or analysis of the source files or source file intermediates. Multiple ObservationVisitor objects can be run. All of them analyze the File Graph and extend it with information, like interim measurements or results. Partial results in leaf nodes can be aggregated upwards the tree to construct the final result.

The MeasurementCollection class and the LocationCollection class are implementations of the Collection interface. The first is used to store all measurement results. A measurement result is stored in the Measurement class. The second stores all source code locations of interest. The specific location details like file name and line number are stored in a Location object.

### 4.1.1 Info Graph

The Info Graph has two important purposes. In the first place it can be considered as a model of the program incorporating several views that serve as intermediates to simplify analysis. Secondly, it is a data structure used to store measurements or analysis data.

The Info Graph consists of different types of nodes and connected with different types of edges. It can also be seen as a representation of multiple graphs or views combined, where nodes are reused. Different types of nodes represent different program elements, like variables or files. When all variables or other program elements are needed, it can simply be extracted from the Info Graph. The nodes itself can be connected to other nodes by various edges, which represent all kinds of relations between program elements, like superclass-subclass relationship edges.

The root of Info Graph is the System node, which represents a software system. This node stores detail of the software system, like name, etc. The software system consists of sources that can be stored in separate directories. For every directory and source code file there is a node representation in the Info Graph. Measurement data or
other derivatives of a source file can be stored as separate nodes in the corresponding source or directory node.

Analyzing source code directly is difficult in most cases, caused by the relatively low level characteristics of source code. To simplify analysis, the abstraction of the program that is derived from the source code and incorporated in the Info Graph as a graph structure is used instead of the source code. The Info Graph is used as a model of the source code on which analyses are performed. This model is more convenient to perform analysis on, because all irrelevant data is left out and relations between program elements are easier determined using the edges.

Traversing the Info Graph can be done automatically using existing classes like GraphWalker, which walks any graph data structure. Because of the existence of different types of nodes and edges for every type of model view, it is possible to specify what types of nodes and edges to process. Traversing can be done depth-first or breadth-first, which is standard. The use of other customized traversals is also possible by creating a subclass of the AbstractListFringeIterator, which is used by the GraphWalker to determine which node to process next.

### 4.2 Architecture Extensions for Java Analysis

Almost all Java analyses are done on the Abstract Syntax Tree (AST) representations of the program. ASTs are constructed by parsing the code found in program source files. An AST captures the essential structure of the program in the form of a tree structure, while unnecessary syntactic details are omitted. This structure makes it easier to perform analysis on, because it is more structured than source code. It is easier to determine the context of a program element or relations with other program elements, if ASTs are used.

To handle Java source code analysis, the basic architecture described earlier is
extended with Java specific classes. These classes are commonly used by most Java analyses. The extension classes consist of classes for: processing Java source files, constructing Java ASTs, traversing Java ASTs and analyzing Java ASTs. These classes standardize the analysis of Java source files.

The JavaSourceContext class is an implementation of the SourceContext interface. It is used to process Java source files, to construct the File Graph initially and to transform it later into a heterogeneous Info Graph by attaching data structures, commonly used to perform Java analysis, to the File Graph. Among these data structures is the Class Path Info Graph. The Class Path Info Graph is a data structure that stores the hierarchy of the classes, the members of every class and the methods of every class. It is also used by the ScopeInformation to determine the scope.

The AbstractObservationVisitor class is an abstract class that implements the ObservationVisitor interface; therefore it is necessary to extend this class before it can be used. Multiple instances of the ObservationVisitor can be used to process every AST received from the JavaSourceContext object. Every AST is processed using at least two objects: a TreeWalker and one or more extensions of the AbstractActionVisitor abstract class, which are described next.

The TreeWalker class traverses the AST and it also keeps track what kind of node is interesting for which instance of AbstractActionVisitor. When such a node is met, it passes the node to the waiting AbstractActionVisitor.

The AbstractActionVisitor class is used to process individual AST nodes de-
derived from the source code file. Like the ObservationVisitor class, it is possible for several instances of AbstractActionVisitor to process the same node at the same time. So it is necessary to tell the TreeWalker which AbstractActionVisitor has to be finished before processing.

A typical Java analysis has to deal with scope, which is the enclosing context in a certain point of the program. Scope resolving is needed to determine the class a method belongs to, which is essential in static analysis. To handle scope, a standard AbstractActionVisitor called ScopeVisitor is used. Like other sub types of AbstractActionVisitor, the ScopeVisitor object also observes the AST while the TreeWalker object is traversing it. Its purpose is to keep track of the scope while traversing the AST.

Scope information is stored in a ScopeInformation object and is available for other objects. Other instances of the AbstractActionVisitor can depend on this object for scope resolving. But before they can do that they have to tell the TreeWalker before the AST is being traversed.

### 4.2.1 Java Info Graph

The Java Info Graph models a Java software system. An example Java Info Graph is illustrated in figure 4.5. It is slightly more complex than the basic Info Graph shown earlier. Typical Java program elements are extracted from the sources and added to the Info Graph.

As visible in the figure, Java package nodes are added below the system node. Links from their corresponding directories are created. The Java types that are defined in source files are also added as nodes, including the links from the corresponding sources. Method and member nodes are also created for every type. Note that the existing hierarchy links between super and child classes are omitted in the illustration.

![Java Info Graph Diagram](image-url)
4.3 Testing

Testing is done during the development process. The development follows a test-driven paradigm. This means that small test cases are written before the actual implementation is done. If code is added or modified, all the tests should run successfully before the code can be checked in the version control system. The small test classes, which are sub types of the \texttt{TestCase} class, are created and stored in a special test directory. The name of these classes should end with “Test” and the defined methods should begin with “test”. The methods should contain test code, which make use of assertion statements, to check other classes. These checks can instantiate certain classes and check their output.

The test classes are automatically instantiated and their methods are run by Daily-Build, which is done at night. Detected assertion failures are reported on a website, which contains detailed information about the whole software. The developers are also notified by email automatically. Another task of Daily-Build is to run the whole program and check for unexpected exceptions, which are also reported on a website and by email.

```java
public class Car {
    int passengers;
    
    public Car() {
        passengers = 0;
    }
    
    public void addPassenger(int noPassengers) {
        passenger += noPassengers;
    }
    
    public int getPassengers() {
        return passengers;
    }
}
```

\textbf{Code Fragment 4.1: Car class}

The example class in listing 4.1 is a simple class created to illustrate how to create a simple test case. The class represents a car and all it can do is add passengers to it and keep track of the number of passengers it contains. To test this class, the following subclass of test case is created:

```java
public class CarTest {
    public void testCar() {
        Car c = new Car();
        for (int i = 0; i < 4; ++i) {
            c.addPassenger(i);
        }
        assertTrue(c.getPassengers() == 4);
    }
}
```

\textbf{Code Fragment 4.2: Car test class}
In its only method, the Car object is created and four passengers are added. If the addPassenger() method and the getPassengers() method are working accordingly, the following assertion statement should succeed. If it fails, it means that the Car class is broken.
Chapter 5

Integer Type Inference

Type inference is defined as deducing either partially or fully the type of the value derived from the evaluation of an expression [41, 21, 45, 24]. Type inference was first introduced in the SIG, when the SIG needed to track bank account numbers in the software of a large bank. This bank needed to extend the length of the bank account number by one digit. The SIG was hired to find out if the software was capable of coping with the extended account numbers. In order to do this, one needed to know where the bank account numbers are used in the software. For this purpose, integer type inference was used.

This analysis can be extended and modified for detecting input validation vulnerabilities. While the original purpose of the analysis is to find all locations where the account number is used, it should be possible to transform the analysis to find all the locations where a value originated from an untrusted method is used as parameter in a critical method call.

The purpose of this chapter is to get a good understanding of the integer type inference analysis, before discussing the design of the new analyzer for detecting security vulnerabilities, which is based on this analysis. The new analyzer is the subject of the next chapter. It is important to map out all limitations of integer type inference when used for detecting security vulnerabilities, before the design phase of the security analyzer can commence. The final design should provide solutions or workarounds for the problems caused by these limitations. The limitations are discussed at the end of this chapter.

5.1 Integer Type Inference

The idea of type inference is to infer relations between same types. The types are modeled as nodes, and when they qualify as the same, a directed edge is created to connect them. The directed edge indicates the direction of inference, which means that if a node has an edge pointing to another node. The latter node is inferred from the first node. Inference rules are used to determine how to infer types. When the inference rules are applied to the source code of a program, the results will be a collection of graphs that denotes how the types relate to each other. This idea was used by SIG to track down back account numbers in the software of a large bank.
The bank account number is represented by the primitive type \texttt{int}. Only integer variables that represent bank account numbers needed to be tracked to determine the locations where they were used. A way to determine or infer that a particular integer variable represented an account number was needed, as it was not explicitly specified in the source.

First, type inference rules are used to determine relations between integer types. Then integers that are known to be bank account numbers are located in the graph. By traversing the graph, other variables that represent bank account numbers can be identified. There are four rules to infer the bank account number integer variable:

1. Assignment rule
2. Method call parameter rule
3. Return rule
4. Comparison rule

The first rule means that if an integer variable, which is known to store an account number, is assigned to another integer variable, this variable also has to be an account number. The second rule means that if a method or constructor is called with a parameter that is qualified as account number, the variable that is used within the method is also an account number. The third rule makes sure that when an account number is returned by a method, the return type of this particular method is qualified as account number. The last rule states that if an account number is used to compare to another integer variable, this variable is also qualified as account number.

The rules explained above are used to create the Fact Graph, which captures these relationships. This Fact Graph is tailored to perform type inference analysis of integer variables.

\section*{5.2 Fact Graph}

The Fact Graph is a graph data structure, which captures the relationships between variables according to the type inference rules defined earlier. Therefore the Fact Graph is a convenient intermediate data structure to perform type inference based analysis as it leaves irrelevant detail out. If two nodes are connected, it means that they are of the same type by inference. The Fact Graph is a separate data structure, which means that it is not an extension of the Info Graph. The reason not to extend the Info Graph with the Fact Graph is because of the size of the Fact Graph for a large Java program, which is quite large.

Nodes are identified by a type and a name. The type is the type of the variable that the node represents, like: \texttt{String} or \texttt{int}. The name is the location where the variable is defined, like: \texttt{mypackage.MyClass.method().s}, for a variable \texttt{s} defined inside \texttt{method()} of \texttt{MyClass} in package \texttt{mypackage}. The list of all important node types related to integer type inference, including their explanations, can be found in appendix B.1.

The Fact Graph is constructed using several classes, each handling a specific part of the Java AST. When a token of interest is detected, the class that handles this
specific type of token determines if and how this should be represented in the Fact Graph. These classes are subclasses of AbstractActionVisitor, which are used by the TreeWalker to traverse the ASTs.

A customized subclass of AbstractObservationVisitor is needed to create and manage the TreeWalker and AbstractActionVisitor classes. The first class is responsible for the traversal of the ASTs, the latter for the analysis of the ASTs. The subclass is used and initiated by the JavaSourceContext object to start the construction of the Fact Graph. The JavaSourceContext class is used for locating and opening the Java source files, which are then handled by the AbstractObservationVisitor class for further processing. The InferenceFacts class is a final class and it is used by the AbstractActionVisitor subclasses to create the nodes and edges in the Fact Graph. InferenceFacts is also responsible for resolving the fully qualified names of identifiers, locating the right nodes and the automatic creation of nodes that are missing.

The following sections describe how the four type inference rules are implemented by the five classes. Each class handles a specific type inference rule, with the exception of the method call parameter rule, which is handled by two classes. Together they are responsible for the construction of the Fact Graph.

5.2.1 Assignment Statements

The AssignmentExtractionVisitor class handles normal assignments and variable definition assignments. It is a subclass of the AbstractActionVisitor and is responsible for the assignment rule, which makes sure that assignment relationships between variables are captured in the Fact Graph. Variable definition assignments are cases when a variable is declared and a value is assigned to it directly. The global control flow of the AssignmentExtractionVisitor class is illustrated in figure 5.1. There are two types of assignments:

1. String str1 = str2;
2. String str1 = str2.toString();

The first case is the assignment of a variable to a variable directly. When such an assignment statement is observed, an ASSIGNED_TO edge is created between the nodes of str1 and str2. The second case is a bit trickier. First all the possible method calls have to be determined using the ScopeInformation object. The methods are found using the method name. The number of parameters is not checked. Then between every method call return node in the Fact Graph and the variable an ASSIGNED_TO edge is created.

