POLITECNICO DI MILANO

Scuola di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Automatic Detection and Correction of Client-Side and Server-Side Input-Validation Inconsistencies

Relatore: Prof. Carlo Ghezzi

Correlatore: Prof. Alessandro Orso

Correlatore UIC: Prof. Ugo Buy

Correlatore ASP: Prof. Marco Torchiano

Tesi di Laurea Magistrale di:

Mattia Fazzini

Matr. 765340

Anno Accademico 2011 - 2012

This thesis is dedicated to the loving memory of my grandfather, Vinicio Fazzini,

the person who made my graduate studies possible.

ACKNOWLEDGMENTS

First and foremost I would like to thank my Supervisor Prof. Carlo Ghezzi and Assistant Supervisor Prof. Alessandro Orso for their guidance in the development of this work and for their patience, motivation, enthusiasm and knowledge. I would like also to thank Prof. Ugo Buy and Prof. Marco Torchiano for their encouragement and insightful comments.

A great thank you to my labmates Shauvik Roy Choudhary, Wei Jin, Dr. Matteo Miraz and Leandro Sales for the interesting and stimulating research discussions.

Last but not least, I would like to thank my parents, Maria Antonietta Di Lena and Gilberto Fazzini, for always believing in me and a special thank you to Mariko Ueda which, in every situation, is there to support me.

MF

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF ABBREVIATIONS

CFG	Control Flow Graph
DFAs	Deterministic Finite Automata
GB	Gigabyte
GNFA	Generalized Non-deterministic Finite Automata
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JEE	Java Enterprise Edition
JSP	JavaServer Pages
MBDD	Multi-terminal Binary Decision Diagram
MVC	Model View Controller
OWASP	Open Web Application Security Project
PDG	Program Dependence Graph
RAM	Random-Access Memory
SQL	Structured Query Language
XML	Extensible Markup Language
XSS	Cross-site Scripting
URL	Uniform Resource Locator

ABSTRACT

The approach presented in this work identifies and corrects erroneous or insufficient validation of the user inputs by automatically discovering inconsistencies between client- and server-side input validation functions. Inconsistencies among input validation function can lead to efficiency and security problems.

Developers typically perform redundant input validation in both the client and the server components of a web application. Client-side validation is used to improve the responsiveness of the application, as it allows for responding without communicating with the server, whereas server-side validation is necessary for security reasons, as malicious users can easily circumvent client-side checks. The main idea behind this approach is that it is possible to leverage the redundancy in these checks to automatically identify inconsistencies within input validation functions. In fact, the approach extracts client- and server-side input validation functions, models them as deterministic finite automata and compares client- and server-side deterministic finite automata to identify, report and correct the inconsistencies between the two sets of checks.

The evaluation of the approach and its implementation are promising. When applied to a set of real-world web applications, the prototype was able to automatically identify and correct a large number of inconsistencies detected in their input validation functions.

SOMMARIO

Questo lavoro presenta un approccio per identificare e correggere, in maniera automatica, errori nelle funzioni di validazione dell'input nel contesto delle applicazioni web. Gli errori nelle funzioni di validazione dell'input possono portare a problemi di efficienza e sicurezza all'interno delle corrispondenti applicazione web.

Solitamente gli sviluppatori compiono validazione dell'input in maniera ridondante, infatti, lo stesso input viene validato sia a lato client che a lato server. A lato client la validazione dell'input usata per migliorare la reattivit delle applicazioni web. A lato server, invece, l'input viene validato per motivi di sicurezza, infatti, i controlli di validazione fatti a lato client possono essere facilmente aggirati da utenti maligni. L'idea principale di questo lavoro quella di usare questa ridondanza nel processo di validazione dell'input per determinare gli errori nelle funzioni stesse di validazione. L'approccio descritto, estrae le funzioni di validazione dalle applicazioni web, modella tutti gli input accettati da queste funzioni come automi a stati finiti e confronta gli automi a lato client con i corrispondenti automi a lato server per poter identificare e correggere le inconsistenze.

L'implementazione dell'approccio mostra risultati promettenti. Infatti, quando sono state analizzate sette applicazioni web reali, stato possibile identificare e correggere un ampio numero di errori nel processo di validazione dell'input.

CHAPTER 1

INTRODUCTION

Web applications are important and increasingly widespread. In fact, nowadays, it is common to use web applications for daily activities such as reading the news, doing online banking, and watching movies. Together with the growth of their user base, the complexity of web applications has also grown. Whereas early web applications were mostly static, today's applications are highly dynamic, with complex logic on both the client and the server sides. In general, web applications follow a three-tier architecture that consists of a front-end component, typically a web browser running on the user machine, a back-end component, i.e., a remote web server, and a back-end data store which could be represented by a database. Because of their nature, web applications have to deal with a large range of issues and problems. Among these issues it is possible to find the set of problems related to the user input. In this set, it is possible to identify two classes of issues: security and efficiency problems. Security is an issue because unlike traditional desktop applications, web applications can be accessed by any user who is connected to the Internet, which exposes a web application to a large base of potential attackers. Second, the back-end database contains data that is often sensitive and confidential, holding information about a potentially large number of users. Finally, the communication between the different layers of a web application occurs through directives that often embed user input and are written in many languages, such as XML, SQL, and HTML. For these reasons, it is of paramount importance to properly validate and sanitize user input, to make sure

that malicious input cannot result in unintended execution of harmful commands and, as a consequence, provide attackers with access to sensitive information. Unfortunately, it is not uncommon for developers to perform either faulty or incomplete input checks, which can leave the web application susceptible to input validation inconsistencies which might lead to security vulnerabilities. Popular examples of attacks that leverage such vulnerabilities are SQL injection, cross-site scripting, locale and Unicode attacks, file system attacks, and buffer overflows. These type of attacks are among the most common and dangerous attacks for web applications, as it is possible to understand from (1) and (2). The second class of issues is represented by efficiency problems. In fact, if inputs are not properly validated in the front-end component, these inputs will be then rejected by the back-end component decreasing in this way the overall performances of the web application and increasing the network load as well. To address this set of problems, this work presents a novel approach for automatically identifying and correcting input validation inconsistencies in web applications. The key insight of the approach presented in this work is based on the observation that developers often introduce redundant checks both in the front-end, client, and the back-end, server, components of a web application. More in detail, client-side checks are mainly introduced for performance reasons, as they can save one network round-trip and the additional server-side processing that would be incurred when invalid input is sent to and subsequently rejected by the web application. Hence, to improve the user experience and provide instant feedback, web applications should thus validate inputs at the client side before making the actual request to the server. On the other hand, since client-side validation can often be circumvented by malicious users, the server cannot trust the

inputs coming from the client side, and all input checks performed on the client side must be repeated on the server side before user input is processed and possibly passed to security sensitive functions. The main idea behind this approach is that, because checks performed at the client and server sides should enforce the same set of constraints on the inputs, it is possible to leverage the redundancy in these checks to automatically identify inconsistencies within input validation functions. If client-side checks are more restrictive, the web application may accept inputs that legitimate clients can never produce, which is problematic because malicious users can in fact bypass client-side checks. If server-side checks are more restrictive, conversely, the client may produce requests that are subsequently rejected by the server, which will result in poor performance and reduce the responsiveness of the web application. Based on this insight, the approach compares the checks performed on one side to the checks performed on the other side and identifies, reports and corrects inconsistencies between them. This way of finding and correcting inconsistencies deals also with the problem that in most of the cases there is no proper specification for input validation inside a web application. From a high-level point of view the characteristics of the approach are the followings. The approach considers the input, for a given input field, to be a string value. It then tries to understand all the possible values that the string can assume by extracting, from both client and server sides, string operations belonging to input validation functions and related to the given input. This extraction process is performed by a context- and path-sensitive analysis which collects the string operations into summaries of input validation functions. More precisely the analysis collects two summaries for the same input field. One representing the string operations performed at client side and

the other representing the string operations performed at server side. After having these two summaries for each of the input fields, the approach transforms them into DFAs. This transformation is possible through the string analysis technique which is detailed in Section 5.2. At the end of this transformation process there will be, for each of the input fields, a DFA representing all possible values which are considered as valid at client side and a DFA representing all possible values which are considered as valid at server side. After these two automata are obtained, the approach compares them and detects if checks on one of the two side are missing for a given input field. If checks on one of the two side are missing, it generates a string proving this inconsistency and creates the related code correction. Summing up, the steps of the approach are:

- Extraction and mapping of input validation functions.
- Modeling input validation functions as DFAs.
- Identifying, report and correct inconsistencies by DFAs analysis.

To evaluate the effectiveness and practicality of the approach, it has been implemented a prototype that can analyze web applications developed using JEE related frameworks. By using the prototype, seven real-world web applications have been analyzed. The rest of this thesis is organized as follows. Chapter 2 gives a general overview on the basic concepts that are necessary in order to understand the rest of the work. Chapter 3 presents a set of definitions that allow a better understanding of the domain of the problem tackled in this work and presents two motivating examples. Chapter 4 illustrates how this work relates with the ones in the stat of the art. In Chapter 5 there is the presentation of the approach which represents on of the

main contributions of this work. Then in Chapter 6 it is considered the implementation of the approach. Chapter 7 presents the evaluation of the approach through empirical results. Finally, Chapter 8 provides concluding remarks and future research directions.

CHAPTER 2

BACKGROUND

In this chapter will be presented a brief overview of the main concepts which are the base of this work. The chapter will start by giving an introduction on what web applications are and which are their main technologies. In addition, while presenting them, it will be shown which are the problems that could lead to efficiency and security issues. Web applications are one of the core elements within this work because they represent the type of software for which the approach, presented in Chapter 5, is developed for. In the second part of the chapter it will be given a brief introduction on automata theory and finite state machines. The understanding of finite state machines is essential for grasping the main concepts of the approach presented in this work. Finally, it will be presented a brief introduction to the principles of software testing and program analysis. Software testing is important because represents the natural environment of this work.

2.1 Web Application

A web application is an application that is accessed through a network. The commonly used network in relation to web applications is the Internet. A web application follows the client-server architecture and its software it is composed by two main parts. The first part is coded in a browser-supported language and in a browser-rendered markup language and therefore it is reliant on a web browser in order to be executable. The second part is composed by the

software that is running on an application server. The second part of the web application is usually the component which also generates the first part of the web application. Web applications are popular due to the ubiquity of web browsers, and the convenience of using a web browser as a client. The use of a web browser and therefore the ability to update and maintain web applications without distributing and installing software on potentially thousands of client computers is a key reason for their popularity, as is the inherent support for cross-platform compatibility (3). In Figure 1 it is possible to see the browser interface of a web application used in the evaluation chapter of this work.

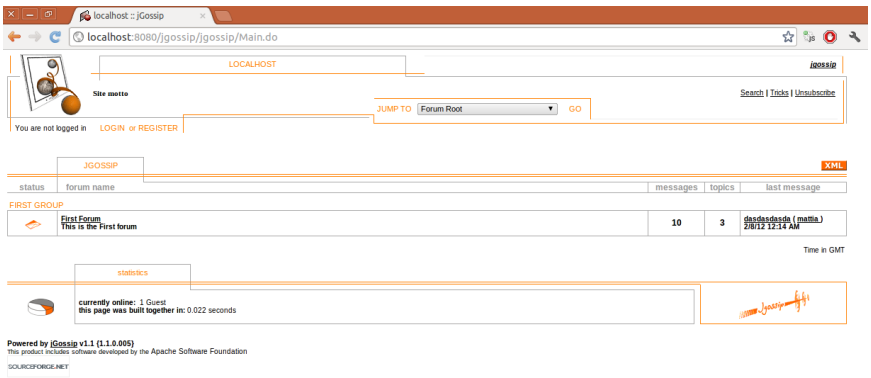

Figure 1. Web application browser interface.

2.1.1 Web Application Architecture

Web applications are usually divided into logical parts. These parts are also known as tiers and to each tier it is assigned a specific role. The most common structure for a web application is a three-tiered structure. In its most common form, the three tiers are called presentation tier, logic tier and data tier. The presentation tier of the web application it is located in its front-end component, while the logic tier is located in its back-end component and the data tier is located in its back-end data component. The front-end component is commonly know as client, the back-end component is commonly known as server and the back-end data component is commonly known as database. Figure 2 shows the partitioning of a web application into its three components.

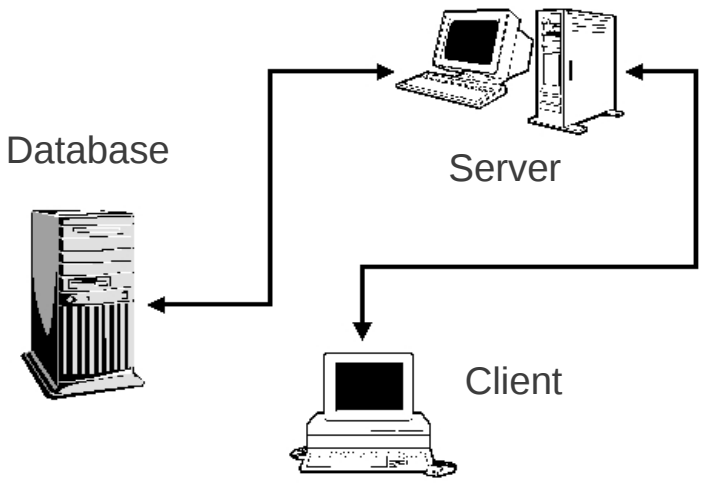

Figure 2. Web application and its three main components.

2.1.2 Web Application Working Mechanism

The possibility of user interaction and all the aspects related to it are one of the main characteristics of a web application. User interaction relies on the web application working mechanism. The following is the general and high-level working mechanism of a web application. The user, through the interface of a web application, can express the intention of using one of the functionalities of the web application. This intention of using a functionality is translated into a HTTP request. The HTTP request is then sent to the server which processes it using the server web application code and possibly by interacting with the database. The result of the processing task is a HTTP response. The HTTP response includes the content of the functionality required by the user. This HTTP response is then sent to the web browser and its content is made accessible to the user through the usage of the web browser. Figure 3 illustrates the working mechanism of a web application.

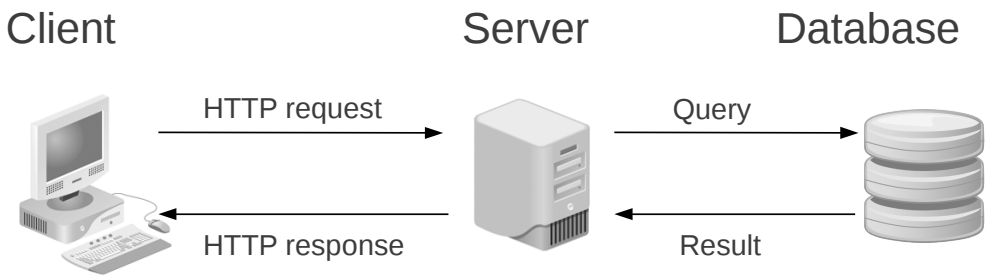

Figure 3. Web application working mechanism.

2.1.3 HTTP Protocol

As written in (4), the hypertext transfer protocol is the communication protocol which is used to access the World Wide Web. This protocol is widely used by all the common web applications. This protocol has been originally developed to retrieve static text-based resources and since then it has been extended in order to support a wider range of applications, including web application. HTTP is strongly connected to the client-server architecture of web applications. In fact, HTTP uses a message-based model in which client sends request messages to the server and the server returns response messages as answer to request messages.

2.1.3.1 HTTP Request

An HTTP request consists of one or more headers followed by a mandatory blank line and possibly by a message body. In Figure 4 it is possible to see an example of HTTP request.

```
1 GET foo.jsp?foo=bar HTTP/1.1
2 Accept: image/gif, image/jpeg
3 application/xshockwaveflash, application/vnd.msexcel,
4 application/vnd.mspowerpoint, application/msword, */*
5 Referer: http://site.com/foo.jsp
6 Accept-Language: en-gb,en-us;q=0.5
7 Accept-Encoding: gzip, deflate
8 User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
9 Host: site.com
10 Cookie: lang=en;
```

Figure 4. Example of HTTP request.

Some of the key elements of a HTTP request, extracted from (4) are: the *verb* indicating the HTTP method, the *requested* URL, the *Referer* header, the *Host* header and the *Cookie* header. Example of HTTP methods are *GET* and *POST*. In case the *GET* method is used, there will be no further data after the blank line following the message header. The *requested* URL represents the name of the resource which is requested through the HTTP request. In addition to the name of the resource, inside the *requested* URL, there might be a query string containing parameters that the client is passing to that resource. The query string starts from the symbol *?* and it is followed by parameter value pairs in the form *param=value*. Parameter value pairs are separated from each other through the & symbol. The *Referer* header is used to express where the request has been originated from. The *Host* header, instead, is used to specify the host name that appeared in the full URL being accessed. This part of the request results to be necessary in case multiple web sites are hosted on the same server. Finally the *Cookie* header is used to submit additional parameter that the server has previously issued to the client.

2.1.3.2 HTTP Response

A HTTP response, as it was happening for a HTTP request, consists of one or more headers followed by a mandatory blank line and possibly by a message body. In the case of a HTTP response, even if not precisely stated by the protocol, the message body is almost always present in the message. In Figure 5, it is possible to see an example of HTTP response. Some of the key elements of a HTTP response, extracted from (4) are: the numeric *status code*, the *Set-cookie* header, the message body, the *Content-Type* header and the *Content-Length* header.

The numeric *status code* indicates the outcome of the response result to a given request. The *Set-cookie* header is signaling to the browser to add a new cookie to the protocol. This cookie will be sent back in the next requests of the client to given the server. The message body is the most important part of the HTTP response, in fact, it results to be in almost all the HTTP responses. Inside the *Content-Type* header, instead, it is written the type of content which has been used as body of the message. Finally the *Content-Length* header expresses, in bytes, the length of the body of the message.

```
1  HTTP/1.1 200 OK
2  Date: Thu, 12 May 2012 13:49:37 GMT
3  Server: IBM_HTTP_SERVER/1.3.26.2 Apache/1.3.26 (Unix)
4  Set-Cookie: foo=bar
5  Pragma: no-cache
6  Expires: Thu, 01 Jan 2020 00:00:00 GMT
7  Content-Type: text/html;charset=ISO-8859-1
8  Content-Language: en-US
9  Content-Length: 24246
10
11 <!DOCTYPE html PUBLIC -//W3C//DTD HTML 4.01 Transitional//EN>
12
13 <html lang="en">
14 <head>
15 <meta http-equiv="Content-Type" content="text/html;
16 charset=iso-8859-1">
17 ...
```

Figure 5. Example of HTTP response.