5.2.2 Method Call Statements

Method calls are processed by two classes: one for processing the method call identifier and parameters, the other for handling parameters that are used as variables within the method call itself. These classes implement the call parameter rule, which makes it possible to infer the type of method call parameters and the type of variables inside the actual method call body. Method calls can be categorized into four types:
5.2 Fact Graph

1. Constructor call
2. Normal method call
3. Super constructor call
4. New object call

The MethodCallExtractionVisitor class handles all cases when a method call is made. It is a subclass of AbstractActionVisitor class. It is also responsible for distinguishing between the two types of parameters: variables and methods. The global control flow of the MethodCallExtractionVisitor is illustrated in figure 5.2.

For every variable used as parameter of the method call, the fully qualified names and their locations in the Fact Graph are resolved. These locations are connected to the right (external) parameter nodes of the method call with an ASSIGNED_TO edge. For every method call used as parameter, the corresponding call is resolved. The return value node is resolved and connected to the external parameter node of the method call with an ASSIGNED_TO edge.

The ParameterUseExtractionVisitor class handles method parameter definitions. It is a subclass of the AbstractActionVisitor. Every parameter of a method will show up in the Fact Graph as two nodes with a PARAM_USE edge between them. One node is for external use by external objects and one for internal use in the method body itself. The reason for using this construction, is because when a call to this method is detected, the exact variable name of the parameter used in the method call definition is unknown. To avoid the necessity of a parameter lookup service, a keyword that includes a number is introduced to indicate which parameter it is. This special node is called the intermediate parameter. In the body of the method call definition, the variable name is used, therefore an edge is used to connect both method call parameter representatives to indicate their relationship.
The two cases are illustrated in code fragment 5.1. The first method is using the \texttt{param} variable internally, by using the parameter of the method in its computation. The variable name is used to refer to the parameter in this case. While the second method is using the \texttt{param} variable externally, which is calling a method with a parameter. The keyword that includes the parameter number is used in the second case.

```java
class MethodCallExtractionVisitor
{
    public void method_call(String param) {
        // Internal use.
        String tmp = param;
    }

    public void call_method() {
        String param = "parameter";
        // External use.
        method_call(param);
    }
}
```

Code Fragment 5.1: Method parameter use cases

### 5.2.3 Return Statements

The \texttt{ReturnExtractionVisitor} class handles return statements and is responsible for the call return rule. It is a subclass of the \texttt{AbstractActionVisitor} class and is used to infer the type of return variables. Like the assignment statements described earlier, two cases are handled:

1. `return str;`
2. `return str.toString();`
5.3 Fact Graph Example

The first case is returning a variable directly. The second case is returning the return value of a method. For both cases, edges should be created to all locations where the method is called. However, it is not sure how to connect the nodes in the Fact Graph, as it is not sure where this return statement returns to. To handle this, a special variable is introduced, which is of the type AbstractVariable and it represents the return variable of the method call. The return statement is then considered like an assignment statement to this special variable. As an example: return str; is interpreted as: return_var = str;. This return_var is represented by a node in the Fact Graph and edge creation proceeds normally like an assignment statement. The edge is be marked RETURN.

5.2.4 Comparison Statements

The ComparisonExtractionVisitor class handles comparison expressions. It is a subclass of AbstractActionVisitor. This is a special case, which is not a direct propagation of a certain value. Variables that are compared together can have the same value. This relationship is represented in the Fact Graph as a COMPARED_TO edge between nodes.

5.3 Fact Graph Example

Each visitor class discussed above handles a specialized part of the AST. When the visitors are combined to process the source code in code fragment 5.2, they will produce the Fact Graph that corresponds to the source code. The Fact Graph of the code in code fragment 5.2 is depicted in figure 5.3.

```
1 public class SampleClass {
2   public int method(int k) {
3       int l;
4       l = k;
5       int m = 0;
6       int v = 0;
7       if (m == l)
8           v = method(v);
9       return m;
10   }
11 }
```

Code Fragment 5.2: Fact Graph example code (SampleClass)

All node names are prefixed with SampleClass.method(int), which is derived from the current scope. Every node also stores the type name of the node. In this example, all the types are int. The prefix and type name are left out from the figure. It is important to notice how the names of the nodes map to the actual source code. Keywords that start with a $-sign and denotes a special property. Parameter nodes that belong to a method end with $arg0, where the last digit refers to the parameter number. Parameter numbering starts from 0, so in this case it is the first parameter. Nodes that represent return values end with a $return.
5.4 Integer Type Inference Limitations

Integer type inference analysis cannot be directly used to detect input validation vulnerabilities. Important key parts of the analysis have to be rewritten or extended to make it suitable for detecting security vulnerabilities. Limitations of the original analysis have been identified and are listed and discussed below.

1. **No support for objects**: Integer type inference analysis can only track, as the name suggests, only integer types, making it unsuitable for detecting input validation vulnerabilities. To detect those vulnerabilities, support for object types is required. This implies the support of manipulations of objects, such as: calling methods of that object, using that object as parameter of another method call and returning the object as return value.

2. **No support for libraries**: Library types are not supported and thus ignored during analysis. This is usually not a problem for the purpose that the analysis originally was designed for. However, for detecting input validation vulnerabilities it is necessary to have support for library types, because input values can be used with library types. Examples: storing values in a container class like `HashMap`, or passing values as parameters to library methods. This value can be returned at some point and used again. Therefore tracking of values through libraries is required.

   Subtyping and polymorphism of library types are also not handled. Support of subtyping and polymorphism of library types is needed to determine the exact method call of an object. Methods of interest could be a library method inherited by non-library types, therefore it is necessary to determine the (library) superclasses and their methods.

3. **Limited type resolving**: The original type resolver can only resolve the types defined in the analyzed source code. This function is too limited and makes it unusable to specify methods accurately. An accurate type resolving mecha-
5.4 Integer Type Inference Limitations

A mechanism that can also handle libraries is necessary in order to detect security related methods.

4. **Unhandled expressions:** The `int` variables analyzed in the original analysis have no meaning in normal arithmetic operations like adding or subtracting, because performing these operations on a bank account number would make no sense. Therefore expressions used as the right-hand side of an assignment, a parameter of a method call or return statement are checked if they consist only of one `int` variable only. If not, they are discarded. For security analysis purposes, this is not sufficient. For example, string values can be concatenated in an expression, before being returned. Clearly, expressions should be split in order to determine the separate elements, which have been tracked.

5. **No inference history/paths:** The original analysis only prints out the locations of where the inferred types are used. For the new analysis, it is necessary to determine where it originates, in order to determine situations where data that depend on an untrusted method being used in a critical method.
Chapter 6

Object Oriented Type Inference

The previous chapter made clear that the original Fact Graph is designed to support integer type inference, making it not directly suitable for detecting input validation vulnerabilities as discussed earlier. A number of changes and extensions are needed to make it suitable for detecting input validation vulnerabilities. This chapter describes in detail the changes made to the integer type inference analysis. The modified analysis supports object oriented type inference and is responsible for the construction of the Security Fact Graph. This graph captures object oriented type inference relationships according to the type inference rules of objects. The Security Fact Graph does not have the limitations of integer type inference that are described in chapter 5 and can therefore be used to detect input validation vulnerabilities.

6.1 Object Oriented Type Inference

To detect input validation vulnerabilities, it is necessary to capture data dependencies between objects to determine the data flow, and not determining types that are probably the same. Determining variables that depend on other variables by inference is the key here. For example, the simplest form of data dependency is the assignment statement. This can be seen as copying the content of one memory location to another memory location, which means that the content of one variable is also found in another variable. The Security Fact Graph is created to capture these data dependencies, so it is possible to infer the origin or destinations of data elements found in a program. This means by analyzing the graph, it is possible to detect cases when data originated from untrusted methods is used in a critical method. When such a case is detected, a potential security vulnerability is found.

Support for only integer variables is not sufficient. Data that originates from an untrusted method are normally not integers, but various kinds of objects including String variables, like explained in the last part of the previous chapter. Rules are needed to infer data dependencies between object types. These objects should also be represented in the Security Fact Graph for the detection of input validation vulnerabilities.

Object oriented type inference has to deal with various kinds of types or classes. For some of these, the corresponding source files are available, some are in libraries, while others are unresolvable due to missing source files or libraries. The latter implies
that the software system to be analyzed is not compilable. For clarity, the definitions about type resolvability that are listed below.

**Definition 3 (Local Type or Method)** A type or method is local, if the source file of the type or method is included in the analysis.

**Definition 4 (External Type or Method)** A type or method is external, if no source file of the type or method is available, but the type or method can be found in a library that is included in the analysis.

**Definition 5 (Unresolvable Type or Method)** A type or method is unresolvable, if no source file nor library can be found where the type or method is defined. The software system is therefore not compilable.

Like integer type inference, object oriented type inference is based on inference rules to infer type properties. These rules are basically the same, although the comparison rule is omitted. The inference rules differ in application. The existence of objects and their properties like polymorphism or subtyping complicate the inference. The last rule of integer type inference concerning comparison is omitted, because objects are not generally compared with the "=="-operator, but with the `equals()` method, which is already handled as a method. The "=="-operator compares references when used with objects, which implies that both objects could originate from the same location as they could be the same. This means indirectly that the relationship already exists, which makes the comparison rule redundant. The new list of inference rules:

1. **Assignment rule**
2. **Method call parameter rule**
3. **Return rule**

The first rule means that when an object is assigned to another object variable, the content of this object variable is dependent on the first object as they are the same. This can also be seen as data flowing from one variable to another variable, although the data is not really flowing, only the references are passed on. The method call parameter rule captures data dependency relationships between method call parameters and variables in the method body, which can be seen as the call parameters data flowing to the variables found in the method body. The last rule captures the fact that the data in the method body can flow back to the location where the method is called.

Before it is possible to extend the integer type inference tool for object oriented type inference, it is necessary to identify the usage of objects. The list of how object can be used in a Java program is as follows.

1. **Variable assignment**: `object1 = object2`
2. **Constructor call**: `this()`
3. **Super constructor call**: `super()`
4. **New object call**: `new SampleClass()`
5. New object method call: `new SampleClass().method()`

6. Static method call: `System.out.println()`

7. Object method call: `object.method()`

8. Object super method call: `object.superMethod()`


10. Super method call: `superMethod()`

Variables, methods, parameters and method return values are represented by nodes. These elements have to be linked together with directed edges when they depend on each other. Connecting these elements is not as straightforward as it seems. Special cases occur when objects and external inheritance are considered. These situations will be described in detail in the Security Fact Graph example.

When a method of an object is called, it is possible that the data containing in the object depends on the parameter. An example is a method call on a container class to add another object, such as adding an object to a `Vector` with the method call `add(anObject)`. Clearly, the data within the container class depends directly on the method call parameter. Therefore an edge from the parameter object to the method and a bidirectional edge between the method and the object that owns it, are needed to capture this relationship. The object might be a subtype of an external type and the inherited method call is defined in the super type, which might be an untrusted or critical method. This is the reason why the exact method call definition has to be located to determine if that is the case. In the same way, an edge should exist between parameters of a new object call and the new object.

### 6.1.1 Splitting Expressions

In order to track the individual expression elements, expressions have to be split into individual parts. The class `ExpressionSplitter` is responsible for splitting expressions. It uses predefined operators, listed in appendix C, as separators to determine individual expression elements, which are also listed below and in the appendix. These elements are interesting, because they can contain or return dangerous values. Other expression elements are ignored.

1. Normal method calls: `method(i, j)`

2. Constructor calls: `this()`

3. Super constructor calls: `super()`

4. Identifiers: `anObject`

The operators in the appendix are chosen, because the values of their operands or result can be found in the evaluated result of the expression. Take as example, the expression `string1 + string2`, where two strings are concatenated. It is easy to see that the string values of both strings can be found intact in the evaluation result.
Special handling is needed for cases where values of expression elements do not show up in the evaluation result. This is the case when the following expression is considered: `method() ? 1 : 2`. The return value of the method call does not get copied into the result of the expression in any way, so splitting this expression up will yield an empty set.