2.1.3.3 URL

An uniform resource locator represents an unique identifier for a resource that can be found on the web. Through the usage of this identifier it is possible to retrieve its corresponding resource. In Figure 6 it is possible to find the URL format. Several components in this format are optional. The optional components are enclosed within square brackets. One of these optional components is the *port* number. In fact, the *port* number is usually expressed when is not used the default one.

```
1 protocol://hostname[:port]/[path/]file[?param=value]
```

Figure 6. URL format.

2.1.3.4 Cookies

One of the most important aspects of HTTP protocol are cookies. Web applications highly rely on this feature of the HTTP protocol. In fact, the cookie mechanism enables the server to send information to the client and automatically receive back these information. A cookie is created at server-side and continues to be automatically resubmitted in each subsequent request created by the client. A server can issue a cookie by using the *Set-Cookie* header of a HTTP response. After receiving the *Set-Cookie* header, the client automatically adds the content of

the header, i.e., the cookie value, to the related subsequent requests. In addition, the cookie mechanism allows to use multiple cookies at the same time. A cookie is mainly made by a variable-value pair. In addition to this it can have other properties. Some of these properties are: the *expires* property, the *domain* property and the *path* property. The *expires* property expresses until which date the cookie can be considered as valid. The *domain* property, instead, is used to know which is the domain associated to the given cookie. Finally, the *path* property specifies which is the URL path for which the cookie is considered as valid in a given domain.

2.1.4 Client-side

The client-side of a web application is represented by all the technologies that allow the user to input information and perform actions in order to interact with the web application. The common technologies which can be found on a client are HTML and JavaScript.

2.1.4.1 HTML

The most popular technology used to build the client part of a web application is HTML. HTML is a tag-based language which is used to structure the document that is shown to the user. HTML is capable to create complex and extremely functional user interfaces. Some of the key feature of HTML are hyperlinks, web forms and input fields.

2.1.4.2 Hyperlink

Hyperlinks are used from the user as an instrument for exchanging information between the client and the server part of a web application. Example of this is the possibility to exchange parameters between client and server. However, in most of the cases, these parameters are

not entered by the user but handled internally by the web application itself. An example of hyperlink which contains two parameters can be seen in Figure 7.

```
1 <a href="/item/showItem?id=05022012&lang=en">Show item!</a>
```

Figure 7. Hyperlink and parameter example.

```
1 <form action="/login.jsp" method="post">
2 Username: <input type="text" name="username"><br>
3 Password: <input type="password" name="password">
4 <input type="submit" name="submit" value="Login">
5 </form>
```

Figure 8. Example of web form and its properties.

2.1.4.3 Web Form

This part of a web application is in strong connection with the work presented in this thesis. In fact, web forms and input fields are the key elements for which the analysis, in the following chapters, is developed for. Web forms are the way to gather inputs from the users and send

these inputs to the server. A web form is usually made by several input fields. Input fields are the atomic elements into which it is possible to input information. There exists several type of input fields. The most common are: *text*, *password*, *submit*, *button* and *hidden*. An example of a web form can be seen in Figure 8. In this case, when the user clicks on the submit button, a HTTP request is sent to the server. This HTTP request will contain the values which have been inserted by the user into the username and password input fields. In addition, because *POST* method is used as method of the HTTP request, the input parameters will be sent to the server in the request body and not as parameter of a query string. The target URL of the request is the one expressed in the *action* property of the web form.

2.1.4.4 JavaScript

Hyperlinks and web forms can be used to gather most kind of inputs from the user and can give access to most of the functionalities of a web application. However, these type of elements can not perform complex processing of data. In fact, their processing functionalities are limited to the delivery of the data they handle to the server part of the web application. In case more complex processing of data is needed, JavaScript is the technology which comes into play. Complex processing of data is done mainly for two primary reasons. First of all, it can improve the application's performances. In fact, with JavaScript technology, certain tasks can be carried out entirely on the client component, without the need to make a round trip of request and response to the server. In addition, JavaScript, can enhance usability, because parts of the user interface can be dynamically updated in response to user actions, without the need to load an entirely new HTML page which is delivered by the server. Among the multiple

tasks that JavaScript can perform it is possible to find three main functionalities. The first one is to validate user input data. This task is performed before submitting data to the server in order to avoid unnecessary requests in case the data contain errors. This observation is in strong connection with the aim of this work. In fact, by detecting input validation inconsistencies, unnecessary requests can be totally avoided. The second operation is to dynamically modify the user interface. This task is performed as response to some user actions such as menu navigation and photo visualization. The third operation is instead to query the document object model in order to retrieve information which are then used in the logic of the user interface. In fact, some of the logic in the user interface could be related to other elements in the document object model.

2.1.5 Server-side

In modern web applications, the content presented to the user, is mostly generated dynamically. In fact, when an user requests a resource, this resource is usually created on the fly. The task of dynamically creating the content of these resources is performed by the server part of the web application. The server part of a web application is code which is running on a server machine. Usually, when the user submit a request for a dynamic resource, also a set of parameters is sent together with the request. These parameters enable the server part of the web application to generate a dynamic content which is usually tailored on the inputs which have been given by the user. In addition, this content could also depend on the interaction with a database. There are three main ways to submit parameters together with an HTTP request: they can be passed as values in the URL query string, they can be passed as HTTP cookies and

they can be part of the body of the request in case of using the *POST* method in the HTTP request. There are several technologies which could be employed as the server part of a web application. Because of the connection with this work it will be presented the JEE technology.

2.1.5.1 Java Enterprise Edition

Java Enterprise Edition is one of the most used platform to develop web applications. This technology is well suited for modular development and code reuse. The following are the most popular components of a web application developed through JEE technology. An *Enterprise Java Bean* is a heavyweight software components which encapsulate the logic of a specific business function of the web application. A business function contains the logic to cover a functionality of the web application. A *Plain Old Java Object* is instead a normal Java object used within the web application which does not reflect the characteristics of any other specific category of object. A *Java Servlet* is an object which resides on an application server and handles HTTP requests. The *Java Servlet* is also the object which takes care of returning an HTTP response to the client. Finally the *web container* is an engine that provides a runtime environment for all the Java based components which have been previously mentioned.

2.1.6 Web Application Security

Because of the large development of the web and its easiness of access, web applications result to be an easy target for attackers. For this reason, web application vulnerabilities have to be taken into serious consideration. A web application vulnerability is a software flaw which allows an attacker to reduce the system's information assurance (5). There are several type of web application vulnerabilities, however, the most interesting, in relation to this work, are the

input-based vulnerabilities. This type of vulnerabilities arises because web applications need to be able to handle different types of user input and these inputs are usually of unknown nature. The main principle adopted when dealing with input, inside the code of a web application, is that all user inputs need to be considered as unsafe. More in detail, it is not possible to trust user input. Input validation is one of the technique in order to ensure that a given input does not present an attack to the web application. Input validation is one of the main topics of this work and its definition is given in Chapter 3. A great variety of attacks against web applications can be achieved by submitting unexpected inputs to the web applications. Example of these attacks are SQL injection and cross-site scripting.

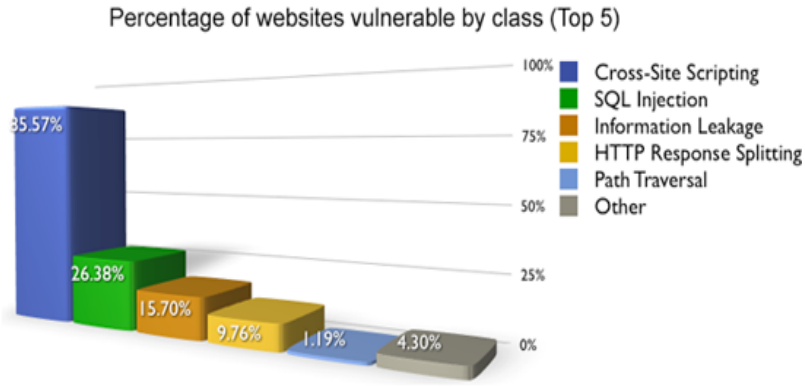

Figure 9. Classification of the most popular web application attacks.

Figure 9 shows that, up to 2007 (6), the two type of previously mentioned attacks were among the most popular attacks to web applications. As reported form (1), in 2010, injection flaws are the most popular type of attacks followed by cross-site scripting.

2.1.6.1 SQL Injection

SQL injection is part of a category of attacks called injection flaws. SQL injection attacks occur when user-supplied data are sent to the SQL interpreter as part of a command or query. An attacker can take advantage of this situation by tricking the interpreter to execute unintended commands (6). The execution of this type of attack can bring the attacker to delete important data or to escalate his privileges to the ones of the administrator.

2.1.6.2 Cross-site Scripting

Cross-site scripting attacks can be performed whenever a web application takes the user input and sends it to a web browser without first validating or encoding its content. Cross-site scripting attacks allow attackers to execute scripts in the victim's browser. The execution of these scripts can lead to multiple problems such as hijacking of user sessions, defacing of web sites or possibly introducing worms (6).

2.2 Automata Theory

Automata theory is the field of theoretical computer science which aims to study mathematical objects called abstract machine or automata (7). In addition, automata theory studies the computational problems which might be solved by these mathematical abstractions. Automata theory is also related to the field of formal languages. In fact, an automaton is a finite representation of a formal language. In formal language theory there are several classes of formal

languages and for this reason, there exist several types of automata. Each of these automata is able to recognize a different class of languages. The automata that are of interest for this work are deterministic finite automata, which belong to the class of finite state machines.

2.2.1 Deterministic Finite Automaton

A deterministic finite automaton is a finite state machine that accepts or rejects finite strings of symbols (8). These finite strings are accepted or rejected by this type of machine in a deterministic way. More in detail, this means that the automaton produces a unique computation while accepting or rejecting each of strings given as input. DFAs recognize all the languages which belong to the set of regular languages. A regular language is a language over the alphabet Σ which is defined recursively as follows: the empty language $\emptyset$ is a regular language, for each $a \in \Sigma$ the singleton a is a regular language, if A and B are regular languages then the union, $A \cup B$, and concatenation, $A \bullet B$, and A^* are regular languages and no other language over Σ^* is a regular language. DFAs are an equivalent model to non-deterministic finite automata because DFAs can be build from non-deterministic finite automata. In addition DFAs are equivalent to regular expressions as well. Even if DFAs are a mathematical abstraction they can be implemented in software. Because of this characteristic, an application of DFA can be found in relation with web applications. In fact DFA can determine whether an online user input for an email address is valid or not (8). This example strongly motivate the approach presented in Chapter 5. A DFA can be analyzed under several perspectives. The most popular perspective are the graphical and the mathematical ones. An example of graphical representation of a deterministic finite automaton can be seen in Figure 10. In general, a DFA is composed by

states and transitions between states. In the graphical representation of Figure 10, the states are graphically illustrated by circles while transitions are represented by arrows. There are three categories of states: normal, initial and accepting states. An initial state is a state from which the recognition of strings begins. An accepting state determines, instead, if an input string can be accepted or not. In Figure 10 state number 1 is the initial state while state number 3 is the accepting state. Transitions are the connections between states and each of them is related to a label. Each of the labels, usually, contains a character which is used by the computation mechanism in order to decide in which state to move after reading a character from the input string. Sometimes, more complex representations are used for labels. For example, in the case of this work, a range of characters is used as label of a given transition.

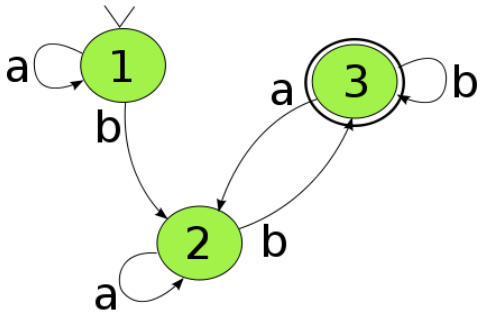

Figure 10. Deterministic finite automaton representation.

In case DFA are analyzed under the mathematical perspective, a deterministic finite automaton can be represented by a tuple composed by five elements $\langle Q, \Sigma, \delta, q_0, F \rangle$. Q represents the set of the states of the automaton, Σ is the finite set of input symbols, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function which expresses the transition between states, q_0 represents the initial state and F represents the set of accepting states.

2.2.2 Regular Expression

As said in the previous part of the chapter, regular expressions and deterministic finite automata are equivalent representations. In fact, a regular expression provides a way to specify and recognize strings (9) that belong to a regular language. More in detail a regular expression is an expression that specifies a set of strings. A regular expression is built by using some basic operations and tokens. Tokens represent the string literals used in the regular expression. Among the operators it is possible to find: the boolean or, the round brackets, ?, + and $*$. The boolean or, represented by the symbol |, is used to separate alternatives. Round brackets (*and*) identify groups and are used to define the scope and precedence of operators. ? is a quantifier which is used after a token or a group and specifies that the preceding element has to occur zero or once. + is a quantifier as well and it is used after a token or a group and specifies that the preceding element has to occur once or more times. Finally, $*$ is a quantifier which is used after a token or a group and specifies that the preceding element can occur any number of times.

2.3 Software Testing

The general aim of software testing is to assess and ensure a certain level of quality for the properties of a software (10). There is no perfect or best testing technique in the field

of software testing. In fact, all the techniques are inside a complex space of trade-offs and have complementary strengths and weaknesses. For this reason the technique or techniques to employ during software testing always depend on the situation and on the constraints of the problem to solve. Software testing techniques can be applied at any time in the development process. In addition, has been shown that applying software testing techniques in the early part of the life-cycle of a software results to be more effective and less expensive. As said before, software testing contains a variety of techniques. Some of them are used to determine if a software meets the requirements which have guided its design and development and some other techniques aim to improve the software by finding its defects. In order to find software defects, program analysis might be employed. In addition, based on the possibility of having or not the source code of a software, white-box or black-box testing might be employed.

2.3.1 White-box Testing

White-box testing is a technique of software testing which tests the internal structures of a software as opposed to the act of testing its functionalities (11). In general, white-box techniques, while performing their testing activities, have direct access to the source code of the software. White-box testing can be applied at several levels of the software structure. It can be used to test single units of the software but it can be also used to perform testing at a system level. White-box testing has also some drawbacks. In fact, it might not detect missing requirements.

2.3.2 Black-box Testing

Black-box testing is a technique of software testing which tests the functionalities of a software as opposed to the act of testing its internal structures (12). In general, black-box testing performs its testing activity without having the need of accessing the software source code. In fact, this technique is only aware of what the software is supposed to do and not how it actually does. In general, black-box testing is used for testing activities on the high level part of a software system. However it is also possible to use this technique for testing the single unit of a software system.

2.3.3 Program Analysis

Program analysis is a technique used in order to automatically analyze the behavior of a computer software. There are two main approaches to program analysis: static and dynamic program analysis. Static program analysis has the advantage that the whole code base is checked, but it has the drawback that the analysis may report warnings for correct code. Such false positives quickly undermine the trust into the correctness of a tool. Hence, when performing static program analysis, it is crucial to perform precise analysis. In addition, static code analysis relates to this work because it is the technique used to analyze server-side input validation functions. Dynamic analysis techniques, instead, perform their computations while actually executing the analyzed code. As a result, dynamic analysis does not generate false positives, since every detected path corresponds to a true path that the program can take at runtime. The disadvantage of dynamic scanners is that they experience problems regarding the coverage of all possible paths through the program. The number of executable paths is

generally unbounded, and grows exponentially with each branch in the program. Hence, it is easy to miss important information about program defects due to program paths that were not taken into account.

CHAPTER 3

PROBLEM STATEMENT

Given the introduction of Chapter 1, it is now necessary to detail some concepts in order to properly define the problem domain which has been tackled in this work. As it is possible to understand from the title of this work, the objective is to detect and correct inconsistencies among client- and server-side input validation functions within the context of web applications. An input validation function is a function which aims to check whether the value, given by the web application user, for a generic input field, is a valid value or not. A valid input is an input value that is within the range of expected values from its related web application. Input values, in most of the cases, can be represented by strings. Examples of these strings are values corresponding to usernames, passwords or emails. In order to validate an input field, the validation functions operate checks on the value representing the input. These checks, in most of the cases, are string operations. These string operations verify and express constraints on the possible values that the string can assume. If the constrains are not satisfied, then the input value is rejected. For this reason there are two possible outcomes when an input validation function is applied to a given input field. The input field value is accepted or rejected. Input validation function can be found on both the client- and server-side of a web application. The key idea of this work is that, for the most general cases, input validation functions should perform, for the same input field, the same checks on both the client- and the server-side. If this is not the case an inconsistency arises. An inconsistency is therefore a difference in the string

operation checks between client- and server-side input validation functions. If an inconsistency is present, one of the two sides will accept a wider range of string values if compared to the other. This situation can lead to two serious problems as mentioned in Chapter 1. If, for an input field, the client-side accepts a wider range of values than the server-side, then there is an efficiency problem among the web application. In fact, in this case, a value will be sent, through an HTTP request, from the client part of the web application to its server side. However, due to the situation, the server-side will reject this value and therefore the request containing the input field value will be not processed. In addition, in most of the cases, the server will communicate this failure to the web application user generating a HTTP response which will go through the Internet again. It is straightforward to understand that these communications could be easily avoided if the the two type of validation functions where accepting the same range of inputs. These communications lead to two categories of problems. The first category is represented by an increased number of requests to the server, while the second one is represented by an increased load of the network. If, instead, for an input-field, the server-side accepts a wider range of values than the client-side, then this inconsistency can lead to a security vulnerability of the web application or to application failures. This type of inconsistencies, might be exploited by a malicious user by disabling client-side validation, which results to be possible usually by disabling JavaScript functionalities on the web browser. Disabling client-side validation allows a malicious user to directly send unexpected input values to server part of the web application. These unexpected values could then lead to attacks toward the web application. Classes of this type of attacks, as said in Chapter 1, are SQL injection, XSS or buffer overflow. These

two type of inconsistencies and their related problems motivate this work. In fact, this work aims to detect and solve these inconsistencies. In the next section, two motivating examples are presented in order to further explain the problems which could arise from these type of inconsistencies and to concretely motivate why this work results to be useful in detecting and solving these inconsistencies.

3.1 Motivating Examples

In order to motivate and further explain this work, this section presents two examples of inconsistencies which could appear in real-world web applications. The first example is extracted from a web application called *JGossip*, which is a message board application written using Java technology. Figure 11 provides a high-level, intuitive view of the web application.