Another special case is the typecast, where the variable or method typecasted is extracted from the typecast. The constant integer values will not be found in the set, because constants are ignored. Constants are not considered to be dangerous, because they cannot be tampered by user input.

### 6.1.2 Classes and Objects

An object class is handled in two ways in object oriented type inference. One is the inner code that defines the class and consists of all code found in the class body. The inner code performs calculation and manipulates objects.

An example of this example can be found in code fragment 6.1. This includes class members, constructors and method definitions. The other way of handling object classes is the object itself, which is manipulated by other code, as in code fragment 6.2. Only the last situation applies in handling the `int` type in integer type inference, because it has no inner code.

The special property of objects makes the object oriented type inference more difficult to handle compared to integer type inference, because the analyzer has to take into account that program code belongs to a class. This class can be instantiated to an object, which is then manipulated by other classes.

This difference plays an important role in object oriented type inference and has consequences in creating the Security Fact Graph, which has to model both objects and classes. These consequences are discussed later, when the implementation of the inference rules are discussed.

```java
public class TestClass {
    public TestClass(Object o) {
        int i;
        i = o.getInt();
        ++i;
        o.setInt(i);
    }
    // Other methods are not displayed.
}

public class TestClass {
    public TestClass(Object o) {
        int i;
        i = o.getInt();
        ++i;
        o.setInt(i);
    }
    // Other methods are not displayed.
}
```

**Code Fragment 6.1: Object inner code**

```java
TestClass testClass = new TestClass(o);
testClass.method1(i);
TestClass anotherTestClass = testClass;
int j = anotherTestClass.method2();
```

**Code Fragment 6.2: Object manipulation**
6.1.3 Type Resolving and Libraries

The original type resolver is not capable of resolving all external types. To have an accurate analysis, it is required to determine the correct fully qualified name of types and methods. The fully qualified name is used to determine if it concerns an untrusted or critical method when a method call is made. To be able to determine the correct fully qualified name, it is necessary to extract the name out of libraries, which consist of Java class files or jars.

The other missing part is related to subtyping of library classes. Method calls that are not locally defined, but inherited from a library class or defined in a library interface, have to be resolved to their fully qualified name also. For the same reason, this method call might be an untrusted or critical method.

The need for a library that can access and extract information out of jars and class files is evident. The Byte Code Engineering Library (BCEL) [1], which is part of the Apache Jakarta Project [6] is used for this purpose. Using BCEL, developers can analyze, create and manipulate Java class files. The library is mostly used for projects such as compilers, optimizers, obfuscators, code generators and other analysis tools. Object oriented type inference uses BCEL to compose the fully qualified name of a type and to determine where an inherited method is implemented.

A wrapper class called LibraryInformation is created to provide an interface to the functionality of BCEL. The LibraryInformation class is also responsible for extracting jars, which BCEL is uncapable of. It also provides storing and caching of library information. Retrieving information of a type can be parameterized with a list of import declarations, similar to those typically found at the beginning of a Java source file. The use of this list by the LibraryInformation class helps to solve ambiguity that occurs when two types have the same name. The import declaration list is used to locate the correct information of a type.

Type resolving and determining where a method is defined are the responsibility of the new class called TypeInformation. When given a simple type name, with an optional list of import declarations, it tries to determine the fully qualified name locally with the help of the InheritanceInformation class, which keeps track of local inheritance of types. When unable, it delegates this task to LibraryInformation, which tries to find the fully qualified name in the libraries.

As an example, consider resolving a type Vector with the new TypeInformation class. When the class Vector is local, it will resolve to mypackagename.Vector, where mypackagename is the name of the package. The second case is when the Vector class is external, which will resolve to java.util.Vector, where java.util is the package name. The package java.util is found in a library. The last case is when Vector is not found in the sources, nor in the libraries. The fully qualified name cannot be constructed and is then prefixed with a keyword: \texttt{$\$unresolvable$.Vector}. The \texttt{$\$unresolvable$} keyword marks the type as unresolvable.

6.2 Security Fact Graph

The Security Fact Graph captures data dependency relationships between program elements. This makes it possible to infer where data is coming from or going to. Similar to the integer type inference, the construction of the Security Fact Graph is the task
of the AbstractActionVisitor subclasses. These new classes are the extended versions of the ones used in integer type inference.

The class InferenceFacts, which is responsible for creating nodes and edges of the Fact Graph, is replaced by the newer class called SecurityRelations. The SecurityRelations class is capable of creating all additional types of nodes required by object oriented type inference. New keywords that are used in the name of nodes are introduced. Notably are the ones that denote external or unresolvable types and methods. The complete list of all important new node types, including their explanations, can be found in appendix B.2.

The purpose of the Security Fact Graph is to use the captured data dependency relationships to detect paths from untrusted methods to critical methods, which are identified beforehand. These paths represent data flow paths, taken by dangerous input originated from an untrusted input method, to a critical method. How to determine these paths is the subject of chapter 7.

The extended visitor classes that are separately described later on, are fully backwards compatible if the boolean variable useObjectRules is set to false. This is the case when only integer inference is desired. Type resolving is delegated to the newer TypeInformation class, instead of the class ScopeInformation that cannot handle library types. The ScopeInformation class is still used to track the current scope in the source code. Nodes and edges are now created by the class SecurityRelations, which replaces the class InferenceFacts. The SecurityRelations class supports external and unresolvable objects and the creation of node that are used to construct the Security Fact Graph.

From the object use cases from the list on page 32, the unsupported cases are identified, which are listed next. The support for these cases is required to handle objects and library types. The support is implemented in all the AbstractActionVisitor subclasses that are described later on.

- **New object call**: `new SampleClass()`
- **New object method call**: `new SampleClass().method()`
- **Static external method call**: `System.out.println()`
- **External object method call**: `externalObject.method()`
- **External super method call**: `externalObject.superMethod()`
- **Local object external super method call**: `localObject.externalMethod()`

### 6.2.1 Assignment Statements

The extended AssignmentExtractionVisitor class is illustrated in figure 6.1. First, it is determined if the assignment statement is a normal assignment statement or a declaration assignment statement. The latter case is when a declared variable is assigned a value directly. The next step is to extract the individual expression elements from the right-hand side of the assignment statement. Each element is handled separately. This element might be a variable or a method call. If a method call is the case, it determines the the type of method call. This might be one of the newly supported method calls.
6.2 Security Fact Graph

Figure 6.1: Flow diagram new AssignmentExtractionVisitor

from the list above or a normal method call. The last step is to create the nodes and the ASSIGNED_TO edges to capture the relationships.

6.2.2 Method Call Statements

The extended MethodCallExtractionVisitor class is the most complex class of all visitors. This complexity is reflected in figure 6.2. This class is relatively complex, because it has to determine what kind of method is actually called, before it can process every expression element found in every parameter of the particular method call. Again, if a parameter contains a method, the class has to determine what type of method call it is, before it can be processed.

The first action the MethodCallExtractionVisitor class takes, is to determine what kind of method call is actually called. All normal methods and the newly supported method types listed earlier, are handled. The next step is to extract the expression elements from each parameter for further handling. For each expression element found in the parameter, the type of parameter is determined. The element might be a variable or a type of newly supported method call. It is the responsibility of this class
6.2 Security Fact Graph  
Object Oriented Type Inference

Figure 6.2: Flow diagram new MethodCallExtractionVisitor

to link the expression elements found in the parameters to the method that is called. It does so also using an `ASSIGNED_TO` edge.

The older related `ParameterUseExtractionVisitor` class that handles method call parameters needs a minor update. The older class does not resolve external parameter types correctly. The class uses the simple name to create the node, while the fully qualified name is required. The extended version is able to resolve external types to their fully qualified name, which is necessary as types are identified by their fully qualified names in object oriented analysis.

Another update is needed to both classes that are responsible for handling method calls. The two edges that link the parameter variable of a method call, the parameter intermediate and the parameter known in the method body are unidirectional. Object oriented type inference requires bidirectional edges, because the parameter variable used could be changed within the method body.
6.2.3 Return Statements

The extended ReturnExtractionVisitor class, which is illustrated in figure 6.3, is the smallest class of the three visitors. This is because the return node is a fixed node for a given scope, and generating the name of the node is straightforward.

Just like the AbstractActionVisitor subclasses discussed before, this class also handles all the new types of method calls correctly. The RETURN edge is used to represent this relationship.

6.3 Security Fact Graph Example

To get a clear understanding of how the AbstractActionVisitor subclasses work together to create the Security Fact Graph, two simple classes are specially created in order to demonstrate the important key functionalities. The classes have no real useful purpose and are meant for demonstration only. The source code of the classes can be found as code fragments 6.3 and 6.4.

In order to show the key functionality of the analysis, the use of external and unresolvable elements are included in the example classes. The classes make use of two classes, which can be found in the standard java.util library. The external classes that are used are the classes Vector and Stack. The library is included in the analysis, in order to show how the library is used to get more accurate results. The
methods Untrusted.Method() and Critical.Method() are unresolvable and are used to demonstrate how unresolvable methods are handled.

The construction of the Security Fact Graph is explained step by step to highlight important details about the workings of the AbstractActionVisitor subclasses.

```java
import java.util.*;

public class NewSampleClass {
    public void method(int s) {
        Vector v = new Vector(s);
        v.add(Untrusted.Method());
        Object o = v;
        NewStack newStack = new NewStack();
        newStack.push(o);
        Critical.Method(newStack.pop());
    }
}
```

Code Fragment 6.3: Security Fact Graph example code (NewSampleClass)

```java
import java.util.*;

public class NewStack extends Stack {
    public NewStack() {
        super();
    }
}
```

Code Fragment 6.4: Security Fact Graph example code (NewStack)

The resulting Security Fact Graph can be found as figure 6.10. As before, the prefix NewSampleClass.method(int). of all nodes is removed. This example illustrates the following:

1. Using local types that are subtypes of an external type: NewStack.

The definition of the method method() in NewSampleClass is processed by the ParameterUseExtractionVisitor class. The result is two connected nodes, which represent the externally known parameter and the internally known parameter as seen in figure 6.4.

The method starts with the creation of a new java.util.Vector instance, with variable s as parameter. The instance is referenced by variable v. This statement is handled by two classes. The MethodCallExtractionVisitor class, that handles the method call, and the AssignmentExtractionVisitor class that handles the assignment. The first class connects the parameter s and the instance, represented by the NewSampleClass.method(int).$object.java.util.Vector node. The second class ensures that an edge is created between the instance and the variable v.
$object$ keyword indicates that this node is an instance of the java.util.Vector class, which name is appended in fully qualified form. This result is visible in figure 6.5. The newly added nodes are slightly darker.

Vector v = new Vector(s);

The second statement is using static method Untrusted.Method() as parameter of a method call that belongs to variable v. The static method is unresolvable, so the $unresolvable$ keyword is used together with the $method$ keyword. The last keyword means that the node is an external method call, while the first one means that the method call is unresolvable. The static method call has to be linked to the node that represents the method call of an external type v variable. The name of this node is constructed using the name of the variable and the fully qualified name of the method call, separated by the $method$ keyword. The fully qualified method name shows which call is actually called, and makes the method easier to identify, when it is an untrusted or external method call. This node is linked to the variable node, because the call might influence the object it belongs to by the parameters.

The variable assignment on line 7 is handled normally like the integer type inference. The right-hand variable v is linked to the left-hand variable o.

The statement that follows is making an instance of a local class, which inherits methods of an external class. The same as instantiating an external class, an $object$ node is created that links to the variable. In this situation no parameters are given, therefore there are no incoming edges to the $object$ node. The two nodes are not connected to the previous nodes.

The next statement is calling an externally inherited method of a local variable. This node is identified with the $extsupercall$ keyword and the fully qualified name
of the method. This node is bi-directional connected to the variable node, because the method call can influence the data in the variable and the data in the variable can influence the return value of the method call. The o variable is the parameter of the call, therefore it has to be connected to the method call. Please note, that the resulting figure 6.9 is reordered a bit.