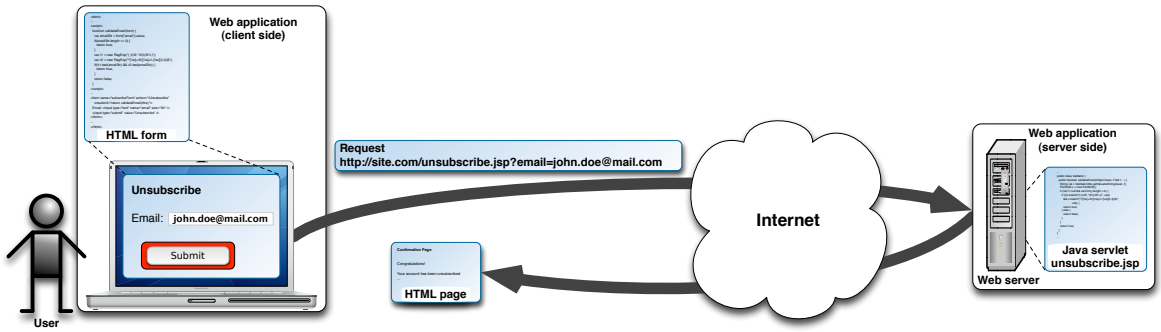

Figure 11. High-level of *JGossip* motivating example.

The parts of the web application shown allow users to unsubscribe their email address by entering it into a form and submitting it to a web back-end hosted on a server at `site.com`. `site.com` is the name used to represent the server location of the web application. As explained in Section 2.1, in order to access a functionality of the server side, the client side issues an HTTP request that contains a set of input elements expressed as $\langle name, value \rangle$ pairs. When this happens, the different input elements are marshaled, i.e., packaged together, and the resulting bundle is passed to the server-side code as an input. The server code then accesses these inputs by name by invoking library functions provided by the language or framework used. These functions parse the HTTP request containing the inputs and return the values of the requested input. The server then processes the retrieved input values, normally performs some form of input validation and then uses them in possibly critical or sensitive functionality of the web application. Figure 12 and Figure 13 show two snippets of client- and server-side validation code, respectively, from *JGossip*. These snippets are the ones used in the first motivating example. Please note that the code has been slightly simplified to make it more readable and self-contained. In the functionality shown in Figure 11 the user should fill the client-side form, shown on lines from 18 to 22 of Figure 12, by providing an email address to the HTML input element with name *email* and by clicking on the submit button. When this button is clicked, the browser invokes the JavaScript function *validateEmail*, which is assigned to the *onsubmit* event of the form. This function first fetches the email address supplied by the user from the corresponding form field. It then checks if this address has zero length and, if so, accepts the empty address on line 6. Otherwise, on lines 9 and 10, the function creates two

regular expressions. The first one specifies three patterns that the email address should not match: a single space character, a string with the @ symbol on both ends, and the string *"@."*. The second one specifies a pattern that the email address should match: start with a set of alphanumeric characters, followed by symbol @, further followed by another set of alphanumeric characters, and finally terminated by a dot followed by two to four additional alphanumeric characters.

```
<html>
...
<script>
  function validateEmail(form) {
    var emailStr = form["email"].value;
    if(emailStr.length == 0) {
      return true;
    }
    var r1 = new RegExp("( )|(@.*@)|(@\\.)");
    var r2 = new RegExp("^[\\w]+@([\\w]+\\.[\\w]{2,4})$");
    if(!r1.test(emailStr) && r2.test(emailStr)) {
      return true;
    }
    return false;
  }
</script>
...
<form name="subscribeForm" action="/Unsubscribe"
    onsubmit="return validateEmail(this);">
  Email: <input type="text" name="email" size="64" />
  <input type="submit" value="Unsubscribe" />
</form>
...
</html>
```

Figure 12. Client-side validation code in first motivating example.

If the email address does not match the first regular expression and matches the second one, this function returns *true*, indicating acceptance of the email address, line 12, and the form data is sent to the server. Otherwise, the function rejects the email address by returning false on line 14. This results in an alert message to inform the user that the email provided is invalid. When the form data is received by the server, it is first passed to the server-side validation function, before it is processed. For the specific form in this example, method *validateEmail*, shown in Figure 13, from the *Validator* class is used.

```
1 public class Validator {
2   public boolean validateEmail(Object bean, Field f, ..) {
3     String val = ValidatorUtils.getValueAsString(bean, f);
4     Perl5Util u = new Perl5Util();
5     if (!(val == null || val.trim().length == 0)) {
6       if ((!u.match("/( )|(@.*@)|(@\\.)/", val))
7         && u.match("/^[\\w]+@([\\w]+\\.[\\w]{2,4})$/",
8                    val)) {
9         return true;
10      } else {
11        return false;
12      }
13    }
14    return true;
15  }
16 ...
17 }
```

Figure 13. Server-side validation code in first motivating example.

This method calls a routine on line 3 to extract the value contained in the email field from the form object, *bean*, and stores it in variable *val*. It then uses library *Perl5Util* to

perform the regular expression match operations, which allows for using the same Perl style regular expression syntax used in the client. First, the method checks whether the email string is *null* or has zero length after applying the *trim* function, on line 5. If so, it accepts the string. Otherwise, it checks the email address using the same regular expressions used on the client side. As shown on lines from 6 to 12, the address is accepted if it satisfies these regular expression checks, and it is used for further processing on the server side, e.g., it may be sent as a query string to database. Otherwise it is rejected, and the user is taken back to the form by a response message sent by the server. This example is interesting for two reasons. The first one is because the regular expression checks are similar on both ends and therefore it is clearly shown that the intention of validation is to allow or reject the same set of input on both sides. The second reason is because, even if the intention is to have the same validation on both sides, the example has an inconsistency. There is an inconsistency because the server-side validation function accepts a wider range of values for the *email* input field. In fact, the client-side validation program shown in Figure 12 rejects a sequence of one or more white space characters, e.g., the string " ", for which the condition on line 6 evaluates to *false* and the regular expression check on line 11 fails, thereby resulting in the function returning *false* and hence not accepting the input value. However, for the same input, the second condition on line 5 of the server-side validation method, Figure 13, evaluates to *false*, due to the *trim* function call, and the string is therefore accepted by the server. This would lead to white spaces being accepted as email addresses by the server, which might in turn lead to failures, e.g., the web application might try to send an email to the user, which would fail due to an invalid email

address. Avoiding this type of inconsistencies would for sure improve the general behavior of the web application. The second example has been extracted from a web application called *Tudu* which has been also used in the evaluation chapter of this work. *Tudu* is a web application for managing to do lists. Figure 14 shows, from a high-level point of view, the functionality of the web application taken into consideration in this example.

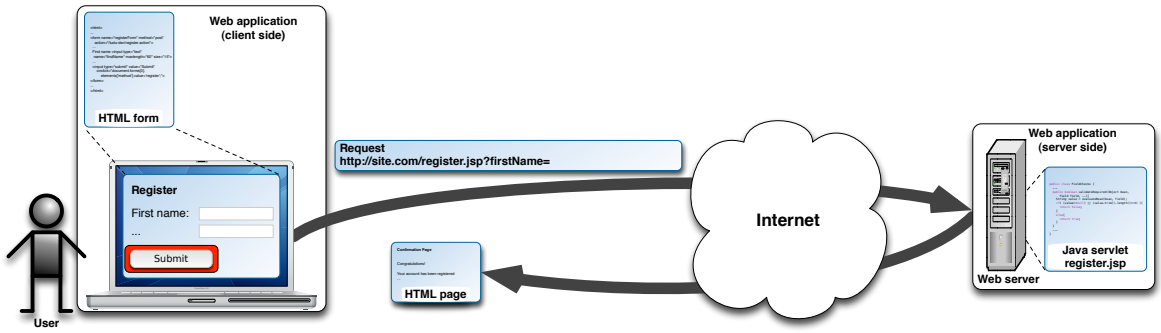

Figure 14. High-level of *Tudu* motivating example.

In this example the user has to complete the registration process by filling personal data into a web form. Among the data of the web form there is the possibility to insert the first name of the user, which is the input field considered in order to show its validation process. As it was happening in the case of the previous example, the input fields which have been inserted into the web form, are passed to the server-side of the web application expressed as $\langle name, value \rangle$

pairs. These pairs are passed to the server-side after the user expresses a submission request, usually done pressing the submit button. Before being actually submitted, the input field might be validated by the client side, and if the validation process is successful the $\langle name, value \rangle$ pairs are passed to the server through an HTTP request. The server processes this request reading its contained input values. After reading the input values, these values are validated and then used in possibly crital functionalities of the web application. Figure 15 and Figure 16 show the snippets of validation code for the input field *firstName* respectively at client- and server-side. The code, as it has been done in the previous example, has been simplyfied in order to be self-contained. Figure 15 represent the HTML code related to the client side-part of the web application.

```
1  <html>
2  ...
3  <form name="registerForm" method="post" action="/tudu-dwr/register.action">
4    ...
5    First name <input type="text" name="firstName" maxlength="60" size="15">
6    ...
7    <input type="submit" value="Submit"
8          onclick="document.forms[0].elements['method'].value='register';">
9  </form>
10 ...
11 </html>
```

Figure 15. Client-side validation code in second motivating example.

As it is possible to see there is no validation code for the *firstName* input field of line 5. For this reason, the user can submit any value for this input field and the request will be sent in any case to the server part of the web application. The user can also decide to not input any value for the input field and the request will be sent to the server anyway. When the user presses the *Submit* button of line 7 and 8 the HTTP request is sent to the server side of the web application and therefore it makes a communication over the network. When the server receives the web form it passes the form content to the server-side validation functions. For the input field which have been considered in this example, method *validateRequired* from *FieldChecks* class is used as validator. Its code can be seen in Figure 16.

```
1 public class FieldChecks {
2   ...
3   public boolean validateRequired(Object bean, Field field, ..){
4     String value = evaluateBean(bean, field);
5     if( (value==null) || (value.trim().length()==0) ){
6       return false;
7     }
8     else{
9       return true;
10    }
11  }
12  ...
13 }
```

Figure 16. Server-side validation code in second motivating example.

The method calls a routine on line 4 which extracts the value for the *firstName* input field and stores this string value in variable *value*. The validation function then performs two checks

on the string value. These checks are performed on line 5. The first check tests if the string value is equivalent to *null*, meaning that the method checks whether a value has been given at client side to the *firstName* input field or not. The second check tests if the string value, removed the initial and final white spaces, has length equal to zero. If one of the two tests is successful then the input value is rejected. In case of rejection, a HTTP response is sent back to the client-side of the web application saying that the first name input field is a required input field. In case both tests are not successful, the input value is accepted and it is further processed in the logic of the web application. This example shows an inconsistency between the client- and server-side validation functions. In fact, if the user decides not to input any value for the *firstName* input field of the web form, the form will be submitted to the server part of the web application. Once the *firstName* input field of the web form reach the server-side of the web application it is validated against *validateRequired* function and its *null* value is rejected. The server then generates an HTTP response saying that the value for *firstName* input field is not valid. This is a type of inconsistency where the client-side of the web application is accepting a wider range of values with respect to the server-side. This situation brings the server to receive a HTTP request which could be avoided, and the network is loaded by a HTTP request and a HTTP response which could be prevented as well. Detecting and correcting this type of inconsistencies could therefore improve application responsiveness and decrease the network load.

CHAPTER 4

RELATED WORK

This work aims to detect and correct inconsistencies among client- and server-side validation functions of web applications. These inconsistencies can lead to software issues, such as software faults and software vulnerabilities, and can decrease performances of web applications. In order to detect software faults and software vulnerabilities of web application, dynamic and static program analysis techniques can be employed. Both dynamic and static program analysis relate to the approach presented in this work. For this reason, it will be presented how, in previous works, dynamic and static program analyses have been employed in both the client and server part of of a web application. The chapter aims to cover also the state of the art of static string analysis and parameter tampering opportunities detection. In addition, throughout the chapter it will be presented how this work is positioned with respect to the previous work and which are the improvements and novelties introduced.

4.1 Dynamic Analysis of Web Applications

Example of works which try to identify web application vulnerabilities through dynamic program analysis of the server side part of a web application are (13), (14). In (13) the authors propose an automated approach for protecting web applications against SQL injection. In this work, the approach is based on the idea of positive tainting and the concept of syntax-aware evaluation. Positive tainting is a dynamic tainting technique which identifies and marks

trusted and untrusted data during the SLQ injection analysis. Syntax-aware evaluation instead considers the context in which trusted and untrusted data can be used. (14) presents an approach to harden web application against common vulnerabilities. The approach in the paper is based on the idea to precisely track taintedness of data and check specifically for dangerous content only in parts of commands and output that came from untrustworthy sources. In addition, the approach, instead of tainting everything which is derived from tainted input, precisely tracks taintedness within appropriate data values. These two works, (13) and (14), identify vulnerabilities by analyzing only the server part of a web application. This work, instead, aims to detect and correct inconsistencies which could lead to these type of vulnerabilities by taking into consideration also the client part of a web application and not just its server part. Even if dynamic program analysis might present some shortcomings, it has been widely used for analyzing the client side part of a web application. This is because JavaScript, the most used language for client-side programming, is notoriously difficult to analyze statically due to its highly dynamic and loosely-typed nature (15). For example in (16) the authors use dynamic slicing on JavaScript code of client-side input validation functions in order to extract the string operations made on the input input field of a web application. (16) is highly related to the approach presented in Chapter 5 because it is the technique employed to analyze client-side input validation functions and because it is explained how to translate string operations into a representation which make possible to analyze the values which an input field can assume. In (17), the authors propose dynamic analysis techniques to systematically discover client-side validation vulnerabilities a new class of emerging web application vulnerabilities. The technique

in (17) is related to this work and to (16) because of the similarities in the client-side validation function extraction process.

4.2 Static Analysis of Web Applications

Xie and Aiken (18) addressed the problem of statically detecting SQL injection vulnerabilities in PHP scripts by means of a three-tier architecture. In this architecture, information is computed bottom-up for the intra-block, intraprocedural, and interprocedural scope. As a result, their analysis is flow-sensitive and interprocedural. This is comparable in power to Pixy (19). Both systems use traditional data flow analysis to determine whether unchecked user inputs can reach security-sensitive functions, also known as sinks, without being properly checked. However, they do not calculate any information about the possible strings that a variable might hold, which is instead done in this work. Thus, they can neither detect all types of vulnerabilities, such as subtle SQL injection bugs nor determine whether sanitization routines work properly. In addition these approaches relate to this work because they perform data-flow analysis which will be taken into consideration in Section 5.1.

4.3 Static String Analysis

Static string analysis is one of the approach to determine the possible values that a string variable may hold at a given program point. An important work which performs static string analysis of Java programs is given by Java String Analyzer (20). In this work the authors perform static string analysis by analyzing *.class* file and from these file a context-free grammar is generated for every string expression. The context-free grammar is then widened into a regular language by using an algorithm previously employed for speech recognition. The collection of

resulting regular languages is compactly represented as a special kind of multi-level automaton from which information about the string values at a given program point might be extracted. The authors then show example of the application of this technique in SQL queries analysis. This technique relates to the approach presented in this work because both of them perform static string analysis. The common ideas of the two works are that they both extract representations of all possible string values for a variable at a given program point and that they both map string operations to regular expression. However, among the differences, there is the fact that the string analysis technique presented in this work is path-sensitive while Java String Analyzer has low support for path-sensitivity. Static analysis of strings has been an active research area in the context of web applications. The goal is to find and eliminate security vulnerabilities and software defects caused by misuse of string variables. The knowledge about the content of string variables, in fact, can be leveraged to eliminate vulnerabilities such as SQL injection and XSS attacks. In (21), multi-track DFAs, known as *transducers*, are used to model replacement operations in conjunction with a grammar-based string analysis approach. The resulting tool has been effective in detecting vulnerabilities in PHP programs. Wassermann et al. (22) and (23) propose grammar-based static string analysis to detect SQL injections and XSS, following Minamide's approach. A more recent approach in static string analysis has been the use of finite state automata as a symbolic representation for encoding possible values that string expressions can take at each program point (24) and (25). This is the same type of representation that has been used in this work in order to express the values that a string can assume at a given program point. Constraint-based, or symbolic-execution-based, tech-

niques represent another approach for static string analysis. Such techniques have been used for the verification of string operations in JavaScript (26) and the detection of security flaws in sanitization libraries (27).