The externally inherited pop() method call is used as parameter of an unresolvable static method call in the last statement. The nodes that represent them are connected to capture this relationship. The final Security Fact Graph is illustrated in figure 6.10.
6.4 Object Oriented Type Inference Limitations

Object oriented type inference is an approximation of the real program, therefore it has limitations like almost all static analyses. The limitations are briefly discussed in this section. Example code fragments are included in several cases for clarity.

1. **Limited differentiation of class instances**: The Security Fact Graph does not model different instances of the same class within the same scope. In code fragment 6.5, the Security Fact Graph does not differentiate the two different class instances referenced by the variables `one` and `two`. The class instance in a particular scope is modeled as one node in the Security Fact Graph. However, it does differentiate the instances of the variable `zero` and the other two. This is because the scope is used in the identification of class instances.

As a future work, this problem can be solved by using a counter or symbol table to track different class instances. The new instance method call has to be
6.4 Object Oriented Type Inference Limitations

Object Oriented Type Inference

Figure 6.10: Security Fact Graph complete

carefully tracked to avoid creating edges to the wrong nodes. This means it is required to differentiate both new AbObject() method call by tracking the file positions.

1 AnObject zero = new AnObject();
2 public void method() {
3     AnObject one = new AnObject();
4     AnObject two = new AnObject();
5 }

Code Fragment 6.5: Class instances

2. **Generalization of static method calls**: Static method calls are generalized to normal method calls and handled that way. The static method call in code fragment 6.6 is handled like a normal method call on a local object X, which means that the assumption is made that the node Y.method(int i).object.X should exist. Therefore the (missing) node is created when the static method call is handled. When this ghostly local object is tainted, it means that the instance
that owns the method is tainted. The taint can possibly propagate further in the Security Fact Graph, when another static method that also belongs to the same class is called. This behaviour does not represent the real system correctly, because static methods do not belong to a class instance that can store data.

```java
public class X {
    public static void method(int s) {
        ...
    }
}

class Y {
    public void method(int i) {
        X. method(i);
    }
}
```

Code Fragment 6.6: Static method calls

3. **Incorrect class members scope**: The Java Security Analysis does not determine the exact location of the object being called. It assumes the object is defined in the current scope. The method call in `method2()` on member object `x` in code fragment 6.7 therefore results in one faulty node. The incorrect node is called `X.method().$object.X`, which has to be `X.$object.X` to be correct.

```java
public class X {
    X x = new X();
    public void method(int y) {
        x. method2(y);
    }
    public void method2(int i) {
        ...
    }
}
```

Code Fragment 6.7: Class members

4. **No support for arrays**: There is no support for arrays. Arrays are ignored in the Java Security Analysis. One way to create support for arrays is to treat arrays as one object where all individual objects it contains maps to. For example, `variable[1]` and `variable[$i]` map to the same node if they are in the same current scope.

5. **No support for complex chained method calls**: Long and complex chained method calls such as `var.method().method().method()` are unsupported. As such method calls are not commonly found in software, creating full support does not justify the costs of development. Found cases are written to a log file in the current Java Security Analysis.
6. **Generalized external methods:** External methods with the same name that differ in parameters are mapped to the same node, which means there is no difference between the method calls `Vector.addAll(Collection c)` and, practically the same, `Vector.addAll(int index, Collection c)`. In this case it does not matter much, but in other cases this might result in an inaccurate analysis with inaccurate results as a consequence.

7. **Multiple data flow paths:** Most of the control flow information is lost in Java Security Analysis, which has its influence on the result when there are loops or shortcuts in the Security Fact Graph. The example in code fragment 6.8 has one data flow path during execution, which contains a loop. The loop is also modeled in the Security Fact Graph, which results in multiple paths in the graph that are listed below. Although one of the paths is not correct, it provides the main data flow and it does not affect the goal of detecting input validation vulnerabilities.

```java
1  j = Untrusted.Method();
2  j = method(j);
3  Critical.Method(j);
```

**Code Fragment 6.8: Data path with loop**

8. **Incorrect handling of new object method call:** Method calls that are from the same construct as the `new TestClass().method()` call are not handled correctly for local classes. Cases of local classes are in this situation handled the same way as external cases, which means that the keyword `$arg0` is not automatically generated to be used in the name of the node. This is caused by the underlying framework, which cannot be modified in this project. Modification of the underlying framework is needed to handle the local case correctly according to the standard.

9. **Generalized data flow within library classes:** data flow in library classes are generalized. When data is used as parameter of a method of a class, the assumption is made that the data influences the class and all return values of the methods it owns. The example in code fragment 6.9 illustrates this clearly.
Object Oriented Type Inference  
6.4 Object Oriented Type Inference Limitations

1  String s = Untrusted.Method();
2  Vector v = new Vector();
3  v.add(s);
4  String notInHere = "Capacity: " + v.capacity();

Code Fragment 6.9: Approximation library data flow

The variable notInHere is according to object oriented type inference dependent on the Untrusted.Method() method in the example. In reality, this is not true. The v.capacity() method returns the capacity of the object, which is not affected by the Untrusted.Method() method. Object oriented type inference is unable to determine this, because it does not know the data flow within the library classes. Therefore, it conservatively assumes that all method calls of library classes are dependent on all the parameters of methods owned by the library classes.
Chapter 7

Taint Analysis

The Security Fact Graph captures all data dependency relations between relevant program elements, which makes it possible to determine which variables are dependent on a certain variable. The next phase is to detect paths from untrusted methods to critical methods, which represent data flow between the two methods. Such a path can be a potential security vulnerability in the software. The analysis method suitable for this task, which is described in this chapter, is called taint analysis.

Furthermore, the implementation of taint analysis for this project is also presented. The implementation of taint analysis consists of three subclasses of a general graph visitor AbstractLinkVisitor. The Tainter class is responsible for creating and propagating taints. The Stripper class is responsible for removing redundant nodes to keep the size of the Security Fact Graph manageable. The TaintedPathFinder class is used to find tainted paths from untrusted methods to critical methods. These three classes all work on the Security Fact Graph.

7.1 Taint Analysis

Taint analysis is inspired by the Perl programming language [46], which can be run in taint mode. When Perl scripts are run in this mode, input derived from outside the program is marked as tainted and tracked through the whole program. Tainted input has to be untainted before it can be used as input to another program or system call for the same security reasons discussed in this paper. Data that are derived from tainted data are also tainted. This taint propagation idea is also applied to detect security vulnerabilities in other projects [40, 27, 39, 51, 50, 47, 33], both in static and in dynamic analysis.

In this project, the same taint propagation principle is used during the analysis of the Security Fact Graph. The objective is to detect input validation vulnerabilities in the program. The taint starts at the untrusted method nodes and propagates further to other nodes through the directed edges. The tainting process shows all the nodes that are dependent on the untrusted method. The goal is to detect cases where the taint reaches critical method nodes, since they represent potential dangerous data flow paths. Therefore it is important to find such paths to detect potential security vulnerabilities in the real program.
7.2 Tainter

The Tainter class is created to handle the creation and propagation of taint to other nodes in the Security Fact Graph. The class relies on the SecurityMethods class, which contains all specified untrusted and critical methods. The Tainter class can use the SecurityMethods class to determine if a node represents an untrusted method or a critical method. If an untrusted method node is detected, it is tainted. Nodes are also tainted, if they are directly connected to other tainted nodes with a directed edge. The tainting process proceeds in iterations. When a node is tainted, an additional iteration is needed to propagate the taint to other nodes. If nothing happens in an iteration, the tainting process is finished.

Tainting nodes happens by creating an instance of the Taint class, which is stored in the node to indicate that it is tainted. Besides serving as a marker that indicates that a node is tainted, this object has also an information storage function. The Taint object stores additional information about the taint, like the origin of the taint. This information is needed to find the paths the taint takes, which represent potential insecure data flow paths.

The SecurityMethods object is created at the start of the analysis. Untrusted and critical methods that are specified in the configuration file, are stored in this object. The SecurityMethods object automatically resolves equivalent parent and child methods of these security methods, which are also stored inside. The resolving is done by the TypeInformation class. Handling local subtyping, which means that local types are subtypes of other local types, is delegated to the InheritanceInformation class. Handling library subtyping, which means that local types are subtypes of library types, is delegated to the LibraryInformation class. Together, these three classes are responsible for the resolving of any supertype or subtype that is needed by the analysis.

7.2.1 Preventing Invalid Taint Paths

Unfortunately, not all paths in the Security Fact Graph are valid data flow paths in the real program. This is because the goal of the Security Fact Graph is to captures data dependency relations, which does not take into account the different instances of a class in different locations of the program. To understand this problem, consider the following classes in the code fragments 7.1, 7.2 and 7.3.

```java
public class Container {
    private String value = null;

    public void set(String v) {
        value = v;
    }

    public String get() {
        return value;
    }
}
```

Code Fragment 7.1: Container class
public class A {
    public void method() {
        Container c = new Container();
        c.set(Untrusted.Method());
    }
}

Code Fragment 7.2: A class

public class B {
    public void method() {
        Container c = new Container();
        Critical.Method(c.get());
    }
}

Code Fragment 7.3: B class

Although the code contains an untrusted and a critical method, there is no dangerous data flow path. The parameter of the Critical.Method() method can never be originated from the Untrusted.Method() method. The corresponding Security Fact Graph of these classes, which is tainted without restrictions, is illustrated in figure 7.1. Irrelevant nodes are omitted. The taint that originates from the Untrusted.Method() node, reaches the Critical.Method() node. This is caused by the fact that different instances of a class are mapped on the same set of nodes that represent the class in the Security Fact Graph. The two instances of the Container class are mapped on the Container prefixed nodes in the Security Fact Graph.

Figure 7.1: Security Fact Graph of classes A, B and Container

Invalid paths can be detected using the prefix of the node names. The prefix is basically the scope of the program element, which can be a method call or variable. The scope changes only when entering a local method call or it is returned from it.
Take for example the \texttt{c.set(\texttt{Untrusted.Method()})} statement in the class \texttt{A}, which can be found in code fragment 7.3. The scope of parameter \texttt{Untrusted.Method()} is \texttt{A.method()}, which means in \texttt{method()} of class \texttt{A}. When the \texttt{c.set()} is entered, the scope prefix is changed to \texttt{Container.set(java.lang.String)}. This is consistent in the Security Fact Graph, where the parameter prefix is \texttt{A.method()} and the prefix of the connected method call node is \texttt{Container.set(java.lang.String)}. The scope or prefix change is reversed, when returning from the method.

This property can be used to check paths for validity. A thorough example can be found at the end of this section. The check starts from the origin of the tainted path in the Security Fact Graph. When a method call is made, the current scope is remembered and traversal continues normally. When a return node is traversed, checks are needed to determine if further traversal is allowed. Further traversal is only permitted to nodes with the remembered scope. This is because a method call can only return to the same point where it is called. When returned, the scope is immediately forgotten. Due to the possibility of method calls in source code, a stack data structure is needed to remember the scope of the locations where every local method call is made.

A special situation exists when a return node is traversed and there is no scope remembered (stack empty), which means that another class is calling the method where the taint is started. Traversal is in this case unrestricted, as the taint can propagate to every caller. This example can be found in code fragments 7.4 and 7.5.

```java
public class Caller {
    public String method() {
        Callee callee = new Callee();
        return callee.method();
    }
}
```

Code Fragment 7.4: Caller class

```java
public class Callee {
    public String method() {
        return Untrusted.Method;
    }
}
```

Code Fragment 7.5: Callee class

Clearly, reporting invalid paths is not desired and needed. Two approaches exist to implement the idea above. One approach is to propagate the taint normally. After the taint propagation phase, all paths are retrieved and invalid paths are filtered out. The second approach is to restrict taint propagation during the tainting process. Invalid taint propagation to neighbor nodes that are directly connected, is not allowed. The final paths found are valid data flow paths.