4.4 Parameter Tampering

NoTamper (28) and WAPTEC (29) represent approaches which are in close connection to this work. In fact, this work can be placed in a complementary position with respect to these two approaches. NoTamper (28) presents an approach to automatically detect potential parameter tampering opportunities. Parameter tampering opportunities possibly expose the server to attacks such as SQL injection and Cross-site scripting attacks. The technique presented in (28) is based on white-box analysis on the client side and black-box analysis on the server side of a web application. The key idea of the work is that the input validation checks done on the server side should be at least as strict as the checks done on the client part of the web application. For this reason, the authors state that it is possible to consider the validation checks done in the client part of a web application as a specification for the validation checks done in its server part. This means that if the server side does not perform at least the same checks as the ones done in the client side there might be a parameter tampering opportunity in the web application. In order to verify if client side checks are done also on the server side of a web application, the approach in (28) starts by extracting constraints on the possible values of the input fields of a web form from the client part of a web application through concrete-symbolic execution. The collected constraints correspond to the validation checks on the inputs on the client side. From these constraints, using logical tools, hostile and benign input values are

generated. Hostile input values are input which are rejected from the client side of the web application and should be rejected also from its server side. Benign input values instead are inputs which are accepted by the client side of the web application and should be also accepted by its server side. Hostile and benign inputs are then sent directly to the server-side of the web application and the server responses are collected. If, for the same input field, hostile and benign values produce the same server response then a potential parameter tampering opportunity is detected. A potential parameter tampering opportunity is detected because the server should generate different type of responses for hostile and benign inputs. This set of inputs and the parameter tampering opportunities report are then shown to the developer which has the task to verify the results. WAPTEC (29) is an improvement of the technique presented in (28). Like in (28), WAPTEC aims to identify parameter tampering opportunities, i.e., input fields for which there is at least one value that the client rejects but the server accepts. WAPTEC uses white-box program analysis for both the client and server sides of a web application. The approach presented in (29) it is made by two main steps. The first one aims to find constraints of paths on the server side that if taken, result in the input being accepted, i.e., the input reaches a sensitive sink. The second step aims to find inputs which are rejected from the client part of the web application but reach a sensitive sinks on the server part, identifying in this way parameter tampering opportunities. Like in (28), the key idea of the work is to use client input validation checks as a specification of their server counterparts. For this reason WAPTEC starts by extracting input constrains from the client-side code. These constrains correspond to input validation checks done on the client part of the web application. Once these constraints

have been found, the approach uses its constraint solver to generate input values which satisfy these constraints, i.e., input values which are accepted by the client. After the generation of these input values, each of them is used as input of the server part of the web application and the corresponding execution traces are analyzed. The analysis of the execution traces allows to find if an input has reached a sensitive sink on the server side. If an input reaches a sensitive sink, it means that a benign input is found and the constraints of the path which has lead to the sensitive sink are stored in the constraint set of the server. The server constraint set contains all the constraints which lead to the acceptance of an input field on the server side. If the input reaches a sensitive sink, in addition, nearby sensitive sinks and their related path constraints are searched by negating some of the constraints of the the path which has lead to the first sensitive sink. If any sinks are found in this way, the constraints of the executed paths are added to the server set of constraints. If the input does not reach a sensitive sink the constraints of the path in the execution trace are extracted and their negated version is added to the constraints of client creating in this way an augmented set of constraints of the client. This augmented set of constraints is then used to generated values in order to find inputs which lead to a sensitive sink on the server side. Once the process of finding the set of constrains on the server side of the web application is over, then the approach tries to find parameter tampering opportunities by generating hostile inputs from the negation of the constraints of the client side an see if they reach sensitive sink in the server part of the application, i.e. these hostile inputs satisfy the set of constraints on the server. If such an input is found then a parameter tampering opportunity is detected. The idea of having client-side code as specification of the server-side is the same

as the one adopted in this work. This work, in addition, uses the server-side as a specification of the client-side. In (29), the authors state that might be possible to find cases where some server-side checks can not be performed by the client part of the web application because the server part has access to more information with respect to the client part, i.e., information that depend on the server state of the web application. Even in this work it is believed that this type of information can not be found at client-side and in fact they are not considered in the validation process. In addition, as far as it is possible to understand form the literature which has been analyzed, this work results to be the first one which presents an approach to positively apply the idea of using the server-side as a specification of the client-side. If the idea of having the server-side as a specification of the client-side would be used following the approach illustrated in (28) and (29) in order to solve the client-side inconsistency problem illustrated in Chapter 3, there could be some difficulties. These difficulties arise in case that client specification would be totally missing. In fact, in this case, the generation of benign input to find sinks at the server side might have a search space which might result to be too big in size. This assume more importance considering that the case in which client side-validation is missing unfortunately appears with high frequency in real world web applications. In addition, to perform server-side analysis, the authors have to transform the server-side code of the web application in order to extract the traces of executed statements. In the approach proposed in this work, because it has been adopted static program analysis to analyze the server, there is no need for instrumentation and therefore the server part of the application can be analyzed as it is. Finally, even if it might be possible, the authors of (29) do not propose any mechanism

to automatically generate code correction for the problems they detect. In this work, for both client- and server-side inconsistencies, it is possible to generate the related code corrections.

CHAPTER 5

APPROACH

Like it has been discussed in the previous chapters, the basic intuition behind this work is that redundant checks in the client- and server-side input validation functions of web applications can be leveraged in order to detect inconsistencies between the two. Like it has been said before, it is important to detect and correct these inconsistencies because they may lead to security vulnerabilities, inefficiencies, and general misbehaviors. More precisely, this approach uses the checks performed by the client as a specification to verify the checks performed by the server and vice versa. Server-side input field checks should never be less strict than their client-side counterparts, as client-side checks can be bypassed by malicious users to take advantage of these types of inconsistency. In fact, less strict server checks may lead to common web application vulnerabilities that the Open Web Application Security Project considers to be of very high severity and with a very high likelihood of being exploited (30). Analogously, it is possible to have client-side checks which are less strict than server-side checks. This could lead to unnecessary network communications and computation on the server side. Adding such checks can therefore improve the responsiveness of web applications and reduce the overall network load. In order to identify these types of inconsistency and solve them, the approach:

1. Extracts input validation functions from both client and server sides.
2. Models input validation functions as DFAs.

3. Compares DFAs in order to identify, report and correct inconsistencies.

Figure 17 provides a high-level view of the three steps that the approach aims to perform.

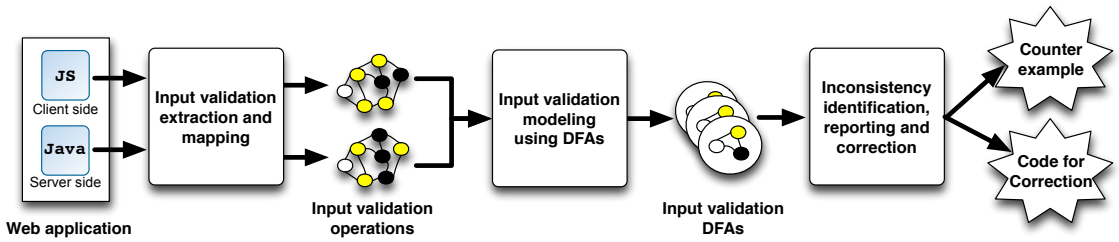

Figure 17. Overall approach schema.

5.1 Input Validation Extraction

The goal of this phase of the approach is to automatically extract and map to each other, the snippets of input validation code, for a given input field, from both the client and the server sides of a web application. This analysis presented in this phase aims to applicable in most cases as possible, however, because web applications can be developed using a wide number of languages and technologies, some of the concepts illustrated in the following sections might be more suitable for the technologies and languages which have been taken into consideration. As far as languages are concerned, the target of this work, are the web applications that use

Java for server-side input validation and JavaScript for client-side input validation. It has been chosen to focus on these languages because they are widely used languages for development of web applications. As far as technologies are concerned, the focus of this work is on web applications based on Java Enterprise Edition web framework (31). This technology has been selected because is is well known and widely adopted among web developers. Given a small introduction on which might be the most suitable technologies and languages for this phase of the approach, it will be now explained how to generally locate the source code of input validation functions, how to select the validation code and how to map client- and server-side extracted code. It is important to find the validation code and therefore the string operations expressed by the validation functions because form the string operations that are applied on a given input field, it is possible to build a representation of all the valid values for the specific input field. This representation and how to obtain it is the main topic of Section 5.2.

5.1.1 Input Validation Identification

Very first step of the approach is to identify which are the points, of a given web application, from where client- and server-side input validation functions could be extracted. Based on the specification of JEE framework (31) the most general approach, in order to gather this information, requires to analyze the JSPs and the configuration file of a given web application. From JSPs it is possible to extract information about web forms, input fields, their client-side validation and the servlet which is supposed to handle the requests generated from a given web form. Detailed information about the servlet can be then found inside the configuration file. JSPs and the servlet methods that handle web form requests can be considered as entry points of

the validation analysis. An entry point is therefore a method or the page into which it could be found validation code for a given input field. Summing up, entry points for server-side validation are represented by methods, while for client-side validation they are identified by JSPs and the client-side validation code related to their web forms. In the implementation of the approach, the process of identifying entry points resulted to be of manageable complexity because the Struts and Spring MVC frameworks, on which the implementation is based, contain all the necessary information for finding entry points in the *web.xml* file and in the other configuration file related to it. This is because Struts and Spring MVC are specialization of the JEE framework and collect most of the web application information in file connected to the *web.xml*. Having collected information about entry points is then possible to pass this information to the client- and server-side input validation analyses.

5.1.2 Client-side Input Validation Analysis

The client-side input validation analysis aims to extract the validation operations performed on an input field at client-side. All of these validation operations are then put together in order to build a summary of the validation functions. A summary of a validation function contains all the relevant statements that contribute to the validation of a given input field. The language which has been taken into consideration for input validation at client-side is JavaScript. Being JavaScript of highly dynamic and loosely typed nature (15), it results to be difficult to be analyze statically. For this reason, the approach extracts the relevant client-side input validation code using dynamic slicing (32). Specifically, the technique executes in sequence all the validation functions associated with a given input field i and collects the traces

produced by the resulting executions. The approach performs these two steps for each of the input fields using inputs that are chosen from a pool of representative values. The values are generated using heuristics that are based on the type of the input field. The trace collection is performed using a modified JavaScript interpreter. In particular, the modified interpreter converts all accesses to objects and arrays to accesses to specific memory locations, which avoids imprecision due to the use of objects, arrays, and aliasing. While slicing along a set of traces, this technique handles internal function calls by inlining the code of the callees. External calls, instead, are treated as uninterpreted functions and are passed to the subsequent string analysis without further expansion. At the end of this phase, the slice for a given input field, is a sequence of statements that are in relations to the input field under consideration. From this slice all the string statements that lead to the acceptance of the input value are then extracted to create the summary of validation function for input field i. The client-side input validation analysis technique is the same one that has been adopted in (16).

5.1.3 Server-side Input Validation Analysis

Once all the server-side entry points of the web application have been identified, these entry points are used in order to extract the validation code performed by the server-side part of the web application. Considering the technologies taken into consideration, the input validation functions used at server-side are written in Java. The Java language is considerably more amenable to static analysis than JavaScript. Therefore, the approach identifies the summaries of validation functions by using static, rather than dynamic slicing (32). Using static analysis, could be avoided possible problems of incompleteness related to the fact that a dynamic analysis

might not exercise all possible behaviors of the validation functions. Similarly on what happen at client-side, the approach has to compute a slice containing all the validation statements at server-side for each of the input fields of the web application. In order to do so, starting from the information related to the entry points, the following steps need to be performed:

1. Find source points.
2. Perform interprocedural data-flow analysis.
3. Find sink points.
4. Build summaries of validation functions.

5.1.3.1 Finding Source Points

A source point is a statement in the server side code in which an input field is read from the request that has arrived to the server. For each of the entry points, the server-side code is then statically analyzed in order to find its source points. These source points are the starting points for the next phase of the analysis. A source point can be considered to be a starting point because it is the location inside the server-side code where the input field is received as it has been sent from the client, i.e., it corresponds to the value which has been inputted into the web application by the user. More in detail, the source point is an assignment to a string variable. In addition to the source point statement, the analysis needs to keep track also of the context in which the source point is located and therefore the analysis is context-sensitive. The context contains all the information that are necessary to be able to continue the analysis from the source point till the end of the entry point methods. More in detail, the context is made by

all the method-statement pairs which have lead to a given source point. For example, consider the situation in which the method represented by the entry point calls a second method in which the source point is found. If no context would be kept, the data-flow analysis performed in the next phase of the approach would be limited to the scope of the method in which the source point is found. Instead, if the call point of the entry point method is kept, it is possible to continue the data flow analysis also in the entry point method having in this way a more detailed and precise analysis. This situation, and the information stored inside the context is well represented by Figure 18.

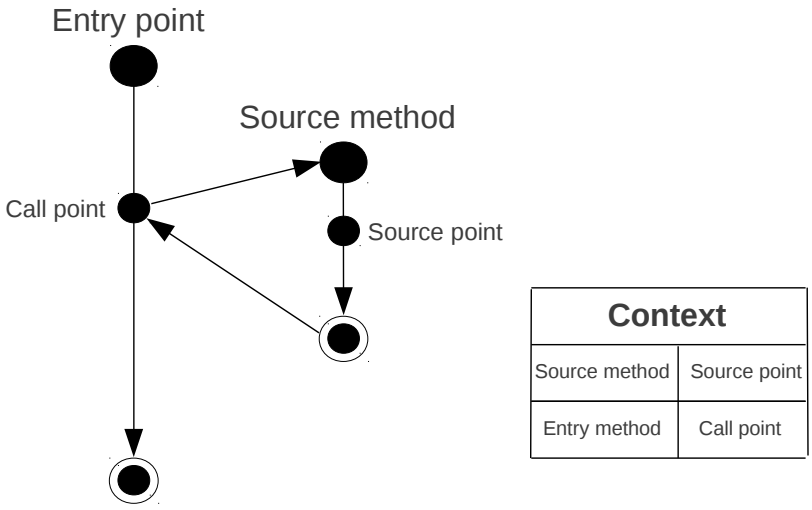

Figure 18. Source point identification and context keeping.

5.1.3.2 Performing Interprocedural Data-flow Analysis

From each of the source points identified by the previous phase, it is then needed to perform interprocedural data-flow analysis. This analysis aims to select all the statements in the server-side code which affect the value related to the source point and therefore the value of the input field taken into consideration. This analysis is then stored and used from the next two phases of the approach. From a general point of view, the data-flow analysis has to be interprocedural because of the situation illustrated in Figure 18. Due to the fact that the analysis is better to be interprocedural, it is clearer now why the information related to the context is important.

5.1.3.3 Finding Sink Points

On the basis of the data-flow analysis performed in the previous step it is then necessary to find the sink points associated to each of the source points. A sink point is a statement inside the server-side code where validation is considered to be over for a given input field and therefore for the source point associated to it. In addition of specifying where validation ends, a sink point, is the point from where an input field can be considered to be correctly validated, i.e., the server-side code from that point on considers the value associated to the input field as a valid input to the web application. This means that if it is considered a path from a source point to its related sink point, this path contains all the operation that lead to the acceptance of the input field. In addition, each of the source points could be related to multiple sink points because each of the sink points could be located on different paths starting from the given source point. Each of the sink points is in connection to its source point because it is located inside the data-flow dependencies which has as origin the given source point. For this reason,

in order to find which are the sink points for a given source, it is necessary to start from a given source point and then through reachability analysis, on the data-flow dependencies, find the statements which represent a sink point. During this process it is necessary to be path sensitive because different sinks could be in different paths and this is one of the reason way the analysis is path sensitive. Of course, after finding a sink point on a given path there is no need to continue the reachability analysis on the same path and therefore the search stops for the given path. An example of multiple sink points related to the same source point is given by Figure 19.

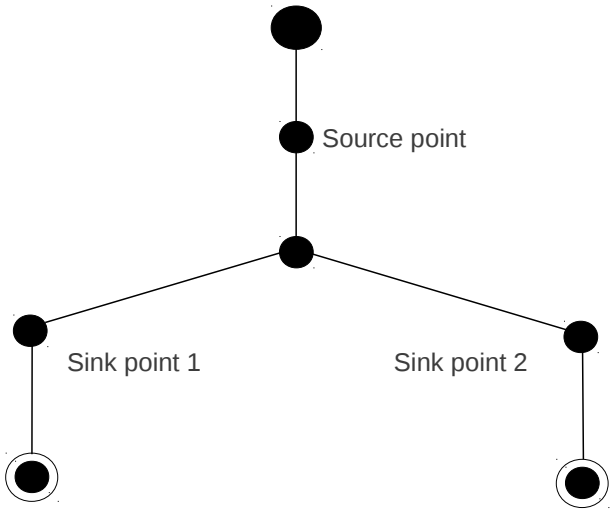

Figure 19. Example of multiple sinks for a given source point.

In addition, being more precise, sink points are those statements which, being related to the source point, can either:

- Store information to file.
- Merge the value of an input field with another string value.
- Send information back to the user containing the value related to the input field.

5.1.3.4 Build Summaries of Validation Functions

After identifying, for each of the source points, its sink points, it is necessary to calculate a summary of the validation function. Like in the case of the client-side code analysis, a summary of the validation function contains all the relevant statements that contribute to the validation of a given input field. More precisely, this phase identifies, for all the source points and their related input fields i, all and only the string statements that operate on i, directly or indirectly. More in detail, these statements result to be string manipulation operations on i and conditional statements which affect i's value. The rest of the code is disregarded because it is irrelevant for the validation of i and it is not of interest for building the representation discussed in Section 5.2. This part of the analysis is path-sensitive as well and stops when a sink point is reached. An example of this process is given by Figure 20. From the figure, it is possible to see how the string operations, gray dots, are kept in the summary of the validation function while other statements, which are not the source or sink points, are discarded. The final result of this first step of the approach is the following. For each validated input of the web application, the approach produces two summary of validation functions: one for the client side and one

for the server side. In case no validation is performed at one of the two side the approach generates, for the required side, a summary validation function which do not perform any type of validation, i.e., accepts every possible string value given as input to the web application. When for every input field there are both client and server summaries of validation functions, the approach associates the client-side summary validation function with the corresponding one at the server-side. These pairs of summaries of validation functions are the input of the subsequent phase of the technique.

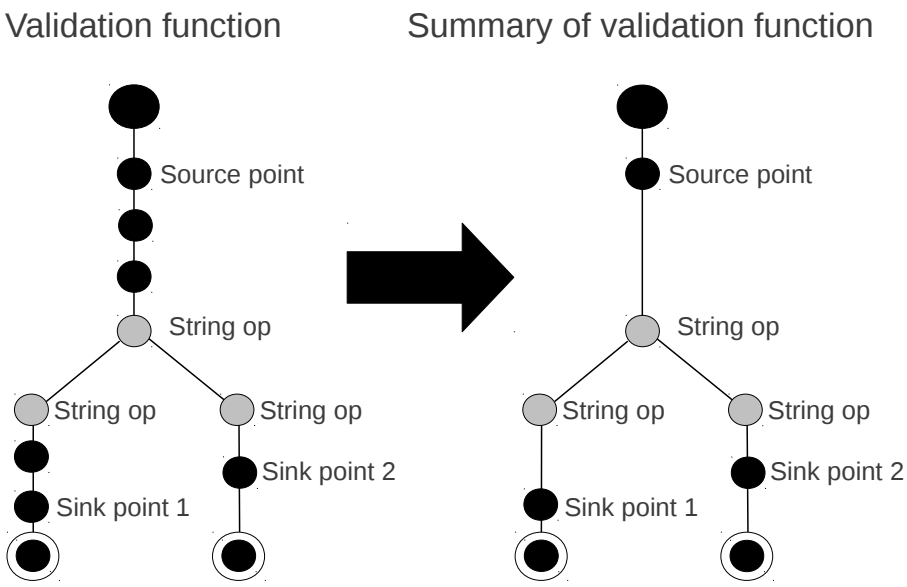

Figure 20. Calculation of summaries of validation functions.