The first approach can be implemented by checking the paths using a stack to keep track of scopes. Only one stack is needed for checking every path. The second approach stores the stack in the nodes itself within the \texttt{Taint} object. The taint propagation starts by retrieving the \texttt{Taint} object. The stack is used to track current
scope and to check if propagation to a neighbor node is allowed. If taint propagation is allowed, the Taint object that include the stack gets copied to the neighbor node.

The second approach is selected for implementation in this project. The main advantage of this approach is that useless data gets filtered out early, which prevents wasting resources on processing and analyzing them. It also saves memory space and prevents having an additional separate phase, in which invalid taint paths are filtered out.

As an example, the second approach is applied to the Security Fact Graph on figure 7.1. The taint starts at an untrusted method node, which is in this case the A.method().$unresolvable.$method.Untrusted.Method() node. The first taint propagation to the only neighbor node results in a scope transition from A.method() to Container.set(java.lang.String), which is allowed because it is a local method call. A local method call can be recognized by the $arg0 tail of the node name. The Taint object is copied to Container.set(java.lang.String).$arg0 and the scope A.method() is saved on the stack of the newly copied Taint object to remember the location of the last local method call. The next taint propagations occur normally, because the scope prefix stays the same. The last transition involves a return node, so it is necessary to check if the destination scope is the same as the last scope of the last local method call, which is saved on top of the stack. In this case the last remembered scope is A.method(), which is not the same as the destination scope B.method(). This means that the taint propagation is not allowed and stops at the return node. The taint in the example Security Fact Graph does not reach a critical method, therefore there is no security vulnerability in the source code.

7.3 Stripper

The complete Security Fact Graph is too large to handle. This is the reason why the Stripper class is in certain conditions used to strip the Security Fact Graph from redundant nodes. Only nodes that are relevant for the analysis are left in the Security Fact Graph. Stripping the Security Fact Graph can be switched off in the XML configuration file.

The Stripper class works also in iterations besides the Tainter class. When the Tainter class is tainting nodes, the Stripper class removes nodes that satisfy the conditions explained below. These nodes are not needed for the analysis. After the tainting process is finished, the Stripper class continues removing nodes until no unnecessary nodes can be removed. The result is a compact graph, which is totally tainted.

The stripping process can be separated into two phases. The first phase coincides with the tainting process. The second phase begins when the tainting process is finished and stops when no nodes are deleted in an iteration. The extra phase is needed, because some nodes can only be determined to be deleted after the tainting process is finished. Nodes that satisfy one of the following conditions are deleted.

- **The node is an untrusted method node with no outgoing edges and not a critical method with ingoing edges.** An untrusted method, which is not a critical method, is only interesting if it has outgoing edges. It cannot propagate taints to other nodes if the node has outgoing edges. The only situation this node is
part of a tainted path is when the node is also a critical method with ingoing edges.

- The **node is a critical method node with no ingoing edges and no not a untrusted method with outgoing edges.** The reverse of the condition above applies to this case. The only situation a critical method node is part of a tainted path is when it has ingoing edges or it is an untrusted method node with outgoing edges.

- The **node has no in or outgoing edges.** The node has no edges, therefore it cannot be part of a tainted path and is allowed to be deleted. This node does not have to be an untrusted nor a critical method.

- The **node has only one edge that is pointing to itself and the node is neither an untrusted method nor a critical method.** This node cannot be part of a tainted path, because it has only one edge that is pointing to itself. One special condition exists, which is when the node is both an untrusted method and critical method. This node is in itself a dangerous tainted path.

- **In the second phase: the node is not tainted.** After the tainting process is finished this rule applies. When a node is not tainted, it is deleted, because only tainted nodes can be part of a tainted path.

Other nodes may be part of a dangerous tainted path, therefore they cannot be removed without losing results. The stripped graph is a relatively small graph containing only tainted nodes, which makes it easier for further analysis, storage or visualization if necessary.
To illustrate the workings of the Stripper class, the Security Fact Graph in figure 7.2 is stripped using the criteria above. In the first iteration of the first phase, node 1, 4, 5 and 6 are deleted immediately, which results in two connected graphs. Then node 3 gets deleted in the next iteration. The last node deleted in phase 1 is node 2. At the start of phase 2, only the looped graph consisting of node 7, 8 and 9 is left, which are deleted in the first iteration of phase 2, because those nodes are not tainted. All nodes are deleted in the Security Fact Graph of this example.

### 7.4 TaintedPathFinder

This phase takes the tainted Security Fact Graph as input, which is preferably stripped. The goal is to find tainted paths that may be input validation vulnerabilities by analyzing the Security Fact Graph. A tainted path is considered dangerous if it has a critical method as end point. The class that is responsible for this task is called the TaintedPathFinder. This class is used to traverse the nodes in the Security Fact Graph and in order to find the tainted paths.

It performs the following steps to find the tainted path. The pseudo code of this algorithm can be found in code fragment 7.6. First locate every tainted critical method node. The node can be tainted by multiple taints, which means that data from different untrusted methods reaches the node. Tainted critical method nodes are sure to be part of one or more tainted paths, or else they are not tainted. Second, backtrack for every taint using the origin node name, which is stored in the Taint object within the node. It is possible to have multiple neighbor source nodes for the taint in a node, which means that the taint reaches this node through different paths. Backtrack through both source nodes and save the paths in a list.

```python
FUNCTION find all tainted paths
FOR every tainted critical method node
DO
  FOR every unique taint in node
  DO
    FOR every node connected by incoming edge
    DO
      IF node has same taint
      DO
        BACKTRACK to taint origin
        SAVE path
      DONE
    DONE
  DONE
DONE
DONE
```

Code Fragment 7.6: Pseudo code tainted path finding

### 7.4.1 Loop Handling

Backtracking by just following the origin stored in every taint is problematic if the path contains a loop like the example Security Fact Graph in figure 7.3.
The backtrack process starts with the critical method node, which is numbered 6 in the figure. The taint is also found in node 3, so this node is the source of the taint. The same routine is applied and the backtrack process determines that node 2 is the source. Node 2 has two ingoing edges from nodes with the same taint, which means there are at least two ways to taint node 2. Node 4 is tainted because the taint process starts from node 1 and follows the directed edges to taint other nodes. One path ends with node 1 and the other continues backtracking with node 4, which reaches node 2 for the second time. The backtrack process continues endlessly in the loop formed by node 2, 3 and 4.

A way is needed to detect loops and to prevent backtracking through the loops. A loop is detecting by checking if the node is already in the path obtained so far. Is so, the node is skipped. The result is a path without loops. The Security Fact Graph in the example contains the path: 1 - 2 - 3 - 5, which may not entirely correspond to the real data flow path. The reason why is already explained on page 46.
Chapter 8

Java Security Analysis

Object oriented type inference (chapter 6) and taint analysis (chapter 7) are combined to perform the Java Security Analysis. The subject of this chapter is the way how this combination works in order to detect security vulnerabilities. The architecture of the analysis and the analysis process are described in detail. The details of the underlying analysis methods are discussed in the chapters mentioned earlier. This chapter gives an overall view and description of the analysis.

In the second part of this chapter, the Java Security Analysis is evaluated. A simple guestbook Java web application is used to test the Java Security Analysis to verify the workings of the analysis. This example accurately reflects simple web applications, but complex business Java web applications often make use of existing frameworks in their architecture. Therefore it is needed to determine the effectiveness of the analysis on Java web application based on these frameworks. Several popular Java web application frameworks are inspected for this reason in chapter 9.

8.1 The Security Analysis Architecture

The architecture of the Java Security Analysis is illustrated in figure 8.1. Interface classes and several other supertypes that belong to the SAT framework are omitted to avoid cluttering in the figure. The architecture includes the classes discussed in earlier chapters. The heart of the architecture is the Security Fact Graph, which role is twofold. First, the graph is constructed while the Java source files are analyzed. Second, the graph is analyzed to detect input validation vulnerabilities.

The Java classes in the Java Security Analysis are divided into Java packages. Related classes belong to the same package, often packages contain classes that inherit from the same superclass. There are 5 packages in total.

- **typeinformation**: This package contains classes that are used to resolve program entities like (super)classes, variables and methods.
- **obsvisitors**: This package contains subclasses of the ObservationVisitors class, which are used to traverse source files. An AST is constructed for every source file, which is then analyzed by using the TreeWalker class.
- **astvisitors**: This package contains subclasses of the AbstractActionVisitor class, which are used by the TreeWalker class to traverse the ASTs.
8.2 The Security Analysis Process

The security analysis process consists of 7 main steps, which are listed and described in the list below. The steps 4, 5 and 6 are described briefly, as these steps are already discussed in detail in the previous chapters about object type inference (chapter 6) and taint analysis (chapter 7).

1. **Initialization**: This step is started by the Spring Framework, which reads the XML configuration file. The framework makes sure that objects are created with the right settings and that the `SecurityMethods` class is given a list of untrusted and critical methods. The security method list is a standard list, which already contains known untrusted and critical methods. To obtain the best results, this list should be customized for every software system. The general `NetworkRunner` class is created and then started, which in turn creates and starts the `JavaSecurityNetwork`. The real analysis is started by this class.

2. **Build Info Graph**: This step builds the standard Info Graph, which is a model derived from the available source files. The Info Graph is used to retrieve information of local classes, such as resolving local types or methods. The class responsible for building the Info Graph is called `SourceGraphMaker`. The Info Graph is not illustrated in the figure, because it is a data structure used by the SAT framework. There is no direct interaction between the Info Graph and the Java Security Analysis.

3. **Get Inheritance Information**: This step is needed to retrieve inheritance from external types by local types. The external types are ignored, when the Info Graph is built. This means that the external inheritance information cannot be found in the Info Graph, therefore this step is needed to make it possible to determine the external type inherited by local types. The class that is specially constructed to make this possible is called the `InheritanceObservationVisitor` class, which is a subtype of `ObservationVisitor`. The `ObservationVisitor` class is used to retrieve information from the source files. For this task, it uses the `InheritanceExtractionVisitor` class, which is a subtype of the `AbstractActionVisitor` class. The inheritance information is stored in the `InheritanceInformation` class, which is used by the `TypeInfo` class to resolve external supertypes of local types.

4. **Build Security Fact Graph**: This is an important phase of the analysis. The Security Fact Graph is constructed with the subtypes of `AbstractActionVisitor` class described in 6 about object type inference. These visitors capture data dependency relations based on the object oriented type inference rules, which are then stored in the Security Fact Graph.
8.2 The Security Analysis Process

Figure 8.1: Java Security Analysis architecture
5. **Taint and Strip Security Fact Graph:** The Security Fact Graph that resulted from the previous step is processed in this step. Untrusted method nodes are tainted and propagated further in the Security Fact Graph. Redundant nodes are removed. The result is a relatively small Security Fact Graph, which consists only of tainted nodes and it might contain dangerous data flow paths.

6. **Detect Dangerous Tainted Paths:** The goal in this step is to detect tainted paths from untrusted methods to critical methods. Such paths can be mapped to real data flow paths taken by dangerous input. Other tainted paths are not considered dangerous. It does so by backtracking tainted critical method nodes to the origin of the taint, which is an untrusted method. This step is covered in detail in chapter 7.

7. **Report Results:** The actual result is the file that contains the dangerous tainted paths. The file is the result of the previous step. As an extra, the tainted and stripped Security Fact Graph is also saved as a DOT file that can be opened with the visualization program Graphviz [5].

The results can be used to manually verify the existence of real security vulnerabilities by checking if input is validated correctly in the source code. If a real security vulnerability is found, it can be fixed by validating the input using filter or check methods.

### 8.3 Experimental Results

A guestbook web application is created to show the workings of the analysis on a real Java web application. The source code of the guestbook can be found in appendix D. The guestbook has two basic functionalities. One is adding a new guestbook entry to the database and the other is retrieving all the entries from the database for display. A MySQL [49] database is used to store guestbook entries. The guestbook contains several SQL injection vulnerabilities. The untrusted method is specified as `javax.servlet.ServletRequest.getParameter()`, which returns the user supplied parameter. In order to make the security vulnerability complete, the critical method `java.sql.Statement.executeQuery()` is used, which executes a SQL query.

The stripped Security Fact Graph that corresponds to the guestbook can be found in figure 8.2. All the nodes of the stripped Security Fact Graph are tainted. Redundant nodes are removed, without influencing the outcome of the analysis. The names of the edges are not displayed. Like expected, several dangerous paths from an untrusted method to a critical method are found. In total, there are three paths found. The paths originate from the `getParameter()` method call, which is used to retrieve the user input. The paths end with the `executeQuery()` method call on the `Statement` object. The specific tainted paths can be found in appendix E, which contains the literal output file content.

The analysis does not recognize methods or algorithms used to validate input, which means that dangerous tainted paths are also found if the input is validated correctly. This is the reason why tainted paths found by the Java Security Analysis have to be verified manually.
Figure 8.2: Security Fact Graph of Guestbook
8.3 Experimental Results

8.3.1 Prepared Statements

The security vulnerability in the guestbook can be easily prevented by validating the input or by using the PreparedStatement class instead of the Statement class. The variables used as parameters to prepared statements are automatically escaped in code fragment 8.1.

```
String mysqlStatement =
"INSERT INTO guestbook (name, email, message) " +
"VALUES('?', '?', '?');

PreparedStatement prepStmt = con.prepareStatement(mysqlStatement);
prepStmt.setString(1, name);
prepStmt.setString(2, email);
prepStmt.setString(3, message);

ResultSet rs = prepStmt.executeQuery();
```

Code Fragment 8.1: Example use prepared statements 1

Using prepared statements alone is not a guarantee of preventing security vulnerabilities, as in code fragment 8.2. In this example, SQL injections are still possible. The binding and escape facilities are circumvented as normal string concatenation is used to compose the statement.

```
String name = request.getParameter("name");
String email = request.getParameter("email");
String message = request.getParameter("message");

PreparedStatement prepStmt = con.prepareStatement("INSERT INTO guestbook (name, email, message) " +
"VALUES(' + name + ', ' + email + ', ' + message + ')";

ResultSet rs = prepStmt.executeQuery();
```

Code Fragment 8.2: Example use prepared statements 2

If the prepared statements are not used correctly, security vulnerabilities may still exist. This is the reason why the following prepared statement methods are included in the critical method list.

- java.sql.PreparedStatement.execute()
- java.sql.PreparedStatement.executeQuery()
- java.sql.PreparedStatement.executeUpdate()
This chapter discusses the influence of use of several popular frameworks by the application on the detection of Java security vulnerabilities. The frameworks that are often used for web applications are selected, because web applications often need to be more secure than the average applications, because anyone can interact with them. An attempt is made to determine if the use of the selected frameworks prevents or restricts input validation vulnerabilities. The second objective is to locate untrusted and critical methods in these frameworks.

The advantage of using frameworks is that common design problems are already addressed by a framework, such as encryption and session management. This way, developers can focus on implementing the business process, rather than designing and implementing (utility) classes that are needed.

Most Java web applications developers work with JavaServer Pages (JSP). Such files can be recognized by the .jsp extension. JSP allows developers to dynamically generate HTML code easier than with Java servlets. The JSP source files are compiled into Java Servlets by a JSP compiler.

The analysis cannot handle JSP pages, therefore it is needed to convert JSP files to normal Java files before the analysis can commence. Several web application frameworks described next, which include Apache Struts and JavaServer Faces, are often combined with JSP. This means that JSP files in such web applications also have to be converted to standard Java source files before the analysis.

### 9.1 Apache Struts

The goal of the Apache Struts [17] framework is to enforce a Model-View-Controller (MVC) [20] architecture in the web application. This means the system is separated in three different components. The model component represents the data and logic in the application, the view component is responsible for displaying data, and the controller defines ways to interact with the model.

Apache Struts makes it possible for JSP files to externalize flow control. No static links to other pages are embedded in the source code. All access to the application is controlled by the ActionServlet class that handles all incoming requests. Individual requests are handled by the Action class, which performs the requested actions. HTML forms are represented by an ActionForm class, which contains and validates
the fields of the form. This means the developer does not have to retrieve the form fields directly, but to use setters and getters to access them.

Apache Struts provides an abstraction layer between the user and the application. The get methods of the ActionForm can be identified as untrusted methods, but because they are defined by the developer, they have to be located manually and specified before the analysis. The ActionForm also promotes the use of input validation, which means that the input from the ActionForm will probably pose no real threat to the system and will result in a harmless tainted path. No additional critical methods are identified.

### 9.2 JavaServer Faces

JavaServer Faces (JSF) [7] also allow the developer to follow the MCV style for web applications by separating the user interface from the underlying logic. The framework contains primarily a collection of classes that makes it possible to create user interfaces in JSP files, unlike Apache Struts that has no user interface component. Flow control is externalized by predefining the flow in an XML navigation file.

Like Apache Struts, JSF also provides input validation classes that can be used to develop secure web applications. These classes are called validator classes and if the standard classes are unsuitable for a certain input, a custom validator class can be created. User input is also retrieved by custom setters and getters. The same procedure applies as in the analysis of Apache Struts based web applications: untrusted methods and critical methods have to be located manually and specified.

### 9.3 WebObjects (Enterprise Objects Framework)

WebObjects [37] is a free Java web application server by Apple Computer. It consists of tools and frameworks to develop and deploy web applications. In the context of the Java Security Analysis, only the Enterprise Objects Framework (EOF) is inspected. EOF converts database rows into an object graph, which can be converted with EO-Modeler to a Java Enterprise object (EO). EOF manages the EOs automatically by committing changes to the object to the database. It has a graphical user interface to work with, which can also be used to create user interfaces for web applications. The EOF is an extensive framework that not only allows object persistance, but it can also be used to create rich user interfaces.

Untrusted methods are surely to be expected, because the framework provides methods to handle user interfaces and input form data. Critical methods could also be expected, because of the ORM functionality the framework provides. There are in total four untrusted methods found, which retrieve user input.

- `com.webobjects.appserver.WOMessage.cookies()`
- `com.webobjects.appserver.WORequest.formValueForKey()`
- `com.webobjects.appserver.WORequest.formValues()`
- `com.webobjects.appserver.WORequest.stringFormValueForKey()`
9.4 Object-Relational Mapper (ORM)

Program data is not persistent, which means that data is lost when the program crashes. That is the reason why many business applications store their data in databases such as MySQL, which is a relational database, to make the data persistent. The object oriented paradigm of Java differs from the relational model of MySQL or other relational databases. The relational model uses relations, tuples and sets, while the object oriented paradigm uses objects, attributes and associations between objects. The difference between the two paradigms causes a gap that has to be bridged by the developer. An object-relational mapper (ORM) that converts data between the two paradigms can be used to bridge that gap.

9.4.1 Hibernate

Hibernate [28] is a popular ORM, which is often used in Java applications. A mapping file is used to store the mapping information between objects and the database data. The mapping information is used by Hibernate to convert data between the two paradigms. Interacting with the database by executing SQL queries directly in the Java application is not necessary anymore, and highly discouraged.

A database system used in an application is a common source of critical methods, but in this case the database is wrapped by Hibernate and generally not accessed directly. As objects are stored and handled by Hibernate, dangerous input is filtered out in the conversion process. This means that queries cannot be manipulated to breach the database. Hibernate does allow custom queries to be created with the two method calls, which take the SQL query as parameter. These methods that are listed below require validated input for secure operations, therefore they are listed in the critical methods list.

- org.hibernate.Session.createQuery()
- org.hibernate.classic.Session.createQuery()
around. The framework does so by executing the right query in the mapping file and with the parameters filled in. Dangerous characters in parameters have no dangerous effect on the query, because the whole parameter is interpreted literally.

No untrusted methods are found in the Apache iBatis framework, because no user input is directly dealt with. SQL queries are prepared correctly and cannot contain dangerous input characters that changes the structure and the semantics of the query. The framework also does not provide a method to execute SQL queries directly, nor does it allow the developer to use the prepared statements incorrectly, therefore no method in the framework can be identified as critical method.

9.4.3 Oracle TopLink

Oracle TopLink [2] is the third ORM discussed. Unlike Hibernate and Apache iBatis, TopLink is not opensource and the framework is a commercial product of Oracle, which is well known for their relational database. It also has a extensive graphical user interface to manage the mapping between Java objects and SQL data.

No untrusted or critical methods are found for the same reason as Apache iBatis. There is no functionality provided to access the database directly by executing custom SQL queries and incorrectly prepared statements cannot be abused. Like Hibernate, Oracle TopLink provides a thick abstraction layer on top of the SQL database, which means that there are no SQL queries defined as mapping information.

9.4.4 Spring Framework JDBC

The JDBC package that is part of the Spring Framework [10] is the last ORM discussed. The Spring package makes it easier to access the database by providing an abstraction layer on top of the standard Java JDBC API. The layer consists of a JdbcTemplate class, which manages the database resources and errors. The mapping information is stored in a file, which contains SQL queries that are used for mapping.

All SQL queries are prepared, which eliminates most dangers of possible dangerous characters in the SQL query. Incorrect use of prepared statements is still possible, so the SQL methods in JDBC below are added to the list of critical methods.

- org.springframework.jdbc.object.RdbmsOperation.setSql()
- org.springframework.jdbc.object.SqlUpdate()
- org.springframework.jdbc.object.BatchSqlUpdate()
- org.springframework.jdbc.object.SqlOperation.newPreparedStatementCreator()
- org.springframework.jdbc.object.SqlCall()
- org.springframework.jdbc.object.SqlQuery()
- org.springframework.jdbc.object.MappingSqlQueryWithParameters()
- org.springframework.jdbc.object.MappingSqlQuery()
• org.springframework.jdbc.object.SqlFunction()

• org.springframework.jdbc.object.UpdatableSqlQuery()
Chapter 10

Related Work

There are many approaches to detect security vulnerabilities in software, which can be classified into two main categories: dynamic analysis and static analysis. The first approach, which includes penetration testing and runtime monitoring, is briefly summarized and compared to the static analysis approach that is the approach taken by this project. The second approach is discussed by comparing this project to several other projects that fall under this category.

10.1 Dynamic Analysis

Dynamic analysis is analysis performed on a running program. Dynamic analysis is characterized by precision, but it is often not exhaustive [15]. The precision is due to the fact that no abstraction and approximation are made of the program, which are often the case with static analysis. Dynamic analysis is also slow compared to static analysis, especially when the analysis tries to be exhaustive, which means all possible execution paths are considered. The exhaustiveness of the analysis depends on the inputs of the running program. A larger range of different inputs gives a more exhaustive analysis.

10.1.1 Runtime Monitoring

Runtime monitoring is just as the name says: monitoring the program in execution. Different runtime monitoring methods exist to prevent security vulnerabilities of being exploited. One way is to provide a wrapper to the program, which filters out dangerous input values [42]. This way, dangerous input can never reach the program and exploit the security vulnerability. The security vulnerability in the program is left in and may never be detected. Like the previous approach, program knowledge is needed to determine the values allowed by the wrapper. Another form of runtime monitoring requires adding additional monitoring code in the source code of the source code or recompiling the libraries. The wrapper is in this case integrated into the software.

Runtime monitoring prevents possible security vulnerabilities being exploited. It does not detect real security vulnerabilities and sometimes it can only detect abnormal runtime behavior without determining the cause. Crash detection is an example of the latter case. It can determine that the program is crashed, but not the reason that causes it to crash. Preventing security vulnerabilities of being exploited is not the goal of this...
project. The goal is to detect security vulnerabilities, which can be removed to create a better quality and more secure software.

The dynamic form of taint analysis is used in many other research projects [50, 51, 40]. This form of taint analysis tracks data during execution, such as the taint execution mode of the Perl interpreter. If the Perl interpreter is run in this mode, variables that contain external values are tainted and checked if the values are safe (untainted) before being used in critical Perl functions that are already known by the interpreter.

The drawbacks of dynamic analysis applies to this form of taint analysis. The incompleteness causes not all possible security vulnerabilities to be found. On top of that, runtime components such as: source code, libraries, Operating System or even hardware have to be modified to make dynamic taint analysis work. This makes this technique inflexible and sometimes very difficult to use.