5.2 Input Validation Modeling Using Deterministic Finite Automata

The approach, as second step, performs automata-based string analysis to model the input checks performed by client- and server-side input validation functions. The string analysis is based on the validation operations contained in the summaries of validation functions computed in the previous step. Given a summary, the technique computes the string values at each program point using a flow and path-sensitive, automata-based symbolic string analysis algorithm which has been already adopted for JavaScript by (16). More in detail, the approach uses the algorithm in (16) and completes it by adding support for handling also the summaries produced by the server-side input validation analysis. The string analysis algorithm represents possible values of a string variable at a given point of the summary of the validation functions using a deterministic finite automaton. The automaton representation that has been used is a symbolic representation where the transitions are encoded as a multi-terminal binary decision diagram. In addition, the analysis might need the usage of an automata widening operator to reach convergence (33). The usage of the widening operator might lead the algorithm to calculate an over-aproximation of the values which a string can assume at a given program point and not only its exact values. The string analysis algorithm, Algorithm 1, starts by receiving the summary of the validation functions associated to a given input field i. Each statement inside the summary can be seen as a node of a control flow graph. In the next part of this section, the focus will be on the analysis of the algorithm and the explanation of the characteristic of the types of nodes/statements which are crucial for the analysis.

Algorithm 1 StringAnalysis(CFG)

```
1: initParams();
2: queue WQ := NULL;
3: WQ.enqueue(CFG.entrynode);
4: while (WQ ≠ NULL) do
5:    node := WQ.dequeue();
6:    IN:= ⋃_{node'∈PredNodes(node)} OUT_{node'};
7:    if (node ≡ IF pred THEN) then
8:       tmp_{on_T} := tmp_{on_F} := IN;
9:       if (numOfVars(pred) = 1) then
10:         var := getPredVar(pred);
11:         predVal := EvalPred(pred);
12:         tmp_{on_T}[var] : = IN[var] ∩ predVal;
13:         tmp_{on_F}[var] : = IN[var] ∩ (Σ* - predVal);
14:      end if
15:      tmp_{on_T} := (tmp_{on_T} ∪ OUT_{on_T})∇OUT_{on_T};
16:      tmp_{on_F} := (tmp_{on_F} ∪ OUT_{on_F})∇OUT_{on_F};
17:      if (tmp_{on_T} ⊈ OUT_{on_T}) then
18:         OUT_{on_T} := tmp_{on_T}; OUT_{on_F} := tmp_{on_F};
19:         WQ.enqueue(Succ(node));
20:      end if
21:   else
22:      tmp := IN;
23:      tmp[var] := EvalExp(exp, IN);
24:      tmp := (tmp ∪ OUT)∇OUT;
25:      if (tmp ⊈ OUT) then
26:         OUT := tmp;
27:         WQ.enqueue(Succ(node));
28:      end if
29:   end if
30: end while
31: for (node ≡ SINK POINT ) do
32:    return OUT_{node}[InputField]
33: end for
```

The algorithm keeps track of the CFG nodes that still need to be processed through the usage of a work-list. Each statement, which is analyzed, is associated with two arrays of DFAs: *IN* and *OUT*. Both *IN* and *OUT* have one DFA for each variable of the summary validation function. Given variable v, and the *IN* array for a statement, $IN[v]$ is a DFA that accepts all string values that variable v can take at the program point just before the execution of that statement. Similarly, $OUT[v]$ is a DFA that accepts all string values that variable v can take at the program point just after the execution of that statement. The *tmp* array is used to store the temporary values, i.e. DFAs, computed by the transfer function before joining these values with the previous ones. The algorithm starts by initializing, if any, all the parameters of the summary. It assigns Σ^* to their value in the *IN* array of the entry statement. The same type of assignment will be then performed by the algorithm after each new string variable declaration is encountered. This operation, of assigning Σ^* to a string variable, indicates that the given string value can contain, initially, any value. Then, the algorithms inserts the CFG node for the entry statement into the work-list. At each iteration of the algorithm a CFG node is taken out of the work-list and the transfer function for the corresponding statement is computed. The next two subsections explain what is a transfer function and illustrate in detail how they are computed. After computing the transfer function using the *IN* array, the *OUT* array for the current statement is updated using the join and widening operators. The analysis converges when the work list becomes empty. After the convergence is obtained, the *OUT* value for the string variable associated to the input field (*InputField*) at the *SINK POINT* statement corresponds to the DFA that accepts the set of input values that the validation

function identifies as valid, and the algorithm returns that DFA. As stated above, in the next two subsections, there is a description of the transfer functions used to compute the OUT array for two main types of statements taken into consideration. These statements are assignment statements and conditional statements.

5.2.1 Assignment Statement

In this type of statement it is assigned a value of an expression to a variable on the left hand side of the statement. The function *EvalExp* is used in order to compute the set of string values that an expression can take. This function takes two inputs: an expression, which is on the right hand side of an assignment and an *IN* array, which is the *IN* array of the assignment statement where the expression is. The function evaluates the expressions as follows:

- Variable. In this case the content on the right hand side of an assignment is a *variable*. For this reason, the set of values for the *variable* in the `IN` array is returned, i.e., the DFA $IN[variable]$ is returned.
- String constant. In this case the content on the right hand side of an assignment is a *string constant*. For this reason, a singleton set that only contains the value of the *string constant* is returned, i.e., it is returned a DFA that recognizes only the *string constant*.
- Replace. In this case the content on the right hand side of an assignment is a *variable*, a *pattern* and a *string constant*. For this reason, it is computed the result of replacing all string values in $IN[variable]$ that match the *pattern*, given as a regular expression, with the *string constant*. There are two types of pattern matching, partial match and full match.

The match operation used is chosen based on the *pattern* value as follows. First, if the value starts with the symbol ^ an ends with the symbol $ this means that it is necessary to do a full match, in this case it is needed to replace a string in `IN`[*variable*] only if it fully matches the regular expression given by the *pattern*. This is done by taking the difference between the language in `IN`[*variable*] and the language $L(pattern)$ and then adding the replace *string constant* to the result. If the previous is not the case, partial match is used and the results is computed by using the language-based replacement algorithm described in (24).

- Function call. Since the analysis is intra-procedural with respect to the summaries of validation functions, the technique only analyzes one function at a time without following any function call. However, for the commonly used functions, such as replace and its variations, have been constructed function models that can be used during the analysis. So, in case of a function call inside the summary validation function, there are three options: the function is inlined if it is possible, the function model it is used if it is available or, if the first two options are not available then the algorithm returns Σ^* indicating an unknown string value.

5.2.2 Conditional Statement

This type of statement represents the branch conditions in a number of language constructs including *if statement*, *for loop*, *while loop* and *do while loop*. Conditional statement consists of a predicate on variables and constants. Since it represents a branch in the program, unlike other statements, it is followed by two statements, one after the *ON_TRUE* branch and the other one

after the *ON_FALSE* branch. The predicate in a conditional statement constrains the values of its variables in each of the two branches of execution following the conditional statement. If the predicate evaluates to true, the execution will continue in the *ON_TRUE* branch, otherwise it will take the *ON_FALSE* branch. This behavior is represented in the analysis by having two OUT arrays reflecting the possible future values on each of the two branches of execution. OUT_{on_T} represents the values for the *ON_TRUE* branch and OUT_{on_F} represents the values for the *ON_FALSE* branch. In order to compute these arrays the algorithm first computes the DFA that accepts the set of string values that a variable can take that would make the predicate to evaluate to true using the function *EvalPred*. Then it computes the OUT array of the *ON_TRUE* branch of the conditional statement by intersecting the *IN* DFA for the variable with the DFA for the set of strings that make the predicate to evaluate to true. On the other hand for the *ON_FALSE* branch the algorithm intersects the *IN* DFA with the DFA that is the complement of the DFA that corresponds to the strings that make the predicate to evaluate to true. Function *EvalPred* recursively traverses the predicate while constructing the DFA for each subexpression in the predicate. Logical operations are handled using automata union, intersection and complement and all other expressions are mapped to regular expressions (16). Note that, the function *EvalPred* only handles predicates that contain a single variable. If there is a branch condition on multiple variables, the analysis loses precision since the path sensitivity at that branch location has been lost. In practice, however, this does not cause significant precision loss since this type of branch conditions with multiple variables has not been encountered throughout the development of this work.

5.3 Inconsistencies Detection, Reporting and Correction

As results of the previous two phases there are two automata for each input field i: $A_c(i)$ the automaton which recognizes all the possible input values for i based on the input validation at client side and $A_s(i)$ the automaton which recognizes all the possible input values for i based on the input validation at server side. From $A_c(i)$ and $A_s(i)$ it is possible to define $L(A_c(i))$ and $L(A_s(i))$ where:

- $L(A_c(i))$ represents the language accepted by automaton $A_c(i)$ which represents the set of string values that are accepted by the client-side validation functions for input field i;
- $L(A_s(i))$ represents the language accepted by automaton $A_s(i)$ which represents the set of string values that are accepted by the server-side validation functions for input field i;

Now, based on $A_c(i)$, $A_s(i)$, $L(A_c(i))$ and $L(A_s(i))$, three tasks need to be performed: inconsistentcies detection, inconsistencies reporting and inconsistencies correction.

5.3.1 Inconsistencies Detection

In this subsection it is presented how to detect inconsistencies among client- and serve-side input validation functions. First of all, by using $A_c(i)$ and $A_s(i)$ it is possible to construct two new automata $A_{c-s}(i)$ and $A_{s-c}(i)$ which can be called difference signatures. The difference signatures are important because, in relation to these two automata, there are $L(A_{c-s}(i))$ and $L(A_{s-c}(i))$ which respectively represents the language accepted by the client and rejected by the server and the language which is accepted by the server and rejected by the client for the input field i. More in detail:

- $L(A_{c-s}(i))$ contains strings that are accepted at client side but rejected at server side.
- $L(A_{s-c}(i))$ contains strings that are accepted at server side but rejected at client side.

$L(A_{c-s}(i))$ and $L(A_{s-c}(i))$ are obtained through the following equations:

- $L(A_{c-s}(i)) = L(A_c(i)) - L(A_s(i)) = L(A_c(i)) \bigcap \sim L(A_s(i))$.
- $L(A_{s-c}(i)) = L(A_s(i)) - L(A_c(i)) = L(A_s(i)) \bigcap \sim L(A_c(i))$.

After the calculation of $L(A_{c-s}(i))$ and $L(A_{s-c}(i))$ different results can arise based on the properties of $L(A_c(i))$ and $L(A_s(i))$. The different scenarios can be represented through the usage of set theory. These scenarios are represented by Figure 21, Figure 22, Figure 23, Figure 24 and Figure 25. The scenario in Figure 21 shows the situation in which the language accepted by the client is identical to the language accepted by the server for input field i. In this situation $L(A_{c-s}(i)) = \emptyset$ and $L(A_{s-c}(i)) = \emptyset$. The scenario in Figure 22 shows, instead, the situation in which the language accepted by the client is less strict than the one accepted by the server for input field i. In this situation $L(A_{c-s}(i)) \neq \emptyset$ while $L(A_{s-c}(i)) = \emptyset$. The scenario in Figure 23 shows the situation in which the language accepted by the server is less strict than the one accepted by the client for input field i. In this situation $L(A_{s-c}(i)) \neq \emptyset$ while $L(A_{c-s}(i)) = \emptyset$. The scenario in Figure 24 shows, instead, the situation in which the language accepted by the client and the server for input field i is only partially overlapping. This is different from the previous two cases because now there is no language containing the other. In this situation $L(A_{s-c}(i)) \neq \emptyset$ and $L(A_{c-s}(i)) \neq \emptyset$.

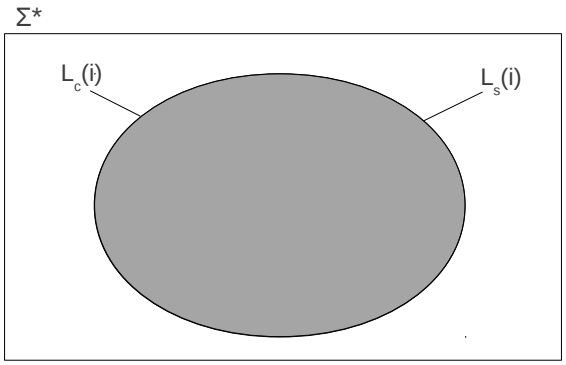

Figure 21. First difference scenario.

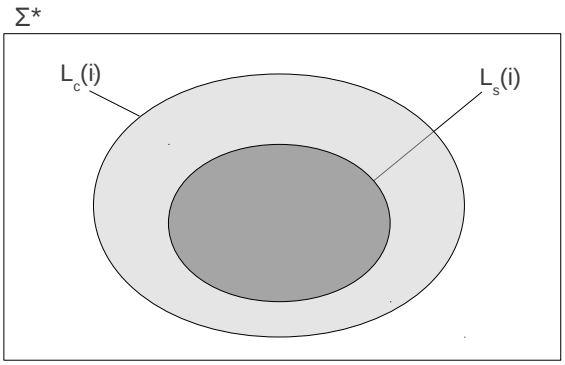

Figure 22. Second difference scenario.

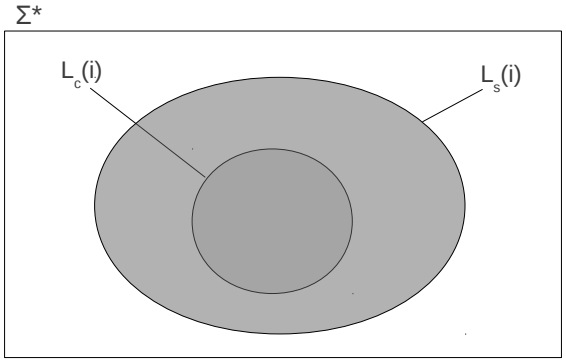

Figure 23. Third difference scenario.

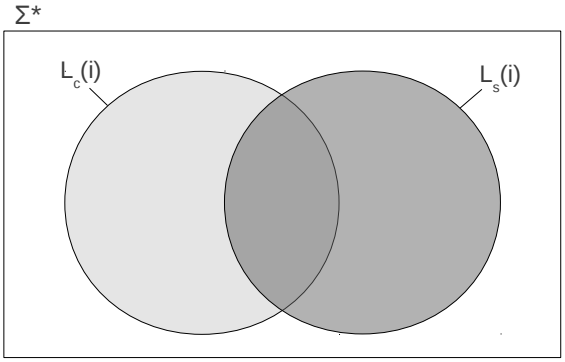

Figure 24. Fourth difference scenario.

The scenario in Figure 25 shows the situation in which there is no overlapping between the language accepted by the client and the one accepted by the server for the input field i. In this situation $L(A_{s-c}(i)) \neq \emptyset$ and $L(A_{c-s}(i)) \neq \emptyset$.

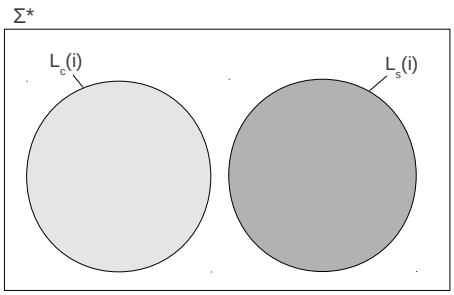

Figure 25. Fifth difference scenario.

From the previous scenarios it is possible to understand that inconsistencies detection, which aims to analyze both $L(A_{c-s}(i))$ and $L(A_{s-c}(i))$, could lead to the results represented by the following cases:

- $L(A_{c-s}(i)) = \emptyset$ and $L(A_{s-c}(i)) = \emptyset$.
- $L(A_{c-s}(i)) \neq \emptyset$ and $L(A_{s-c}(i)) = \emptyset$.
- $L(A_{c-s}(i)) = \emptyset$ and $L(A_{s-c}(i)) \neq \emptyset$.
- $L(A_{c-s}(i)) \neq \emptyset$ and $L(A_{s-c}(i)) \neq \emptyset$.

The previous four cases can be split into the basic cases which are:

- $L(A_{c-s}(i)) = \emptyset$.
- $L(A_{s-c}(i)) = \emptyset$.
- $L(A_{c-s}(i)) \neq \emptyset$.
- $L(A_{s-c}(i)) \neq \emptyset$.

When $L(A_{c-s}(i)) = \emptyset$ or $L(A_{s-c}(i)) = \emptyset$ is part of the output of the analysis, this means that the analysis could not identify any difference between the client and server-side validation functions, hence there are no errors to report. However, due to the possibility of having over-approximations in the analysis, as explained in Section 5.2, this does not mean that the client and server-side validation functions are proved to be equivalent. It just means that the approach could not identify an inconsistency. When $L(A_{c-s}(i)) \neq \emptyset$ is part of the output of the analysis then there might be an error in the client-side validation function. In this situation the client-side validation function is accepting a string value which is then rejected by the server. Doing so, response time of web application and network load increase, worsening web application performances. However, due to the possibility of having over-approximations in the analysis, this result could be a false positive. To prevent the generation of false alarms, the error is validated as described in Subsection 5.3.2. Finally, when $L(A_{s-c}(i)) \neq \emptyset$ is part of the output of the analysis then there might be an error in the server-side validation function. Server-side input validation function should not accept a string value that is rejected by the client-side input validation function. If it is the case, this could lead to security vulnerabilities. However, like in the previous situation, due to the possibility of having over-approximations in the analysis, this result could be a false positive. To prevent the generation of false alarms the error is validated as described in Subsection 5.3.2.

5.3.2 Inconsistencies Reporting

This subsection represents the part of the analysis where the errors, detected by the the previous phase of the analysis, are validated. This part of the analysis is needed in order to

eliminate the false positive which could be the result of the possible over-approximations as explained in Section 5.2. When an inconsistency of type $L(A_{c-s}(i)) \neq \emptyset$ or $L(A_{s-c}(i)) \neq \emptyset$ is detected, a string $s \in L(A_{c-s}(i))$ and $s \in L(A_{s-c}(i))$ is generated respectively. Subsequently the string s is used as input for the input field i. If $s \in L(A_{c-s}(i))$ and server-side function rejects and client-side function accepts the generated string s, then it is sure that there is a problem with the application and the string s is reported as a counter-example behavior demonstrating the inconsistency between the client and server-side validation functions altogether with the code correction generated in Subsection 5.3.3. If instead $s \in L(A_{s-c}(i))$ and client-side function rejects and server-side function accepts the generated string s, then it is sure that there is a problem with the application and the string s is reported as a counter-example behavior demonstrating the inconsistency between the client and server-side validation functions altogether with the code correction generated in Subsection 5.3.3.