10.1.2 Penetration Testing

Penetration Testing [16, 23] is the most used approach to find security vulnerabilities in software. It is a special form of dynamic analysis, where the system is considered a blackbox. Internal control/data flow and data structures are unknown. This analysis requires a set of input values, which can be composed manually or generated automatically. The set of input values, which often contains known malicious values, is then given as input to the running program. The resulting program behavior is then evaluated.

This approach often reveals a small percentage of all security vulnerabilities in a program. This is caused by the incompleteness of execution, which means that not all execution paths are taken, and the incompleteness of the set of input values. Determining a set of input values that contain all dangerous values and that tests all execution paths, requires considerable program knowledge. In the case of detecting new vulnerabilities, this approach is not effective, because the result depends on the set of input values. This approach is effective in detecting known security vulnerabilities in systems that are not patched, as the specific input is already known.

10.2 Static Analysis

Static analysis is characterized by its imprecision, but the exhaustiveness of some analyzers and flexibility make this a powerful type of analysis. Static analysis approaches differ greatly in approach in complexity [18]. Some use simple lexical scanning techniques, while others use advanced syntactical/semantical analysis techniques that are control or data path sensitive. Simple lexical analysis is not suitable for Java, because the object oriented complexities cause very inaccurate results. This is the reason why lexical scanners are not used for comparison. For the rest of this chapter, only three static security analyzers that are based on static taint analysis, are selected for comparison. Due to the small number of static taint analyzers for Java, two closely related PHP security analyzers and one for Java are selected. Object oriented features support of both PHP analyzers is limited and not as important as in Java. Everything is an object in Java, while object oriented features in PHP are an extra.
10.2 Static Analysis

10.2.1 WebSSari

WebSSari [52] uses static analysis to detect security vulnerabilities in PHP source code. Static analysis is performed to construct the AST, CFG and a symbol table that tracks the state of variables. The CFG is used to determine dangerous tainted paths by type inference and to locate dangerous functions that can use tainted values. Special validation code is inserted in the corresponding location in the source code. The code performs checks and prevents security vulnerabilities being exploited.

Similar to this project, a list of security functions is used to identify untrusted and critical PHP functions. User submitted values are considered untrusted and are sources of taint. The second similarity is that WebSSari also derives a model from the AST first, which is then analyzed instead of the source or AST directly. Although the model differs greatly. The third similarity is that arrays are not supported. Although array support can easily be created, because of the modular design of the underlying SAT framework.

The model analyzed in the case of WebSSari, is the CFG, while the Java Security Analysis uses a data dependency graph called Security Fact Graph that is obtained by type inference. The second difference is the purpose of using a variant of type inference. WebSSari uses it to propagate taint, while the Java Security Analysis uses it to construct the Security Fact Graph. Other important differences are the lack of object oriented features, references and libraries support. The parser used of WebSSari does not parse all PHP files correctly and rejects them. In contrast to the Java Security Analysis, which parses all Java source files successfully.

10.2.2 Pixy

Pixy [39] is another security analyzer for PHP. It first constructs a CFG, which is used to perform the analysis on. Pointer analysis is used to determine which variables point to the same memory location. If one variable is tainted other variables that point to the same memory location are also tainted. The goal is to detect tainted variables, also originated from untrusted PHP functions, reaching critical PHP functions. Such functions are predefined like the analyzer described in this paper.

Like the previous security analyzer, Pixy does not support object oriented features. Each use of object members and methods are treated in an optimistic way, which means that taint does not propagate from these object members. The other difference with the Java Security Analysis and the security analyzer described earlier, is that Pixy uses pointer analysis to determine taint propagation, which typically trades off scalability for precision. Type inference is not used to determine taint propagation.

10.2.3 The Griffin Software Security Project

The two static security analyzers discussed earlier are used to analyze PHP source files to detect the same type of security vulnerabilities as this project. Although PHP and Java share some programming language properties as their syntax is similar and both languages support objects. However, they are not quite the same.

An analyzer that also tries to detect security vulnerabilities in Java source files is the analyzer of the Griffin Software Security Project [31, 33], which is a project of the
SUIF group, a research department of the Stanford University. The similarities and differences between the two analyzers are evaluated.

The static analyzer of the SUIF group shares the same goal with this project, which is to detect input validation vulnerabilities by detecting dangerous data flow paths. Like the Java Security Analysis, the untrusted methods that they call source and critical methods sink, have to be specified before the analysis starts. The analyzers then try to detect paths between these methods. Both analyzers are written in Java and work on generated ASTs to perform the taint analysis. The limitation both analyzers have is that they are unable to distinguish individual objects in container classes. The conservative assumption is therefore made that container classes that can contain tainted objects always return tainted objects.

The first difference is the interface of the analyzer of the SUIF group, which is integrated in Eclipse, while there is practically no user interface in the Java Security Analysis. Configuration happens with a configuration file and output is also written to files. This allows the Java Security Analysis being run as a background process on a remote server.

The other big difference is in specifying the untrusted and critical methods. The Java Security Analysis requires such methods to be specified in fully qualified names. In the analyzer of the SUIF group, this is not necessary, which causes ambiguity of types. The specification has to be done in a custom language called PQL [36], which is also used to specify propagation rules. The propagation rules in Java Security Analysis are already elicited and integrated in the analysis. Specifying additional propagation rules is not possible.

Another structural difference is that the analyzer of the SUIF group analyzes byte-code. The availability of source files is not necessary and analyzing libraries directly is also possible. The analyzer can also be modified to support the analysis of, for example, the Microsoft Intermediate Language (MSIL), which makes the analysis of C# software possible. The Java Security Analysis has limited support for library analysis, it uses BCEL to retrieve the most important type information from libraries.

The underlying propagation structure of both analyzers also differ. The analyzer of the SUIF group uses pointer analysis [48] to handle an essential part of the taint propagation, while the Java Security Analysis uses a variant of type inference to determine the possible taint propagations. Pointer analysis determines if a certain variable could point to another variable some time during the execution of the program. Pointer analysis approaches typically trade scalability for precision. The analyzer of the SUIF group uses binary decision diagrams (BDDs) in the pointer analysis to maintain precision without the scalability trade off. Pointer analysis is not used in this project, because the existing SAT framework allows an existing type inference analysis to be reused and modified for the detection of input validation vulnerabilities. The overall result of both analyses do not differ much, although the approaches are quite different.

The last difference is the propagation of taint. In the Java Security Analysis the taint is a real object that propagates through a graph. In the analyzer of the SUIF group, the taint propagation is just a concept, no taint is really propagated, but the objects the taint can propagate to are determined, which means no taint object is needed.

The practical advantage of the Java Security Analysis is the fact that it can run on a fast remote server as batch or background process during the night. Results can be collected as files the next day, which can be conveniently processed. It is not meant
to be a developer tool such as the analyzer of the SUIF group, which is integrated in
Eclipse to assist the developer during development. The advantage of the Java Secu-

rity Analysis is the advanced support of polymorphism. Supertypes and subtypes are
automatically determined if all the libraries are available. This prevents the hassle of
having to determine the hierarchy of untrusted or critical methods and it allows an ac-
curate analysis. Real differences in performance and precision have to be determined
by testing both analyzers to the same extensive set of software that contains input val-
idation vulnerabilities.
Chapter 11

Conclusions and Future Work

This chapter starts with the summarization of the contributions of the project, which is then followed by the conclusion and discussion sections that discuss the results and the development process of the project. Finally, this chapter ends with some ideas for future improvements of the Java Security Analysis.

11.1 Contributions

The Java Security Analysis allows SIG to detect security vulnerabilities in software of clients. The list of services to clients can be extended by a security check or assessment service, which makes it interesting for clients who want their Java web application to be checked for security vulnerabilities.

The other contributions are actually side products, which are essential components of the analysis. The first one is the extended type inference framework, which now supports object types. This is in contrast with the former type inference framework, which can only deal with integer types. The second is the BCEL wrapper that is used to resolve library types. This wrapper is proven to be useful and it can be used to increase accuracy of the existing analyses used by SIG.

The analysis is strongly data flow oriented, which means that it can easily be modified for other purposes than detecting security vulnerabilities, for instance, to identify all locations where a certain value is used. This way, dependencies of classes or modules on that value can be identified to separate software architecture modules. This information can be used to improve program understanding, which is in line with the SWERL research area.

This project shows a way how type inference can be used to capture data dependency relationships between variables. These relationships are then used to perform taint analysis in order to detect input validation vulnerabilities. It also shows a way to deal with objects, which can be defined in libraries. In addition to normal source code analysis, the use of byte code analysis is described to improve accuracy.

11.2 Conclusions

The Java Security Analysis allows SIG to find input validation vulnerabilities in software written in Java. The analysis is implemented by extending existing classes found
in the SAT framework as much as possible. The overall structure confirms to the SAT standard. False positive results still exist, because the analysis is unable to detect validation of values. Therefore manual verifying results is still necessary to confirm real security vulnerabilities.

The rate of false positives depends on the list of untrusted and critical methods. On the conservative side, all methods found in appendix A can be included in the analysis. The methods that in theory can lead to security vulnerabilities but often not, can cause false positives being found. To prevent false positives, the methods that are unlikely to cause a security vulnerability should be excluded. Some program knowledge is desired for tailoring the analysis for optimal results.

Several limitations exist, as described at the end of chapter 6. Some unhandled cases are logged. The total number of unhandled cases differ per system, but they will not influence the result greatly as they do not occur frequently.

11.3 Discussion

BCEL is used to extract type information of superclasses and subclasses. Information about methods are also extracted using BCEL to determine if it concerns an untrusted or critical method. However, much can be gained in accuracy by using BCEL to inspect the inner code in the body of classes and methods also. This code can also be incorporated into the Security Fact Graph to avoid the approximation of library elements, which results in false positives in some cases.

11.4 Future work

The first improvement that comes to mind is the ability to detect filter or validation methods, such as the use of pattern matching. However, there is no standard method to do this, because the format of a validated value has to be defined. It is difficult to determine if a value is validated or not. An idea is to specify the validation method before the analysis and the analysis will determine if a tainted value is passing through a validation method, which will untaint it.

Java web applications are often developed in JSP files, which is a convenient and popular way to develop them. These files are not supported by the Java Security Analysis directly. These files have to be converted manually to standard Java source files first, before the analysis can start. This functionality can easily be integrated in the analysis to convert JSP files automatically.

The limitations of object oriented type inference that are described in chapter 6 means that not all code is analyzed. It mostly concerns complex method calls and arrays. These limitations can be elimited by rewriting parts of the analysis. This will result in more code being analyzed, which increases the precision. The unhandled cases are logged during earlier tests on real business software systems. These extreme cases are rare and most of the time not worth the effort to support it.

The discussion section already indicates that large parts of class files found in libraries are unused, which means that a wealth of useful information in the class files is unused, such as code found in the body of classes and methods. The bytecode found in class files can be analyzed together with the normal source files to improve precision.
Bytecode can be used for example, to eliminate limitations of taint propagation in library classes. In the same way of handling normal source code, program elements found in the bytecode should also be modeled as nodes in the Security Fact Graph. The current analysis generalizes taint propagation for library classes to a limited set of nodes, because the code inside classes is not used to refine taint propagation within the library classes.
Bibliography


Appendix A

List of Methods

The standard untrusted methods and critical methods are listed here. The list is composed after careful inspection of the standard libraries and several popular frameworks. The prefix indicates the type the method belongs to, which can be a class or interface. This means it is not used to denote the return type of the method. Untrusted input methods are Java methods used to get user input, which can be dangerous and should not be trusted. Critical methods are used to delegate tasks to subsystems.

A.1 Untrusted Methods

In the package java.awt:

TextComponent.getText(): Get the text string.

TextComponent.getSelectedText(): Get the selected text string.

In the package java.io:

BufferedReader.readLine(): Read a line of text.

Reader.read(): Read a character sequence.

In the package javax.servlet:

ServletRequest.getParameter(): Get the value of the parameter.

http.Cookie.getValue(): Get the value stored in the cookie.

In the package com.webobjects.appserver:

WOMessage.cookies(): Get the all cookies.

WORequest.formValueForKey(): Get value of a form, identified by name.

WORequest.formValues(): Get all values of form elements.