5.3.3 Inconsistencies Correction

This subsection aims to explain how it is possible to correct the validation code in case an inconsistency has been detected. If an inconsistency is detected it means that $L(A_{c-s}(i)) \neq \emptyset$ or $L(A_{s-c}(i)) \neq \emptyset$. $L(A_{c-s}(i))$ and $L(A_{s-c}(i))$ are the languages of the difference signatures, namely $A_{c-s}(i)$ and $A_{s-c}(i)$. Remembering that the difference signatures are deterministic finite automata and that DFAs and regular expressions are equivalent models (34), the aim of this phase is to translate a non empty difference signature into the corresponding regular expression. Having then the regular expression associated to the inconsistency, code can be generated in order to correct the inconsistency found.

5.3.3.1 From Deterministic Finite Automaton to Regular Expression

One of the approach to translate a DFA into its equivalent regular expression (35) requires the use of a generalized non-deterministic finite automaton which is a non-deterministic finite automaton, The transition of a GNFA are represented by labels that are regular expression. For this reason it is possible to travel among two states of the automaton if the regular expression on a given transition is satisfied. In order to simplify the transformation process, the GNFA needs to meet some criteria:

- There are no transition into the initial state.
- There are no transition going out from the only accept state.
- The accept state is distinct from the initial state.
- Except for the initial and the accept state, all other states are connected to all other via a transition.

When it is not possible to go between two states, a GNFA has a transition labeled with $\star$ which will not match any string of input characters neither the empty string. These transitions will not be drawn in the following diagrams. The operations to be performed in order to transform a DFA into the corresponding regular expression are:

1. Convert DFA to a GNFA adding new initial and final states.
2. Remove all states one by one until only the initial and accept states are remaining.
3. Extract the result which is the regular expression corresponding to the label of the only transition remaining in the final GNFA.

The first step consists of adding the new initial state q_{init} which is connected to the old initial state via an ϵ-transition. Next it is needed to add the new final state q_{final} such that all the final states in the original DFA are connected to q_{final} via an ϵ-transition. Doing so, criterion one, two and three from the previous list are satisfied. Then, every pair of states is considered and if there is no transition between them a transition labeled with $\star$ is introduced in order to satisfy criterion four of the previous list. The second step consists of simplifying the GNFA in order to finally obtain only one transition between the initial and accept state. From a GNFA with N states it is possible to rip out one state obtaining an equivalent GNFA but with $N - 1$ states. Considering the example in Figure 26, in order to transform every transition through q_{rip} it is necessary to consider each possible pair of states q_i and r_i and then create a direct transition between them. In order to understand how this works, consider Figure 27. In this case there is q_{rip} and two specific states q_{in} and q_{out}. The state q_{rip} has a self loop with regular expression R_{rip} associated with it. Consider now a fragment of an accepting path which goes through q_{rip}. This path has a transition from q_{in} with regular expression R_{in} and a transition that travels out of q_{rip} and into state q_{out} with regular expression R_{out} associated to it. The path corresponds to the regular expression R_{in} followed by 0 or more times of traveling on the self loop R_{rip} and then followed by the regular expression R_{out}. For this reason it is possible to introduce a direct transition from q_{in} to q_{out} with regular expression equivalent to $R = R_{in}(R_{rip})^* R_{out}$. This process of connecting directly q_{in} and q_{out} could lead to the creation of parallel edges. Parallel edges are replaced by only one edge having as regular expression $R = R_{par1} + R_{par2}$. The third step is the final step of the process and indicates when

stopping the ripping procedure. In fact, when only the single transition between the initial and accepting state is obtained, this transition has as label the regular expression representing the initial automaton. This regular expression is then stored and used in the code generation phase.

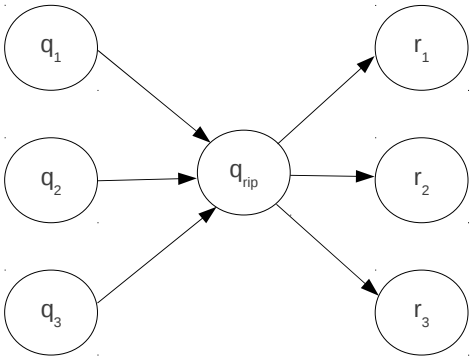

Figure 26. Node ripping global situation.

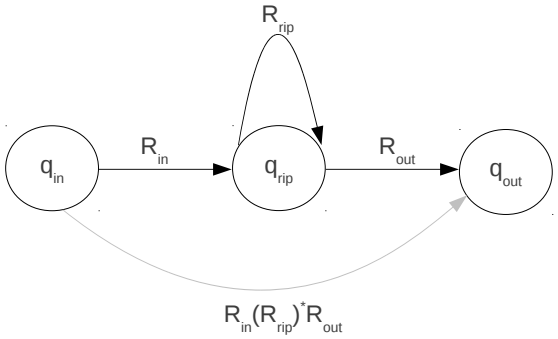

Figure 27. Node ripping individual situation.

5.3.3.2 Code Generation

The aim of code generation is to correct the inconsistencies found through previous steps of the analysis. Considering that the approach is able to find inconsistencies both at client and server sides, the generated code will be dependent on the language which has been used at client or server side. Even if the language of the generated code will be different from case to case, the principle which drives the code generation is only one. In fact, having an inconsistency of type $L(A_{c-s}(i)) \neq \emptyset$ or $L(A_{s-c}(i)) \neq \emptyset$ means that at client or server side has been done some

validation checks which have not been replicated on the other side. The easiest way to replicate these checks, on the side in which they are missing, is to use the regular expression calculated in the previous step. In fact, all the missing checks can be summarized by the calculated regular expression, because the regular expression represents the language accepted by one of the two sides and rejected by the the other side. After having the regular expression is then just needed to add to the source code a *string match* on the the input field i. All the input values that match the regular expression are rejected while all the input values which do not match are passed to the related validation functions. As said before, the *string match* is language dependent. Considering the two type of inconsistencies that can be detected, the following are the two task which are performed:

- If $L(A_{c-s}(i)) \neq \emptyset$ add the *string match* containing the regular expression to the client-side validation code.
- If $L(A_{s-c}(i)) \neq \emptyset$ add the *string match* containing the regular expression to the server-side validation code.

CHAPTER 6

IMPLEMENTATION

In this chapter, it is possible to find all the details related to the implementation of this work. All the modules of the prototype which have been developed are described in the following part of the chapter. In addition, it is also illustrated their relation with the general approach of Chapter 5. Chapter 5 is divided in three main sections: input validation extraction, input validation modeling using deterministic finite automata and inconsistencies detection, reporting and correction. Also this chapter will be divided in a similar fashion, in fact every main section of the approach will be considered under an implementation perspective. The implementation has been driven by the technologies and frameworks taken into consideration, which are JEE (31), Struts (36) and Spring MVC (37) web frameworks. These frameworks have been analyzed because they are largely adopted among the developer community of web applications. These technologies extensively use two languages: JavaScript and Java. More in detail, JavaScript is the language for client-side input field validation while Java is the language used for server-side input field validation. The prototype which has been implemented is written in Java language and makes use of some other technologies which will be described in the rest of the chapter.

6.1 Extraction Implementation

Input validation extraction is the first task that has to be performed according to what has been illustrated in Chapter 5. Due to the properties of web applications based on Struts

or Spring MVC, which are the frameworks on which the implementation of the prototype is specialized on, the point of the web application where it is possible to find the initial information about input validation is the web deployment descriptor.

6.1.1 Web Deployment Descriptor Analyzer

According to the JEE (38), Struts and Spring MVC specifications each web application must provide a web deployment descriptor file. This file is also known as *web.xml*. In this first step, the prototype analyzes this file to understand the different client and server-side validation components used within the web application. More in detail, the aim of the search is to identify which are the entry points for the validation of input fields on both client and server-side. In order to complete this task, a XML parser has been implemented. This parser, as said before, starts by reading the information inside the *web.xml* file. The first information which is of interest to the parser is the configuration file name of the web application. This file can not be missing because it is required by Struts and Spring MVC web frameworks. Figure 28 shows a portion of a *web.xml* which contains this type of information. This snippet of code and all the ones which will be shown in the next part of the chapter are real examples extracted from one of the application evaluated in Chapter 7. After the information related to the location and the name of the framework configuration file have been extracted, the configuration file is parsed and analyzed as well. Inside this file it is possible to detect all the names of the web forms of the web application and their JSP location. In addition to this, the configuration file contains the information related to the input validation adopted in the web application.

This information is a link to two different file. Figure 29 illustrates all the previous mentioned information contained in the configuration file.

```
...
<web-app>
        ...
        <servlet>
                <servlet-name>action</servlet-name>
                <servlet-class>
                        org.apache.struts.action.ActionServlet
                </servlet-class>
                <init-param>
                        <param-name>config/jgossip</param-name>
                        <param-value>
                                /WEB-INF/struts-config-jgossip.xml
                        </param-value>
                </init-param>
                <load-on-startup>1</load-on-startup>
        </servlet>
        ...
</web-app>
```

Figure 28. Content of the web deployment descriptor file.

From the first of the two validation file, *validation-jgossip.xml*, it is possible to understand which are the functions used to validate a specific input field of a given web form. Inside this file there is no validation code but just the symbolic name which is used to identify the different validation functions. One example of this type of information is represented by Figure 30. From the second of the two validation file, *validator-rules-jgossip.xml*, it is possible to understand

which is the exact JavaScript function or Java method that has been used in association with the symbolic name found in the previous validation file.

```
...
<struts-config>
        ...
        <form-beans>
                ...
                <form-bean name="logonForm" type="org.jresearch.gossip.forms.LogonForm" />
                ...
        </form-beans>
        ...
        <action-mappings>
                ...
                <action path="/Logon"
                        type="org.jresearch.gossip.actions.user.LogonAction" name="logonForm"
                        scope="request" validate="true" input="/showLogon.do">
                        <description>
                                Perform Logon for registered users
                        </description>
                        <forward name="welcome" path="/Main.do" redirect="true" />
                </action>
                ...
        </action-mappings>
        ...
        <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
                <set-property property="pathnames"
                        value="/WEB-INF/validation-jgossip.xml,/WEB-INF/validator-rules-jgossip.xml"/>
                <set-property property="stopOnFirstError" value="false" />
        </plug-in>
        ...
</struts-config>
```

Figure 29. Example of Struts configuration file.

Most of the web application that are based on the these type of frameworks use the Struts validator framework therefore the implementation of these functions could be found inside the *org.apache.struts.validator* or *org.apache.commons.validator* libraries for both JavaScript

and Java validation functions. Even if standard validation functions are offered, it is possible to specify inside this file custom validation functions. Last step in the identification of the validation functions is to check whether the client-side validation functions are actually used within the web application. It is possible to understand this by looking for a specific tag, *:javascript*, inside the JSPs which are associated to the web forms of the web application. These JSPs are the same ones which have been identified in the configuration file. Figure 31 shows the relation between symbolic name and validation function.

```
...
<form-validation>
        <global>
                ...
                <constant>
                            <constant-name>logon_pattern</constant-name>
                            <constant-value>^[0-9a-zA-Z_]{3,32}$</constant-value>
                </constant>
                ...
        </global>
        <formset>
                ...
                <form name="logonForm">
                        <field property="username" depends="required,mask">
                                <arg0 key="forum.U_NAME"/>
                                <var>
                                        <var-name>mask</var-name>
                                        <var-value>${logon_pattern}</var-value>
                                </var>
                        </field>
                        ...
                </form>
                ...
        </formset>
</form-validation>
```

Figure 30. Example of validation file.

```
...
<form-validation>
        ...
           <global>
                ...
                <validator name="required"
                          classname="org.apache.struts.validator.FieldChecks"
                                method="validateRequired"
                          methodParams="java.lang.Object,
                                org.apache.commons.validator.ValidatorAction,
                                org.apache.commons.validator.Field,
                                org.apache.struts.action.ActionMessages,
                                org.apache.commons.validator.Validator,
                                javax.servlet.http.HttpServletRequest"
                           msg="errors.required"
                         jsFunction="org.apache.commons.validator.javascript.validateRequired"/>
                ...
        </global>
<form-validation>
```

Figure 31. Example of validator rules file.

During the parsing process all the information are stored in appropriate data structures. More in detail, for each of the input fields it is created an association between the input field itself and the validation functions adopted both at client and server-side. In addition to this, it is stored also the web page where the input field is used. These validation functions, associated to each of the input fields, represent, on the server side, the entry points specified in Subsection 5.1.1. These functions can be considered as entry points because they identify all the validation operations which are performed on the input field and, in addition, they are the logical place of the web application where to specify these type of checks. What has been said in Subsection 5.1.1 was that for each of the entry points there could be multiple source points. In this situation however, for each of the entry points, there will be only one source

point related to it. All the information that has been gathered by the parser are then passed to the part of the prototype that will take care of client- and server-side input validation function analysis.

6.1.2 Client-side Input Validation Analyzer

In order to perform the client-side input validation analysis it has been used a tool produce by Alkhalaf et al. in (16). The prototype has been then adapted to the case of the client-side validation functions that are part of the class of web applications under analysis. In fact, these validation functions do not take as input the target field that is under validation but they take the whole web form related to it. Given this form as input, a validation function first extracts the field that it wants to validate from the form object and then validates the value of that field. This creates a problem for the analysis since the approach aims to calculate the summary validation function for a single field, not the whole form. As mentioned in Subsection 5.1.2, given the difficulties associated with static analysis of JavaScript (15), static extraction of the JavaScript validation code for a single field with enough precision is not feasible. This problem has been solved by the Alkhalaf tool adopted in this work by executing the validation function using different input values and extracting the validation code for the target field using dynamic slicing, as described in (16). A dynamic slice contains all the statements that access the targeted field along with all the other statements they depend on. In order to make this possible it is needed to use HtmlUnit (39), which is a browser simulator that uses Rhino (40), a JavaScript interpreter, to facilitate the dynamic extraction. Using HtmlUnit, it is simulated the process of filling out a form and submitting it. The aim of filling out and submitting a form

is to capture which are the statements of the validation code that are actually executed. The process of filling out a given form is accomplished by using a profile of values which have been calculated using heuristics on the type of value of a given input field. In order to capture the validation statements which are executed when the form is submitted the JavaSript interpreter has been instrumented. When the form is actually submitted all the executed statements and all other statements that they depend on are outputted. If there are function calls for non-native JavaScript functions , these functions need to be inlined such that the final code consists of only one validation function for the target field which ends with a *return true* statement. This *return true* statement has the same function as the sink point in the server-side analysis. Because some information related to validation could be find inside the XML configuration file these information needs to be manually placed inside validation function before doing the dynamic slicing. One example of this are validation mask, i.e. regular expression, which are specified inside the *validation.xml* file. Output of this phase of the analysis are the summaries of validation functions related to each of the input fields. In addition to these, the tool employed calculates the client-side automata which will be the input of the third phase of the approach. These resulting automata are written in *DOT* language. The client-side automata together with the summaries of validation functions are stored for further processing.

6.1.3 Server-side Input Validation Analyzer

Input of this phase are the validation functions which have been identified by the previous step of the analysis. In order to build the summaries which are the necessary input for the next phase of the analysis, it is necessary to perform code analysis on the validation functions used

within the web application. Because precision is essential for the approach, it has been decided to perform static program analysis. Static program analysis, in the case of Java language, can be performed through the usage of Soot framework (41). Soot is a Java optimization framework which helps to analyze and transform Java bytecode. The validation functions received as input from the previous phase represent the entry points from where the static program analysis has to begin. As it is possible to understand from the validation identification analysis, there could be the possibility that the same input field could be validated by different functions. In the implementation of the prototype each of these function is analyzed by itself and the results of this analysis are put together by the next phase of the approach. For this reason will be now explained how the analysis for a single function has been implemented. First step is to use the Soot framework in order to translate the validation function into 3-address code. This representation, also known as Jimple, is useful because its statements only reference at most 3 local variables or constants, simplifying the analysis procedure. In order to perform the analysis for a single validation function, the *.class* file containing the validation function it is given as input to the Soot framework. After this operation, static analysis is performed on the Jimple code. First objective, as the approach requires, it is to locate the source point from the given entry point. This is done by analyzing each statement of the Jimple code. More in detail when speaking of Jimple format each statement is represented by an *unit*. For this reason, unit and statements can be intended as synonym from this point on. In order to identify which is the source point it has been implemented a list containing all possible signatures which identify a source point. For the frameworks which are analyzed by this work a possible

signature which identifies a source point is given by Table I. Once the source point has been identified data-flow analysis related to the validation function is required. In order to obtain this information an intraprocedural data-flow and control-flow analyzer has been implemented. Within Subsection 5.1.3, however, has been stated that interprocedural data-flow analysis was the information which was retained to be necessary. This is still true and applicable to a wider range of situation, however based on the structure of these functions only intraprocedural analysis resulted to be necessary. This is because it has been possible to model interprocedural calls to libraries with the corresponding string operations which they were performing. This can be considered as a form of function inlining. Two examples of the previous concept are shown in Table II.

TABLE I

SOURCE POINT SIGNATURE.

Signature
$evaluateBean(java.lang.Object, org.apache.commons.validator.Field);$

All the function calls which result not to be within the formulated mappings are not considered within the analysis. This seems to introduce imprecision in the implementation of the approach, however the Chapter 7 shows that this is not the case.

TABLE II

LIBRARY FUNCTION MAPPING.

Library Function	**Function Mapping**
$GenericValidator.isBlankOrNull(value)$	$value == null \mid\mid value.trim().length() == 0$
$GenericValidator.matchRegexp(value, mask)$	$Perl5Utilmatcher = newPerl5Util();$ $matcher.match("/" + mask + "/", value);$

Once the implemented data-flow and control-flow analyzer calculates the dependencies information of a given validation function a program dependence graph is built and associated to its function. After the PDG has been calculated the sink points within the validation function are searched. These sink points correspond, in the case of the validator frameworks which have been considered, to all the *return true* statements within the validation function. A *return true* statement identify the point from where the input value is considered as valid inside the web application. This is in perfect accordance with the definition of sink point given in Subsection 5.1.3. Once the sink points are detected, the implemented backward slicer is used. Backward slicing starts from every sink point and selects all the statements and especially, string operations, which affect the acceptance of a given input parameter. After the backward slice is computed all the string operations are selected and stored into the summary of the validation function. During this process, the string operations are stored maintaining path sensitivity, i.e., the knowledge about different paths is kept in the summary of a validation function. Example of sink points search and backward slicing on validation functions is given by Figure 32. The

summary of validation function has been implemented as an XML file. The idea of having a summary of a given validation function in a common language, such as XML, it is really important for the extensibility of this prototype. In fact, this part of the implementation is the only one which is language dependent. If the same code analysis would be performed for PHP based web application, the summary of the validation function expressed in XML language could be given as well as input to the next phase of the analysis. In this way it is easily possible to extend the range of the web application which are analyzed by this prototype.