WORequest.stringFormValueForKey(): Get the value of a form as a string.
A.2 Critical Methods

In the package `java.sql`:

- `Statement.execute()`: Execute a SQL query.
- `Statement.executeQuery()`: Execute a SQL query.
- `Statement.executeUpdate()`: Execute a SQL update query.

In the package `java.io`:

- `Writer.write()`: Write a line.
- `PrintWriter.println()`: Print a line.
- `PrintWriter.print()`: Print a line.
- `File()`: Create a new file.

In the package `java.lang`:

- `Runtime.exec()`: Execute a program.
- `Runtime.getRuntime()`: Get a runtime class for program execution.

In the package `com.webobjects.eoaccess`:

- `EOUtilities.rawRowsForSQL()`: Execute the specified SQL and return the raw rows.
- `EOAdaptorChannel()`: This class is an abstract class for performing database operations.

In the package `org.hibernate`:

- `Session.createSQLQuery()`: Execute a Hibernate query and return results.
- `classic.Session.createSQLQuery()`: Execute a Hibernate query and return results.

In the package `org.springframework`:

- `jdbc.object.RdbmsOperation.setSql()`: Set the SQL query executed by this object.
- `jdbc.object.SqlUpdate()`: This class represents a SQL update query.
- `jdbc.object.BatchSqlUpdate()`: This class performs batch update operations.
- `jdbc.object.SqlOperation.newPreparedStatementCreator()`: This class is used to create prepared statement.
- `jdbc.object.SqlCall()`: This class represents a SQL based call.
- `jdbc.object.SqlQuery()`: This class represents a SQL query.
jdbc.object.MappingSqlQueryWithParameters(): This class represents a query, in which concrete subclasses must implement the abstract mapRow(ResultSet, int) method to map each row of the JDBC ResultSet into an object.

jdbc.object.MappingSqlQuery(): This class is a simplified MappingSqlQueryWithParameters class.

jdbc.object.SqlFunction(): This class is a SQL function wrapper for a query that returns a single row of results.

jdbc.object.UpdatableSqlQuery(): This class is a query representation, in which concrete subclasses must implement the abstract updateRow(ResultSet, int, context) method to update each row of the JDBC ResultSet.
Appendix B

Type Inference Nodes

The node types used in integer and object oriented type inference are listed here. A node consists of an identifier string and a type string. The combination of identifier and type is unique and used for identification of the node. This appendix only lists the identifiers. The example class used is called `Class`, which belongs to the package called `package`. Keywords can be easily recognized, because they start with a `$`-sign.

The first list contains the node types that can be found in the Fact Graph, which is constructed with integer type inference. These nodes are also used by object oriented type inference, therefore they can also be found in the Security Fact Graph.

### B.1 Integer Type Inference Nodes

- **package.Class.var**: a member of a class.
- **package.Class.method(int i).var**: a local variable of a method.
- **package.Class.method(int i).$arg0**: the first parameter of the method call.
- **package.Class.method(int i, int j).$arg1**: the second parameter of the method call.
- **package.Class.method(int i).$return**: the return variable that represents the return value of the method.
- **package.Class.$constructor(int i, int j).$arg0**: the first parameter of the constructor method.

### B.2 Object Oriented Type Inference Nodes

The following list are the new node types introduced by object oriented type inference. The nodes can only be found in the Security Fact Graph, which is constructed with object oriented type inference. The example library class used is the `Vector` class, which can be found in the standard `java.util` package. A special new keyword is introduced to denote the resolvability of types and methods. Library types and (inherited) methods are normally identified by their fully qualified name. If the type or method is not found in a library, the type or method is prefixed with `$unresolvable`. The scope prefix is omitted in this listing because of the length of the whole name.
$object.X$: an instance of class $X$.

$object.java.util.Vector$: an instance of the class Vector, which is found in the library package java.util.

$unresolvable.$method.$System.out.println(): an unresolvable static method call.

$v$: the variable $v$. The type (not displayed) is java.util.Vector. This node represents a Vector object.

$v.$method.$java.util.Vector.removeAllElements(): a method call on variable $v$ of the class Vector.

$v.$extsuper.$call.$java.util.Vector.elementAt(): a super method call on variable $v$ of the class Vector.

$v.$unresolvable.$extsuper.$call.$java.util.Vector.elementAt(): an unresolvable super method call on variable $v$ of the class Vector.
Appendix C

Expression Operators

The lists of expression operators and expression elements can be found here. The expression elements of interest, which can be found in expressions are listed first. Followed by the list of expression operators that are used to distinguish and separate individual expression elements found in an expression.

C.1 Expression Elements

Normal method calls: method(i, j)

Constructor calls: this()

Super constructor calls: super()

New object calls: new SampleClass(x, y, z)

Identifiers: System.out

C.2 Expression Separators

/ The division operator.
/= The division assignment operator.
- The subtraction operator.
-= The subtraction assignment operator.
% The remainder operator.
% The remainder assignment operator.
+ The addition operator.
+= The addition assignment operator.
<< The left shift operator.
<<= The left shift assignment operator.
C.2 Expression Separators  Expression Operators

>> The right shift operator.

>>= The right shift assignment operator.

* The multiplication operator.

* The multiplication assignment operator.

-- The prefix and postfix decrement.

++ The prefix and postfix increment.

? The question operator. Only the variable \( a \) and \( b \) are considered as expression elements in the expression \((\text{expression } ? \ a \ : \ b)\). Everything in \text{expression} is ignored.

(Type) The typecast. The expression typecasted is processed further to extract expression elements.
Appendix D

Guestbook Source Code

The Java guestbook web application that is used to demonstrate the Java Security Analysis. The guestbook can be used stand-alone and is fully functional. The guestbook supports adding a new guestbook entry and displaying all entries. The guestbook contains a SQL injection vulnerability that can be detected using the Java Security Analysis.

```java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;

public class Guestbook extends HttpServlet {

    // Handles request.
    public void doGet(HttpServletRequest req,
                      HttpServletResponse res)
        throws ServletException, IOException {
            PrintWriter out = res.getWriter();
            String name = req.getParameter("name");
            String email = req.getParameter("email");
            String message = req.getParameter("message");

            printHeader(out);

            if (name == null)
                printForm(out);
            else
                addNewEntry(name, email, message);
            printAll(out);

            printFooter(out);
    }

    // Prints the HTML header.
    public void printHeader(PrintWriter out) {
        out.println("<html>");
        out.println("<head><title>Guestbook</title></head>");
        out.println("<body>");
        out.println("<h1>Java Guestbook</h1>");
    }
```
### \Guestbook Source Code

```java
guestbook source code

    /// Prints the footer.
    public void printFooter(PrintWriter out) {
        out.println("</body>");
        out.println("</html>");
    }

    /// Prints out form.
    public void printForm(PrintWriter out) {
        out.println("<form method="get" action="Guestbook">")
        out.println("<input type="text" name="name">");
        out.println("<input type="text" name="email">");
        out.println("<input type="text" name="message">");
        out.println("<input type="submit">");
        out.println("</form>");
    }

    /// Adds new guestbook entry.
    public void addNewEntry(String name, String email, String message) {
        Connection con = null;
        Statement stmt = null;

        try {
            con = DriverManager.getConnection("jdbc:mysql://localhost:3306/guestbook", "login", "pass");
            stmt = con.createStatement();

            // Execute the query.
            stmt.executeQuery(" INSERT INTO guestbook (name, email, message) VALUES ('" + name + ", '" + email + ", '" + message + ")");

            // Close the connection to the database.
            con.close();
        } catch(Exception e) {}
    }

    /// Prints all guestbook entries.
    public void printAll(PrintWriter out) {
        Connection con = null;
        Statement stmt = null;
        ResultSet rs = null;

        try {
            // Get a connection to the MySQL database.
            con = DriverManager.getConnection("jdbc:mysql://localhost:3306/guestbook", "login", "pass");
            stmt = con.createStatement();

            // Close the connection to the database.
            con.close();
        } catch(Exception e) {}
    }
```

---

92
// Execute the query and get the results.
rs = stmt.executeQuery("SELECT name, message " +
    "FROM guestbook");

// Display the results.
out.println("<ul>");
while (rs.next()) {
    out.println("<li>");
    out.println(rs.getString("name") + ": ");
    out.println(rs.getString("message");
}
out.println("</ul>");

// Close the connection to the database.
con.close();
} catch (Exception e) {}
Appendix E

Tainted Paths in Guestbook

The tainted paths from untrusted methods to critical methods found in the guestbook are listed here. This is the original output of the Java Security Analysis. Nothing is modified or simplified.

PATH:
- Guestbook.doGet(HttpServletRequest, HttpServletResponse).req.$extsupercall.javax.servlet.ServletRequest.getParameter()
- Guestbook.doGet(HttpServletRequest, HttpServletResponse).req
- Guestbook.doGet(HttpServletRequest, HttpServletResponse).req.$extsupercall.javax.servlet.ServletRequest.getParameter()
- Guestbook.doGet(HttpServletRequest, HttpServletResponse).name
- Guestbook.addNewEntry(java.lang.String, java.lang.String, java.lang.String).$arg0
- Guestbook.addNewEntry(java.lang.String, java.lang.String, java.lang.String).name

PATH:
- Guestbook.doGet(HttpServletRequest, HttpServletResponse).req.$extsupercall.javax.servlet.ServletRequest.getParameter()
- Guestbook.doGet(HttpServletRequest, HttpServletResponse).req
- Guestbook.doGet(HttpServletRequest, HttpServletResponse).req.$extsupercall.javax.servlet.ServletRequest.getParameter()
- Guestbook.doGet(HttpServletRequest, HttpServletResponse).email
- Guestbook.addNewEntry(java.lang.String, java.lang.String, java.lang.String).$arg1

PATH:
- Guestbook.doGet(HttpServletRequest, HttpServletResponse).req.$extsupercall.javax.servlet.ServletRequest.getParameter()
- Guestbook.doGet(HttpServletRequest, HttpServletResponse).req
- Guestbook.doGet(HttpServletRequest, HttpServletResponse).req
- Guestbook.doGet(HttpServletRequest, HttpServletResponse).req.
  $extsupercall.java.servlet.ServletRequest.getParameter()
- Guestbook.doGet(HttpServletRequest, HttpServletResponse).message
- Guestbook.addNewEntry(java.lang.String, java.lang.String,
  java.lang.String).$arg2
- Guestbook.addNewEntry(java.lang.String, java.lang.String,
  java.lang.String).message
- Guestbook.addNewEntry(java.lang.String, java.lang.String,
Number of tainted paths found: 3.
Appendix F

Glossary

An overview of frequently used terms and abbreviations in this paper, is listed here.

**Abstract Syntax Tree (AST):** a tree data structure, where the internal nodes are labeled by operators and the leaf nodes represent variables or constants. It is often constructed by compilers to represent a computer program. Compilers use this data structure to optimize and to generate code [12].

**Control Flow Graph (CFG):** a graph structured representation of all execution paths that can be taken by a computer program. The nodes represent basic blocks and the edges represent (conditional) jumps [53].

**Data flow analysis:** an analysis method to compute the relations between data, the dependencies and values. Data flow analysis operates on the Control Flow Graph. It uses the paths in a control flow graph to compute the values of variables and their flow [22, 53, 25].

**Dynamic analysis:** an analysis performed by analyzing the software during execution [15].

**Fully qualified name:** a name of a type that includes the full package name. This name is used to identify and resolve Java types.

**HyperText Markup Language (HTML):** markup language used to display information on the web.

**Integrated Development Environment (IDE):** software to assist the computer programmer to develop software. It integrates important software development software like: a compiler, an editor and a debugger into one user interface, to ease software development.

**Legacy software:** old but difficult to replace software used by organizations. This type of software usually assists core business operations.

**Phishing:** a type of electronic fraud that uses an imitation of a trusted website in attempt to steal sensitive information from the users [35].
**Software framework:** a software framework is a reusable design of software. This is a set of (abstract) classes and is intended to be reused for a specific type of software [30].

**Stack:** a data structure commonly used in software, where the first-in-first-out (FIFO) principle applies. The first element stored, will be the first returned.

**Static analysis:** an analysis performed by analyzing the software without actually executing the software [14].