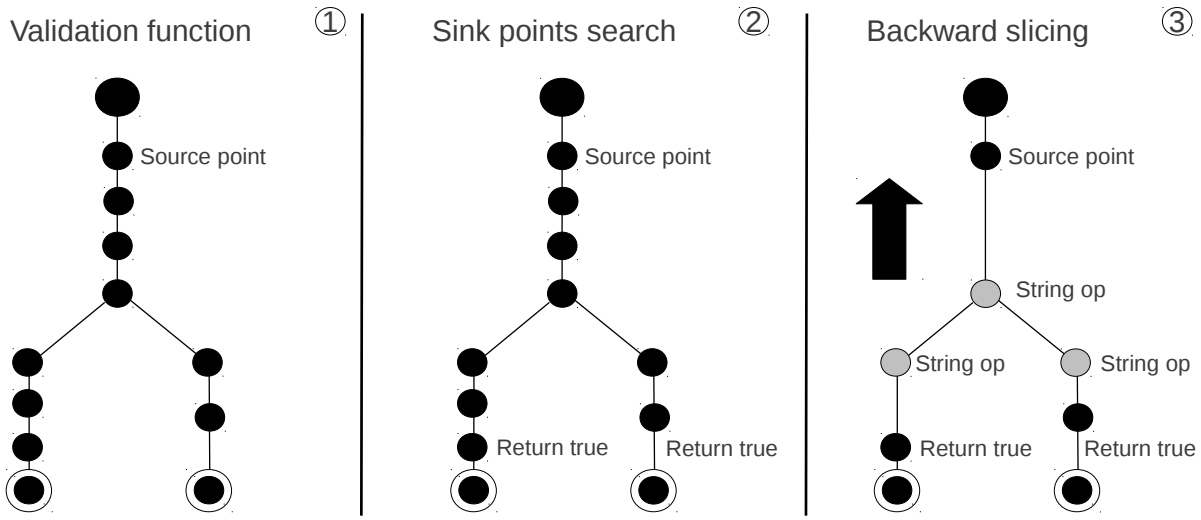

Figure 32. Example of sink points search and backward slicing.

As result of this first module of the implementation there are, for each input field, summaries of the validation functions on both client and server-side and the automata corresponding to the client-side summaries. This information are the input for the next phases of the analysis. In addition, it is important to point out that, at server-side there could be more than one summary of validation function for a given input field. This will be handled by the next module of the implemented prototype.

6.2 Input Validation Modeler Using Deterministic Finite Automata

Aim of this phase of the analysis, as said in Section 5.2, is to generate an automaton for each of the validation functions of an input field. This automaton represents all possible inputs which can be considered as valid for a given input field. The construction of this automaton is based on the summary of validation function generated in the previous phase. Due to the capability of the software used for the client-side validation function extraction the automaton representing all the possible inputs for a given input field at client-side is already generated. For this reason, the implementation of this module of the prototype aims to generate the corresponding server-side automaton for each of the input-fields. The implementation, even if it is focused on the results obtained from the server-side input validation extraction, it is easily extensible also to the client-side part because it is based on an XML input which could be also used to express the client-side summaries of validation functions. This extension, of outputting client-side summary functions in XML format, is part of the future work. In order to be able to model summaries of validation functions into DFAs it has been used the StrangerAutomaton library (42), which

internally uses the Mona tool (43) to represent DFAs. There are three important features in the implementation of the prototype, they are:

- String operation mapping.
- Algorithm implementation.
- Final automaton creation.

6.2.1 String Operation Mapping

In order to be able to transform a summary of a validation function into the automaton that expresses all the valid string for a given input field, it has been implemented a mapping between string operations and regular expressions. This has been done because, using StrangerAutomaton library, it is possible to translate a regular expression into the corresponding automaton. Examples of this mapping can be seen in Table III.

TABLE III

STRING OPERATIONS AND REGULAR EXPRESSIONS.

String operation	**Regular Expression**
$equals(value)$	$/\hat{}value\$/$
$match(regexp)$	$/\hat{}regexp\$/$
$value.lenth() > 0$	$/\hat{}(.)+\$/$

6.2.2 Algorithm Implementation

In order to calculate which is the automaton corresponding to a given input field at a given sink point it has been implemented the algorithm illustrated in Section 5.2. The implementation parses the XML summary of a validation function and generates an automaton into the format given by the StrangerAutomaton library.

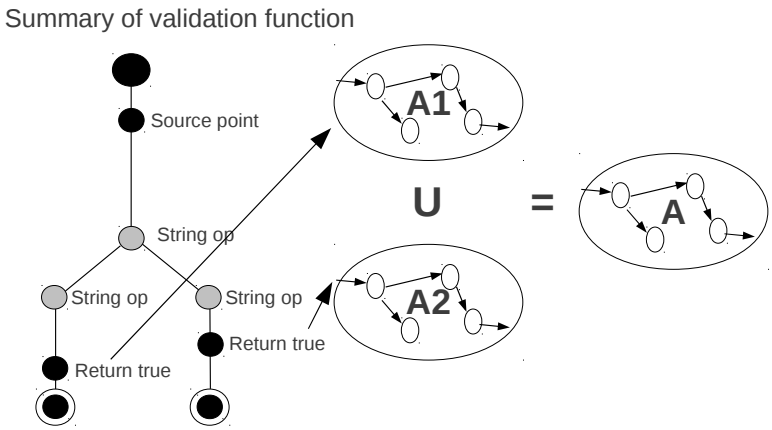

Figure 33. Union operation between automata.

6.2.3 Final Automaton Creation

The real output of this phase of the approach is generated by refining the results obtained after the execution of the previous algorithm. In fact two operation might be necessary. The first one concerns the possibility to have multiple sink points within the same summary of validation

function. For this reason it has been implemented a routine which takes as input the automata corresponding to these sink points and creates a new automaton which represents the union of the possible inputs which could lead to all of these sink points. This union operation can be better understood through Figure 33. After having only one automaton for each summary of validation a second step might be needed. The second operation it is associated to the fact that multiple summaries of validation functions could be associated to the same input field. In order to be valid, an input field has to be accepted by all of the validation function associated to it. For this reason all the valid values for a given input field are the ones represented by the automaton that is the intersection of all the automata that represents all the summaries of validation function. This intersection operation can be better understood through Figure 34.

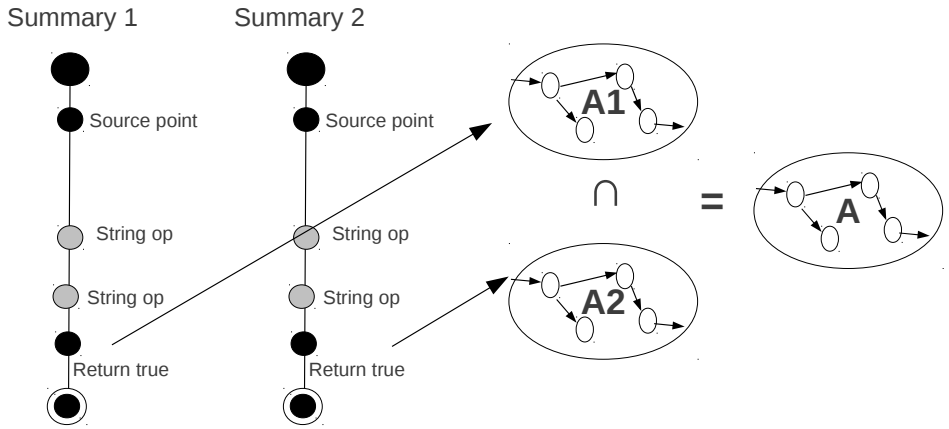

Figure 34. Intersection operation between automata.

As output of this module of the prototype there is one automaton associated to both the client and the server side input validation functions. As required from the approach, each of these two automata represents all the possible string values which are considered to be valid for a given input field. The automata are saved to file into *DOT* language. An example of automaton generated by the prototype can be seen in Figure 35 while its translation into *DOT* language can be seen in Figure 36. From Figure 35 it is possible to note that some of the labels of the transitions specify a range of values instead of only one value. The automata generated within this phase of the analysis will be the input for the next module of the tool.

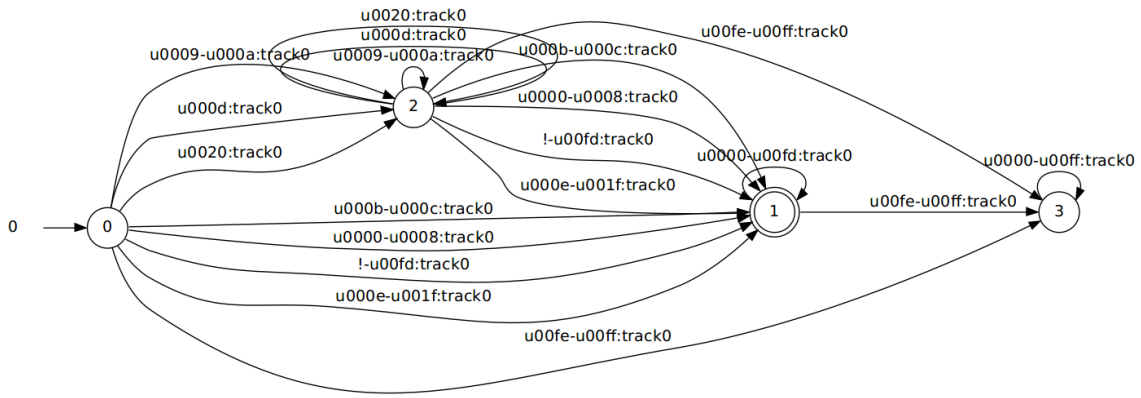

Figure 35. Automaton representation for possible values of an input field.

```
digraph Automaton {
  rankdir = LR;
  0 [shape=circle,label="0"];
  initial [shape=plaintext,label="0"];
  initial -> 0
  0 -> 3 [label="\u0009-\u000a:track0"]
  0 -> 3 [label="\u000d:track0"]
  0 -> 1 [label="\u000b-\u000c:track0"]
  0 -> 1 [label="\u0000-\u0008:track0"]
  0 -> 3 [label="\u0020:track0"]
  0 -> 1 [label="!-\u00fd:track0"]
  0 -> 1 [label="\u000e-\u001f:track0"]
  0 -> 2 [label="\u00fe-\u00ff:track0"]
  1 [shape=doublecircle,label="1"];
  1 -> 1 [label="\u0000-\u00fd:track0"]
  1 -> 2 [label="\u00fe-\u00ff:track0"]
  2 [shape=circle,label="3"];
  2 -> 2 [label="\u0000-\u00ff:track0"]
  3 [shape=circle,label="2"];
  3 -> 3 [label="\u0009-\u000a:track0"]
  3 -> 3 [label="\u000d:track0"]
  3 -> 1 [label="\u000b-\u000c:track0"]
  3 -> 1 [label="\u0000-\u0008:track0"]
  3 -> 3 [label="\u0020:track0"]
  3 -> 1 [label="!-\u00fd:track0"]
  3 -> 1 [label="\u000e-\u001f:track0"]
  3 -> 2 [label="\u00fe-\u00ff:track0"]
}
```

Figure 36. Automaton representation expressed in *DOT* language.

6.3 Inconsistencies Detection, Reporting and Correction Implementation

In order to identify the inconsistencies between client- and server-side input validation functions it has been implemented a module which starts to read the automata form their *DOT* language file and converts them into StrangerAutomaton format. Once this step has been executed for both client and server-side automata it is possible to compare the two automata associated to a given input field.

6.3.1 Inconsistencies Detection Implementation

The first part of this module of the prototype checks, for every input field, if there are some input values which are accepted by one of the two automaton and not accepted by the other according to the approach explained in Subsection 5.3.1. The comparison between client- and server-side automata is performed through complement and intersection operations of Strager-Automaton library. During the comparison process two new automata are computed: $A_{c-s}(i)$ and $A_{s-c}(i)$. After computing these two automata, the prototype, checks whether these two automata are empty or not. Having an empty automaton means that there is no string value that is accepted by one of the automata and not accepted by the other. If $A_{c-s}(i)$ or $A_{s-c}(i)$ is not empty then this new automaton is stored and passed to the second and third part of this module of the prototype.

6.3.2 Inconsistencies Reporting Implementation

If an inconsistency has been detected then the prototype generates a string value which demonstrates the inconsistency. This string will be accepted by one of the two validation side but rejected from the other. After the generation of this string value, the string it is first used to verify if the inconsistency it is a false positive, i.e., an inconsistency which actually does not exists. This check is manually performed by running the application and using the proposed value. If the string value is actually rejected by both sides it means that there is a false positive. If the string value actually correspond to an inconsistency, it is then also used to verify, after the code has been corrected, that the code correction of the web application it is actually working. The string value it is generated through the usage of the StrangerAutomaton

library. The string value is associated with the corresponding input field, the side on which has been identified an inconsistency, and the code correction generated by the third part of this module of the prototype.

6.3.3 Inconsistencies Correction Implementation

This part of the prototype aims to correct the web application code in case an inconsistency has been found. In case an inconsistency of type $A_{c-s}(i)$ is detected, client side code for input field i needs to be corrected. While, if an inconsistency of type $A_{s-c}(i)$ is detected, then server side code for input field i needs to be corrected. First step is to generate the regular expression associated to the non empty automaton $A_{c-s}(i)$ or $A_{s-c}(i)$. Once this has been done code generation can be executed. Code generation is language dependent and, based on the languages taken into consideration by this work, JavaScript code is generated if client code needs to be corrected, while Java code is generated if server code needs to be corrected. The principle of code generation is however the same, in fact, a string match has to be generated. In Figure 37 it is possible to see the output of JavaScript code correction, while in Figure 38 it is possible to see the output of Java code correction.

```
function correctionFunction(value){
        var regexp = /^generatedregexp$/;
        if (regexp.test(value)){
                return false;
         }
         return true;
}
```

Figure 37. Client-side code correction.

```
public boolean correctionFunction(String value){
        String regexp = "generatedregexp";
        Perl5Util matcher = new Perl5Util();
        if(matcher.match("/^" + regexp + "$/", value)){
                return false;
        }
        return true;
}
```

Figure 38. Server-side code correction.

CHAPTER 7

EVALUATION

To assess the usefulness of the approach, it has been used the implementation discussed in Chapter 6. The prototype that has been implemented it has been used to perform an empirical evaluation on a set of real-world web applications. During the evaluation process several aspects have been tested. First of all it has been evaluated if the approach and its implementation can actually identify inconsistencies in client- and server-side input validation functions. If inconsistencies are not present, it has been evaluated if the prototype is able to establish an equivalence relation between the client- and server-side input validation functions. Because the implementation is able to generate an input value example when an inconsistency is found, it has been evaluated whether the detected inconsistencies are actual inconsistencies. In addition to this, it is also reported the number of inconsistencies which should have been reported but actually have not been detected form the prototype. After these two type of evaluation, it has been tested the effectiveness of the code correction module implemented by the prototype. This is possible because the tool produces as output both an input value example when an inconsistency is detected and its related code correction. Finally it is reported if the implementation of the approach is efficient enough to analyze real-world web applications. This evaluation is based on the prototype's execution time and memory usage. In the next part of the chapter are also presented the subjects which have been taken into consideration during the evaluation process and the results related to them.

7.1 Experimental Subjects

For the empirical evaluation of the approach, seven real-world web applications have been selected. These web applications can be found at `http://www.julien-dubois.com/` and in two open source code repositories, Sourceforge `http://sourceforge.net` and Google Code `http://code.google.com`. The two repositories have been taken into consideration because they offer web applications base on JEE, Struts and Spring MVC framework. Please note that these are the frameworks that the current implementation can handle. In addition, note that projects with a small user base or with a low activity level have been discarded. In this way have been privileged web applications that were more likely to be widely used and well maintained. Table IV shows the list of web applications that have been used for the experimental evaluation. In the table it is also possible to find the URL where the subjects have been obtained. The first four applications in the list are written using the Struts framework (36):*JGossip* is a messaging board application, *Vehicle* is an application to manage vehicles owned by a company, *MeoDist* is an application for managing information about sport club members, and *MyAlumni* is a social network application for school alumni. The last three applications are written using the Spring MVC framework (37):*Consumer* is a customer relationship management application, *Tudu* is an on-line application for managing to do lists and *JcrBib* is a virtual library application that supports user collaboration. Based on their description, these web applications cover a wide spectrum of types. Moreover, because of the way they were selected, most of these applications are popular and widely used in practice. In fact, *JGossip*, for instance, has been downloaded almost $30,000$ times from its Sourceforge page.

TABLE IV

WEB APPLICATIONS USED IN THE EMPIRICAL EVALUATION.

Name	*URL*
JGossip	`http://sourceforge.net/projects/jgossipforum/`
Vehicle	`http://code.google.com/p/vehiclemanage/`
MeoDist	`http://code.google.com/p/meodist/`
MyAlumni	`http://code.google.com/p/myalumni/`
Consumer	`http://code.google.com/p/consumerbasedenforcement/`
Tudu	`http://www.julien-dubois.com/tudu-lists`
JcrBib	`http://code.google.com/p/jcrbib/`

7.2 Extraction Evaluation

For conducing the experiments and therefore having an empirical evaluation, it has been used an Ubuntu Linux machine with an Intel Core Duo 2.4Ghz processor and 2GB of RAM running Java 1.6. According to what has been said in Chapter 6, for each web application, the prototype first analyzes the application's configuration file to identify its input fields and corresponding client- and server-side validation functions. It then builds the client- and server-side summaries of validation functions for each input fields. Relevant data for this part of the analysis are shown in Table V. The first column in the table lists the application name, followed by the number of forms extracted, Frm, and the total number of inputs across all forms, $Inputs$. Column VI_C, respectively VI_S, lists the number of inputs for which a client-side, respectively server-side, validation function is specified in the configuration file. Similarly, column ET_C, respectively ET_S, lists the time taken, in seconds, to extract the summaries of validation functions for these

inputs on the client side, respectively server-side. For example, web application *Consumer* contains 3 forms, for a total of 21 input fields. Of these input fields, 14 are validated on the client side, whereas all of 21 of them are validated on the server-side. It took 68.4 and 1.1 seconds to extract the summaries of validation functions on the client- and server-side, respectively. Note that the time required to compute the client-side summary validation functions is much higher than the time to extract the server-side summaries of validation functions. This difference is due to the additional time required to perform dynamic slicing on the client side, which in turn requires the prototype to load and run JavaScript functions in the browser simulator. In addition, it is important to add that the extraction task on the client side has been performed by the same people of (16) to whom has been asked to provide the generated automaton and the data in Table V.

TABLE V

RELEVANT DATA ON INPUT VALIDATION EXTRACTION.

Subject	Frm	$Inputs$	VI_C	ET_C(s)	VI_S	ET_S(s)
JGossip	25	83	74	329.80	83	4.38
Vehicle	17	41	41	155.48	41	2.04
MeoDist	18	62	62	192.20	62	1.93
MyAlumni	46	141	0	0.00	141	4.28
Consumer	3	21	14	68.40	21	1.10
Tudu	3	11	0	0.00	11	0.78
JcrBib	21	45	0	0.00	45	1.51

From Table V it is possible to observe that the extraction phase of the prototype took between 0.78 and 4.38 seconds for the server-side validation functions. As said before, client-side extraction is more expensive however still between acceptable limits. The extraction overall could be considered to be extremely efficient because it is a process which requires to be executed only once and therefore even an extraction time of hours could be considered to be fine. Here the extraction time stays in the range of few minutes proving in this way its efficiency.

7.3 Input Validation Modeling Evaluation

After building the client- and server-side summaries of validation functions for each input field, the prototype constructs the corresponding DFAs, as described in Section 5.2. Table VI shows details about this part of the approach. For each application, and both for the client and the server sides, the table shows: the average size of the automata in megabytes, followed by the minimum, maximum, and average number of states, column S, and symbolic transitions in the automata, column T. The number of transitions actually represent the size of the symbolic representation of the automata's transition relation. In fact, associated to one of these symbolic transitions could correspond a multiple number of transitions of a standard automaton. As an example, the application *Tudu* has DFAs with an average of 4 states and 10 symbolic transitions in the client-side automata, whereas it has an average of 8 states and 68 symbolic transitions in the server-side automata. Note that, when client-side validation is absent for an input, the DFA for that input is a Σ^* automaton. Hence, *Tudu* has a client-side automaton even though it has no client-side validated inputs, see Table V. The results proposed in this section illustrates

the main space cost of the prototype. As it is possible to understand form Table V the space needed to store the automata is negligible, as it is less than seven megabytes in all cases.

TABLE VI

RELEVANT DATA ON INPUT VALIDATION MODELING.

Subject	*Client − side DFA*							*Server − side DFA*						
	AvgSize	*min*		*max*		*avg*		*AvgSize*	*min*		*max*		*avg*	
	(*mb*)	*S*	*T*	*S*	*T*	*S*	*T*	(*mb*)	*S*	*T*	*S*	*T*	*S*	*T*
JGossip	6.03	4	10	35	706	6	39	6.05	4	24	35	706	6	41
Vehicle	4.83	4	24	7	41	5	26	4.84	4	24	7	41	5	26
MeoDist	5.67	5	25	5	25	5	25	5.67	5	25	5	25	5	25
MyAlumni	3.17	4	10	4	10	4	10	3.16	3	24	5	25	5	25
Consumer	5.34	4	10	17	132	5	25	5.34	4	24	17	132	7	41
Tudu	6.12	4	10	4	10	4	10	6.12	3	24	23	264	8	68
JcrBib	5.37	4	10	4	10	4	10	5.38	5	25	5	25	5	25

7.4 Inconsistencies Detection Evaluation

At this point of the experimental evaluation, the prototype compares client- and server-side automata to identify possible inconsistencies among them. The results of this comparison for the subjects taken into consideration is shown in Table VII. For each application, the table reports the time it took to the prototype to perform differential string analysis, in milliseconds, and the number of inputs with identified inconsistencies. Specifically, column A_{C-S} shows the number of inputs for which the client side accepts strings that would be rejected by the server

side, whereas column A_{S-C} shows the opposite. For *JGossip*, for instance, the differential string analysis took around 3 seconds and identified 9 client-side inconsistencies and 2 server-side inconsistencies.

TABLE VII

RESULTS OF INCONSISTENCY IDENTIFICATION.

Subject	*Time* (ms)	A_{C-S}	A_{S-C}
JGossip	3220	9	2
Vehicle	1486	0	0
MeoDist	1745	0	0
MyAlumni	2853	141	0
Consumer	1019	7	0
Tudu	595	11	0
JcrBib	1168	45	0

The data shown in Table VII, even if can give a general idea of the performances of the prototype in terms of inconsistency detection do not show all the necessary information in order to properly evaluate the accuracy of the approach and its implementation. This is mainly due to one reason. The reason is that the approach illustrated in Section 5.2 might compute an over-approximation of the automata considered in the analysis. This possible over-approximation therefore could lead to the detection of inconsistencies which are not actual inconsistencies, false positive, or could avoid the detection of inconsistencies which actually should have been

identified as real inconsistencies. Please note that an over-approximation could also arise from possible oversimplification done in the implementation of the analysis. In order to identify if an inconsistency is a false positive it has been taken the string example generated by the prototype and used as input to the web application. False negative instead have been checked manually inspecting the code. True negative are input fields which do not show any inconsistency and they correctly do not have been reported. Table VIII shows the TP true positive, FP false positive, FN false negative and TN true negative for the inconsistency detection when calculating A_{C-S}. The results are reported for each of the web application taken into consideration. Table IX, instead, shows the same results but for the inconsistency detection when calculating A_{S-C}. The numbers given in the two tables are related to the input fields which have been analyzed. The results are positively surprising because no false positive or false negative have been detected, hence precision and recall are at their maximum value. This is because the widening operator discussed in Section 5.2 did not introduce any over-approximation and the simplifications which have been made in the implementation did not damage the analysis.

7.5 Inconsistencies Correction Evaluation

In order to evaluate the code correction mechanism proposed by the prototype all the applications have been checked in case an inconsistency was found. The technique adopted takes the string value generated in case of inconsistency detection and manually test whether the inconsistency was still present after the code correction was applied to the given web application. The results of the code correction are shown in Table X. The table shows that no inconsistencies were detected after code correction.

TABLE VIII

A_{C-S} CONFUSION MATRIX

Subject	TP	FP	FN	TN
JGossip	9	0	0	74
Vehicle	0	0	0	41
MeoDist	0	0	0	62
MyAlumni	141	0	0	0
Consumer	7	0	0	14
Tudu	11	0	0	0
JcrBib	45	0	0	0

7.6 Evaluation Conclusion

From the results of the previous tables it is possible to understand that the prototype was able to find and correct both types of inconsistencies: client checks that are more strict than server checks and vice versa. For *JGossip*, in particular, it has been found two instances of the inconsistency A_{S-C} which could lead to security vulnerabilities. Considering a common security policy for web applications which states that server-side input validation checks should always be at least as strong as the corresponding client-side checks then, cases that violate this security policy should be considered vulnerabilities. For this reason the inconsistencies of type A_{S-C} identified by this approach can be considered as actual vulnerabilities and they might be exploited by a malicious user. In case of *JGossip* these two vulnerability let attackers to force the server to send emails to invalid email addresses.

TABLE IX

A_{S-C} CONFUSION MATRIX

Subject	TP	FP	FN	TN
JGossip	2	0	0	81
Vehicle	0	0	0	41
MeoDist	0	0	0	62
MyAlumni	0	0	0	141
Consumer	0	0	0	21
Tudu	0	0	0	11
JcrBib	0	0	0	45

This could lead to cascading errors in the mail server or, in the worst situation, to a denial-of-service problem. These two inconsistencies have been further examined and their source of error is due to an inconsistency in the Struts framework inside the *commons-validator* library. For the remaining applications, four out of six contain input validation inconsistencies on the client side. A special case is that of *MyAlumni*, which has 141 inputs that are inconsistently validated at the client side. For this application, the developers provided no validation whatsoever on the client side, and thus all the 141 inputs that are checked on the server side are inconsistently validated. In addition two of the applications, *Vehicle* and *MeoDist* perform perfect validation proving that this implementation can lead also to the identification of equivalence relation between client- and serve-side input validation. Overall, these results provide clear evidence that the approach and its implementation are both practical and useful.

TABLE X

INCONSISTENCIES AFTER CODE CORRECTION.

Subject	A_{C-S}	A_{S-C}
JGossip	0	0
Vehicle	0	0
MeoDist	0	0
MyAlumni	0	0
Consumer	0	0
Tudu	0	0
JcrBib	0	0

CHAPTER 8

CONCLUSION

Two main classes of issues have been identified in relation to the input of web applications. The classes are represented by security and efficiency problems. Security problems arise because web applications store sensitive user information and, in addition, they are easily accessible. The easiness in their access make them to be a common target of attackers. In addition, some of the most insidious attacks against web applications are those that take advantage of input validation inconsistencies which let attackers to submit malicious inputs to an application, often with catastrophic consequences. Efficiency problems arise because input validation inconsistencies can create unnecessary HTTP requests and responses between the client and server part of a web application. Unfortunately, automatically checking client and server validating functions would require a complete specification of the legal inputs for an application, which is rarely available. In these thesis has been therefore detailed and implemented a novel technique that leverages differential string analysis to identify and correct potentially erroneous or insufficient validation of user inputs in a web application on both client- and server-side. The approach is based on the insight that developers typically perform redundant input validation on the client and server sides of a web application, and it is therefore possible to use the validation performed on one side as a specification for the validation performed on the other side. The approach operates by automatically extracting client- and server-side input validation functions, modeling them as deterministic finite automata, and comparing client- and server-side automata to de-

tect, report and correct inconsistencies between the two sets of checks. To assess the efficiency and effectiveness of the approach it has been implemented a prototype which aims to analyze web applications built using JEE based frameworks. Each of the aspects of the approach has been implemented from scratch or have been covered by using already existing technologies. The prototype has been then used to analyze a set of real-world web applications. In total, as reported in Chapter 7, have been evaluated seven web applications. More in detail a total of 133 web forms and 404 input fields have been analyzed. Five web applications presented inconsistencies in their input validation functions. Overall have been detected 215 inconsistencies. 2 of these inconsistencies were cases in which the server-side input validation function were accepting a wider range of values with respect to the client-side validation function. The rest of the inconsistencies were cases in which the client-side input validation function were accepting a wider range of values with respect to the server-side validation function. These results prove that the proposed approach is able to detect both type of inconsistencies. By manual verification it has been shown the absence of false positive and false negative. This result highlights the correctness in the modeling of the problem by the proposed approach. In addition, this result, prove that the prototype is able to perfectly create an equivalence relation between input validation functions in case no inconsistency is present. In fact, this is the case for the two web applications for which it has been not identified any inconsistencies. The code correction mechanism works for all the cases in which an inconsistency has been detected. The results of this initial evaluation are therefore promising and motivate further research in this direction also because the analysis was extremely fast, which demonstrates the practical appli-

cability of the approach. In addition, further research is motivated due to the fact that, up to the literature which has been explored, this is the first approach which consider client-side inconsistencies and has a code correction mechanism for the two type of inconsistencies.

8.1 Future Work

There are several possible directions that might be considered for future work. In the short term, it is possible to extend the implementation so that it can handle a larger number of types of web applications, e.g., applications written in different languages and more general applications not belonging to the frameworks which have been taken into consideration in this work. At the same time it would be positive to include, within this prototype, the tool used to extract client-side validation functions. In addition, it is possible to include part of the work done in (44) in order to not only detect inconsistencies which might lead to security vulnerabilities, but use regular expression signatures, which encode common security attacks, to detect, form the possible values of a given input field, if it is possible to generate a successful attack. If so, code could be generated to fix this type of problem as well. One more general direction for future work has to deal with the solution space for the string analysis technique on which the approach relies. There are many dimensions that characterize string analysis techniques one example of these is static vs. dynamic. In the current approach, it is considered only a specific solution in this space. In the future, it is possible to study the many trade-offs between different approaches and investigate whether a different approach may be advantageous in terms of efficiency, precision, expressiveness, or ability to compare resulting models. Finally,

it might be possible to include also non-string constraints during the analysis process to better guide the string analysis technique.

CITED LITERATURE

1. The OWASP Foundation: Top ten most critical web application vulnerabilities, 2010. http://www.owasp.org/documentation/topten.html.

2. The OWASP Foundation: Data Validation, 2010. http://www.owasp.org/index.php/Data_Validation.

3. Wikipedia: Web application. http://en.wikipedia.org/wiki/Web_application.

4. Stuttard, D. and Pinto, M.: The web application hacker's handbook: discovering and exploiting security flaws. New York, NY, USA, John Wiley & Sons, Inc., 2007.

5. Wikipedia: Vulnerability in Computing. http://en.wikipedia.org/wiki/Vulnerability_(computing).

6. The OWASP Foundation: Top 10 2007. https://www.owasp.org/index.php/Top_10_2007.

7. Wikipedia: Automata Theory. http://en.wikipedia.org/wiki/Automata_theory.

8. Wikipedia: Deterministic finite automaton. http://en.wikipedia.org/wiki/Deterministic_finite_automaton.

9. Wikipedia: Regular expression. http://en.wikipedia.org/wiki/Regular_expression.

10. Young, M. and Pezze, M.: Software Testing and Analysis: Process, Principles and Techniques. John Wiley & Sons, 2005.

11. Wikipedia: White-box testing. http://en.wikipedia.org/wiki/White-box_testing.

12. Wikipedia: Black-box testing. http://en.wikipedia.org/wiki/Black-box_testing.

13. Halfond, W., Orso, A., and Manolios, P.: WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. IEEE Transactions on Software Engineering (TSE), 34(1):65–81, 2008.

CITED LITERATURE (Continued)

14. Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., and Evans, D.: Automatically Hardening Web Applications Using Precise Tainting. In IFIP Security, 2005.

15. Richards, G., Lebresne, S., Burg, B., and Vitek, J.: An analysis of the dynamic behavior of JavaScript programs. In Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM.

16. Alkhalaf, M., Bultan, T., and Gallegos, J. L.: Verifying Client-Side Input Validation Functions Using String Analysis. In Proceedings of the 34th International Conference on Software Engineering (ICSE 2012), 2012 (to appear).

17. Saxena, P., Hanna, S., Poosankam, P., and Song, D.: FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In NDSS, 2010.

18. Xie, Y. and Aiken, A.: Static Detection of Security Vulnerabilities in Scripting Languages. In 15th USENIX Security Symposium (USENIX'06), 2006.

19. Jovanovic, N., Kruegel, C., and Kirda, E.: Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In Proceedings of the IEEE Symposium on Security and Privacy, 2006.

20. Christensen, A. S., Møller, A., and Schwartzbach, M. I.: Precise Analysis of String Expressions. In Proc. 10th International Static Analysis Symposium (SAS), volume 2694 of LNCS, pages 1–18. Springer-Verlag, June 2003.

21. Minamide, Y.: Static Approximation of Dynamically Generated Web Pages. In Proceedings of the 14th International World Wide Web Conference (WWW), pages 432–441, 2005.

22. Wassermann, G. and Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI), pages 32–41, 2007.

23. Wassermann, G. and Su, Z.: Static detection of cross-site scripting vulnerabilities. In Proceedings of the 30th International Conference on Software Engineering (ICSE), pages 171–180, 2008.

CITED LITERATURE (Continued)

24. Yu, F., Bultan, T., Cova, M., and Ibarra, O. H.: Symbolic String Verification: An Automata-Based Approach. In 15th International SPIN Workshop on Model Checking Software (SPIN), pages 306–324, 2008.

25. Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kruegel, C., Kirda, E., and Vigna, G.: Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In Proceedings of the Symposium on Security and Privacy (S and P), 2008.

26. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., and Song, D.: A Symbolic Execution Framework for JavaScript. In Proceedings of the 31st IEEE Symposium on Security and Privacy (Oakland 2010), 2010.

27. Hooimeijer, P., Livshits, B., Molnar, D., Saxena, P., and Veanes, M.: Fast and Precise Sanitizer Analysis with Bek. In Usenix Security Symposium, 2011.

28. Bisht, P., Hinrichs, T., Skrupsky, N., Bobrowicz, R., and Venkatakrishnan, V. N.: NoTamper: automatic blackbox detection of parameter tampering opportunities in web applications. In ACM Conference on Computer and Communications Security, pages 607–618, 2010.

29. Bisht, P., Hinrichs, T., Skrupsky, N., and Venkatakrishnan, V. N.: WAPTEC: whitebox analysis of web applications for parameter tampering exploit construction. In ACM Conference on Computer and Communications Security, pages 575–586, 2011.

30. The OWASP Foundation: Validation performed in client, 2010. `http://www.owasp.org/index.php/Validation\_performed\_in\_client`.

31. Oracle: Java EE. `http://www.oracle.com/technetwork/java/javaee/overview/index.html`.

32. Tip, F.: A Survey of Program Slicing Techniques. Journal of Programming Languages, 3:121–189, 1995.

33. Bartzis, C. and Bultan, T.: Widening Arithmetic Automata. In Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004), eds. R. Alur and D. Peled, volume 3114 of Lecture Notes in Computer Science, pages 321–333. Springer-Verlag, July 2004.

CITED LITERATURE (Continued)

34. Reghizzi, S. C.: Formal Languages and Compilation. Springer Publishing Company, Incorporated, 1 edition, 2009.

35. Har-Peled and Parthasarathy, M.: From DFAs/NFAs to Regular Expressions. 2009.

36. The Apache Software Foundation: Struts. `http://struts.apache.org/`.

37. Spring source community: SpringMVC. `http://www.springsource.org`.

38. Coward, N. and Yoshida, Y.: Java Servlet Specification Version 2.4. Technical report, November 2003.

39. Gargoyle Software: HtmlUnit: headless browser for testing web applications. `http://htmlunit.sourceforge.net/`.

40. Mozilla Foundation: Rhino: Javascript for Java. `http://www.mozilla.org/rhino/`.

41. McGill University: Soot: a Java Optimization Framework. `http://www.sable.mcgill.ca/soot/`.

42. Vlab: Stranger Automaton Library. `http://www.cs.ucsb.edu/~vlab/stranger/`.

43. BRICS: The MONA Project. `http://www.brics.dk/mona/`.

44. Yu, F., Alkhalaf, M., and Bultan, T.: Stranger: An Automata-Based String Analysis Tool for PHP. In TACAS, pages 154–157, 2010.

45. Arni Einarsson and Janus Dam Nielsen: A Survivor's Guide to Java Program Analysis with Soot. `http://www.sable.mcgill.ca/soot/`.